


# wxErlang

Copyright © 2009-2026 Ericsson AB. All Rights Reserved.
wxErlang 2.4.1.1
October 31, 2026

**Copyright © 2009-2026 Ericsson AB. All Rights Reserved.**
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

**October 31, 2026**

# 1 wxErlang User's Guide

The **wxErlang** application is an api for writing graphical user interfaces with wxWidgets.

## 1.1 wx the erlang binding of wxWidgets

The **wx** application is an erlang binding of **wxWidgets**. This document describes the erlang mapping to wxWidgets and it's implementation. It is not a complete users guide to wxWidgets. If you need that, you will have to read the wxWidgets documentation instead. **wx** tries to keep a one-to-one mapping with the original API so that the original documentation and examples shall be as easy as possible to use.

Wx examples and test suite can be found in the erlang src release. They can also provide some help on how to use the API.

This is currently a very brief introduction to **wx**. The application is still under development, which means the interface may change, and the test suite currently have a poor coverage ratio.

### 1.1.1 Contents


- Introduction
- Multiple processes and memory handling
- Event Handling
- Acknowledgments


### 1.1.2 Introduction

The original **wxWidgets** is an object-oriented (C++) API and that is reflected in the erlang mapping. In most cases each class in wxWidgets is represented as a module in erlang. This gives the **wx** application a huge interface, spread over several modules, and it all starts with the **wx** module. The **wx** module contains functions to create and destroy the GUI, i.e. `wx:new/0`, `wx:destroy/0`, and some other useful functions.

Objects or object references in **wx** should be seen as erlang processes rather than erlang terms. When you operate on them they can change state, e.g. they are not functional objects as erlang terms are. Each object has a type or rather a class, which is manipulated with the corresponding module or by sub-classes of that object. Type checking is done so that a module only operates on it's objects or inherited classes.

An object is created with **new** and destroyed with **destroy**. Most functions in the classes are named the same as their C++ counterpart, except that for convenience, in erlang they start with a lowercase letter and the first argument is the object reference. Optional arguments are last and expressed as tagged tuples in any order.

For example the **wxWindow** C++ class is implemented in the **wxWindow** erlang module and the member **wxWindow::CenterOnParent** is thus **wxWindow:centerOnParent**. The following C++ code:

```
wxWindow MyWin = new wxWindow();
MyWin.CenterOnParent(wxVERTICAL);
...
delete MyWin;
```

would in erlang look like:

```
MyWin = wxWindow:new(),
wxWindow:centerOnParent(MyWin, [{dir,?wxVERTICAL}]),
...
wxWindow:destroy(MyWin),
```

When you are reading wxWidgets documentation or the examples, you will notice that some of the most basic classes are missing in **wx**, they are directly mapped to corresponding erlang terms:

**wxPoint** is represented by {Xcoord,Ycoord}
**wxSize** is represented by {Width,Height}
**wxRect** is represented by {Xcoord,Ycoord,Width,Height}
**wxColour** is represented by {Red,Green,Blue[,Alpha]}
**wxString** is represented by unicode:charlist()
**wxGBPosition** is represented by {Row,Column}
**wxGBSpan** is represented by {RowSpan,ColumnSPan}
**wxGridCellCoords** is represented by {Row,Column}

In the places where the erlang API differs from the original one it should be obvious from the erlang documentation which representation has been used. E.g. the C++ arrays and/or lists are sometimes represented as erlang lists and sometimes as tuples.

Colours are represented with {Red,Green,Blue[,Alpha]}, the Alpha value is optional when used as an argument to functions, but it will always be returned from **wx** functions.

Defines, enumerations and global variables exists in `wx.hrl` as defines. Most of these defines are constants but not all. Some are platform dependent and therefore the global variables must be instantiated during runtime. These will be acquired from the driver with a call, so not all defines can be used in matching statements. Class local enumerations will be prefixed with the class name and a underscore as in `ClassName_Enum`.

Additionally some global functions, i.e. non-class functions, exist in the `wx_misc` module.

**Wx** is implemented as a (threaded) driver and a rather direct interface to the C++ API, with the drawback that if the erlang programmer does an error, it might crash the emulator.

Since the driver is threaded it requires a **smp** enabled emulator, that provides a thread safe interface to the driver.

## 1.1.3 Multiple processes and memory handling

The intention is that each erlang application calls wx:new() once to setup it's GUI which creates an environment and a memory mapping. To be able to use **wx** from several processes in your application, you must share the environment. You can get the active environment with `wx:get_env/0` and set it in the new processes with `wx:set_env/1`. Two processes or applications which have both called wx:new() will not be able use each others objects.

```
wx:new(),
MyWin = wxFrame:new(wx:null(), 42, "Example", []),
Env = wx:get_env(),
spawn(fun() ->
         wx:set_env(Env),
         %% Here you can do wx calls from your helper process.
         ...
      end),
...
```

When `wx:destroy/0` is invoked or when all processes in the application have died, the memory is deleted and all windows created by that application are closed.

The **wx** application never cleans or garbage collects memory as long as the user application is alive. Most of the objects are deleted when a window is closed, or at least all the objects which have a parent argument that is non null. By using

`wxCLASS:destroy/1` when possible you can avoid an increasing memory usage. This is especially important when **wxWidgets** assumes or recommends that you (or rather the C++ programmer) have allocated the object on the stack since that will never be done in the erlang binding. For example `wxDC` class or its sub-classes or `wxSizerFlags`.

Currently the dialogs show modal function freezes wxWidgets until the dialog is closed. That is intended but in erlang where you can have several GUI applications running at the same time it causes trouble. This will hopefully be fixed in future **wxWidgets** releases.

### 1.1.4 Event Handling

Event handling in **wx** differs most from the original API. You must specify every event you want to handle in **wxWidgets**, that is the same in the erlang binding but you can choose to receive the events as messages or handle them with callback **funs**.

Otherwise the event subscription is handled as **wxWidgets** dynamic event-handler connection. You subscribe to events of a certain type from objects with an **ID** or within a range of **ID**s. The callback **fun** is optional, if not supplied the event will be sent to the process that called **connect/2**. Thus, a handler is a callback **fun** or a process which will receive an event message.

Events are handled in order from bottom to top, in the widgets hierarchy, by the last subscribed handler first. Depending on if `wxEvent:skip()` is called the event will be handled by the other handler(s) afterwards. Most of the events have default event handler(s) installed.

Message events looks like #wx{id=integer(), obj=wx:wxObject(), userData=term(), event=Rec }. The **id** is the identifier of the object that received the event. The **obj** field contains the object that you used **connect** on. The **userData** field contains a user supplied term, this is an option to **connect**. And the **event** field contains a record with event type dependent information. The first element in the event record is always the type you subscribed to. For example if you subscribed to **key_up** events you will receive the `#wx{event=Event}` where **Event** will be a **wxKey** event record where `Event#wxKey.type = key_up`.

In **wxWidgets** the developer has to call `wxEvent:skip()` if he wants the event to be processed by other handlers. You can do the same in **wx** if you use callbacks. If you want the event as messages you just don't supply a callback and you can set the **skip** option in **connect** call to true or false, the default it is false. True means that you get the message but let the subsequent handlers also handle the event. If you want to change this behavior dynamically you must use callbacks and call `wxEvent:skip()`.

Callback event handling is done by using the optional callback **fun/2** when attaching the handler. The **fun(#wx{},wxObject()** must take two arguments where the first is the same as with message events described above and the second is an object reference to the actual event object. With the event object you can call `wxEvent:skip()` and access all the data. When using callbacks you must call `wxEvent:skip()` by yourself if you want any of the events to be forwarded to the following handlers. The actual event objects are deleted after the **fun** returns.

The callbacks are always invoked by another process and have exclusive usage of the GUI when invoked. This means that a callback **fun** cannot use the process dictionary and should not make calls to other processes. Calls to another process inside a callback **fun** may cause a deadlock if the other process is waiting on completion of his call to the GUI.

### 1.1.5 Acknowledgments

Mats-Ola Persson wrote the initial **wxWidgets** binding as part of his master thesis. The current version is a total re-write but many ideas have been reused. The reason for the re-write was mostly due to the limited requirements he had been given by us.

Also thanks to the **wxWidgets** team that develops and supports it so we have something to use.

# 2 Reference Manual

The **wxErlang** application is an api for writing graphical user interfaces with wxWidgets.

# wx

Erlang module

A port of **wxWidgets**.

This is the base api of **wxWidgets**. This module contains functions for starting and stopping the wx-server, as well as other utility functions.

wxWidgets is object oriented, and not functional. Thus, in Wx a module represents a class, and the object created by this class has an own type, wxCLASS(). This module represents the base class, and all other wxMODULE's are sub-classes of this class.

Objects of a class are created with wxCLASS:new(...) and destroyed with wxCLASS:destroy(). Member functions are called with wxCLASS:member(Object, ...) instead of as in C++ Object.member(...).

Sub class modules inherit (non static) functions from their parents. The inherited functions are not documented in the sub-classes.

This erlang port of wxWidgets tries to be a one-to-one mapping with the original wxWidgets library. Some things are different though, as the optional arguments use property lists and can be in any order. The main difference is the event handling which is different from the original library. See wxEvtHandler.

The following classes are implemented directly as erlang types: wxPoint={x,y},wxSize={w,h},wxRect={x,y,w,h},wxColour={r,g,b [,a]}, wxString=unicode:chardata(), wxGBPosition={r,c},wxGBSpan={rs,cs},wxGridCellCoords={r,c}.

wxWidgets uses a process specific environment, which is created by wx:new/0. To be able to use the environment from other processes, call get_env/0 to retrieve the environment and set_env/1 to assign the environment in the other process.

Global (classless) functions are located in the wx_misc module.

## DATA TYPES

wx_colour() = {R::byte(), G::byte(), B::byte()} | wx_colour4()

wx_colour4() = {R::byte(), G::byte(), B::byte(), A::byte()}

wx_datetime() = {{Year::integer(), Month::integer(), Day::integer()}, {Hour::integer(), Minute::integer(), Second::integer()}}

In Local Timezone

wx_enum() = integer()

Constant defined in wx.hrl

wx_env() = #wx_env{}

Opaque process environment

wx_memory() = binary() | #wx_mem{}

Opaque memory reference

wx_object() = #wx_ref{}

Opaque object reference

wx_wxHtmlLinkInfo() = #wxHtmlLinkInfo{href=unicode:chardata(), target=unicode:chardata()}

wx_wxMouseState() = #wxMouseState{x=integer(), y=integer(), leftDown=boolean(), middleDown=boolean(), rightDown=boolean(), controlDown=boolean(), shiftDown=boolean(), altDown=boolean(), metaDown=boolean(), cmdDown=boolean()}

See #wxMouseState{} defined in wx.hrl

## Exports

```
parent_class(X1) -> term()
```

```
new() -> wx_object()
```

Starts a wx server.

```
new(Options::[Option]) -> wx_object()
```

Types:

**`Option = {debug, list() | atom()} | {silent_start, boolean()}`**

Starts a wx server. Option may be {debug, Level}, see debug/1. Or {silent_start, Bool}, which causes error messages at startup to be suppressed. The latter can be used as a silent test of whether wx is properly installed or not.

```
destroy() -> ok
```

Stops a wx server.

```
get_env() -> wx_env()
```

Gets this process's current wx environment. Can be sent to other processes to allow them use this process wx environment.

**See also:** set_env/1.

```
set_env(Wx_env::wx_env()) -> ok
```

Sets the process wx environment, allows this process to use another process wx environment.

```
subscribe_events() -> ok
```

Adds the calling process to the list of of processes that are listening to wx application events.

At the moment these are all MacOSX specific events corresponding to `MacNewFile()` and friends from wxWidgets **wxApp**:

- `{new_file, ""}`
- `{open_file, Filename}`
- `{print_file, Filename}`
- `{open_url, Url}`
- `{reopen_app, ""}`

The call always returns ok but will have sent any already received events to the calling process.

```
null() -> wx_object()
```

Returns the null object

```
is_null(Wx_ref::wx_object()) -> boolean()
```

Returns true if object is null, false otherwise

```
equal(Wx_ref::wx_object(), X2::wx_object()) -> boolean()
```

Returns true if both arguments references the same object, false otherwise

```
getObjectType(Wx_ref::wx_object()) -> atom()
```

Returns the object type

```
typeCast(Old::wx_object(), NewType::atom()) -> wx_object()
```

Casts the object to class NewType. It is needed when using functions like wxWindow:findWindow/2, which returns a generic wxObject type.

```
batch(Fun::function()) -> term()
```

Batches all `wx` commands used in the fun. Improves performance of the command processing by grabbing the wxWidgets thread so that no event processing will be done before the complete batch of commands is invoked.

**See also:** foldl/3, foldr/3, foreach/2, map/2.

```
foreach(Fun::function(), List::list()) -> ok
```

Behaves like lists:foreach/2 but batches wx commands. See batch/1.

```
map(Fun::function(), List::list()) -> list()
```

Behaves like lists:map/2 but batches wx commands. See batch/1.

```
foldl(Fun::function(), Acc::term(), List::list()) -> term()
```

Behaves like lists:foldl/3 but batches wx commands. See batch/1.

```
foldr(Fun::function(), Acc::term(), List::list()) -> term()
```

Behaves like lists:foldr/3 but batches wx commands. See batch/1.

```
create_memory(Size::integer()) -> wx_memory()
```

Creates a memory area (of Size in bytes) which can be used by an external library (i.e. opengl). It is up to the client to keep a reference to this object so it does not get garbage collected by erlang while still in use by the external library.

This is far from erlang's intentional usage and can crash the erlang emulator. Use it carefully.

```
get_memory_bin(Wx_mem::wx_memory()) -> binary()
```

Returns the memory area as a binary.

```
retain_memory(Wx_mem::wx_memory()) -> ok
```

Saves the memory from deletion until release_memory/1 is called. If release_memory/1 is not called the memory will not be garbage collected.

```
release_memory(Wx_mem::wx_memory()) -> ok
```

```
debug(Debug::Level | [Level]) -> ok
```

Types:

**`Level = none | verbose | trace | driver | integer()`**

Sets debug level. If debug level is 'verbose' or 'trace' each call is printed on console. If Level is 'driver' each allocated object and deletion is printed on the console.

# wx_object

Erlang module

wx_object - Generic wx object behaviour

This is a behaviour module that can be used for "sub classing" wx objects. It works like a regular gen_server module and creates a server per object.

NOTE: Currently no form of inheritance is implemented.

The user module should export:

init(Args) should return
{wxWindow, State} | {wxWindow, State, Timeout} | ignore | {stop, Reason}

Asynchronous window event handling:
handle_event(#wx{}, State) should return
{noreply, State} | {noreply, State, Timeout} | {stop, Reason, State}

The user module can export the following callback functions:

handle_call(Msg, {From, Tag}, State) should return
{reply, Reply, State} | {reply, Reply, State, Timeout} | {noreply, State} | {noreply, State, Timeout} | {stop, Reason, Reply, State}

handle_cast(Msg, State) should return
{noreply, State} | {noreply, State, Timeout} | {stop, Reason, State}

If the above are not exported but called, the wx_object process will crash. The user module can also export:

Info is message e.g. {'EXIT', P, R}, {nodedown, N}, ...
handle_info(Info, State) should return , ...
{noreply, State} | {noreply, State, Timeout} | {stop, Reason, State}

If a message is sent to the wx_object process when handle_info is not exported, the message will be dropped and ignored.

When stop is returned in one of the functions above with Reason = normal | shutdown | Term, terminate(State) is called. It lets the user module clean up, it is always called when server terminates or when wx_object() in the driver is deleted. If the Parent process terminates the Module:terminate/2 function is called.
terminate(Reason, State)

Example:

```
-module(myDialog).
-export([new/2, show/1, destroy/1]).  %% API
-export([init/1, handle_call/3, handle_event/2,
         handle_info/2, code_change/3, terminate/2]).
         new/2, showModal/1, destroy/1]).  %% Callbacks

%% Client API
new(Parent, Msg) ->
   wx_object:start(?MODULE, [Parent,Id], []).

show(Dialog) ->
   wx_object:call(Dialog, show_modal).

destroy(Dialog) ->
   wx_object:call(Dialog, destroy).

%% Server Implementation ala gen_server
init([Parent, Str]) ->
   Dialog = wxDialog:new(Parent, 42, "Testing", []),
   ...
   wxDialog:connect(Dialog, command_button_clicked),
   {Dialog, MyState}.

handle_call(show, _From, State) ->
   wxDialog:show(State#state.win),
   {reply, ok, State};
...
handle_event(#wx{}, State) ->
   io:format("Users clicked button~n",[]),
   {noreply, State};
...
```

## DATA TYPES

request_id() = term()

server_ref() = wx:wx_object() | atom() | pid()

## Exports

`start(Name, Mod, Args, Options) -> wxWindow:wxWindow() | {error, term()}`

Types:

**Name = {local, atom()}**
**Mod = atom()**
**Args = term()**
**Flag = trace | log | {logfile, string()} | statistics | debug**
**Options = [{timeout, timeout()} | {debug, [Flag]}]**

Starts a generic wx_object server and invokes Mod:init(Args) in the new process.

`start_link(Mod, Args, Options) -> wxWindow:wxWindow() | {error, term()}`

Types:

**Mod = atom()**
**Args = term()**
**Flag = trace | log | {logfile, string()} | statistics | debug**

**Options = [{timeout, timeout()} | {debug, [Flag]}]**

Starts a generic wx_object server and invokes Mod:init(Args) in the new process.

```
start_link(Name, Mod, Args, Options) -> wxWindow:wxWindow() | {error, term()}
```

Types:

**Name = {local, atom()}**
**Mod = atom()**
**Args = term()**
**Flag = trace | log | {logfile, string()} | statistics | debug**
**Options = [{timeout, timeout()} | {debug, [Flag]}]**

Starts a generic wx_object server and invokes Mod:init(Args) in the new process.

```
stop(Obj) -> ok
```

Types:

**Obj = wx:wx_object() | atom() | pid()**

Stops a generic wx_object server with reason 'normal'. Invokes terminate(Reason,State) in the server. The call waits until the process is terminated. If the process does not exist, an exception is raised.

```
stop(Obj, Reason, Timeout) -> ok
```

Types:

**Obj = wx:wx_object() | atom() | pid()**
**Reason = term()**
**Timeout = timeout()**

Stops a generic wx_object server with the given Reason. Invokes terminate(Reason,State) in the server. The call waits until the process is terminated. If the call times out, or if the process does not exist, an exception is raised.

```
call(Obj, Request) -> term()
```

Types:

**Obj = wx:wx_object() | atom() | pid()**
**Request = term()**

Make a call to a wx_object server. The call waits until it gets a result. Invokes handle_call(Request, From, State) in the server

```
call(Obj, Request, Timeout) -> term()
```

Types:

**Obj = wx:wx_object() | atom() | pid()**
**Request = term()**
**Timeout = integer()**

Make a call to a wx_object server with a timeout. Invokes handle_call(Request, From, State) in server

```
send_request(Obj, Request::term()) -> request_id()
```

Types:

**Obj = wx:wx_object() | atom() | pid()**

Make an send_request to a generic server. and return a RequestId which can/should be used with wait_response/[1|2]. Invokes handle_call(Request, From, State) in server.

```
wait_response(RequestId::request_id()) -> {reply, Reply::term()} | {error,
{term(), server_ref()}}
```

Wait infinitely for a reply from a generic server.

```
wait_response(Key::request_id(), Timeout::timeout()) -> {reply,
Reply::term()} | timeout | {error, {term(), server_ref()}}
```

Wait 'timeout' for a reply from a generic server.

```
check_response(Msg::term(), Key::request_id()) -> {reply, Reply::term()} |
false | {error, {term(), server_ref()}}
```

Check if a received message was a reply to a RequestId

```
cast(Obj, Request) -> ok
```

Types:

**Obj = wx:wx_object() | atom() | pid()**

**Request = term()**

Make a cast to a wx_object server. Invokes handle_cast(Request, State) in the server

```
get_pid(Obj) -> pid()
```

Types:

**Obj = wx:wx_object() | atom() | pid()**

Get the pid of the object handle.

```
set_pid(Obj, Pid::pid()) -> wx:wx_object()
```

Types:

**Obj = wx:wx_object() | atom() | pid()**

Sets the controlling process of the object handle.

```
reply(X1::{pid(), Tag::term()}, Reply::term()) -> pid()
```

Get the pid of the object handle.

# wxAcceleratorEntry

Erlang module

An object used by an application wishing to create an accelerator table (see `wxAcceleratorTable`).

See: `wxAcceleratorTable`, `wxWindow:setAcceleratorTable/2`

wxWidgets docs: **wxAcceleratorEntry**

## Data Types

```
wxAcceleratorEntry() = wx:wx_object()
```

## Exports

```
new() -> wxAcceleratorEntry()
```

```
new(Options :: [Option]) -> wxAcceleratorEntry()
new(Entry) -> wxAcceleratorEntry()
```

Types:

```
Entry = wxAcceleratorEntry()
```

Copy ctor.

```
getCommand(This) -> integer()
```

Types:

```
This = wxAcceleratorEntry()
```

Returns the command identifier for the accelerator table entry.

```
getFlags(This) -> integer()
```

Types:

```
This = wxAcceleratorEntry()
```

Returns the flags for the accelerator table entry.

```
getKeyCode(This) -> integer()
```

Types:

```
This = wxAcceleratorEntry()
```

Returns the keycode for the accelerator table entry.

```
set(This, Flags, KeyCode, Cmd) -> ok
```

Types:

```
This = wxAcceleratorEntry()
Flags = KeyCode = Cmd = integer()
```

```
set(This, Flags, KeyCode, Cmd, Options :: [Option]) -> ok
```

Types:

```
This = wxAcceleratorEntry()
Flags = KeyCode = Cmd = integer()
Option = {item, wxMenuItem:wxMenuItem()}
```

Sets the accelerator entry parameters.

```
destroy(This :: wxAcceleratorEntry()) -> ok
```

Destroys the object.

# wxAcceleratorTable

Erlang module

An accelerator table allows the application to specify a table of keyboard shortcuts for menu or button commands.

The object ?wxNullAcceleratorTable is defined to be a table with no data, and is the initial accelerator table for a window.

Example:

Remark: An accelerator takes precedence over normal processing and can be a convenient way to program some event handling. For example, you can use an accelerator table to enable a dialog with a multi-line text control to accept CTRL-Enter as meaning 'OK'.

Predefined objects (include wx.hrl): ?wxNullAcceleratorTable

See: `wxAcceleratorEntry`, `wxWindow:setAcceleratorTable/2`

wxWidgets docs: **wxAcceleratorTable**

## Data Types

```
wxAcceleratorTable() = wx:wx_object()
```

## Exports

```
new() -> wxAcceleratorTable()
```

Default ctor.

```
new(N, Entries) -> wxAcceleratorTable()
```

Types:

```
N = integer()
Entries = [wxAcceleratorEntry:wxAcceleratorEntry()]
```

Initializes the accelerator table from an array of `wxAcceleratorEntry`.

```
destroy(This :: wxAcceleratorTable()) -> ok
```

Destroys the `wxAcceleratorTable` object.

See overview_refcount_destruct for more info.

```
ok(This) -> boolean()
```

Types:

```
This = wxAcceleratorTable()
```

See: `isOk/1`.

```
isOk(This) -> boolean()
```

Types:

```
This = wxAcceleratorTable()
```

Returns true if the accelerator table is valid.

# wxActivateEvent

Erlang module

An activate event is sent when a window or application is being activated or deactivated.

Note: Until wxWidgets 3.1.0 activation events could be sent by wxMSW when the window was minimized. This reflected the native MSW behaviour but was often surprising and unexpected, so starting from 3.1.0 such events are not sent any more when the window is in the minimized state.

See: **Overview events**, `wxApp::IsActive` (not implemented in wx)

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxActivateEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxActivateEventType` to subscribe to events of this type.

## Data Types

```
wxActivateEvent() = wx:wx_object()
wxActivate() =
    #wxActivate{type = wxActivateEvent:wxActivateEventType(),
                active = boolean()}
wxActivateEventType() = activate | activate_app | hibernate
```

## Exports

```
getActive(This) -> boolean()
```

Types:

```
   This = wxActivateEvent()
```

Returns true if the application or window is being activated, false otherwise.

# wxArtProvider

Erlang module

`wxArtProvider` class is used to customize the look of wxWidgets application.

When wxWidgets needs to display an icon or a bitmap (e.g. in the standard file dialog), it does not use a hard-coded resource but asks `wxArtProvider` for it instead. This way users can plug in their own `wxArtProvider` class and easily replace standard art with their own version.

All that is needed is to derive a class from `wxArtProvider`, override either its `wxArtProvider::CreateBitmap()` (not implemented in wx) and/or its `wxArtProvider::CreateIconBundle()` (not implemented in wx) methods and register the provider with `wxArtProvider::Push()` (not implemented in wx):

If you need bitmap images (of the same artwork) that should be displayed at different sizes you should probably consider overriding `wxArtProvider::CreateIconBundle` (not implemented in wx) and supplying icon bundles that contain different bitmap sizes.

There's another way of taking advantage of this class: you can use it in your code and use platform native icons as provided by `getBitmap/2` or `getIcon/2`.

Identifying art resources

Every bitmap and icon bundle are known to `wxArtProvider` under an unique ID that is used when requesting a resource from it. The ID is represented by the ?wxArtID type and can have one of these predefined values (you can see bitmaps represented by these constants in the page_samples_artprov):

Additionally, any string recognized by custom art providers registered using `wxArtProvider::Push` (not implemented in wx) may be used.

Note: When running under GTK+ 2, GTK+ stock item IDs (e.g. `"gtk-cdrom"`) may be used as well: For a list of the GTK+ stock items please refer to the **GTK+ documentation page**. It is also possible to load icons from the current icon theme by specifying their name (without extension and directory components). Icon themes recognized by GTK+ follow the freedesktop.org **Icon Themes specification**. Note that themes are not guaranteed to contain all icons, so `wxArtProvider` may return ?wxNullBitmap or ?wxNullIcon. The default theme is typically installed in `/usr/share/icons/hicolor`.

Clients

The `client` is the entity that calls `wxArtProvider`'s `getBitmap/2` or `getIcon/2` function. It is represented by wxClientID type and can have one of these values:

Client ID serve as a hint to `wxArtProvider` that is supposed to help it to choose the best looking bitmap. For example it is often desirable to use slightly different icons in menus and toolbars even though they represent the same action (e.g. wxART_FILE_OPEN). Remember that this is really only a hint for `wxArtProvider` - it is common that `getBitmap/2` returns identical bitmap for different client values!

See: **Examples**, `wxArtProvider`, usage; stock ID list

wxWidgets docs: **wxArtProvider**

## Data Types

```
wxArtProvider() = wx:wx_object()
```

## Exports

```
getBitmap(Id) -> wxBitmap:wxBitmap()
```

Types:

```
   Id = unicode:chardata()
```

```
getBitmap(Id, Options :: [Option]) -> wxBitmap:wxBitmap()
```

Types:

```
   Id = unicode:chardata()
   Option =
       {client, unicode:chardata()} |
       {size, {W :: integer(), H :: integer()}}
```

Query registered providers for bitmap with given ID.

Return: The bitmap if one of registered providers recognizes the ID or wxNullBitmap otherwise.

```
getIcon(Id) -> wxIcon:wxIcon()
```

Types:

```
   Id = unicode:chardata()
```

```
getIcon(Id, Options :: [Option]) -> wxIcon:wxIcon()
```

Types:

```
   Id = unicode:chardata()
   Option =
       {client, unicode:chardata()} |
       {size, {W :: integer(), H :: integer()}}
```

Same as `getBitmap/2`, but return a `wxIcon` object (or ?wxNullIcon on failure).

# wxAuiDockArt

Erlang module

`wxAuiDockArt` is part of the wxAUI class framework. See also overview_aui.

`wxAuiDockArt` is the art provider: provides all drawing functionality to the wxAui dock manager. This allows the dock manager to have a pluggable look-and-feel.

By default, a `wxAuiManager` uses an instance of this class called `wxAuiDefaultDockArt` (not implemented in wx) which provides bitmap art and a colour scheme that is adapted to the major platforms' look. You can either derive from that class to alter its behaviour or write a completely new dock art class. Call `wxAuiManager:setArtProvider/2` to force wxAUI to use your new dock art provider.

See: `wxAuiManager`, `wxAuiPaneInfo`

wxWidgets docs: **wxAuiDockArt**

## Data Types

```
wxAuiDockArt() = wx:wx_object()
```

## Exports

```
getColour(This, Id) -> wx:wx_colour4()
```

Types:

```
This = wxAuiDockArt()
Id = integer()
```

Get the colour of a certain setting.

`id` can be one of the colour values of `wxAuiPaneDockArtSetting`.

```
getFont(This, Id) -> wxFont:wxFont()
```

Types:

```
This = wxAuiDockArt()
Id = integer()
```

Get a font setting.

```
getMetric(This, Id) -> integer()
```

Types:

```
This = wxAuiDockArt()
Id = integer()
```

Get the value of a certain setting.

`id` can be one of the size values of `wxAuiPaneDockArtSetting`.

```
setColour(This, Id, Colour) -> ok
```

Types:

```
This = wxAuiDockArt()
Id = integer()
Colour = wx:wx_colour()
```

Set a certain setting with the value `colour`.

`id` can be one of the colour values of `wxAuiPaneDockArtSetting`.

`setFont(This, Id, Font) -> ok`

Types:

```
This = wxAuiDockArt()
Id = integer()
Font = wxFont:wxFont()
```

Set a font setting.

`setMetric(This, Id, New_val) -> ok`

Types:

```
This = wxAuiDockArt()
Id = New_val = integer()
```

Set a certain setting with the value `new_val`.

`id` can be one of the size values of `wxAuiPaneDockArtSetting`.

# wxAuiManager

Erlang module

`wxAuiManager` is the central class of the wxAUI class framework.

`wxAuiManager` manages the panes associated with it for a particular `wxFrame`, using a pane's `wxAuiPaneInfo` information to determine each pane's docking and floating behaviour.

`wxAuiManager` uses wxWidgets' sizer mechanism to plan the layout of each frame. It uses a replaceable dock art class to do all drawing, so all drawing is localized in one area, and may be customized depending on an application's specific needs.

`wxAuiManager` works as follows: the programmer adds panes to the class, or makes changes to existing pane properties (dock position, floating state, show state, etc.). To apply these changes, `wxAuiManager`'s `update/1` function is called. This batch processing can be used to avoid flicker, by modifying more than one pane at a time, and then "committing" all of the changes at once by calling `update/1`.

Panes can be added quite easily:

Later on, the positions can be modified easily. The following will float an existing pane in a tool window:

Layers, Rows and Directions, Positions

Inside wxAUI, the docking layout is figured out by checking several pane parameters. Four of these are important for determining where a pane will end up:

Styles

This class supports the following styles:

See: **Overview aui**, `wxAuiNotebook`, `wxAuiDockArt`, `wxAuiPaneInfo`

This class is derived (and can use functions) from: `wxEvtHandler`

wxWidgets docs: **wxAuiManager**

## Events

Event types emitted from this class: `aui_pane_button`, `aui_pane_close`, `aui_pane_maximize`, `aui_pane_restore`, `aui_pane_activated`, `aui_render`

## Data Types

```
wxAuiManager() = wx:wx_object()
```

## Exports

```
new() -> wxAuiManager()

new(Options :: [Option]) -> wxAuiManager()
```

Types:

```
   Option =
       {managed_wnd, wxWindow:wxWindow()} | {flags, integer()}
```

Constructor.

```
destroy(This :: wxAuiManager()) -> ok
```

Dtor.

```
addPane(This, Window) -> boolean()
```

Types:

```
   This = wxAuiManager()
   Window = wxWindow:wxWindow()
```

```
addPane(This, Window, Options :: [Option]) -> boolean()
addPane(This, Window, Pane_info) -> boolean()
```

Types:

```
   This = wxAuiManager()
   Window = wxWindow:wxWindow()
   Pane_info = wxAuiPaneInfo:wxAuiPaneInfo()
```

`addPane/4` tells the frame manager to start managing a child window.

There are several versions of this function. The first version allows the full spectrum of pane parameter possibilities. The second version is used for simpler user interfaces which do not require as much configuration. The last version allows a drop position to be specified, which will determine where the pane will be added.

```
addPane(This, Window, Pane_info, Drop_pos) -> boolean()
```

Types:

```
   This = wxAuiManager()
   Window = wxWindow:wxWindow()
   Pane_info = wxAuiPaneInfo:wxAuiPaneInfo()
   Drop_pos = {X :: integer(), Y :: integer()}
```

```
detachPane(This, Window) -> boolean()
```

Types:

```
   This = wxAuiManager()
   Window = wxWindow:wxWindow()
```

Tells the `wxAuiManager` to stop managing the pane specified by window.

The window, if in a floated frame, is reparented to the frame managed by `wxAuiManager`.

```
getAllPanes(This) -> [wxAuiPaneInfo:wxAuiPaneInfo()]
```

Types:

```
   This = wxAuiManager()
```

Returns an array of all panes managed by the frame manager.

```
getArtProvider(This) -> wxAuiDockArt:wxAuiDockArt()
```

Types:

```
   This = wxAuiManager()
```

Returns the current art provider being used.

See: `wxAuiDockArt`

```
getDockSizeConstraint(This) ->
                         {Widthpct :: number(),
                          Heightpct :: number()}
```

Types:

```
This = wxAuiManager()
```

Returns the current dock constraint values.

See `setDockSizeConstraint/3` for more information.

```
getFlags(This) -> integer()
```

Types:

```
This = wxAuiManager()
```

Returns the current ?wxAuiManagerOption's flags.

```
getManagedWindow(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxAuiManager()
```

Returns the frame currently being managed by `wxAuiManager`.

```
getManager(Window) -> wxAuiManager()
```

Types:

```
Window = wxWindow:wxWindow()
```

Calling this method will return the `wxAuiManager` for a given window.

The `window` parameter should specify any child window or sub-child window of the frame or window managed by `wxAuiManager`.

The `window` parameter need not be managed by the manager itself, nor does it even need to be a child or sub-child of a managed window. It must however be inside the window hierarchy underneath the managed window.

```
getPane(This, Name) -> wxAuiPaneInfo:wxAuiPaneInfo()
getPane(This, Window) -> wxAuiPaneInfo:wxAuiPaneInfo()
```

Types:

```
This = wxAuiManager()
Window = wxWindow:wxWindow()
```

`getPane/2` is used to lookup a `wxAuiPaneInfo` object either by window pointer or by pane name, which acts as a unique id for a window pane.

The returned `wxAuiPaneInfo` object may then be modified to change a pane's look, state or position. After one or more modifications to `wxAuiPaneInfo`, `update/1` should be called to commit the changes to the user interface. If the lookup failed (meaning the pane could not be found in the manager), a call to the returned `wxAuiPaneInfo`'s IsOk() method will return false.

```
hideHint(This) -> ok
```

Types:

```
This = wxAuiManager()
```

`hideHint/1` hides any docking hint that may be visible.

```
insertPane(This, Window, Insert_location) -> boolean()
```

Types:

```
   This = wxAuiManager()
   Window = wxWindow:wxWindow()
   Insert_location = wxAuiPaneInfo:wxAuiPaneInfo()
```

```
insertPane(This, Window, Insert_location, Options :: [Option]) ->
              boolean()
```

Types:

```
   This = wxAuiManager()
   Window = wxWindow:wxWindow()
   Insert_location = wxAuiPaneInfo:wxAuiPaneInfo()
   Option = {insert_level, integer()}
```

This method is used to insert either a previously unmanaged pane window into the frame manager, or to insert a currently managed pane somewhere else.

`insertPane/4` will push all panes, rows, or docks aside and insert the window into the position specified by `insert_location`.

Because `insert_location` can specify either a pane, dock row, or dock layer, the `insert_level` parameter is used to disambiguate this. The parameter `insert_level` can take a value of wxAUI_INSERT_PANE, wxAUI_INSERT_ROW or wxAUI_INSERT_DOCK.

```
loadPaneInfo(This, Pane_part, Pane) -> ok
```

Types:

```
   This = wxAuiManager()
   Pane_part = unicode:chardata()
   Pane = wxAuiPaneInfo:wxAuiPaneInfo()
```

`loadPaneInfo/3` is similar to LoadPerspective, with the exception that it only loads information about a single pane.

This method writes the serialized data into the passed pane. Pointers to UI elements are not modified.

Note: This operation also changes the name in the pane information!

See: `loadPerspective/3`

See: `savePaneInfo/2`

See: `savePerspective/1`

```
loadPerspective(This, Perspective) -> boolean()
```

Types:

```
   This = wxAuiManager()
   Perspective = unicode:chardata()
```

```
loadPerspective(This, Perspective, Options :: [Option]) ->
                  boolean()
```

Types:

```
This = wxAuiManager()
Perspective = unicode:chardata()
Option = {update, boolean()}
```

Loads a saved perspective.

A perspective is the layout state of an AUI managed window.

All currently existing panes that have an object in "perspective" with the same name ("equivalent") will receive the layout parameters of the object in "perspective". Existing panes that do not have an equivalent in "perspective" remain unchanged, objects in "perspective" having no equivalent in the manager are ignored.

See: `loadPaneInfo/3`

See: `loadPerspective/3`

See: `savePerspective/1`

```
savePaneInfo(This, Pane) -> unicode:charlist()
```

Types:

```
This = wxAuiManager()
Pane = wxAuiPaneInfo:wxAuiPaneInfo()
```

`savePaneInfo/2` is similar to SavePerspective, with the exception that it only saves information about a single pane.

Return: The serialized layout parameters of the pane are returned within the string. Information about the pointers to UI elements stored in the pane are not serialized.

See: `loadPaneInfo/3`

See: `loadPerspective/3`

See: `savePerspective/1`

```
savePerspective(This) -> unicode:charlist()
```

Types:

```
This = wxAuiManager()
```

Saves the entire user interface layout into an encoded `wxString` (not implemented in wx), which can then be stored by the application (probably using wxConfig).

See: `loadPerspective/3`

See: `loadPaneInfo/3`

See: `savePaneInfo/2`

```
setArtProvider(This, Art_provider) -> ok
```

Types:

```
This = wxAuiManager()
Art_provider = wxAuiDockArt:wxAuiDockArt()
```

Instructs `wxAuiManager` to use art provider specified by parameter `art_provider` for all drawing calls.

This allows pluggable look-and-feel features. The previous art provider object, if any, will be deleted by `wxAuiManager`.

See: `wxAuiDockArt`

```
setDockSizeConstraint(This, Widthpct, Heightpct) -> ok
```

Types:

```
This = wxAuiManager()
Widthpct = Heightpct = number()
```

When a user creates a new dock by dragging a window into a docked position, often times the large size of the window will create a dock that is unwieldy large.

`wxAuiManager` by default limits the size of any new dock to 1/3 of the window size. For horizontal docks, this would be 1/3 of the window height. For vertical docks, 1/3 of the width.

Calling this function will adjust this constraint value. The numbers must be between 0.0 and 1.0. For instance, calling SetDockSizeContraint with 0.5, 0.5 will cause new docks to be limited to half of the size of the entire managed window.

```
setFlags(This, Flags) -> ok
```

Types:

```
This = wxAuiManager()
Flags = integer()
```

This method is used to specify ?wxAuiManagerOption's flags.

`flags` specifies options which allow the frame management behaviour to be modified.

```
setManagedWindow(This, Managed_wnd) -> ok
```

Types:

```
This = wxAuiManager()
Managed_wnd = wxWindow:wxWindow()
```

Called to specify the frame or window which is to be managed by `wxAuiManager`.

Frame management is not restricted to just frames. Child windows or custom controls are also allowed.

```
showHint(This, Rect) -> ok
```

Types:

```
This = wxAuiManager()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
```

This function is used by controls to explicitly show a hint window at the specified rectangle.

It is rarely called, and is mostly used by controls implementing custom pane drag/drop behaviour. The specified rectangle should be in screen coordinates.

```
unInit(This) -> ok
```

Types:

```
This = wxAuiManager()
```

Dissociate the managed window from the manager.

This function may be called before the managed frame or window is destroyed, but, since wxWidgets 3.1.4, it's unnecessary to call it explicitly, as it will be called automatically when this window is destroyed, as well as when the manager itself is.

```
update(This) -> ok
```

Types:

```
This = wxAuiManager()
```

This method is called after any number of changes are made to any of the managed panes.

`update/1` must be invoked after `addPane/4` or `insertPane/4` are called in order to "realize" or "commit" the changes. In addition, any number of changes may be made to `wxAuiPaneInfo` structures (retrieved with `getPane/2`), but to realize the changes, `update/1` must be called. This construction allows pane flicker to be avoided by updating the whole layout at one time.

# wxAuiManagerEvent

Erlang module

Event used to indicate various actions taken with `wxAuiManager`.

See `wxAuiManager` for available event types.

See: `wxAuiManager`, `wxAuiPaneInfo`

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxAuiManagerEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxAuiManagerEventType` to subscribe to events of this type.

## Data Types

```
wxAuiManagerEvent() = wx:wx_object()
wxAuiManager() =
    #wxAuiManager{type =
                      wxAuiManagerEvent:wxAuiManagerEventType(),
                  manager = wxAuiManager:wxAuiManager(),
                  pane = wxAuiPaneInfo:wxAuiPaneInfo(),
                  button = integer(),
                  veto_flag = boolean(),
                  canveto_flag = boolean(),
                  dc = wxDC:wxDC()}
wxAuiManagerEventType() =
    aui_pane_button | aui_pane_close | aui_pane_maximize |
    aui_pane_restore | aui_pane_activated | aui_render |
    aui_find_manager
```

## Exports

```
setManager(This, Manager) -> ok
```

Types:

```
   This = wxAuiManagerEvent()
   Manager = wxAuiManager:wxAuiManager()
```

Sets the `wxAuiManager` this event is associated with.

```
getManager(This) -> wxAuiManager:wxAuiManager()
```

Types:

```
   This = wxAuiManagerEvent()
```

Return: The `wxAuiManager` this event is associated with.

```
setPane(This, Pane) -> ok
```

Types:

```
   This = wxAuiManagerEvent()
   Pane = wxAuiPaneInfo:wxAuiPaneInfo()
```

Sets the pane this event is associated with.

```
getPane(This) -> wxAuiPaneInfo:wxAuiPaneInfo()
```

Types:

```
   This = wxAuiManagerEvent()
```

Return: The pane this event is associated with.

```
setButton(This, Button) -> ok
```

Types:

```
   This = wxAuiManagerEvent()
   Button = integer()
```

Sets the ID of the button clicked that triggered this event.

```
getButton(This) -> integer()
```

Types:

```
   This = wxAuiManagerEvent()
```

Return: The ID of the button that was clicked.

```
setDC(This, Pdc) -> ok
```

Types:

```
   This = wxAuiManagerEvent()
   Pdc = wxDC:wxDC()
```

```
getDC(This) -> wxDC:wxDC()
```

Types:

```
   This = wxAuiManagerEvent()
```

```
veto(This) -> ok
```

Types:

```
   This = wxAuiManagerEvent()
```

```
veto(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxAuiManagerEvent()
   Option = {veto, boolean()}
```

Cancels the action indicated by this event if `canVeto/1` is true.

```
getVeto(This) -> boolean()
```

Types:

```
   This = wxAuiManagerEvent()
```

Return: true if this event was vetoed.

See: veto/2

setCanVeto(This, Can_veto) -> ok

Types:

```
This = wxAuiManagerEvent()
Can_veto = boolean()
```

Sets whether or not this event can be vetoed.

canVeto(This) -> boolean()

Types:

```
This = wxAuiManagerEvent()
```

Return: true if this event can be vetoed.

See: veto/2

# wxAuiNotebook

Erlang module

wxAuiNotebook is part of the wxAUI class framework, which represents a notebook control, managing multiple windows with associated tabs.

See also overview_aui.

wxAuiNotebook is a notebook control which implements many features common in applications with dockable panes. Specifically, wxAuiNotebook implements functionality which allows the user to rearrange tab order via drag-and-drop, split the tab window into many different splitter configurations, and toggle through different themes to customize the control's look and feel.

The default theme that is used is wxAuiDefaultTabArt (not implemented in wx), which provides a modern, glossy look and feel. The theme can be changed by calling setArtProvider/2.

Styles

This class supports the following styles:

This class is derived (and can use functions) from: wxControl wxWindow wxEvtHandler

wxWidgets docs: **wxAuiNotebook**

## Events

Event types emitted from this class: command_auinotebook_page_close, command_auinotebook_page_closed, command_auinotebook_page_changed, command_auinotebook_page_changing, command_auinotebook_button, command_auinotebook_begin_drag, command_auinotebook_end_drag, command_auinotebook_drag_motion, command_auinotebook_allow_dnd, command_auinotebook_drag_done, command_auinotebook_tab_middle_down, command_auinotebook_tab_middle_up, command_auinotebook_tab_right_down, command_auinotebook_tab_right_up, command_auinotebook_bg_dclick

## Data Types

```
wxAuiNotebook() = wx:wx_object()
```

## Exports

```
new() -> wxAuiNotebook()
```

Default ctor.

```
new(Parent) -> wxAuiNotebook()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxAuiNotebook()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
```

```
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructor.

Creates a wxAuiNotebok control.

```
addPage(This, Page, Caption) -> boolean()
```

Types:

```
   This = wxAuiNotebook()
   Page = wxWindow:wxWindow()
   Caption = unicode:chardata()
```

```
addPage(This, Page, Caption, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxAuiNotebook()
   Page = wxWindow:wxWindow()
   Caption = unicode:chardata()
   Option = {select, boolean()} | {bitmap, wxBitmap:wxBitmap()}
```

Adds a page.

If the `select` parameter is true, calling this will generate a page change event.

```
addPage(This, Page, Text, Select, ImageId) -> boolean()
```

Types:

```
   This = wxAuiNotebook()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
   Select = boolean()
   ImageId = integer()
```

Adds a new page.

The page must have the book control itself as the parent and must not have been added to this control previously.

The call to this function may generate the page changing events.

Return: true if successful, false otherwise.

Remark: Do not delete the page, it will be deleted by the book control.

See: `insertPage/6`

Since: 2.9.3

```
create(This, Parent) -> boolean()
```

Types:

```
   This = wxAuiNotebook()
   Parent = wxWindow:wxWindow()

create(This, Parent, Winid) -> boolean()
create(This, Parent, Winid :: [Option]) -> boolean()
```

Types:

```
   This = wxAuiNotebook()
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Creates the notebook window.

```
create(This, Parent, Winid, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxAuiNotebook()
   Parent = wxWindow:wxWindow()
   Winid = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructs the book control with the given parameters.

```
deletePage(This, Page) -> boolean()
```

Types:

```
   This = wxAuiNotebook()
   Page = integer()
```

Deletes a page at the given index.

Calling this method will generate a page change event.

```
getArtProvider(This) -> wxAuiTabArt:wxAuiTabArt()
```

Types:

```
   This = wxAuiNotebook()
```

Returns the associated art provider.

```
getPage(This, Page_idx) -> wxWindow:wxWindow()
```

Types:

```
   This = wxAuiNotebook()
   Page_idx = integer()
```

Returns the page specified by the given index.

```
getPageBitmap(This, Page) -> wxBitmap:wxBitmap()
```

Types:

```
   This = wxAuiNotebook()
   Page = integer()
```

Returns the tab bitmap for the page.

```
getPageCount(This) -> integer()
```

Types:

```
   This = wxAuiNotebook()
```

Returns the number of pages in the notebook.

```
getPageIndex(This, Page_wnd) -> integer()
```

Types:

```
   This = wxAuiNotebook()
   Page_wnd = wxWindow:wxWindow()
```

Returns the page index for the specified window.

If the window is not found in the notebook, wxNOT_FOUND is returned.

```
getPageText(This, Page) -> unicode:charlist()
```

Types:

```
   This = wxAuiNotebook()
   Page = integer()
```

Returns the tab label for the page.

```
getSelection(This) -> integer()
```

Types:

```
   This = wxAuiNotebook()
```

Returns the currently selected page.

```
insertPage(This, Page_idx, Page, Caption) -> boolean()
```

Types:

```
   This = wxAuiNotebook()
   Page_idx = integer()
   Page = wxWindow:wxWindow()
   Caption = unicode:chardata()

insertPage(This, Page_idx, Page, Caption, Options :: [Option]) ->
              boolean()
```

Types:

```
This = wxAuiNotebook()
Page_idx = integer()
Page = wxWindow:wxWindow()
Caption = unicode:chardata()
Option = {select, boolean()} | {bitmap, wxBitmap:wxBitmap()}
```

`insertPage/6` is similar to AddPage, but allows the ability to specify the insert location.

If the `select` parameter is true, calling this will generate a page change event.

```
insertPage(This, Index, Page, Text, Select, ImageId) -> boolean()
```

Types:

```
This = wxAuiNotebook()
Index = integer()
Page = wxWindow:wxWindow()
Text = unicode:chardata()
Select = boolean()
ImageId = integer()
```

Inserts a new page at the specified position.

Return: true if successful, false otherwise.

Remark: Do not delete the page, it will be deleted by the book control.

See: `addPage/5`

Since: 2.9.3

```
removePage(This, Page) -> boolean()
```

Types:

```
This = wxAuiNotebook()
Page = integer()
```

Removes a page, without deleting the window pointer.

```
setArtProvider(This, Art) -> ok
```

Types:

```
This = wxAuiNotebook()
Art = wxAuiTabArt:wxAuiTabArt()
```

Sets the art provider to be used by the notebook.

```
setFont(This, Font) -> boolean()
```

Types:

```
This = wxAuiNotebook()
Font = wxFont:wxFont()
```

Sets the font for drawing the tab labels, using a bold version of the font for selected tab labels.

```
setPageBitmap(This, Page, Bitmap) -> boolean()
```

Types:

```
This = wxAuiNotebook()
Page = integer()
Bitmap = wxBitmap:wxBitmap()
```

Sets the bitmap for the page.

To remove a bitmap from the tab caption, pass wxNullBitmap.

```
setPageText(This, Page, Text) -> boolean()
```

Types:

```
This = wxAuiNotebook()
Page = integer()
Text = unicode:chardata()
```

Sets the tab label for the page.

```
setSelection(This, New_page) -> integer()
```

Types:

```
This = wxAuiNotebook()
New_page = integer()
```

Sets the page selection.

Calling this method will generate a page change event.

```
setTabCtrlHeight(This, Height) -> ok
```

Types:

```
This = wxAuiNotebook()
Height = integer()
```

Sets the tab height.

By default, the tab control height is calculated by measuring the text height and bitmap sizes on the tab captions. Calling this method will override that calculation and set the tab control to the specified height parameter. A call to this method will override any call to `setUniformBitmapSize/2`.

Specifying -1 as the height will return the control to its default auto-sizing behaviour.

```
setUniformBitmapSize(This, Size) -> ok
```

Types:

```
This = wxAuiNotebook()
Size = {W :: integer(), H :: integer()}
```

Ensure that all tabs have the same height, even if some of them don't have bitmaps.

Passing ?wxDefaultSize as `size` undoes the effect of a previous call to this function and instructs the control to use dynamic tab height.

```
destroy(This :: wxAuiNotebook()) -> ok
```

Destroys the object.

# wxAuiNotebookEvent

Erlang module

This class is used by the events generated by `wxAuiNotebook`.

See: `wxAuiNotebook`, `wxBookCtrlEvent`

This class is derived (and can use functions) from: `wxBookCtrlEvent` `wxNotifyEvent` `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxAuiNotebookEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxAuiNotebookEventType` to subscribe to events of this type.

## Data Types

```
wxAuiNotebookEvent() = wx:wx_object()
wxAuiNotebook() =
    #wxAuiNotebook{type =
                       wxAuiNotebookEvent:wxAuiNotebookEventType(),
                   old_selection = integer(),
                   selection = integer(),
                   drag_source = wxAuiNotebook:wxAuiNotebook()}
wxAuiNotebookEventType() =
    command_auinotebook_page_close |
    command_auinotebook_page_changed |
    command_auinotebook_page_changing |
    command_auinotebook_button | command_auinotebook_begin_drag |
    command_auinotebook_end_drag |
    command_auinotebook_drag_motion |
    command_auinotebook_allow_dnd |
    command_auinotebook_tab_middle_down |
    command_auinotebook_tab_middle_up |
    command_auinotebook_tab_right_down |
    command_auinotebook_tab_right_up |
    command_auinotebook_page_closed |
    command_auinotebook_drag_done | command_auinotebook_bg_dclick
```

## Exports

```
setSelection(This, Page) -> ok
```

Types:

```
   This = wxAuiNotebookEvent()
   Page = integer()
```

Sets the selection member variable.

```
getSelection(This) -> integer()
```

Types:

```
This = wxAuiNotebookEvent()
```

Returns the currently selected page, or `wxNOT_FOUND` if none was selected.

Note: under Windows, `getSelection/1` will return the same value as `getOldSelection/1` when called from the `EVT_BOOKCTRL_PAGE_CHANGING` handler and not the page which is going to be selected.

```
setOldSelection(This, Page) -> ok
```

Types:

```
This = wxAuiNotebookEvent()
Page = integer()
```

Sets the id of the page selected before the change.

```
getOldSelection(This) -> integer()
```

Types:

```
This = wxAuiNotebookEvent()
```

Returns the page that was selected before the change, `wxNOT_FOUND` if none was selected.

```
setDragSource(This, S) -> ok
```

Types:

```
This = wxAuiNotebookEvent()
S = wxAuiNotebook:wxAuiNotebook()
```

```
getDragSource(This) -> wxAuiNotebook:wxAuiNotebook()
```

Types:

```
This = wxAuiNotebookEvent()
```

# wxAuiPaneInfo

Erlang module

`wxAuiPaneInfo` is part of the wxAUI class framework. See also overview_aui.

`wxAuiPaneInfo` specifies all the parameters for a pane. These parameters specify where the pane is on the screen, whether it is docked or floating, or hidden. In addition, these parameters specify the pane's docked position, floating position, preferred size, minimum size, caption text among many other parameters.

See: `wxAuiManager`, `wxAuiDockArt`

wxWidgets docs: **wxAuiPaneInfo**

## Data Types

```
wxAuiPaneInfo() = wx:wx_object()
```

## Exports

```
new() -> wxAuiPaneInfo()
```

```
new(C) -> wxAuiPaneInfo()
```

Types:

```
C = wxAuiPaneInfo()
```

Copy constructor.

```
bestSize(This, Size) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
Size = {W :: integer(), H :: integer()}
```

`bestSize/3` sets the ideal size for the pane.

The docking manager will attempt to use this size as much as possible when docking or floating the pane.

```
bestSize(This, X, Y) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
X = Y = integer()
```

```
bottom(This) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
```

`bottom/1` sets the pane dock position to the bottom side of the frame.

This is the same thing as calling Direction(wxAUI_DOCK_BOTTOM).

```
bottomDockable(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
bottomDockable(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {b, boolean()}
```

`bottomDockable/2` indicates whether a pane can be docked at the bottom of the frame.

```
caption(This, C) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   C = unicode:chardata()
```

`caption/2` sets the caption of the pane.

```
captionVisible(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
captionVisible(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {visible, boolean()}
```

CaptionVisible indicates that a pane caption should be visible.

If false, no pane caption is drawn.

```
centre(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

`Center()` (not implemented in wx) sets the pane dock position to the left side of the frame.

The centre pane is the space in the middle after all border panes (left, top, right, bottom) are subtracted from the layout. This is the same thing as calling Direction(wxAUI_DOCK_CENTRE).

```
centrePane(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

`centrePane/1` specifies that the pane should adopt the default center pane settings.

Centre panes usually do not have caption bars. This function provides an easy way of preparing a pane to be displayed in the center dock position.

```
closeButton(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
closeButton(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {visible, boolean()}
```

`closeButton/2` indicates that a close button should be drawn for the pane.

```
defaultPane(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

`defaultPane/1` specifies that the pane should adopt the default pane settings.

```
destroyOnClose(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
destroyOnClose(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {b, boolean()}
```

`destroyOnClose/2` indicates whether a pane should be destroyed when it is closed.

Normally a pane is simply hidden when the close button is clicked. Setting DestroyOnClose to true will cause the window to be destroyed when the user clicks the pane's close button.

```
direction(This, Direction) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Direction = integer()
```

`direction/2` determines the direction of the docked pane.

It is functionally the same as calling `left/1`, `right/1`, `top/1` or `bottom/1`, except that docking direction may be specified programmatically via the parameter.

```
dock(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

`dock/1` indicates that a pane should be docked.

It is the opposite of `float/1`.

```
dockable(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
dockable(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {b, boolean()}
```

`dockable/2` specifies whether a frame can be docked or not.

It is the same as specifying TopDockable(b).BottomDockable(b).LeftDockable(b).RightDockable(b).

```
fixed(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

`fixed/1` forces a pane to be fixed size so that it cannot be resized.

After calling `fixed/1`, `isFixed/1` will return true.

```
float(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

`float/1` indicates that a pane should be floated.

It is the opposite of `dock/1`.

```
floatable(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
floatable(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {b, boolean()}
```

`floatable/2` sets whether the user will be able to undock a pane and turn it into a floating window.

```
floatingPosition(This, Pos) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Pos = {X :: integer(), Y :: integer()}
```

`floatingPosition/3` sets the position of the floating pane.

```
floatingPosition(This, X, Y) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   X = Y = integer()
```

```
floatingSize(This, Size) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Size = {W :: integer(), H :: integer()}
```

`floatingSize/3` sets the size of the floating pane.

```
floatingSize(This, X, Y) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   X = Y = integer()
```

```
gripper(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
gripper(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {visible, boolean()}
```

`gripper/2` indicates that a gripper should be drawn for the pane.

```
gripperTop(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
gripperTop(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {attop, boolean()}
```

`gripperTop/2` indicates that a gripper should be drawn at the top of the pane.

```
hasBorder(This) -> boolean()
```

Types:

```
   This = wxAuiPaneInfo()
```

`hasBorder/1` returns true if the pane displays a border.

```
hasCaption(This) -> boolean()
```

Types:

```
   This = wxAuiPaneInfo()
```

`hasCaption/1` returns true if the pane displays a caption.

```
hasCloseButton(This) -> boolean()
```
Types:
```
   This = wxAuiPaneInfo()
```
`hasCloseButton/1` returns true if the pane displays a button to close the pane.

```
hasFlag(This, Flag) -> boolean()
```
Types:
```
   This = wxAuiPaneInfo()
   Flag = integer()
```
`hasFlag/2` returns true if the property specified by flag is active for the pane.

```
hasGripper(This) -> boolean()
```
Types:
```
   This = wxAuiPaneInfo()
```
`hasGripper/1` returns true if the pane displays a gripper.

```
hasGripperTop(This) -> boolean()
```
Types:
```
   This = wxAuiPaneInfo()
```
`hasGripper/1` returns true if the pane displays a gripper at the top.

```
hasMaximizeButton(This) -> boolean()
```
Types:
```
   This = wxAuiPaneInfo()
```
`hasMaximizeButton/1` returns true if the pane displays a button to maximize the pane.

```
hasMinimizeButton(This) -> boolean()
```
Types:
```
   This = wxAuiPaneInfo()
```
`hasMinimizeButton/1` returns true if the pane displays a button to minimize the pane.

```
hasPinButton(This) -> boolean()
```
Types:
```
   This = wxAuiPaneInfo()
```
`hasPinButton/1` returns true if the pane displays a button to float the pane.

```
hide(This) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
```
`hide/1` indicates that a pane should be hidden.

```
isBottomDockable(This) -> boolean()
```
Types:

```
This = wxAuiPaneInfo()
```

`isBottomDockable/1` returns true if the pane can be docked at the bottom of the managed frame.

See: `IsDockable()` (not implemented in wx)

```
isDocked(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isDocked/1` returns true if the pane is currently docked.

```
isFixed(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isFixed/1` returns true if the pane cannot be resized.

```
isFloatable(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isFloatable/1` returns true if the pane can be undocked and displayed as a floating window.

```
isFloating(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isFloating/1` returns true if the pane is floating.

```
isLeftDockable(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isLeftDockable/1` returns true if the pane can be docked on the left of the managed frame.

See: `IsDockable()` (not implemented in wx)

```
isMovable(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

IsMoveable() returns true if the docked frame can be undocked or moved to another dock position.

```
isOk(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isOk/1` returns true if the `wxAuiPaneInfo` structure is valid.

A pane structure is valid if it has an associated window.

```
isResizable(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isResizable/1` returns true if the pane can be resized.

```
isRightDockable(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isRightDockable/1` returns true if the pane can be docked on the right of the managed frame.

See: `IsDockable()` (not implemented in wx)

```
isShown(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isShown/1` returns true if the pane is currently shown.

```
isToolbar(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isToolbar/1` returns true if the pane contains a toolbar.

```
isTopDockable(This) -> boolean()
```

Types:

```
This = wxAuiPaneInfo()
```

`isTopDockable/1` returns true if the pane can be docked at the top of the managed frame.

See: `IsDockable()` (not implemented in wx)

```
layer(This, Layer) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
Layer = integer()
```

`layer/2` determines the layer of the docked pane.

The dock layer is similar to an onion, the inner-most layer being layer 0. Each shell moving in the outward direction has a higher layer number. This allows for more complex docking layout formation.

```
left(This) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
```

`left/1` sets the pane dock position to the left side of the frame.

This is the same thing as calling Direction(wxAUI_DOCK_LEFT).

```
leftDockable(This) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
```

```
leftDockable(This, Options :: [Option]) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
   Option = {b, boolean()}
```
`leftDockable/2` indicates whether a pane can be docked on the left of the frame.

```
maxSize(This, Size) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
   Size = {W :: integer(), H :: integer()}
```
`maxSize/3` sets the maximum size of the pane.

```
maxSize(This, X, Y) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
   X = Y = integer()
```

```
maximizeButton(This) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
```

```
maximizeButton(This, Options :: [Option]) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
   Option = {visible, boolean()}
```
`maximizeButton/2` indicates that a maximize button should be drawn for the pane.

```
minSize(This, Size) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
   Size = {W :: integer(), H :: integer()}
```
`minSize/3` sets the minimum size of the pane.

Please note that this is only partially supported as of this writing.

```
minSize(This, X, Y) -> wxAuiPaneInfo()
```
Types:

```
   This = wxAuiPaneInfo()
   X = Y = integer()
```

```
minimizeButton(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
minimizeButton(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {visible, boolean()}
```

`minimizeButton/2` indicates that a minimize button should be drawn for the pane.

```
movable(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
movable(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {b, boolean()}
```

Movable indicates whether a frame can be moved.

```
name(This, N) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   N = unicode:chardata()
```

`name/2` sets the name of the pane so it can be referenced in lookup functions.

If a name is not specified by the user, a random name is assigned to the pane when it is added to the manager.

```
paneBorder(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
paneBorder(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {visible, boolean()}
```

PaneBorder indicates that a border should be drawn for the pane.

```
pinButton(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
pinButton(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {visible, boolean()}
```

`pinButton/2` indicates that a pin button should be drawn for the pane.

```
position(This, Pos) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Pos = integer()
```

`position/2` determines the position of the docked pane.

```
resizable(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
resizable(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {resizable, boolean()}
```

`resizable/2` allows a pane to be resized if the parameter is true, and forces it to be a fixed size if the parameter is false.

This is simply an antonym for `fixed/1`.

```
right(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

`right/1` sets the pane dock position to the right side of the frame.

This is the same thing as calling Direction(wxAUI_DOCK_RIGHT).

```
rightDockable(This) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
```

```
rightDockable(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
   This = wxAuiPaneInfo()
   Option = {b, boolean()}
```

`rightDockable/2` indicates whether a pane can be docked on the right of the frame.

```
row(This, Row) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
Row = integer()
```

`row/2` determines the row of the docked pane.

```
safeSet(This, Source) -> ok
```

Types:

```
This = Source = wxAuiPaneInfo()
```

Write the safe parts of a PaneInfo object "source" into "this".

"Safe parts" are all non-UI elements (e.g. all layout determining parameters like the size, position etc.). "Unsafe parts" (pointers to button, frame and window) are not modified by this write operation.

Remark: This method is used when loading perspectives.

```
setFlag(This, Flag, Option_state) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
Flag = integer()
Option_state = boolean()
```

`setFlag/3` turns the property given by flag on or off with the option_state parameter.

```
show(This) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
```

```
show(This, Options :: [Option]) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
Option = {show, boolean()}
```

`show/2` indicates that a pane should be shown.

```
toolbarPane(This) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
```

`toolbarPane/1` specifies that the pane should adopt the default toolbar pane settings.

```
top(This) -> wxAuiPaneInfo()
```

Types:

```
This = wxAuiPaneInfo()
```

`top/1` sets the pane dock position to the top of the frame.

This is the same thing as calling Direction(wxAUI_DOCK_TOP).

```
topDockable(This) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
```

```
topDockable(This, Options :: [Option]) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
   Option = {b, boolean()}
```
`topDockable/2` indicates whether a pane can be docked at the top of the frame.

```
window(This, W) -> wxAuiPaneInfo()
```
Types:
```
   This = wxAuiPaneInfo()
   W = wxWindow:wxWindow()
```
`window/2` assigns the window pointer that the `wxAuiPaneInfo` should use.

This normally does not need to be specified, as the window pointer is automatically assigned to the `wxAuiPaneInfo` structure as soon as it is added to the manager.

```
getWindow(This) -> wxWindow:wxWindow()
```
Types:
```
   This = wxAuiPaneInfo()
```

```
getFrame(This) -> wxFrame:wxFrame()
```
Types:
```
   This = wxAuiPaneInfo()
```

```
getDirection(This) -> integer()
```
Types:
```
   This = wxAuiPaneInfo()
```

```
getLayer(This) -> integer()
```
Types:
```
   This = wxAuiPaneInfo()
```

```
getRow(This) -> integer()
```
Types:
```
   This = wxAuiPaneInfo()
```

```
getPosition(This) -> integer()
```
Types:
```
   This = wxAuiPaneInfo()
```

```
getFloatingPosition(This) -> {X :: integer(), Y :: integer()}
```
Types:

```
   This = wxAuiPaneInfo()
```

```
getFloatingSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxAuiPaneInfo()
```

```
destroy(This :: wxAuiPaneInfo()) -> ok
```

Destroys the object.

# wxAuiSimpleTabArt

Erlang module

Another standard tab art provider for `wxAuiNotebook`.

`wxAuiSimpleTabArt` is derived from `wxAuiTabArt` demonstrating how to write a completely new tab art class. It can also be used as alternative to `wxAuiDefaultTabArt` (not implemented in wx).

This class is derived (and can use functions) from: `wxAuiTabArt`

wxWidgets docs: **wxAuiSimpleTabArt**

## Data Types

```
wxAuiSimpleTabArt() = wx:wx_object()
```

## Exports

```
new() -> wxAuiSimpleTabArt()
```

```
destroy(This :: wxAuiSimpleTabArt()) -> ok
```

Destroys the object.

# wxAuiTabArt

Erlang module

Tab art provider defines all the drawing functions used by `wxAuiNotebook`.

This allows the `wxAuiNotebook` to have a pluggable look-and-feel.

By default, a `wxAuiNotebook` uses an instance of this class called `wxAuiDefaultTabArt` (not implemented in wx) which provides bitmap art and a colour scheme that is adapted to the major platforms' look. You can either derive from that class to alter its behaviour or write a completely new tab art class.

Another example of creating a new `wxAuiNotebook` tab bar is `wxAuiSimpleTabArt`.

Call `wxAuiNotebook:setArtProvider/2` to make use of this new tab art.

wxWidgets docs: **wxAuiTabArt**

## Data Types

```
wxAuiTabArt() = wx:wx_object()
```

## Exports

```
setFlags(This, Flags) -> ok
```

Types:

```
   This = wxAuiTabArt()
   Flags = integer()
```

Sets flags.

```
setMeasuringFont(This, Font) -> ok
```

Types:

```
   This = wxAuiTabArt()
   Font = wxFont:wxFont()
```

Sets the font used for calculating measurements.

```
setNormalFont(This, Font) -> ok
```

Types:

```
   This = wxAuiTabArt()
   Font = wxFont:wxFont()
```

Sets the normal font for drawing labels.

```
setSelectedFont(This, Font) -> ok
```

Types:

```
   This = wxAuiTabArt()
   Font = wxFont:wxFont()
```

Sets the font for drawing text for selected UI elements.

```
setColour(This, Colour) -> ok
```

Types:

```
This = wxAuiTabArt()
Colour = wx:wx_colour()
```

Sets the colour of the inactive tabs.

Since: 2.9.2

```
setActiveColour(This, Colour) -> ok
```

Types:

```
This = wxAuiTabArt()
Colour = wx:wx_colour()
```

Sets the colour of the selected tab.

Since: 2.9.2

# wxBitmap

Erlang module

This class encapsulates the concept of a platform-dependent bitmap, either monochrome or colour or colour with alpha channel support.

If you need direct access the bitmap data instead going through drawing to it using `wxMemoryDC` you need to use the `wxPixelData` (not implemented in wx) class (either wxNativePixelData for RGB bitmaps or wxAlphaPixelData for bitmaps with an additionally alpha channel).

Note that many `wxBitmap` functions take a `type` parameter, which is a value of the ?wxBitmapType enumeration. The validity of those values depends however on the platform where your program is running and from the wxWidgets configuration. If all possible wxWidgets settings are used:

In addition, `wxBitmap` can load and save all formats that `wxImage` can; see `wxImage` for more info. Of course, you must have loaded the `wxImage` handlers (see ?wxInitAllImageHandlers() and `wxImage::AddHandler` (not implemented in wx)). Note that all available wxBitmapHandlers for a given wxWidgets port are automatically loaded at startup so you won't need to use `wxBitmap::AddHandler` (not implemented in wx).

More on the difference between `wxImage` and `wxBitmap`: `wxImage` is just a buffer of RGB bytes with an optional buffer for the alpha bytes. It is all generic, platform independent and image file format independent code. It includes generic code for scaling, resizing, clipping, and other manipulations of the image data. OTOH, `wxBitmap` is intended to be a wrapper of whatever is the native image format that is quickest/easiest to draw to a DC or to be the target of the drawing operations performed on a `wxMemoryDC`. By splitting the responsibilities between wxImage/wxBitmap like this then it's easier to use generic code shared by all platforms and image types for generic operations and platform specific code where performance or compatibility is needed.

Predefined objects (include wx.hrl): ?wxNullBitmap

See: **Overview bitmap**, **Overview bitmap**, `wxDC:blit/6`, `wxIcon`, `wxCursor`, `wxMemoryDC`, `wxImage`, `wxPixelData` (not implemented in wx)

wxWidgets docs: **wxBitmap**

## Data Types

```
wxBitmap() = wx:wx_object()
```

## Exports

```
new() -> wxBitmap()
```

Default constructor.

Constructs a bitmap object with no data; an assignment or another member function such as `create/4` or `loadFile/3` must be called subsequently.

```
new(Name) -> wxBitmap()
new(Sz) -> wxBitmap()
new(Img) -> wxBitmap()
```

Types:

```
   Img = wxImage:wxImage() | wxBitmap:wxBitmap()

new(Width, Height) -> wxBitmap()
new(Name, Height :: [Option]) -> wxBitmap()
new(Sz, Height :: [Option]) -> wxBitmap()
new(Img, Height :: [Option]) -> wxBitmap()
```

Types:

```
   Img = wxImage:wxImage()
   Option = {depth, integer()}
```

Creates this bitmap object from the given image.

This has to be done to actually display an image as you cannot draw an image directly on a window.

The resulting bitmap will use the provided colour depth (or that of the current system if depth is ? wxBITMAP_SCREEN_DEPTH) which entails that a colour reduction may take place.

On Windows, if there is a palette present (set with SetPalette), it will be used when creating the `wxBitmap` (most useful in 8-bit display mode). On other platforms, the palette is currently ignored.

```
new(Bits, Width, Height) -> wxBitmap()
new(Width, Height, Height :: [Option]) -> wxBitmap()
```

Types:

```
   Width = Height = integer()
   Option = {depth, integer()}
```

Creates a new bitmap.

A depth of ?wxBITMAP_SCREEN_DEPTH indicates the depth of the current screen or visual.

Some platforms only support 1 for monochrome and ?wxBITMAP_SCREEN_DEPTH for the current colour setting.

A depth of 32 including an alpha channel is supported under MSW, Mac and GTK+.

```
new(Bits, Width, Height, Options :: [Option]) -> wxBitmap()
```

Types:

```
   Bits = binary()
   Width = Height = integer()
   Option = {depth, integer()}
```

Creates a bitmap from the given array `bits`.

You should only use this function for monochrome bitmaps (depth 1) in portable programs: in this case the bits parameter should contain an XBM image.

For other bit depths, the behaviour is platform dependent: under Windows, the data is passed without any changes to the underlying CreateBitmap() API. Under other platforms, only monochrome bitmaps may be created using this constructor and `wxImage` should be used for creating colour bitmaps from static data.

```
destroy(This :: wxBitmap()) -> ok
```

Creates bitmap corresponding to the given cursor.

This can be useful to display a cursor as it cannot be drawn directly on a window.

This constructor only exists in wxMSW and wxGTK (where it is implemented for GTK+ 2.8 or later) only.

Since: 3.1.0 Destructor. See overview_refcount_destruct for more info.

If the application omits to delete the bitmap explicitly, the bitmap will be destroyed automatically by wxWidgets when the application exits.

Warning: Do not delete a bitmap that is selected into a memory device context.

```
convertToImage(This) -> wxImage:wxImage()
```

Types:

```
   This = wxBitmap()
```

Creates an image from a platform-dependent bitmap.

This preserves mask information so that bitmaps and images can be converted back and forth without loss in that respect.

```
copyFromIcon(This, Icon) -> boolean()
```

Types:

```
   This = wxBitmap()
   Icon = wxIcon:wxIcon()
```

Creates the bitmap from an icon.

```
create(This, Sz) -> boolean()
```

Types:

```
   This = wxBitmap()
   Sz = {W :: integer(), H :: integer()}
```

```
create(This, Width, Height) -> boolean()
create(This, Sz, Height :: [Option]) -> boolean()
```

Types:

```
   This = wxBitmap()
   Sz = {W :: integer(), H :: integer()}
   Option = {depth, integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
create(This, Width, Height, Options :: [Option]) -> boolean()
create(This, Width, Height, Dc) -> boolean()
```

Types:

```
   This = wxBitmap()
   Width = Height = integer()
   Dc = wxDC:wxDC()
```

Create a bitmap compatible with the given DC, inheriting its magnification factor.

Return: true if the creation was successful.

Since: 3.1.0

```
getDepth(This) -> integer()
```

Types:

```
This = wxBitmap()
```

Gets the colour depth of the bitmap.

A value of 1 indicates a monochrome bitmap.

```
getHeight(This) -> integer()
```

Types:

```
This = wxBitmap()
```

Gets the height of the bitmap in pixels.

See: `getWidth/1`, `GetSize()` (not implemented in wx)

```
getPalette(This) -> wxPalette:wxPalette()
```

Types:

```
This = wxBitmap()
```

Gets the associated palette (if any) which may have been loaded from a file or set for the bitmap.

See: `wxPalette`

```
getMask(This) -> wxMask:wxMask()
```

Types:

```
This = wxBitmap()
```

Gets the associated mask (if any) which may have been loaded from a file or set for the bitmap.

See: `setMask/2`, `wxMask`

```
getWidth(This) -> integer()
```

Types:

```
This = wxBitmap()
```

Gets the width of the bitmap in pixels.

See: `getHeight/1`, `GetSize()` (not implemented in wx)

```
getSubBitmap(This, Rect) -> wxBitmap()
```

Types:

```
This = wxBitmap()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
```

Returns a sub bitmap of the current one as long as the rect belongs entirely to the bitmap.

This function preserves bit depth and mask information.

```
loadFile(This, Name) -> boolean()
```

Types:

```
   This = wxBitmap()
   Name = unicode:chardata()
```

```
loadFile(This, Name, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxBitmap()
   Name = unicode:chardata()
   Option = {type, wx:wx_enum()}
```

Loads a bitmap from a file or resource.

Return: true if the operation succeeded, false otherwise.

Remark: A palette may be associated with the bitmap if one exists (especially for colour Windows bitmaps), and if the code supports it. You can check if one has been created by using the `getPalette/1` member.

See: `saveFile/4`

```
ok(This) -> boolean()
```

Types:

```
   This = wxBitmap()
```

See: `isOk/1`.

```
isOk(This) -> boolean()
```

Types:

```
   This = wxBitmap()
```

Returns true if bitmap data is present.

```
saveFile(This, Name, Type) -> boolean()
```

Types:

```
   This = wxBitmap()
   Name = unicode:chardata()
   Type = wx:wx_enum()
```

```
saveFile(This, Name, Type, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxBitmap()
   Name = unicode:chardata()
   Type = wx:wx_enum()
   Option = {palette, wxPalette:wxPalette()}
```

Saves a bitmap in the named file.

Return: true if the operation succeeded, false otherwise.

Remark: Depending on how wxWidgets has been configured, not all formats may be available.

See: `loadFile/3`

```
setDepth(This, Depth) -> ok
```

Types:

```
This = wxBitmap()
Depth = integer()
```

Deprecated: This function is deprecated since version 3.1.2, dimensions and depth can only be set at construction time.

Sets the depth member (does not affect the bitmap data).

```
setHeight(This, Height) -> ok
```

Types:

```
This = wxBitmap()
Height = integer()
```

Deprecated: This function is deprecated since version 3.1.2, dimensions and depth can only be set at construction time.

Sets the height member (does not affect the bitmap data).

```
setMask(This, Mask) -> ok
```

Types:

```
This = wxBitmap()
Mask = wxMask:wxMask()
```

Sets the mask for this bitmap.

Remark: The bitmap object owns the mask once this has been called.

Note: A mask can be set also for bitmap with an alpha channel but doing so under wxMSW is not recommended because performance of drawing such bitmap is not very good.

See: `getMask/1`, `wxMask`

```
setPalette(This, Palette) -> ok
```

Types:

```
This = wxBitmap()
Palette = wxPalette:wxPalette()
```

Sets the associated palette.

(Not implemented under GTK+).

See: `wxPalette`

```
setWidth(This, Width) -> ok
```

Types:

```
This = wxBitmap()
Width = integer()
```

Deprecated: This function is deprecated since version 3.1.2, dimensions and depth can only be set at construction time.

Sets the width member (does not affect the bitmap data).

# wxBitmapButton

Erlang module

A bitmap button is a control that contains a bitmap.

Notice that since wxWidgets 2.9.1 bitmap display is supported by the base `wxButton` class itself and the only tiny advantage of using this class is that it allows specifying the bitmap in its constructor, unlike `wxButton`. Please see the base class documentation for more information about images support in `wxButton`.

Styles

This class supports the following styles:

See: `wxButton`

This class is derived (and can use functions) from: `wxButton` `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxBitmapButton**

## Events

Event types emitted from this class: `command_button_clicked`

## Data Types

```
wxBitmapButton() = wx:wx_object()
```

## Exports

```
new() -> wxBitmapButton()
```

Default ctor.

```
new(Parent, Id, Bitmap) -> wxBitmapButton()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Bitmap = wxBitmap:wxBitmap()
```

```
new(Parent, Id, Bitmap, Options :: [Option]) -> wxBitmapButton()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Bitmap = wxBitmap:wxBitmap()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {validator, wx:wx_object()}
```

Constructor, creating and showing a button.

Remark: The bitmap parameter is normally the only bitmap you need to provide, and wxWidgets will draw the button correctly in its different states. If you want more control, call any of the functions `SetBitmapPressed()` (not implemented in wx), `wxButton:setBitmapFocus/2`, `wxButton:setBitmapDisabled/2`.

See: `create/5`, `wxValidator` (not implemented in wx)

```
create(This, Parent, Id, Bitmap) -> boolean()
```

Types:

```
   This = wxBitmapButton()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Bitmap = wxBitmap:wxBitmap()
```

```
create(This, Parent, Id, Bitmap, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxBitmapButton()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Bitmap = wxBitmap:wxBitmap()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Button creation function for two-step creation.

For more details, see `new/4`.

```
newCloseButton(Parent, Winid) -> wxBitmapButton()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Winid = integer()
```

Helper function creating a standard-looking "Close" button.

To get the best results, platform-specific code may need to be used to create a small, title bar-like "Close" button. This function is provided to avoid the need to test for the current platform and creates the button with as native look as possible.

Return: The new button.

Since: 2.9.5

```
destroy(This :: wxBitmapButton()) -> ok
```

Destroys the object.

# wxBitmapDataObject

Erlang module

`wxBitmapDataObject` is a specialization of `wxDataObject` for bitmap data. It can be used without change to paste data into the `wxClipboard` or a `wxDropSource` (not implemented in wx). A user may wish to derive a new class from this class for providing a bitmap on-demand in order to minimize memory consumption when offering data in several formats, such as a bitmap and GIF.

This class may be used as is, but `getBitmap/1` may be overridden to increase efficiency.

See: **Overview dnd**, `wxDataObject`, `wxDataObjectSimple` (not implemented in wx), `wxFileDataObject`, `wxTextDataObject`, `wxDataObject`

This class is derived (and can use functions) from: `wxDataObject`

wxWidgets docs: **wxBitmapDataObject**

## Data Types

```
wxBitmapDataObject() = wx:wx_object()
```

## Exports

```
new() -> wxBitmapDataObject()
```

```
new(Options :: [Option]) -> wxBitmapDataObject()
new(Bitmap) -> wxBitmapDataObject()
```

Types:

```
Bitmap = wxBitmap:wxBitmap()
```

Constructor, optionally passing a bitmap (otherwise use `setBitmap/2` later).

```
getBitmap(This) -> wxBitmap:wxBitmap()
```

Types:

```
This = wxBitmapDataObject()
```

Returns the bitmap associated with the data object.

You may wish to override this method when offering data on-demand, but this is not required by wxWidgets' internals. Use this method to get data in bitmap form from the `wxClipboard`.

```
setBitmap(This, Bitmap) -> ok
```

Types:

```
This = wxBitmapDataObject()
Bitmap = wxBitmap:wxBitmap()
```

Sets the bitmap associated with the data object.

This method is called when the data object receives data. Usually there will be no reason to override this function.

```
destroy(This :: wxBitmapDataObject()) -> ok
```

Destroys the object.

# wxBookCtrlBase

Erlang module

A book control is a convenient way of displaying multiple pages of information, displayed one page at a time. wxWidgets has five variants of this control:

This abstract class is the parent of all these book controls, and provides their basic interface. This is a pure virtual class so you cannot allocate it directly.

See: **Overview bookctrl**

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxBookCtrlBase**

## Data Types

```
wxBookCtrlBase() = wx:wx_object()
```

## Exports

```
addPage(This, Page, Text) -> boolean()
```

Types:

```
   This = wxBookCtrlBase()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
```

```
addPage(This, Page, Text, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxBookCtrlBase()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
   Option = {bSelect, boolean()} | {imageId, integer()}
```

Adds a new page.

The page must have the book control itself as the parent and must not have been added to this control previously.

The call to this function will generate the page changing and page changed events if `select` is true, but not when inserting the very first page (as there is no previous page selection to switch from in this case and so it wouldn't make sense to e.g. veto such event).

Return: true if successful, false otherwise.

Remark: Do not delete the page, it will be deleted by the book control.

See: `insertPage/5`

```
insertPage(This, Index, Page, Text) -> boolean()
```

Types:

```
   This = wxBookCtrlBase()
   Index = integer()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()

insertPage(This, Index, Page, Text, Options :: [Option]) ->
              boolean()
```

Types:

```
   This = wxBookCtrlBase()
   Index = integer()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
   Option = {bSelect, boolean()} | {imageId, integer()}
```

Inserts a new page at the specified position.

Return: true if successful, false otherwise.

Remark: Do not delete the page, it will be deleted by the book control.

See: `addPage/4`

```
deletePage(This, Page) -> boolean()
```

Types:

```
   This = wxBookCtrlBase()
   Page = integer()
```

Deletes the specified page, and the associated window.

The call to this function generates the page changing events when deleting the currently selected page or a page preceding it in the index order, but it does `not` send any events when deleting the last page: while in this case the selection also changes, it becomes invalid and for compatibility reasons the control never generates events with the invalid selection index.

```
removePage(This, Page) -> boolean()
```

Types:

```
   This = wxBookCtrlBase()
   Page = integer()
```

Deletes the specified page, without deleting the associated window.

See `deletePage/2` for a note about the events generated by this function.

```
deleteAllPages(This) -> boolean()
```

Types:

```
   This = wxBookCtrlBase()
```

Deletes all pages.

```
getPage(This, Page) -> wxWindow:wxWindow()
```

Types:

```
This = wxBookCtrlBase()
Page = integer()
```

Returns the window at the given page position.

```
getPageCount(This) -> integer()
```

Types:

```
This = wxBookCtrlBase()
```

Returns the number of pages in the control.

```
getCurrentPage(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxBookCtrlBase()
```

Returns the currently selected page or NULL.

```
advanceSelection(This) -> ok
```

Types:

```
This = wxBookCtrlBase()
```

```
advanceSelection(This, Options :: [Option]) -> ok
```

Types:

```
This = wxBookCtrlBase()
Option = {forward, boolean()}
```

Cycles through the tabs.

The call to this function generates the page changing events.

```
setSelection(This, Page) -> integer()
```

Types:

```
This = wxBookCtrlBase()
Page = integer()
```

Sets the selection to the given page, returning the previous selection.

Notice that the call to this function generates the page changing events, use the `changeSelection/2` function if you don't want these events to be generated.

See: `getSelection/1`

```
getSelection(This) -> integer()
```

Types:

```
This = wxBookCtrlBase()
```

Returns the currently selected page, or `wxNOT_FOUND` if none was selected.

Note that this method may return either the previously or newly selected page when called from the `EVT_BOOKCTRL_PAGE_CHANGED` handler depending on the platform and so `wxBookCtrlEvent:getSelection/1` should be used instead in this case.

```
changeSelection(This, Page) -> integer()
```

Types:

```
This = wxBookCtrlBase()
Page = integer()
```

Changes the selection to the given page, returning the previous selection.

This function behaves as `setSelection/2` but does `not` generate the page changing events.

See overview_events_prog for more information.

```
hitTest(This, Pt) -> Result
```

Types:

```
Result = {Res :: integer(), Flags :: integer()}
This = wxBookCtrlBase()
Pt = {X :: integer(), Y :: integer()}
```

Returns the index of the tab at the specified position or `wxNOT_FOUND` if none.

If `flags` parameter is non-NULL, the position of the point inside the tab is returned as well.

Return: Returns the zero-based tab index or `wxNOT_FOUND` if there is no tab at the specified position.

```
getPageText(This, NPage) -> unicode:charlist()
```

Types:

```
This = wxBookCtrlBase()
NPage = integer()
```

Returns the string for the given page.

```
setPageText(This, Page, Text) -> boolean()
```

Types:

```
This = wxBookCtrlBase()
Page = integer()
Text = unicode:chardata()
```

Sets the text for the given page.

# wxBookCtrlEvent

Erlang module

This class represents the events generated by book controls (`wxNotebook`, `wxListbook`, `wxChoicebook`, `wxTreebook`, `wxAuiNotebook`).

The PAGE_CHANGING events are sent before the current page is changed. It allows the program to examine the current page (which can be retrieved with `getOldSelection/1`) and to veto the page change by calling `wxNotifyEvent:veto/1` if, for example, the current values in the controls of the old page are invalid.

The PAGE_CHANGED events are sent after the page has been changed and the program cannot veto it any more, it just informs it about the page change.

To summarize, if the program is interested in validating the page values before allowing the user to change it, it should process the PAGE_CHANGING event, otherwise PAGE_CHANGED is probably enough. In any case, it is probably unnecessary to process both events at once.

See: `wxNotebook`, `wxListbook`, `wxChoicebook`, `wxTreebook`, `wxToolbook`, `wxAuiNotebook`

This class is derived (and can use functions) from: `wxNotifyEvent` `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxBookCtrlEvent**

## Data Types

```
wxBookCtrlEvent() = wx:wx_object()
wxBookCtrl() =
    #wxBookCtrl{type = wxBookCtrlEvent:wxBookCtrlEventType(),
                nSel = integer(),
                nOldSel = integer()}
wxBookCtrlEventType() =
    command_notebook_page_changed |
    command_notebook_page_changing | choicebook_page_changed |
    choicebook_page_changing | treebook_page_changed |
    treebook_page_changing | toolbook_page_changed |
    toolbook_page_changing | listbook_page_changed |
    listbook_page_changing
```

## Exports

```
getOldSelection(This) -> integer()
```

Types:

```
This = wxBookCtrlEvent()
```

Returns the page that was selected before the change, `wxNOT_FOUND` if none was selected.

```
getSelection(This) -> integer()
```

Types:

```
This = wxBookCtrlEvent()
```

Returns the currently selected page, or `wxNOT_FOUND` if none was selected.

Note: under Windows, `getSelection/1` will return the same value as `getOldSelection/1` when called from the `EVT_BOOKCTRL_PAGE_CHANGING` handler and not the page which is going to be selected.

```
setOldSelection(This, Page) -> ok
```

Types:

```
This = wxBookCtrlEvent()
Page = integer()
```

Sets the id of the page selected before the change.

```
setSelection(This, Page) -> ok
```

Types:

```
This = wxBookCtrlEvent()
Page = integer()
```

Sets the selection member variable.

# wxBoxSizer

Erlang module

The basic idea behind a box sizer is that windows will most often be laid out in rather simple basic geometry, typically in a row or a column or several hierarchies of either.

For more information, please see overview_sizer_box.

See: `wxSizer`, **Overview sizer**

This class is derived (and can use functions) from: `wxSizer`

wxWidgets docs: **wxBoxSizer**

## Data Types

```
wxBoxSizer() = wx:wx_object()
```

## Exports

```
new(Orient) -> wxBoxSizer()
```

Types:

```
Orient = integer()
```

Constructor for a `wxBoxSizer`.

`orient` may be either of wxVERTICAL or wxHORIZONTAL for creating either a column sizer or a row sizer.

```
getOrientation(This) -> integer()
```

Types:

```
This = wxBoxSizer()
```

Returns the orientation of the box sizer, either wxVERTICAL or wxHORIZONTAL.

```
destroy(This :: wxBoxSizer()) -> ok
```

Destroys the object.

# wxBrush

Erlang module

A brush is a drawing tool for filling in areas. It is used for painting the background of rectangles, ellipses, etc. It has a colour and a style.

On a monochrome display, wxWidgets shows all brushes as white unless the colour is really black.

Do not initialize objects on the stack before the program commences, since other required structures may not have been set up yet. Instead, define global pointers to objects and create them in `wxApp::OnInit` (not implemented in wx) or when required.

An application may wish to create brushes with different characteristics dynamically, and there is the consequent danger that a large number of duplicate brushes will be created. Therefore an application may wish to get a pointer to a brush by using the global list of brushes ?wxTheBrushList, and calling the member function `wxBrushList::FindOrCreateBrush()` (not implemented in wx).

This class uses reference counting and copy-on-write internally so that assignments between two instances of this class are very cheap. You can therefore use actual objects instead of pointers without efficiency problems. If an instance of this class is changed it will create its own data internally so that other instances, which previously shared the data using the reference counting, are not affected.

Predefined objects (include wx.hrl):

See: `wxBrushList` (not implemented in wx), `wxDC`, `wxDC:setBrush/2`

wxWidgets docs: **wxBrush**

## Data Types

```
wxBrush() = wx:wx_object()
```

## Exports

```
new() -> wxBrush()
```

Default constructor.

The brush will be uninitialised, and `wxBrush:isOk/1` will return false.

```
new(Colour) -> wxBrush()
new(Brush) -> wxBrush()
```

Types:

```
Brush = wxBrush:wxBrush() | wxBitmap:wxBitmap()
```

Copy constructor, uses reference counting.

```
new(Colour, Options :: [Option]) -> wxBrush()
```

Types:

```
Colour = wx:wx_colour()
Option = {style, wx:wx_enum()}
```

Constructs a brush from a colour object and `style`.

```
destroy(This :: wxBrush()) -> ok
```

Destructor.

See overview_refcount_destruct for more info.

Remark: Although all remaining brushes are deleted when the application exits, the application should try to clean up all brushes itself. This is because wxWidgets cannot know if a pointer to the brush object is stored in an application data structure, and there is a risk of double deletion.

```
getColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxBrush()
```

Returns a reference to the brush colour.

See: `setColour/4`

```
getStipple(This) -> wxBitmap:wxBitmap()
```

Types:

```
This = wxBrush()
```

Gets a pointer to the stipple bitmap.

If the brush does not have a `wxBRUSHSTYLE_STIPPLE` style, this bitmap may be non-NULL but uninitialised (i.e. `wxBitmap:isOk/1` returns false).

See: `setStipple/2`

```
getStyle(This) -> wx:wx_enum()
```

Types:

```
This = wxBrush()
```

Returns the brush style, one of the ?wxBrushStyle values.

See: `setStyle/2`, `setColour/4`, `setStipple/2`

```
isHatch(This) -> boolean()
```

Types:

```
This = wxBrush()
```

Returns true if the style of the brush is any of hatched fills.

See: `getStyle/1`

```
isOk(This) -> boolean()
```

Types:

```
This = wxBrush()
```

Returns true if the brush is initialised.

Notice that an uninitialized brush object can't be queried for any brush properties and all calls to the accessor methods on it will result in an assert failure.

```
setColour(This, Colour) -> ok
```

Types:

```
This = wxBrush()
Colour = wx:wx_colour()
```

Sets the brush colour using red, green and blue values.

See: `getColour/1`

```
setColour(This, Red, Green, Blue) -> ok
```

Types:

```
This = wxBrush()
Red = Green = Blue = integer()
```

```
setStipple(This, Bitmap) -> ok
```

Types:

```
This = wxBrush()
Bitmap = wxBitmap:wxBitmap()
```

Sets the stipple bitmap.

Remark: The style will be set to `wxBRUSHSTYLE_STIPPLE`, unless the bitmap has a mask associated to it, in which case the style will be set to `wxBRUSHSTYLE_STIPPLE_MASK_OPAQUE`.

See: `wxBitmap`

```
setStyle(This, Style) -> ok
```

Types:

```
This = wxBrush()
Style = wx:wx_enum()
```

Sets the brush style.

See: `getStyle/1`

# wxBufferedDC

Erlang module

This class provides a simple way to avoid flicker: when drawing on it, everything is in fact first drawn on an in-memory buffer (a `wxBitmap`) and then copied to the screen, using the associated `wxDC`, only once, when this object is destroyed. `wxBufferedDC` itself is typically associated with `wxClientDC`, if you want to use it in your `EVT_PAINT` handler, you should look at `wxBufferedPaintDC` instead.

When used like this, a valid `DC` must be specified in the constructor while the `buffer` bitmap doesn't have to be explicitly provided, by default this class will allocate the bitmap of required size itself. However using a dedicated bitmap can speed up the redrawing process by eliminating the repeated creation and destruction of a possibly big bitmap. Otherwise, `wxBufferedDC` can be used in the same way as any other device context.

Another possible use for `wxBufferedDC` is to use it to maintain a backing store for the window contents. In this case, the associated `DC` may be NULL but a valid backing store bitmap should be specified.

Finally, please note that GTK+ 2.0 as well as macOS provide double buffering themselves natively. You can either use `wxWindow:isDoubleBuffered/1` to determine whether you need to use buffering or not, or use `wxAutoBufferedPaintDC` (not implemented in wx) to avoid needless double buffering on the systems which already do it automatically.

See: `wxDC`, `wxMemoryDC`, `wxBufferedPaintDC`, `wxAutoBufferedPaintDC` (not implemented in wx)

This class is derived (and can use functions) from: `wxMemoryDC` `wxDC`

wxWidgets docs: **wxBufferedDC**

## Data Types

```
wxBufferedDC() = wx:wx_object()
```

## Exports

```
new() -> wxBufferedDC()
```

Default constructor.

You must call one of the `init/4` methods later in order to use the device context.

```
new(Dc) -> wxBufferedDC()
```

Types:

```
Dc = wxDC:wxDC()
```

```
new(Dc, Area) -> wxBufferedDC()
new(Dc, Area :: [Option]) -> wxBufferedDC()
```

Types:

```
Dc = wxDC:wxDC()
Option = {buffer, wxBitmap:wxBitmap()} | {style, integer()}
```

Creates a buffer for the provided dc.

`init/4` must not be called when using this constructor.

```
new(Dc, Area, Options :: [Option]) -> wxBufferedDC()
```

Types:

```
Dc = wxDC:wxDC()
Area = {W :: integer(), H :: integer()}
Option = {style, integer()}
```

Creates a buffer for the provided `dc`.

`init/4` must not be called when using this constructor.

```
destroy(This :: wxBufferedDC()) -> ok
```

Copies everything drawn on the DC so far to the underlying DC associated with this object, if any.

```
init(This, Dc) -> ok
```

Types:

```
This = wxBufferedDC()
Dc = wxDC:wxDC()
```

```
init(This, Dc, Area) -> ok
init(This, Dc, Area :: [Option]) -> ok
```

Types:

```
This = wxBufferedDC()
Dc = wxDC:wxDC()
Option = {buffer, wxBitmap:wxBitmap()} | {style, integer()}
```

```
init(This, Dc, Area, Options :: [Option]) -> ok
```

Types:

```
This = wxBufferedDC()
Dc = wxDC:wxDC()
Area = {W :: integer(), H :: integer()}
Option = {style, integer()}
```

Initializes the object created using the default constructor.

Please see the constructors for parameter details.

# wxBufferedPaintDC

Erlang module

This is a subclass of `wxBufferedDC` which can be used inside of an `EVT_PAINT()` event handler to achieve double-buffered drawing. Just use this class instead of `wxPaintDC` and make sure `wxWindow:setBackgroundStyle/2` is called with wxBG_STYLE_PAINT somewhere in the class initialization code, and that's all you have to do to (mostly) avoid flicker. The only thing to watch out for is that if you are using this class together with `wxScrolled` (not implemented in wx), you probably do `not` want to call `wxScrolledWindow:prepareDC/2` on it as it already does this internally for the real underlying `wxPaintDC`.

See: `wxDC`, `wxBufferedDC`, `wxAutoBufferedPaintDC` (not implemented in wx), `wxPaintDC`

This class is derived (and can use functions) from: `wxBufferedDC` `wxMemoryDC` `wxDC`

wxWidgets docs: **wxBufferedPaintDC**

## Data Types

```
wxBufferedPaintDC() = wx:wx_object()
```

## Exports

```
new(Window) -> wxBufferedPaintDC()
```

Types:

```
Window = wxWindow:wxWindow()
```

```
new(Window, Buffer) -> wxBufferedPaintDC()
new(Window, Buffer :: [Option]) -> wxBufferedPaintDC()
```

Types:

```
Window = wxWindow:wxWindow()
Option = {style, integer()}
```

```
new(Window, Buffer, Options :: [Option]) -> wxBufferedPaintDC()
```

Types:

```
Window = wxWindow:wxWindow()
Buffer = wxBitmap:wxBitmap()
Option = {style, integer()}
```

As with `wxBufferedDC`, you may either provide the bitmap to be used for buffering or let this object create one internally (in the latter case, the size of the client part of the window is used).

Pass wxBUFFER_CLIENT_AREA for the `style` parameter to indicate that just the client area of the window is buffered, or wxBUFFER_VIRTUAL_AREA to indicate that the buffer bitmap covers the virtual area.

```
destroy(This :: wxBufferedPaintDC()) -> ok
```

Copies everything drawn on the DC so far to the window associated with this object, using a `wxPaintDC`.

# wxButton

Erlang module

A button is a control that contains a text string, and is one of the most common elements of a GUI.

It may be placed on a `wxDialog` or on a `wxPanel` panel, or indeed on almost any other window.

By default, i.e. if none of the alignment styles are specified, the label is centered both horizontally and vertically. If the button has both a label and a bitmap, the alignment styles above specify the location of the rectangle combining both the label and the bitmap and the bitmap position set with `wxButton::SetBitmapPosition()` (not implemented in wx) defines the relative position of the bitmap with respect to the label (however currently non-default alignment combinations are not implemented on all platforms).

Since version 2.9.1 `wxButton` supports showing both text and an image (currently only when using wxMSW, wxGTK or wxOSX/Cocoa ports), see `SetBitmap()` (not implemented in wx) and `setBitmapLabel/2`, `setBitmapDisabled/2` &c methods. In the previous wxWidgets versions this functionality was only available in (the now trivial) `wxBitmapButton` class which was only capable of showing an image without text.

A button may have either a single image for all states or different images for the following states (different images are not currently supported under macOS where the normal image is used for all states):

All of the bitmaps must be of the same size and the normal bitmap must be set first (to a valid bitmap), before setting any other ones. Also, if the size of the bitmaps is changed later, you need to change the size of the normal bitmap before setting any other bitmaps with the new size (and you do need to reset all of them as their original values can be lost when the normal bitmap size changes).

The position of the image inside the button be configured using `SetBitmapPosition()` (not implemented in wx). By default the image is on the left of the text.

Please also notice that GTK+ uses a global setting called `gtk-button-images` to determine if the images should be shown in the buttons at all. If it is off (which is the case in e.g. Gnome 2.28 by default), no images will be shown, consistently with the native behaviour.

Styles

This class supports the following styles:

See: `wxBitmapButton`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxButton**

## Events

Event types emitted from this class: `command_button_clicked`

## Data Types

```
wxButton() = wx:wx_object()
```

## Exports

```
new() -> wxButton()
```

Default ctor.

```
new(Parent, Id) -> wxButton()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxButton()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Option =
    {label, unicode:chardata()} |
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {validator, wx:wx_object()}
```

Constructor, creating and showing a button.

The preferred way to create standard buttons is to use default value of `label`. If no label is supplied and `id` is one of standard IDs from this list, a standard label will be used. In other words, if you use a predefined `wxID_XXX` constant, just omit the label completely rather than specifying it. In particular, help buttons (the ones with `id` of `wxID_HELP`) under macOS can't display any label at all and while `wxButton` will detect if the standard "Help" label is used and ignore it, using any other label will prevent the button from correctly appearing as a help button and so should be avoided.

In addition to that, the button will be decorated with stock icons under GTK+ 2.

See: `create/4`, `wxValidator` (not implemented in wx)

```
create(This, Parent, Id) -> boolean()
```

Types:

```
This = wxButton()
Parent = wxWindow:wxWindow()
Id = integer()
```

```
create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
This = wxButton()
Parent = wxWindow:wxWindow()
Id = integer()
Option =
    {label, unicode:chardata()} |
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {validator, wx:wx_object()}
```

Button creation function for two-step creation.

For more details, see `new/3`.

```
getDefaultSize() -> {W :: integer(), H :: integer()}
```

Returns the default size for the buttons.

It is advised to make all the dialog buttons of the same size and this function allows retrieving the (platform, and current font dependent) size which should be the best suited for this.

The optional `win` argument is new since wxWidgets 3.1.3 and allows to get a per-monitor DPI specific size.

```
getDefaultSize(Win) -> {W :: integer(), H :: integer()}
```

Types:

```
Win = wxWindow:wxWindow()
```

```
setDefault(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxButton()
```

This sets the button to be the default item in its top-level window (e.g.

the panel or the dialog box containing it).

As normal, pressing return causes the default button to be depressed when the return key is pressed.

See also `wxWindow:setFocus/1` which sets the keyboard focus for windows and text panel items, and `wxTopLevelWindow::SetDefaultItem()` (not implemented in wx).

Remark: Under Windows, only dialog box buttons respond to this function.

Return: the old default item (possibly NULL)

```
setLabel(This, Label) -> ok
```

Types:

```
This = wxButton()
Label = unicode:chardata()
```

Sets the string label for the button.

```
getBitmapDisabled(This) -> wxBitmap:wxBitmap()
```

Types:

```
This = wxButton()
```

Returns the bitmap for the disabled state, which may be invalid.

See: `setBitmapDisabled/2`

Since: 2.9.1 (available in `wxBitmapButton` only in previous versions)

```
getBitmapFocus(This) -> wxBitmap:wxBitmap()
```

Types:

```
This = wxButton()
```

Returns the bitmap for the focused state, which may be invalid.

See: `setBitmapFocus/2`

Since: 2.9.1 (available in `wxBitmapButton` only in previous versions)

```
getBitmapLabel(This) -> wxBitmap:wxBitmap()
```

Types:

```
This = wxButton()
```

Returns the bitmap for the normal state.

This is exactly the same as `GetBitmap()` (not implemented in wx) but uses a name backwards-compatible with `wxBitmapButton`.

See: `SetBitmap()` (not implemented in wx), `setBitmapLabel/2`

Since: 2.9.1 (available in `wxBitmapButton` only in previous versions)

```
setBitmapDisabled(This, Bitmap) -> ok
```

Types:

```
This = wxButton()
Bitmap = wxBitmap:wxBitmap()
```

Sets the bitmap for the disabled button appearance.

If `bitmap` is invalid, the disabled bitmap is set to the automatically generated greyed out version of the normal bitmap, i.e. the same bitmap as is used by default if this method is not called at all. Use `SetBitmap()` (not implemented in wx) with an invalid bitmap to remove the bitmap completely (for all states).

See: `getBitmapDisabled/1`, `setBitmapLabel/2`, `SetBitmapPressed()` (not implemented in wx), `setBitmapFocus/2`

Since: 2.9.1 (available in `wxBitmapButton` only in previous versions)

```
setBitmapFocus(This, Bitmap) -> ok
```

Types:

```
This = wxButton()
Bitmap = wxBitmap:wxBitmap()
```

Sets the bitmap for the button appearance when it has the keyboard focus.

If `bitmap` is invalid, the normal bitmap will be used in the focused state.

See: `getBitmapFocus/1`, `setBitmapLabel/2`, `SetBitmapPressed()` (not implemented in wx), `setBitmapDisabled/2`

Since: 2.9.1 (available in `wxBitmapButton` only in previous versions)

```
setBitmapLabel(This, Bitmap) -> ok
```

Types:

```
This = wxButton()
Bitmap = wxBitmap:wxBitmap()
```

Sets the bitmap label for the button.

Remark: This is the bitmap used for the unselected state, and for all other states if no other bitmaps are provided.

See: `SetBitmap()` (not implemented in wx), `getBitmapLabel/1`

Since: 2.9.1 (available in `wxBitmapButton` only in previous versions)

```
destroy(This :: wxButton()) -> ok
```

Destroys the object.

# wxCalendarCtrl

Erlang module

The calendar control allows the user to pick a date. The user can move the current selection using the keyboard and select the date (generating EVT_CALENDAR event) by pressing <Return> or double clicking it.

Generic calendar has advanced possibilities for the customization of its display, described below. If you want to use these possibilities on every platform, use wxGenericCalendarCtrl instead of `wxCalendarCtrl`.

All global settings (such as colours and fonts used) can, of course, be changed. But also, the display style for each day in the month can be set independently using `wxCalendarDateAttr` class.

An item without custom attributes is drawn with the default colours and font and without border, but setting custom attributes with `setAttr/3` allows modifying its appearance. Just create a custom attribute object and set it for the day you want to be displayed specially (note that the control will take ownership of the pointer, i.e. it will delete it itself). A day may be marked as being a holiday, even if it is not recognized as one by `wx_datetime()` using the `wxCalendarDateAttr:setHoliday/2` method.

As the attributes are specified for each day, they may change when the month is changed, so you will often want to update them in EVT_CALENDAR_PAGE_CHANGED event handler.

If neither the `wxCAL_SUNDAY_FIRST` or `wxCAL_MONDAY_FIRST` style is given, the first day of the week is determined from operating system's settings, if possible. The native wxGTK calendar chooses the first weekday based on locale, and these styles have no effect on it.

Styles

This class supports the following styles:

Note: Changing the selected date will trigger an EVT_CALENDAR_DAY, MONTH or YEAR event as well as an EVT_CALENDAR_SEL_CHANGED event.

See: **Examples**, `wxCalendarDateAttr`, `wxCalendarEvent`, `wxDatePickerCtrl`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxCalendarCtrl**

## Events

Event types emitted from this class: `calendar_sel_changed`, `calendar_weekday_clicked`

## Data Types

```
wxCalendarCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxCalendarCtrl()
```

Default constructor.

```
new(Parent, Id) -> wxCalendarCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()

new(Parent, Id, Options :: [Option]) -> wxCalendarCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {date, wx:wx_datetime()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Does the same as `create/4` method.

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = wxCalendarCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()

create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxCalendarCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {date, wx:wx_datetime()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Creates the control.

See `wxWindow:new/3` for the meaning of the parameters and the control overview for the possible styles.

```
destroy(This :: wxCalendarCtrl()) -> ok
```

Destroys the control.

```
setDate(This, Date) -> boolean()
```

Types:

```
   This = wxCalendarCtrl()
   Date = wx:wx_datetime()
```

Sets the current date.

The `date` parameter must be valid and in the currently valid range as set by `SetDateRange()` (not implemented in wx), otherwise the current date is not changed and the function returns false and, additionally, triggers an assertion failure if the date is invalid.

```
getDate(This) -> wx:wx_datetime()
```

Types:

```
This = wxCalendarCtrl()
```

Gets the currently selected date.

```
enableYearChange(This) -> ok
```

Types:

```
This = wxCalendarCtrl()
```

```
enableYearChange(This, Options :: [Option]) -> ok
```

Types:

```
This = wxCalendarCtrl()
Option = {enable, boolean()}
```

Deprecated:

This function should be used instead of changing `wxCAL_NO_YEAR_CHANGE` style bit directly. It allows or disallows the user to change the year interactively. Only in generic `wxCalendarCtrl`.

```
enableMonthChange(This) -> boolean()
```

Types:

```
This = wxCalendarCtrl()
```

```
enableMonthChange(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxCalendarCtrl()
Option = {enable, boolean()}
```

This function should be used instead of changing `wxCAL_NO_MONTH_CHANGE` style bit.

It allows or disallows the user to change the month interactively. Note that if the month cannot be changed, the year cannot be changed neither.

Return: true if the value of this option really changed or false if it was already set to the requested value.

```
enableHolidayDisplay(This) -> ok
```

Types:

```
This = wxCalendarCtrl()
```

```
enableHolidayDisplay(This, Options :: [Option]) -> ok
```

Types:

```
This = wxCalendarCtrl()
Option = {display, boolean()}
```

This function should be used instead of changing `wxCAL_SHOW_HOLIDAYS` style bit directly.

It enables or disables the special highlighting of the holidays.

```
setHeaderColours(This, ColFg, ColBg) -> ok
```

Types:

```
This = wxCalendarCtrl()
ColFg = ColBg = wx:wx_colour()
```

Set the colours used for painting the weekdays at the top of the control.

This method is currently only implemented in generic `wxCalendarCtrl` and does nothing in the native versions.

`getHeaderColourFg(This) -> wx:wx_colour4()`

Types:

```
This = wxCalendarCtrl()
```

Gets the foreground colour of the header part of the calendar window.

This method is currently only implemented in generic `wxCalendarCtrl` and always returns `wxNullColour` in the native versions.

See: `setHeaderColours/3`

`getHeaderColourBg(This) -> wx:wx_colour4()`

Types:

```
This = wxCalendarCtrl()
```

Gets the background colour of the header part of the calendar window.

This method is currently only implemented in generic `wxCalendarCtrl` and always returns `wxNullColour` in the native versions.

See: `setHeaderColours/3`

`setHighlightColours(This, ColFg, ColBg) -> ok`

Types:

```
This = wxCalendarCtrl()
ColFg = ColBg = wx:wx_colour()
```

Set the colours to be used for highlighting the currently selected date.

This method is currently only implemented in generic `wxCalendarCtrl` and does nothing in the native versions.

`getHighlightColourFg(This) -> wx:wx_colour4()`

Types:

```
This = wxCalendarCtrl()
```

Gets the foreground highlight colour.

Only in generic `wxCalendarCtrl`.

This method is currently only implemented in generic `wxCalendarCtrl` and always returns `wxNullColour` in the native versions.

See: `setHighlightColours/3`

`getHighlightColourBg(This) -> wx:wx_colour4()`

Types:

```
This = wxCalendarCtrl()
```

Gets the background highlight colour.

Only in generic wxCalendarCtrl.

This method is currently only implemented in generic wxCalendarCtrl and always returns wxNullColour in the native versions.

See: setHighlightColours/3

```
setHolidayColours(This, ColFg, ColBg) -> ok
```

Types:

```
This = wxCalendarCtrl()
ColFg = ColBg = wx:wx_colour()
```

Sets the colours to be used for the holidays highlighting.

This method is only implemented in the generic version of the control and does nothing in the native ones. It should also only be called if the window style includes wxCAL_SHOW_HOLIDAYS flag or enableHolidayDisplay/2 had been called.

```
getHolidayColourFg(This) -> wx:wx_colour4()
```

Types:

```
This = wxCalendarCtrl()
```

Return the foreground colour currently used for holiday highlighting.

Only useful with generic wxCalendarCtrl as native versions currently don't support holidays display at all and always return wxNullColour.

See: setHolidayColours/3

```
getHolidayColourBg(This) -> wx:wx_colour4()
```

Types:

```
This = wxCalendarCtrl()
```

Return the background colour currently used for holiday highlighting.

Only useful with generic wxCalendarCtrl as native versions currently don't support holidays display at all and always return wxNullColour.

See: setHolidayColours/3

```
getAttr(This, Day) -> wxCalendarDateAttr:wxCalendarDateAttr()
```

Types:

```
This = wxCalendarCtrl()
Day = integer()
```

Returns the attribute for the given date (should be in the range 1...31).

The returned pointer may be NULL. Only in generic wxCalendarCtrl.

```
setAttr(This, Day, Attr) -> ok
```

Types:

```
This = wxCalendarCtrl()
Day = integer()
Attr = wxCalendarDateAttr:wxCalendarDateAttr()
```

Associates the attribute with the specified date (in the range 1...31).

If the pointer is NULL, the items attribute is cleared. Only in generic `wxCalendarCtrl`.

```
setHoliday(This, Day) -> ok
```

Types:

```
This = wxCalendarCtrl()
Day = integer()
```

Marks the specified day as being a holiday in the current month.

This method is only implemented in the generic version of the control and does nothing in the native ones.

```
resetAttr(This, Day) -> ok
```

Types:

```
This = wxCalendarCtrl()
Day = integer()
```

Clears any attributes associated with the given day (in the range 1...31).

Only in generic `wxCalendarCtrl`.

```
hitTest(This, Pos) -> Result
```

Types:

```
Result =
    {Res :: wx:wx_enum(),
     Date :: wx:wx_datetime(),
     Wd :: wx:wx_enum()}
This = wxCalendarCtrl()
Pos = {X :: integer(), Y :: integer()}
```

Returns one of wxCalendarHitTestResult constants and fills either `date` or `wd` pointer with the corresponding value depending on the hit test code.

Not implemented in wxGTK currently.

# wxCalendarDateAttr

Erlang module

`wxCalendarDateAttr` is a custom attributes for a calendar date. The objects of this class are used with `wxCalendarCtrl`.

See: `wxCalendarCtrl`

wxWidgets docs: **wxCalendarDateAttr**

## Data Types

```
wxCalendarDateAttr() = wx:wx_object()
```

## Exports

```
new() -> wxCalendarDateAttr()
```

```
new(Border) -> wxCalendarDateAttr()
new(Border :: [Option]) -> wxCalendarDateAttr()
```

Types:

```
Option =
    {colText, wx:wx_colour()} |
    {colBack, wx:wx_colour()} |
    {colBorder, wx:wx_colour()} |
    {font, wxFont:wxFont()} |
    {border, wx:wx_enum()}
```

Constructor for specifying all `wxCalendarDateAttr` properties.

```
new(Border, Options :: [Option]) -> wxCalendarDateAttr()
```

Types:

```
Border = wx:wx_enum()
Option = {colBorder, wx:wx_colour()}
```

Constructor using default properties except the given border.

```
setTextColour(This, ColText) -> ok
```

Types:

```
This = wxCalendarDateAttr()
ColText = wx:wx_colour()
```

Sets the text (foreground) colour to use.

```
setBackgroundColour(This, ColBack) -> ok
```

Types:

```
This = wxCalendarDateAttr()
ColBack = wx:wx_colour()
```

Sets the text background colour to use.

```
setBorderColour(This, Col) -> ok
```

Types:

```
This = wxCalendarDateAttr()
Col = wx:wx_colour()
```

Sets the border colour to use.

```
setFont(This, Font) -> ok
```

Types:

```
This = wxCalendarDateAttr()
Font = wxFont:wxFont()
```

Sets the font to use.

```
setBorder(This, Border) -> ok
```

Types:

```
This = wxCalendarDateAttr()
Border = wx:wx_enum()
```

Sets the border to use.

```
setHoliday(This, Holiday) -> ok
```

Types:

```
This = wxCalendarDateAttr()
Holiday = boolean()
```

If `holiday` is true, this calendar day will be displayed as a holiday.

```
hasTextColour(This) -> boolean()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns true if a non-default text foreground colour is set.

```
hasBackgroundColour(This) -> boolean()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns true if a non-default text background colour is set.

```
hasBorderColour(This) -> boolean()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns true if a non-default border colour is set.

```
hasFont(This) -> boolean()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns true if a non-default font is set.

```
hasBorder(This) -> boolean()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns true if a non-default (i.e. any) border is set.

```
isHoliday(This) -> boolean()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns true if this calendar day is displayed as a holiday.

```
getTextColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns the text colour set for the calendar date.

```
getBackgroundColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns the background colour set for the calendar date.

```
getBorderColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns the border colour set for the calendar date.

```
getFont(This) -> wxFont:wxFont()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns the font set for the calendar date.

```
getBorder(This) -> wx:wx_enum()
```

Types:

```
This = wxCalendarDateAttr()
```

Returns the border set for the calendar date.

```
destroy(This :: wxCalendarDateAttr()) -> ok
```

Destroys the object.

# wxCalendarEvent

Erlang module

The `wxCalendarEvent` class is used together with `wxCalendarCtrl`.

See: `wxCalendarCtrl`

This class is derived (and can use functions) from: `wxDateEvent` `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxCalendarEvent**

## Data Types

```
wxCalendarEvent() = wx:wx_object()
wxCalendar() =
    #wxCalendar{type = wxCalendarEvent:wxCalendarEventType(),
                wday = wx:wx_enum(),
                date = wx:wx_datetime()}
wxCalendarEventType() =
    calendar_sel_changed | calendar_day_changed |
    calendar_month_changed | calendar_year_changed |
    calendar_doubleclicked | calendar_weekday_clicked
```

## Exports

```
getWeekDay(This) -> wx:wx_enum()
```

Types:

```
   This = wxCalendarEvent()
```

Returns the week day on which the user clicked in `EVT_CALENDAR_WEEKDAY_CLICKED` handler.

It doesn't make sense to call this function in other handlers.

```
getDate(This) -> wx:wx_datetime()
```

Types:

```
   This = wxCalendarEvent()
```

Returns the date.

# wxCaret

Erlang module

A caret is a blinking cursor showing the position where the typed text will appear. Text controls usually have their own caret but `wxCaret` provides a way to use a caret in other windows.

Currently, the caret appears as a rectangle of the given size. In the future, it will be possible to specify a bitmap to be used for the caret shape.

A caret is always associated with a window and the current caret can be retrieved using `wxWindow:getCaret/1`. The same caret can't be reused in two different windows.

wxWidgets docs: **wxCaret**

## Data Types

```
wxCaret() = wx:wx_object()
```

## Exports

```
new(Window, Size) -> wxCaret()
```

Types:

```
Window = wxWindow:wxWindow()
Size = {W :: integer(), H :: integer()}
```

```
new(Window, Width, Height) -> wxCaret()
```

Types:

```
Window = wxWindow:wxWindow()
Width = Height = integer()
```

Creates a caret with the given size (in pixels) and associates it with the `window`.

```
create(This, Window, Size) -> boolean()
```

Types:

```
This = wxCaret()
Window = wxWindow:wxWindow()
Size = {W :: integer(), H :: integer()}
```

```
create(This, Window, Width, Height) -> boolean()
```

Types:

```
This = wxCaret()
Window = wxWindow:wxWindow()
Width = Height = integer()
```

Creates a caret with the given size (in pixels) and associates it with the `window` (same as the equivalent constructors).

```
getBlinkTime() -> integer()
```

Returns the blink time which is measured in milliseconds and is the time elapsed between 2 inversions of the caret (blink time of the caret is the same for all carets, so this functions is static).

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
   This = wxCaret()
```

```
getSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxCaret()
```

```
getWindow(This) -> wxWindow:wxWindow()
```

Types:

```
   This = wxCaret()
```

Get the window the caret is associated with.

```
hide(This) -> ok
```

Types:

```
   This = wxCaret()
```

Hides the caret, same as Show(false).

```
isOk(This) -> boolean()
```

Types:

```
   This = wxCaret()
```

Returns true if the caret was created successfully.

```
isVisible(This) -> boolean()
```

Types:

```
   This = wxCaret()
```

Returns true if the caret is visible and false if it is permanently hidden (if it is blinking and not shown currently but will be after the next blink, this method still returns true).

```
move(This, Pt) -> ok
```

Types:

```
   This = wxCaret()
   Pt = {X :: integer(), Y :: integer()}
```

```
move(This, X, Y) -> ok
```

Types:

```
   This = wxCaret()
   X = Y = integer()
```

Move the caret to given position (in logical coordinates).

```
setBlinkTime(Milliseconds) -> ok
```

Types:

```
Milliseconds = integer()
```

Sets the blink time for all the carets.

Warning: Under Windows, this function will change the blink time for all carets permanently (until the next time it is called), even for carets in other applications.

See: `getBlinkTime/0`

```
setSize(This, Size) -> ok
```

Types:

```
This = wxCaret()
Size = {W :: integer(), H :: integer()}
```

```
setSize(This, Width, Height) -> ok
```

Types:

```
This = wxCaret()
Width = Height = integer()
```

Changes the size of the caret.

```
show(This) -> ok
```

Types:

```
This = wxCaret()
```

```
show(This, Options :: [Option]) -> ok
```

Types:

```
This = wxCaret()
Option = {show, boolean()}
```

Shows or hides the caret.

Notice that if the caret was hidden N times, it must be shown N times as well to reappear on the screen.

```
destroy(This :: wxCaret()) -> ok
```

Destroys the object.

# wxCheckBox

Erlang module

A checkbox is a labelled box which by default is either on (checkmark is visible) or off (no checkmark). Optionally (when the wxCHK_3STATE style flag is set) it can have a third state, called the mixed or undetermined state. Often this is used as a "Does Not Apply" state.

Styles

This class supports the following styles:

See: `wxRadioButton`, `wxCommandEvent`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxCheckBox**

## Events

Event types emitted from this class: `command_checkbox_clicked`

## Data Types

```
wxCheckBox() = wx:wx_object()
```

## Exports

```
new() -> wxCheckBox()
```

Default constructor.

See: `create/5`, `wxValidator` (not implemented in wx)

```
new(Parent, Id, Label) -> wxCheckBox()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Label = unicode:chardata()
```

```
new(Parent, Id, Label, Options :: [Option]) -> wxCheckBox()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Label = unicode:chardata()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {validator, wx:wx_object()}
```

Constructor, creating and showing a checkbox.

See: `create/5`, `wxValidator` (not implemented in wx)

```
destroy(This :: wxCheckBox()) -> ok
```

Destructor, destroying the checkbox.

```
create(This, Parent, Id, Label) -> boolean()
```

Types:

```
   This = wxCheckBox()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
```

```
create(This, Parent, Id, Label, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxCheckBox()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Creates the checkbox for two-step construction.

See `new/4` for details.

```
getValue(This) -> boolean()
```

Types:

```
   This = wxCheckBox()
```

Gets the state of a 2-state checkbox.

Return: Returns true if it is checked, false otherwise.

```
get3StateValue(This) -> wx:wx_enum()
```

Types:

```
   This = wxCheckBox()
```

Gets the state of a 3-state checkbox.

Asserts when the function is used with a 2-state checkbox.

```
is3rdStateAllowedForUser(This) -> boolean()
```

Types:

```
   This = wxCheckBox()
```

Returns whether or not the user can set the checkbox to the third state.

Return: true if the user can set the third state of this checkbox, false if it can only be set programmatically or if it's a 2-state checkbox.

```
is3State(This) -> boolean()
```

Types:

```
This = wxCheckBox()
```

Returns whether or not the checkbox is a 3-state checkbox.

Return: true if this checkbox is a 3-state checkbox, false if it's a 2-state checkbox.

```
isChecked(This) -> boolean()
```

Types:

```
This = wxCheckBox()
```

This is just a maybe more readable synonym for `getValue/1`: just as the latter, it returns true if the checkbox is checked and false otherwise.

```
setValue(This, State) -> ok
```

Types:

```
This = wxCheckBox()
State = boolean()
```

Sets the checkbox to the given state.

This does not cause a `wxEVT_CHECKBOX` event to get emitted.

```
set3StateValue(This, State) -> ok
```

Types:

```
This = wxCheckBox()
State = wx:wx_enum()
```

Sets the checkbox to the given state.

This does not cause a `wxEVT_CHECKBOX` event to get emitted.

Asserts when the checkbox is a 2-state checkbox and setting the state to wxCHK_UNDETERMINED.

# wxCheckListBox

Erlang module

A `wxCheckListBox` is like a `wxListBox`, but allows items to be checked or unchecked.

When using this class under Windows wxWidgets must be compiled with wxUSE_OWNER_DRAWN set to 1.

See: `wxListBox`, `wxChoice`, `wxComboBox`, `wxListCtrl`, `wxCommandEvent`

This class is derived (and can use functions) from: `wxListBox` `wxControlWithItems` `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxCheckListBox**

## Events

Event types emitted from this class: `command_checklistbox_toggled`

## Data Types

```
wxCheckListBox() = wx:wx_object()
```

## Exports

```
new() -> wxCheckListBox()
```

Default constructor.

```
new(Parent, Id) -> wxCheckListBox()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxCheckListBox()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {choices, [unicode:chardata()]} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Constructor, creating and showing a list box.

```
destroy(This :: wxCheckListBox()) -> ok
```

Destructor, destroying the list box.

```
check(This, Item) -> ok
```

Types:

```
   This = wxCheckListBox()
   Item = integer()

check(This, Item, Options :: [Option]) -> ok
```

Types:

```
   This = wxCheckListBox()
   Item = integer()
   Option = {check, boolean()}
```

Checks the given item.

Note that calling this method does not result in a `wxEVT_CHECKLISTBOX` event being emitted.

```
isChecked(This, Item) -> boolean()
```

Types:

```
   This = wxCheckListBox()
   Item = integer()
```

Returns true if the given item is checked, false otherwise.

# wxChildFocusEvent

Erlang module

A child focus event is sent to a (parent-)window when one of its child windows gains focus, so that the window could restore the focus back to its corresponding child if it loses it now and regains later.

Notice that child window is the direct child of the window receiving event. Use `wxWindow:findFocus/0` to retrieve the window which is actually getting focus.

See: **Overview events**

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxChildFocusEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxChildFocusEventType` to subscribe to events of this type.

## Data Types

```
wxChildFocusEvent() = wx:wx_object()
wxChildFocus() =
    #wxChildFocus{type =
                      wxChildFocusEvent:wxChildFocusEventType()}
wxChildFocusEventType() = child_focus
```

## Exports

```
getWindow(This) -> wxWindow:wxWindow()
```

Types:

```
   This = wxChildFocusEvent()
```

Returns the direct child which receives the focus, or a (grand-)parent of the control receiving the focus.

To get the actually focused control use `wxWindow:findFocus/0`.

# wxChoice

Erlang module

A choice item is used to select one of a list of strings. Unlike a `wxListBox`, only the selection is visible until the user pulls down the menu of choices.

Styles

This class supports the following styles:

See: `wxListBox`, `wxComboBox`, `wxCommandEvent`

This class is derived (and can use functions) from: `wxControlWithItems wxControl wxWindow wxEvtHandler`

wxWidgets docs: **wxChoice**

## Events

Event types emitted from this class: `command_choice_selected`

## Data Types

```
wxChoice() = wx:wx_object()
```

## Exports

```
new() -> wxChoice()
```

Default constructor.

See: `create/7`, `wxValidator` (not implemented in wx)

```
new(Parent, Id) -> wxChoice()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxChoice()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {choices, [unicode:chardata()]} |
    {style, integer()} |
    {validator, wx:wx_object()}
```

Constructor, creating and showing a choice.

See: `create/7`, `wxValidator` (not implemented in wx)

```
destroy(This :: wxChoice()) -> ok
```

Destructor, destroying the choice item.

```
create(This, Parent, Id, Pos, Size, Choices) -> boolean()
```

Types:

```
   This = wxChoice()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Pos = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Choices = [unicode:chardata()]
```

```
create(This, Parent, Id, Pos, Size, Choices, Options :: [Option]) ->
          boolean()
```

Types:

```
   This = wxChoice()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Pos = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Choices = [unicode:chardata()]
   Option = {style, integer()} | {validator, wx:wx_object()}
```

```
delete(This, N) -> ok
```

Types:

```
   This = wxChoice()
   N = integer()
```

Deletes an item from the control.

The client data associated with the item will be also deleted if it is owned by the control. Note that it is an error (signalled by an assert failure in debug builds) to remove an item with the index negative or greater or equal than the number of items in the control.

If there is a currently selected item below the item being deleted, i.e. if `wxControlWithItems:getSelection/1` returns a valid index greater than or equal to `n`, the selection is invalidated when this function is called. However if the selected item appears before the item being deleted, the selection is preserved unchanged.

See: `wxControlWithItems:clear/1`

```
getColumns(This) -> integer()
```

Types:

```
   This = wxChoice()
```

Gets the number of columns in this choice item.

Remark: This is implemented for GTK and Motif only and always returns 1 for the other platforms.

```
setColumns(This) -> ok
```

Types:

```
This = wxChoice()
```

```
setColumns(This, Options :: [Option]) -> ok
```

Types:

```
This = wxChoice()
Option = {n, integer()}
```

Sets the number of columns in this choice item.

Remark: This is implemented for GTK and Motif only and doesn't do anything under other platforms.

# wxChoicebook

Erlang module

wxChoicebook is a class similar to wxNotebook, but uses a wxChoice control to show the labels instead of the tabs.

For usage documentation of this class, please refer to the base abstract class wxBookCtrl. You can also use the page_samples_notebook to see wxChoicebook in action.

wxChoicebook allows the use of wxBookCtrlBase::GetControlSizer(), allowing a program to add other controls next to the choice control. This is particularly useful when screen space is restricted, as it often is when wxChoicebook is being employed.

Styles

This class supports the following styles:

See: **Overview bookctrl**, wxNotebook, **Examples**

This class is derived (and can use functions) from: wxBookCtrlBase wxControl wxWindow wxEvtHandler

wxWidgets docs: **wxChoicebook**

## Events

Event types emitted from this class: choicebook_page_changed, choicebook_page_changing

## Data Types

```
wxChoicebook() = wx:wx_object()
```

## Exports

```
new() -> wxChoicebook()
```

Constructs a choicebook control.

```
new(Parent, Id) -> wxChoicebook()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxChoicebook()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

```
addPage(This, Page, Text) -> boolean()
```

Types:

```
   This = wxChoicebook()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()

addPage(This, Page, Text, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxChoicebook()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
   Option = {bSelect, boolean()} | {imageId, integer()}
```

Adds a new page.

The page must have the book control itself as the parent and must not have been added to this control previously.

The call to this function will generate the page changing and page changed events if `select` is true, but not when inserting the very first page (as there is no previous page selection to switch from in this case and so it wouldn't make sense to e.g. veto such event).

Return: true if successful, false otherwise.

Remark: Do not delete the page, it will be deleted by the book control.

See: `insertPage/5`

```
advanceSelection(This) -> ok
```

Types:

```
   This = wxChoicebook()

advanceSelection(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxChoicebook()
   Option = {forward, boolean()}
```

Cycles through the tabs.

The call to this function generates the page changing events.

```
assignImageList(This, ImageList) -> ok
```

Types:

```
   This = wxChoicebook()
   ImageList = wxImageList:wxImageList()
```

Sets the image list for the page control and takes ownership of the list.

See: `wxImageList`, `setImageList/2`

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = wxChoicebook()
   Parent = wxWindow:wxWindow()
   Id = integer()

create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxChoicebook()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Create the choicebook control that has already been constructed with the default constructor.

```
deleteAllPages(This) -> boolean()
```

Types:

```
   This = wxChoicebook()
```

Deletes all pages.

```
getCurrentPage(This) -> wxWindow:wxWindow()
```

Types:

```
   This = wxChoicebook()
```

Returns the currently selected page or NULL.

```
getImageList(This) -> wxImageList:wxImageList()
```

Types:

```
   This = wxChoicebook()
```

Returns the associated image list, may be NULL.

See: `wxImageList`, `setImageList/2`

```
getPage(This, Page) -> wxWindow:wxWindow()
```

Types:

```
   This = wxChoicebook()
   Page = integer()
```

Returns the window at the given page position.

```
getPageCount(This) -> integer()
```

Types:

```
   This = wxChoicebook()
```

Returns the number of pages in the control.

```
getPageImage(This, NPage) -> integer()
```

Types:

```
   This = wxChoicebook()
   NPage = integer()
```

Returns the image index for the given page.

```
getPageText(This, NPage) -> unicode:charlist()
```

Types:

```
   This = wxChoicebook()
   NPage = integer()
```

Returns the string for the given page.

```
getSelection(This) -> integer()
```

Types:

```
   This = wxChoicebook()
```

Returns the currently selected page, or `wxNOT_FOUND` if none was selected.

Note that this method may return either the previously or newly selected page when called from the `EVT_BOOKCTRL_PAGE_CHANGED` handler depending on the platform and so `wxBookCtrlEvent:getSelection/1` should be used instead in this case.

```
hitTest(This, Pt) -> Result
```

Types:

```
   Result = {Res :: integer(), Flags :: integer()}
   This = wxChoicebook()
   Pt = {X :: integer(), Y :: integer()}
```

Returns the index of the tab at the specified position or `wxNOT_FOUND` if none.

If `flags` parameter is non-NULL, the position of the point inside the tab is returned as well.

Return: Returns the zero-based tab index or `wxNOT_FOUND` if there is no tab at the specified position.

```
insertPage(This, Index, Page, Text) -> boolean()
```

Types:

```
   This = wxChoicebook()
   Index = integer()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
```

```
insertPage(This, Index, Page, Text, Options :: [Option]) ->
              boolean()
```

Types:

```
This = wxChoicebook()
Index = integer()
Page = wxWindow:wxWindow()
Text = unicode:chardata()
Option = {bSelect, boolean()} | {imageId, integer()}
```

Inserts a new page at the specified position.

Return: true if successful, false otherwise.

Remark: Do not delete the page, it will be deleted by the book control.

See: `addPage/4`

`setImageList(This, ImageList) -> ok`

Types:

```
This = wxChoicebook()
ImageList = wxImageList:wxImageList()
```

Sets the image list to use.

It does not take ownership of the image list, you must delete it yourself.

See: `wxImageList`, `assignImageList/2`

`setPageSize(This, Size) -> ok`

Types:

```
This = wxChoicebook()
Size = {W :: integer(), H :: integer()}
```

Sets the width and height of the pages.

Note: This method is currently not implemented for wxGTK.

`setPageImage(This, Page, Image) -> boolean()`

Types:

```
This = wxChoicebook()
Page = Image = integer()
```

Sets the image index for the given page.

`image` is an index into the image list which was set with `setImageList/2`.

`setPageText(This, Page, Text) -> boolean()`

Types:

```
This = wxChoicebook()
Page = integer()
Text = unicode:chardata()
```

Sets the text for the given page.

`setSelection(This, Page) -> integer()`

Types:

```
This = wxChoicebook()
Page = integer()
```

Sets the selection to the given page, returning the previous selection.

Notice that the call to this function generates the page changing events, use the `changeSelection/2` function if you don't want these events to be generated.

See: `getSelection/1`

```
changeSelection(This, Page) -> integer()
```

Types:

```
This = wxChoicebook()
Page = integer()
```

Changes the selection to the given page, returning the previous selection.

This function behaves as `setSelection/2` but does `not` generate the page changing events.

See overview_events_prog for more information.

```
destroy(This :: wxChoicebook()) -> ok
```

Destroys the object.

# wxClientDC

Erlang module

wxClientDC is primarily useful for obtaining information about the window from outside EVT_PAINT() handler.

Typical use of this class is to obtain the extent of some text string in order to allocate enough size for a window, e.g.

Note: While wxClientDC may also be used for drawing on the client area of a window from outside an EVT_PAINT() handler in some ports, this does not work on all platforms (neither wxOSX nor wxGTK with GTK 3 Wayland backend support this, so drawing using wxClientDC simply doesn't have any effect there) and the only portable way of drawing is via wxPaintDC. To redraw a small part of the window, use wxWindow:refreshRect/3 to invalidate just this part and check wxWindow:getUpdateRegion/1 in the paint event handler to redraw this part only.

wxClientDC objects should normally be constructed as temporary stack objects, i.e. don't store a wxClientDC object.

A wxClientDC object is initialized to use the same font and colours as the window it is associated with.

See: wxDC, wxMemoryDC, wxPaintDC, wxWindowDC, wxScreenDC

This class is derived (and can use functions) from: wxWindowDC wxDC

wxWidgets docs: **wxClientDC**

## Data Types

```
wxClientDC() = wx:wx_object()
```

## Exports

```
new(Window) -> wxClientDC()
```

Types:

```
Window = wxWindow:wxWindow()
```

Constructor.

Pass a pointer to the window on which you wish to paint.

```
destroy(This :: wxClientDC()) -> ok
```

Destroys the object.

# wxClipboard

Erlang module

A class for manipulating the clipboard.

To use the clipboard, you call member functions of the global ?wxTheClipboard object.

See the overview_dataobject for further information.

Call `open/1` to get ownership of the clipboard. If this operation returns true, you now own the clipboard. Call `setData/2` to put data on the clipboard, or `getData/2` to retrieve data from the clipboard. Call `close/1` to close the clipboard and relinquish ownership. You should keep the clipboard open only momentarily.

For example:

Note: On GTK, the clipboard behavior can vary depending on the configuration of the end-user's machine. In order for the clipboard data to persist after the window closes, a clipboard manager must be installed. Some clipboard managers will automatically flush the clipboard after each new piece of data is added, while others will not. The @Flush() function will force the clipboard manager to flush the data.

See: **Overview dnd**, **Overview dataobject**, `wxDataObject`

wxWidgets docs: **wxClipboard**

## Data Types

```
wxClipboard() = wx:wx_object()
```

## Exports

```
new() -> wxClipboard()
```

Default constructor.

```
destroy(This :: wxClipboard()) -> ok
```

Destructor.

```
addData(This, Data) -> boolean()
```

Types:

```
   This = wxClipboard()
   Data = wxDataObject:wxDataObject()
```

Call this function to add the data object to the clipboard.

This is an obsolete synonym for `setData/2`.

```
clear(This) -> ok
```

Types:

```
   This = wxClipboard()
```

Clears the global clipboard object and the system's clipboard if possible.

```
close(This) -> ok
```

Types:

```
This = wxClipboard()
```

Call this function to close the clipboard, having opened it with `open/1`.

```
flush(This) -> boolean()
```

Types:

```
This = wxClipboard()
```

Flushes the clipboard: this means that the data which is currently on clipboard will stay available even after the application exits (possibly eating memory), otherwise the clipboard will be emptied on exit.

Currently this method is implemented in MSW and GTK and always returns false otherwise.

Note: On GTK, only the non-primary selection can be flushed. Calling this function when the clipboard is using the primary selection will return false and not make any data available after the program exits.

Return: false if the operation is unsuccessful for any reason.

```
getData(This, Data) -> boolean()
```

Types:

```
This = wxClipboard()
Data = wxDataObject:wxDataObject()
```

Call this function to fill `data` with data on the clipboard, if available in the required format.

Returns true on success.

```
isOpened(This) -> boolean()
```

Types:

```
This = wxClipboard()
```

Returns true if the clipboard has been opened.

```
open(This) -> boolean()
```

Types:

```
This = wxClipboard()
```

Call this function to open the clipboard before calling `setData/2` and `getData/2`.

Call `close/1` when you have finished with the clipboard. You should keep the clipboard open for only a very short time.

Return: true on success. This should be tested (as in the sample shown above).

```
setData(This, Data) -> boolean()
```

Types:

```
This = wxClipboard()
Data = wxDataObject:wxDataObject()
```

Call this function to set the data object to the clipboard.

The new data object replaces any previously set one, so if the application wants to provide clipboard data in several different formats, it must use a composite data object supporting all of the formats instead of calling this function several times with different data objects as this would only leave data from the last one in the clipboard.

After this function has been called, the clipboard owns the data, so do not delete the data explicitly.

```
usePrimarySelection(This) -> ok
```

Types:

```
This = wxClipboard()
```

```
usePrimarySelection(This, Options :: [Option]) -> ok
```

Types:

```
This = wxClipboard()
Option = {primary, boolean()}
```

On platforms supporting it (all X11-based ports), `wxClipboard` uses the CLIPBOARD X11 selection by default.

When this function is called with true, all subsequent clipboard operations will use PRIMARY selection until this function is called again with false.

On the other platforms, there is no PRIMARY selection and so all clipboard operations will fail. This allows implementing the standard X11 handling of the clipboard which consists in copying data to the CLIPBOARD selection only when the user explicitly requests it (i.e. by selecting the "Copy" menu command) but putting the currently selected text into the PRIMARY selection automatically, without overwriting the normal clipboard contents with the currently selected text on the other platforms.

```
isSupported(This, Format) -> boolean()
```

Types:

```
This = wxClipboard()
Format = wx:wx_enum()
```

Returns true if there is data which matches the data format of the given data object currently `available` on the clipboard.

```
get() -> wxClipboard()
```

Returns the global instance (wxTheClipboard) of the clipboard object.

# wxClipboardTextEvent

Erlang module

This class represents the events generated by a control (typically a `wxTextCtrl` but other windows can generate these events as well) when its content gets copied or cut to, or pasted from the clipboard.

There are three types of corresponding events `wxEVT_TEXT_COPY`, `wxEVT_TEXT_CUT` and `wxEVT_TEXT_PASTE`.

If any of these events is processed (without being skipped) by an event handler, the corresponding operation doesn't take place which allows preventing the text from being copied from or pasted to a control. It is also possible to examine the clipboard contents in the PASTE event handler and transform it in some way before inserting in a control - for example, changing its case or removing invalid characters.

Finally notice that a CUT event is always preceded by the COPY event which makes it possible to only process the latter if it doesn't matter if the text was copied or cut.

Note: These events are currently only generated by `wxTextCtrl` in wxGTK and wxOSX but are also generated by `wxComboBox` without wxCB_READONLY style in wxMSW.

See: `wxClipboard`

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxClipboardTextEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxClipboardTextEventType` to subscribe to events of this type.

## Data Types

```
wxClipboardTextEvent() = wx:wx_object()
wxClipboardText() =
    #wxClipboardText{type =
                         wxClipboardTextEvent:wxClipboardTextEventType()}
wxClipboardTextEventType() =
    command_text_copy | command_text_cut | command_text_paste
```

# wxCloseEvent

Erlang module

This event class contains information about window and session close events.

The handler function for EVT_CLOSE is called when the user has tried to close a a frame or dialog box using the window manager (X) or system menu (Windows). It can also be invoked by the application itself programmatically, for example by calling the `wxWindow:close/2` function.

You should check whether the application is forcing the deletion of the window using `canVeto/1`. If this is false, you `must` destroy the window using `wxWindow:'Destroy'/1`.

If the return value is true, it is up to you whether you respond by destroying the window.

If you don't destroy the window, you should call `veto/2` to let the calling code know that you did not destroy the window. This allows the `wxWindow:close/2` function to return true or false depending on whether the close instruction was honoured or not.

Example of a `wxCloseEvent` handler:

The EVT_END_SESSION event is slightly different as it is sent by the system when the user session is ending (e.g. because of log out or shutdown) and so all windows are being forcefully closed. At least under MSW, after the handler for this event is executed the program is simply killed by the system. Because of this, the default handler for this event provided by wxWidgets calls all the usual cleanup code (including `wxApp::OnExit()` (not implemented in wx)) so that it could still be executed and exit()s the process itself, without waiting for being killed. If this behaviour is for some reason undesirable, make sure that you define a handler for this event in your wxApp-derived class and do not call `event.Skip()` in it (but be aware that the system will still kill your application).

See: `wxWindow:close/2`, **Overview windowdeletion**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxCloseEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxCloseEventType` to subscribe to events of this type.

## Data Types

```
wxCloseEvent() = wx:wx_object()
wxClose() = #wxClose{type = wxCloseEvent:wxCloseEventType()}
wxCloseEventType() =
    close_window | end_session | query_end_session
```

## Exports

```
canVeto(This) -> boolean()
```

Types:

```
   This = wxCloseEvent()
```

Returns true if you can veto a system shutdown or a window close event.

Vetoing a window close event is not possible if the calling code wishes to force the application to exit, and so this function must be called to check this.

```
getLoggingOff(This) -> boolean()
```

Types:

```
   This = wxCloseEvent()
```

Returns true if the user is just logging off or false if the system is shutting down.

This method can only be called for end session and query end session events, it doesn't make sense for close window event.

```
setCanVeto(This, CanVeto) -> ok
```

Types:

```
   This = wxCloseEvent()
   CanVeto = boolean()
```

Sets the 'can veto' flag.

```
setLoggingOff(This, LoggingOff) -> ok
```

Types:

```
   This = wxCloseEvent()
   LoggingOff = boolean()
```

Sets the 'logging off' flag.

```
veto(This) -> ok
```

Types:

```
   This = wxCloseEvent()
```

```
veto(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxCloseEvent()
   Option = {veto, boolean()}
```

Call this from your event handler to veto a system shutdown or to signal to the calling application that a window close did not happen.

You can only veto a shutdown if `canVeto/1` returns true.

# wxColourData

Erlang module

This class holds a variety of information related to colour dialogs.

See: `wx_color()`, `wxColourDialog`, **Overview cmndlg**

wxWidgets docs: **wxColourData**

## Data Types

```
wxColourData() = wx:wx_object()
```

## Exports

```
new() -> wxColourData()
```

Constructor.

Initializes the custom colours to `wxNullColour`, the `data` colour setting to black, and the `choose` full setting to true.

```
destroy(This :: wxColourData()) -> ok
```

Destructor.

```
getChooseFull(This) -> boolean()
```

Types:

```
This = wxColourData()
```

Under Windows, determines whether the Windows colour dialog will display the full dialog with custom colour selection controls.

Has no meaning under other platforms.

The default value is true.

```
getColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxColourData()
```

Gets the current colour associated with the colour dialog.

The default colour is black.

```
getCustomColour(This, I) -> wx:wx_colour4()
```

Types:

```
This = wxColourData()
I = integer()
```

Returns custom colours associated with the colour dialog.

```
setChooseFull(This, Flag) -> ok
```

Types:

```
This = wxColourData()
Flag = boolean()
```

Under Windows, tells the Windows colour dialog to display the full dialog with custom colour selection controls.

Under other platforms, has no effect.

The default value is true.

```
setColour(This, Colour) -> ok
```

Types:

```
This = wxColourData()
Colour = wx:wx_colour()
```

Sets the default colour for the colour dialog.

The default colour is black.

```
setCustomColour(This, I, Colour) -> ok
```

Types:

```
This = wxColourData()
I = integer()
Colour = wx:wx_colour()
```

Sets custom colours for the colour dialog.

# wxColourDialog

Erlang module

This class represents the colour chooser dialog.

Starting from wxWidgets 3.1.3 and currently in the MSW port only, this dialog generates wxEVT_COLOUR_CHANGED events while it is being shown, i.e. from inside its `wxDialog:showModal/1` method, that notify the program about the change of the currently selected colour and allow it to e.g. preview the effect of selecting this colour. Note that if you react to this event, you should also correctly revert to the previously selected colour if the dialog is cancelled by the user.

Example of using this class with dynamic feedback for the selected colour:

See: **Overview cmndlg**, `wx_color()`, `wxColourData`, `wxColourDialogEvent` (not implemented in wx), ? wxGetColourFromUser()

This class is derived (and can use functions) from: `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxColourDialog**

## Data Types

```
wxColourDialog() = wx:wx_object()
```

## Exports

```
new() -> wxColourDialog()
```

```
new(Parent) -> wxColourDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxColourDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
Option = {data, wxColourData:wxColourData()}
```

Constructor.

Pass a parent window, and optionally a pointer to a block of colour data, which will be copied to the colour dialog's colour data.

Custom colours from colour data object will be used in the dialog's colour palette. Invalid entries in custom colours list will be ignored on some platforms(GTK) or replaced with white colour on platforms where custom colours palette has fixed size (MSW).

See: `wxColourData`

```
destroy(This :: wxColourDialog()) -> ok
```

Destructor.

```
create(This, Parent) -> boolean()
```
Types:
```
   This = wxColourDialog()
   Parent = wxWindow:wxWindow()
```

```
create(This, Parent, Options :: [Option]) -> boolean()
```
Types:
```
   This = wxColourDialog()
   Parent = wxWindow:wxWindow()
   Option = {data, wxColourData:wxColourData()}
```
Same as `new/2`.

```
getColourData(This) -> wxColourData:wxColourData()
```
Types:
```
   This = wxColourDialog()
```
Returns the colour data associated with the colour dialog.

# wxColourPickerCtrl

Erlang module

This control allows the user to select a colour. The generic implementation is a button which brings up a `wxColourDialog` when clicked. Native implementation may differ but this is usually a (small) widget which give access to the colour-chooser dialog. It is only available if `wxUSE_COLOURPICKERCTRL` is set to 1 (the default).

Styles

This class supports the following styles:

See: `wxColourDialog`, `wxColourPickerEvent`

This class is derived (and can use functions) from: `wxPickerBase` `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxColourPickerCtrl**

## Events

Event types emitted from this class: `command_colourpicker_changed`

## Data Types

```
wxColourPickerCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxColourPickerCtrl()

new(Parent, Id) -> wxColourPickerCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()

new(Parent, Id, Options :: [Option]) -> wxColourPickerCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {col, wx:wx_colour()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Initializes the object and calls `create/4` with all the parameters.

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = wxColourPickerCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()

create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxColourPickerCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {col, wx:wx_colour()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Creates a colour picker with the given arguments.

Return: true if the control was successfully created or false if creation failed.

```
getColour(This) -> wx:wx_colour4()
```

Types:

```
   This = wxColourPickerCtrl()
```

Returns the currently selected colour.

```
setColour(This, Colname) -> ok
setColour(This, Col) -> ok
```

Types:

```
   This = wxColourPickerCtrl()
   Col = wx:wx_colour()
```

Sets the currently selected colour.

See `wxColour::Set()` (not implemented in wx).

```
destroy(This :: wxColourPickerCtrl()) -> ok
```

Destroys the object.

# wxColourPickerEvent

Erlang module

This event class is used for the events generated by `wxColourPickerCtrl`.

See: `wxColourPickerCtrl`

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxColourPickerEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxColourPickerEventType` to subscribe to events of this type.

## Data Types

```
wxColourPickerEvent() = wx:wx_object()
wxColourPicker() =
    #wxColourPicker{type =
                        wxColourPickerEvent:wxColourPickerEventType(),
                    colour = wx:wx_colour()}
wxColourPickerEventType() = command_colourpicker_changed
```

## Exports

```
getColour(This) -> wx:wx_colour4()
```

Types:

```
   This = wxColourPickerEvent()
```

Retrieve the colour the user has just selected.

# wxComboBox

Erlang module

A combobox is like a combination of an edit control and a listbox.

It can be displayed as static list with editable or read-only text field; or a drop-down list with text field; or a drop-down list without a text field depending on the platform and presence of wxCB_READONLY style.

A combobox permits a single selection only. Combobox items are numbered from zero.

If you need a customized combobox, have a look at `wxComboCtrl` (not implemented in wx), `wxOwnerDrawnComboBox` (not implemented in wx), `wxComboPopup` (not implemented in wx) and the ready-to-use `wxBitmapComboBox` (not implemented in wx).

Please refer to `wxTextEntry` (not implemented in wx) documentation for the description of methods operating with the text entry part of the combobox and to `wxItemContainer` (not implemented in wx) for the methods operating with the list of strings. Notice that at least under MSW `wxComboBox` doesn't behave correctly if it contains strings differing in case only so portable programs should avoid adding such strings to this control.

Styles

This class supports the following styles:

See: `wxListBox`, `wxTextCtrl`, `wxChoice`, `wxCommandEvent`

This class is derived (and can use functions) from: `wxControlWithItems wxControl wxWindow wxEvtHandler`

wxWidgets docs: **wxComboBox**

## Events

Event types emitted from this class: `command_combobox_selected`, `command_text_updated`, `command_text_enter`, `combobox_dropdown`, `combobox_closeup`

## Data Types

```
wxComboBox() = wx:wx_object()
```

## Exports

```
new() -> wxComboBox()
```

Default constructor.

```
new(Parent, Id) -> wxComboBox()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxComboBox()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {value, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {choices, [unicode:chardata()]} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Constructor, creating and showing a combobox.

See: `create/8`, `wxValidator` (not implemented in wx)

```
destroy(This :: wxComboBox()) -> ok
```

Destructor, destroying the combobox.

```
create(This, Parent, Id, Value, Pos, Size, Choices) -> boolean()
```

Types:

```
   This = wxComboBox()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Value = unicode:chardata()
   Pos = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Choices = [unicode:chardata()]

create(This, Parent, Id, Value, Pos, Size, Choices,
       Options :: [Option]) ->
          boolean()
```

Types:

```
   This = wxComboBox()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Value = unicode:chardata()
   Pos = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Choices = [unicode:chardata()]
   Option = {style, integer()} | {validator, wx:wx_object()}

canCopy(This) -> boolean()
```

Types:

```
   This = wxComboBox()
```

Returns true if the selection can be copied to the clipboard.

```
canCut(This) -> boolean()
```
Types:
```
   This = wxComboBox()
```
Returns true if the selection can be cut to the clipboard.

```
canPaste(This) -> boolean()
```
Types:
```
   This = wxComboBox()
```
Returns true if the contents of the clipboard can be pasted into the text control.

On some platforms (Motif, GTK) this is an approximation and returns true if the control is editable, false otherwise.

```
canRedo(This) -> boolean()
```
Types:
```
   This = wxComboBox()
```
Returns true if there is a redo facility available and the last operation can be redone.

```
canUndo(This) -> boolean()
```
Types:
```
   This = wxComboBox()
```
Returns true if there is an undo facility available and the last operation can be undone.

```
copy(This) -> ok
```
Types:
```
   This = wxComboBox()
```
Copies the selected text to the clipboard.

```
cut(This) -> ok
```
Types:
```
   This = wxComboBox()
```
Copies the selected text to the clipboard and removes it from the control.

```
getInsertionPoint(This) -> integer()
```
Types:
```
   This = wxComboBox()
```
Same as `wxTextCtrl:getInsertionPoint/1`.

Note: Under wxMSW, this function always returns 0 if the combobox doesn't have the focus.

```
getLastPosition(This) -> integer()
```
Types:
```
   This = wxComboBox()
```
Returns the zero based index of the last position in the text control, which is equal to the number of characters in the control.

```
getValue(This) -> unicode:charlist()
```

Types:

```
This = wxComboBox()
```

Gets the contents of the control.

Notice that for a multiline text control, the lines will be separated by (Unix-style) `\n` characters, even under Windows where they are separated by a `\r\n` sequence in the native control.

```
paste(This) -> ok
```

Types:

```
This = wxComboBox()
```

Pastes text from the clipboard to the text item.

```
redo(This) -> ok
```

Types:

```
This = wxComboBox()
```

If there is a redo facility and the last operation can be redone, redoes the last operation.

Does nothing if there is no redo facility.

```
replace(This, From, To, Value) -> ok
```

Types:

```
This = wxComboBox()
From = To = integer()
Value = unicode:chardata()
```

Replaces the text starting at the first position up to (but not including) the character at the last position with the given text.

This function puts the current insertion point position at `to` as a side effect.

```
remove(This, From, To) -> ok
```

Types:

```
This = wxComboBox()
From = To = integer()
```

Removes the text starting at the first given position up to (but not including) the character at the last position.

This function puts the current insertion point position at `to` as a side effect.

```
setInsertionPoint(This, Pos) -> ok
```

Types:

```
This = wxComboBox()
Pos = integer()
```

Sets the insertion point at the given position.

```
setInsertionPointEnd(This) -> ok
```

Types:

```
This = wxComboBox()
```

Sets the insertion point at the end of the text control.

This is equivalent to calling `setInsertionPoint/2` with `getLastPosition/1` argument.

### setSelection(This, N) -> ok

Types:

```
This = wxComboBox()
N = integer()
```

Sets the selection to the given item `n` or removes the selection entirely if `n` == `wxNOT_FOUND`.

Note that this does not cause any command events to be emitted nor does it deselect any other items in the controls which support multiple selections.

See: `wxControlWithItems:setString/3`, `wxControlWithItems:setStringSelection/2`

### setSelection(This, From, To) -> ok

Types:

```
This = wxComboBox()
From = To = integer()
```

Same as `wxTextCtrl:setSelection/3`.

### setValue(This, Text) -> ok

Types:

```
This = wxComboBox()
Text = unicode:chardata()
```

Sets the text for the combobox text field.

For normal, editable comboboxes with a text entry field calling this method will generate a `wxEVT_TEXT` event, consistently with `wxTextCtrl:setValue/2` behaviour, use `wxTextCtrl:changeValue/2` if this is undesirable.

For controls with `wxCB_READONLY` style the method behaves somewhat differently: the string must be in the combobox choices list (the check for this is case-insensitive) and `wxEVT_TEXT` is `not` generated in this case.

### undo(This) -> ok

Types:

```
This = wxComboBox()
```

If there is an undo facility and the last operation can be undone, undoes the last operation.

Does nothing if there is no undo facility.

# wxCommandEvent

Erlang module

This event class contains information about command events, which originate from a variety of simple controls.

Note that wxCommandEvents and wxCommandEvent-derived event classes by default and unlike other wxEvent-derived classes propagate upward from the source window (the window which emits the event) up to the first parent which processes the event. Be sure to read overview_events_propagation.

More complex controls, such as `wxTreeCtrl`, have separate command event classes.

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxCommandEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxCommandEventType` to subscribe to events of this type.

## Data Types

```
wxCommandEvent() = wx:wx_object()
wxCommand() =
    #wxCommand{type = wxCommandEvent:wxCommandEventType(),
               cmdString = unicode:chardata(),
               commandInt = integer(),
               extraLong = integer()}
wxCommandEventType() =
    command_button_clicked | command_checkbox_clicked |
    command_choice_selected | command_listbox_selected |
    command_listbox_doubleclicked | command_text_updated |
    command_text_enter | text_maxlen | command_menu_selected |
    command_slider_updated | command_radiobox_selected |
    command_radiobutton_selected | command_scrollbar_updated |
    command_vlbox_selected | command_combobox_selected |
    combobox_dropdown | combobox_closeup | command_tool_rclicked |
    command_tool_enter | tool_dropdown |
    command_checklistbox_toggled | command_togglebutton_clicked |
    command_left_click | command_left_dclick |
    command_right_click | command_set_focus | command_kill_focus |
    command_enter | notification_message_click |
    notification_message_dismissed | notification_message_action
```

## Exports

```
getClientData(This) -> term()
```

Types:

```
This = wxCommandEvent()
```

Returns client object pointer for a listbox or choice selection event (not valid for a deselection).

```
getExtraLong(This) -> integer()
```

Types:

```
This = wxCommandEvent()
```

Returns extra information dependent on the event objects type.

If the event comes from a listbox selection, it is a boolean determining whether the event was a selection (true) or a deselection (false). A listbox deselection only occurs for multiple-selection boxes, and in this case the index and string values are indeterminate and the listbox must be examined by the application.

```
getInt(This) -> integer()
```

Types:

```
This = wxCommandEvent()
```

Returns the integer identifier corresponding to a listbox, choice or radiobox selection (only if the event was a selection, not a deselection), or a boolean value representing the value of a checkbox.

For a menu item, this method returns -1 if the item is not checkable or a boolean value (true or false) for checkable items indicating the new state of the item.

```
getSelection(This) -> integer()
```

Types:

```
This = wxCommandEvent()
```

Returns item index for a listbox or choice selection event (not valid for a deselection).

```
getString(This) -> unicode:charlist()
```

Types:

```
This = wxCommandEvent()
```

Returns item string for a listbox or choice selection event.

If one or several items have been deselected, returns the index of the first deselected item. If some items have been selected and others deselected at the same time, it will return the index of the first selected item.

```
isChecked(This) -> boolean()
```

Types:

```
This = wxCommandEvent()
```

This method can be used with checkbox and menu events: for the checkboxes, the method returns true for a selection event and false for a deselection one.

For the menu events, this method indicates if the menu item just has become checked or unchecked (and thus only makes sense for checkable menu items).

Notice that this method cannot be used with `wxCheckListBox` currently.

```
isSelection(This) -> boolean()
```

Types:

```
This = wxCommandEvent()
```

For a listbox or similar event, returns true if it is a selection, false if it is a deselection.

If some items have been selected and others deselected at the same time, it will return true.

```
setInt(This, IntCommand) -> ok
```

Types:

```
This = wxCommandEvent()
IntCommand = integer()
```

Sets the `m_commandInt` member.

```
setString(This, String) -> ok
```

Types:

```
This = wxCommandEvent()
String = unicode:chardata()
```

Sets the `m_commandString` member.

# wxContextMenuEvent

Erlang module

This class is used for context menu events, sent to give the application a chance to show a context (popup) menu for a `wxWindow`.

Note that if `getPosition/1` returns wxDefaultPosition, this means that the event originated from a keyboard context button event, and you should compute a suitable position yourself, for example by calling `wx_misc:getMousePosition/0`.

Notice that the exact sequence of mouse events is different across the platforms. For example, under MSW the context menu event is generated after `EVT_RIGHT_UP` event and only if it was not handled but under GTK the context menu event is generated after `EVT_RIGHT_DOWN` event. This is correct in the sense that it ensures that the context menu is shown according to the current platform UI conventions and also means that you must not handle (or call `wxEvent:skip/2` in your handler if you do have one) neither right mouse down nor right mouse up event if you plan on handling `EVT_CONTEXT_MENU` event.

See: `wxCommandEvent`, **Overview events**

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxContextMenuEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxContextMenuEventType` to subscribe to events of this type.

## Data Types

```
wxContextMenuEvent() = wx:wx_object()
wxContextMenu() =
    #wxContextMenu{type =
                       wxContextMenuEvent:wxContextMenuEventType(),
                   pos = {X :: integer(), Y :: integer()}}
wxContextMenuEventType() = context_menu
```

## Exports

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
   This = wxContextMenuEvent()
```

Returns the position in screen coordinates at which the menu should be shown.

Use `wxWindow:screenToClient/2` to convert to client coordinates.

You can also omit a position from `wxWindow:popupMenu/4` in order to use the current mouse pointer position.

If the event originated from a keyboard event, the value returned from this function will be wxDefaultPosition.

```
setPosition(This, Point) -> ok
```

Types:

```
This = wxContextMenuEvent()
Point = {X :: integer(), Y :: integer()}
```

Sets the position at which the menu should be shown.

# wxControl

Erlang module

This is the base class for a control or "widget".

A control is generally a small window which processes user input and/or displays one or more item of data.

See: `wxValidator` (not implemented in wx)

This class is derived (and can use functions) from: `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxControl**

## Events

Event types emitted from this class: `command_text_copy`, `command_text_cut`, `command_text_paste`

## Data Types

```
wxControl() = wx:wx_object()
```

## Exports

```
getLabel(This) -> unicode:charlist()
```

Types:

```
This = wxControl()
```

Returns the control's label, as it was passed to `setLabel/2`.

Note that the returned string may contains mnemonics ("&" characters) if they were passed to the `setLabel/2` function; use `GetLabelText()` (not implemented in wx) if they are undesired.

Also note that the returned string is always the string which was passed to `setLabel/2` but may be different from the string passed to `SetLabelText()` (not implemented in wx) (since this last one escapes mnemonic characters).

```
setLabel(This, Label) -> ok
```

Types:

```
This = wxControl()
Label = unicode:chardata()
```

Sets the control's label.

All "&" characters in the `label` are special and indicate that the following character is a `mnemonic` for this control and can be used to activate it from the keyboard (typically by using `Alt` key in combination with it). To insert a literal ampersand character, you need to double it, i.e. use "&&". If this behaviour is undesirable, use `SetLabelText()` (not implemented in wx) instead.

# wxControlWithItems

Erlang module

This is convenience class that derives from both `wxControl` and `wxItemContainer` (not implemented in wx). It is used as basis for some wxWidgets controls (`wxChoice` and `wxListBox`).

See: `wxItemContainer` (not implemented in wx), `wxItemContainerImmutable` (not implemented in wx)

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxControlWithItems**

## Data Types

```
wxControlWithItems() = wx:wx_object()
```

## Exports

```
append(This, Item) -> integer()
```

Types:

```
This = wxControlWithItems()
Item = unicode:chardata()
```

Appends item into the control.

Return: The return value is the index of the newly inserted item. Note that this may be different from the last one if the control is sorted (e.g. has `wxLB_SORT` or `wxCB_SORT` style).

```
append(This, Item, ClientData) -> integer()
```

Types:

```
This = wxControlWithItems()
Item = unicode:chardata()
ClientData = term()
```

Appends item into the control.

Return: The return value is the index of the newly inserted item. Note that this may be different from the last one if the control is sorted (e.g. has `wxLB_SORT` or `wxCB_SORT` style).

```
appendStrings(This, Items) -> integer()
```

Types:

```
This = wxControlWithItems()
Items = [unicode:chardata()]
```

Appends several items at once into the control.

Notice that calling this method is usually much faster than appending them one by one if you need to add a lot of items.

```
appendStrings(This, Items, ClientsData) -> integer()
```

Types:

```
This = wxControlWithItems()
Items = [unicode:chardata()]
ClientsData = [term()]
```

Appends several items at once into the control.

Notice that calling this method is usually much faster than appending them one by one if you need to add a lot of items.

```
clear(This) -> ok
```

Types:

```
This = wxControlWithItems()
```

Removes all items from the control.

`clear/1` also deletes the client data of the existing items if it is owned by the control.

```
delete(This, N) -> ok
```

Types:

```
This = wxControlWithItems()
N = integer()
```

Deletes an item from the control.

The client data associated with the item will be also deleted if it is owned by the control. Note that it is an error (signalled by an assert failure in debug builds) to remove an item with the index negative or greater or equal than the number of items in the control.

If there is a currently selected item below the item being deleted, i.e. if `getSelection/1` returns a valid index greater than or equal to `n`, the selection is invalidated when this function is called. However if the selected item appears before the item being deleted, the selection is preserved unchanged.

See: `clear/1`

```
findString(This, String) -> integer()
```

Types:

```
This = wxControlWithItems()
String = unicode:chardata()
```

```
findString(This, String, Options :: [Option]) -> integer()
```

Types:

```
This = wxControlWithItems()
String = unicode:chardata()
Option = {bCase, boolean()}
```

Finds an item whose label matches the given string.

Return: The zero-based position of the item, or wxNOT_FOUND if the string was not found.

```
getClientData(This, N) -> term()
```

Types:

```
This = wxControlWithItems()
N = integer()
```

Returns a pointer to the client data associated with the given item (if any).

It is an error to call this function for a control which doesn't have typed client data at all although it is OK to call it even if the given item doesn't have any client data associated with it (but other items do).

Notice that the returned pointer is still owned by the control and will be deleted by it, use `DetachClientObject()` (not implemented in wx) if you want to remove the pointer from the control.

Return: A pointer to the client data, or NULL if not present.

```
setClientData(This, N, Data) -> ok
```

Types:

```
This = wxControlWithItems()
N = integer()
Data = term()
```

Associates the given typed client data pointer with the given item: the `data` object will be deleted when the item is deleted (either explicitly by using `delete/2` or implicitly when the control itself is destroyed).

Note that it is an error to call this function if any untyped client data pointers had been associated with the control items before.

```
getCount(This) -> integer()
```

Types:

```
This = wxControlWithItems()
```

Returns the number of items in the control.

See: `isEmpty/1`

```
getSelection(This) -> integer()
```

Types:

```
This = wxControlWithItems()
```

Returns the index of the selected item or `wxNOT_FOUND` if no item is selected.

Return: The position of the current selection.

Remark: This method can be used with single selection list boxes only, you should use `wxListBox:getSelections/1` for the list boxes with wxLB_MULTIPLE style.

See: `setSelection/2`, `getStringSelection/1`

```
getString(This, N) -> unicode:charlist()
```

Types:

```
This = wxControlWithItems()
N = integer()
```

Returns the label of the item with the given index.

Return: The label of the item or an empty string if the position was invalid.

```
getStringSelection(This) -> unicode:charlist()
```

Types:

```
This = wxControlWithItems()
```

Returns the label of the selected item or an empty string if no item is selected.

See: `getSelection/1`

```
insert(This, Item, Pos) -> integer()
```

Types:

```
This = wxControlWithItems()
Item = unicode:chardata()
Pos = integer()
```

Inserts item into the control.

Return: The return value is the index of the newly inserted item. If the insertion failed for some reason, -1 is returned.

```
insert(This, Item, Pos, ClientData) -> integer()
```

Types:

```
This = wxControlWithItems()
Item = unicode:chardata()
Pos = integer()
ClientData = term()
```

Inserts item into the control.

Return: The return value is the index of the newly inserted item. If the insertion failed for some reason, -1 is returned.

```
insertStrings(This, Items, Pos) -> integer()
```

Types:

```
This = wxControlWithItems()
Items = [unicode:chardata()]
Pos = integer()
```

Inserts several items at once into the control.

Notice that calling this method is usually much faster than inserting them one by one if you need to insert a lot of items.

Return: The return value is the index of the last inserted item. If the insertion failed for some reason, -1 is returned.

```
insertStrings(This, Items, Pos, ClientsData) -> integer()
```

Types:

```
This = wxControlWithItems()
Items = [unicode:chardata()]
Pos = integer()
ClientsData = [term()]
```

Inserts several items at once into the control.

Notice that calling this method is usually much faster than inserting them one by one if you need to insert a lot of items.

Return: The return value is the index of the last inserted item. If the insertion failed for some reason, -1 is returned.

```
isEmpty(This) -> boolean()
```

Types:

```
This = wxControlWithItems()
```

Returns true if the control is empty or false if it has some items.

See: `getCount/1`

```
select(This, N) -> ok
```

Types:

```
This = wxControlWithItems()
N = integer()
```

This is the same as `setSelection/2` and exists only because it is slightly more natural for controls which support multiple selection.

```
setSelection(This, N) -> ok
```

Types:

```
This = wxControlWithItems()
N = integer()
```

Sets the selection to the given item `n` or removes the selection entirely if `n` == `wxNOT_FOUND`.

Note that this does not cause any command events to be emitted nor does it deselect any other items in the controls which support multiple selections.

See: `setString/3`, `setStringSelection/2`

```
setString(This, N, String) -> ok
```

Types:

```
This = wxControlWithItems()
N = integer()
String = unicode:chardata()
```

Sets the label for the given item.

```
setStringSelection(This, String) -> boolean()
```

Types:

```
This = wxControlWithItems()
String = unicode:chardata()
```

Selects the item with the specified string in the control.

This method doesn't cause any command events to be emitted.

Notice that this method is case-insensitive, i.e. the string is compared with all the elements of the control case-insensitively and the first matching entry is selected, even if it doesn't have exactly the same case as this string and there is an exact match afterwards.

Return: true if the specified string has been selected, false if it wasn't found in the control.

# wxCursor

Erlang module

A cursor is a small bitmap usually used for denoting where the mouse pointer is, with a picture that might indicate the interpretation of a mouse click. As with icons, cursors in X and MS Windows are created in a different manner. Therefore, separate cursors will be created for the different environments. Platform-specific methods for creating a `wxCursor` object are catered for, and this is an occasion where conditional compilation will probably be required (see `wxIcon` for an example).

A single cursor object may be used in many windows (any subwindow type). The wxWidgets convention is to set the cursor for a window, as in X, rather than to set it globally as in MS Windows, although a global `wx_misc:setCursor/1` function is also available for MS Windows use.

Creating a Custom Cursor

The following is an example of creating a cursor from 32x32 bitmap data (down_bits) and a mask (down_mask) where 1 is black and 0 is white for the bits, and 1 is opaque and 0 is transparent for the mask. It works on Windows and GTK+.

Predefined objects (include wx.hrl):

See: `wxBitmap`, `wxIcon`, `wxWindow:setCursor/2`, `wx_misc:setCursor/1`, ?wxStockCursor

This class is derived (and can use functions) from: `wxBitmap`

wxWidgets docs: **wxCursor**

## Data Types

```
wxCursor() = wx:wx_object()
```

## Exports

```
new() -> wxCursor()
```

Default constructor.

```
new(CursorName) -> wxCursor()
new(Image) -> wxCursor()
new(CursorId) -> wxCursor()
```

Types:

```
CursorId = wx:wx_enum()
```

Constructs a cursor using a cursor identifier.

```
new(CursorName, Options :: [Option]) -> wxCursor()
```

Types:

```
CursorName = unicode:chardata()
Option =
    {type, wx:wx_enum()} |
    {hotSpotX, integer()} |
    {hotSpotY, integer()}
```

Constructs a cursor by passing a string resource name or filename.

The arguments `hotSpotX` and `hotSpotY` are only used when there's no hotspot info in the resource/image-file to load (e.g. when using `wxBITMAP_TYPE_ICO` under wxMSW or `wxBITMAP_TYPE_XPM` under wxGTK).

```
destroy(This :: wxCursor()) -> ok
```

Destroys the cursor.

See reference-counted object destruction for more info.

A cursor can be reused for more than one window, and does not get destroyed when the window is destroyed. wxWidgets destroys all cursors on application exit, although it is best to clean them up explicitly.

```
ok(This) -> boolean()
```

Types:

```
This = wxCursor()
```

See: `isOk/1`.

```
isOk(This) -> boolean()
```

Types:

```
This = wxCursor()
```

Returns true if cursor data is present.

# wxDC

Erlang module

A `wxDC` is a `"device context"` onto which graphics and text can be drawn. It is intended to represent different output devices and offers a common abstract API for drawing on any of them.

wxWidgets offers an alternative drawing API based on the modern drawing backends GDI+, CoreGraphics, Cairo and Direct2D. See `wxGraphicsContext`, `wxGraphicsRenderer` and related classes. There is also a `wxGCDC` linking the APIs by offering the `wxDC` API on top of a `wxGraphicsContext`.

`wxDC` is an abstract base class and cannot be created directly. Use `wxPaintDC`, `wxClientDC`, `wxWindowDC`, `wxScreenDC`, `wxMemoryDC` or `wxPrinterDC` (not implemented in wx). Notice that device contexts which are associated with windows (i.e. `wxClientDC`, `wxWindowDC` and `wxPaintDC`) use the window font and colours by default (starting with wxWidgets 2.9.0) but the other device context classes use system-default values so you always must set the appropriate fonts and colours before using them.

In addition to the versions of the methods documented below, there are also versions which accept single {X,Y} parameter instead of the two wxCoord ones or {X,Y} and {Width,Height} instead of the four wxCoord parameters.

Beginning with wxWidgets 2.9.0 the entire `wxDC` code has been reorganized. All platform dependent code (actually all drawing code) has been moved into backend classes which derive from a common wxDCImpl class. The user-visible classes such as `wxClientDC` and `wxPaintDC` merely forward all calls to the backend implementation.

Device and logical units

In the `wxDC` context there is a distinction between `logical` units and `device` units.

`Device` units are the units native to the particular device; e.g. for a screen, a device unit is a `pixel`. For a printer, the device unit is defined by the resolution of the printer (usually given in `DPI:` dot-per-inch).

All `wxDC` functions use instead `logical` units, unless where explicitly stated. Logical units are arbitrary units mapped to device units using the current mapping mode (see `setMapMode/2`).

This mechanism allows reusing the same code which prints on e.g. a window on the screen to print on e.g. a paper.

Support for Transparency / Alpha Channel

In general `wxDC` methods don't support alpha transparency and the alpha component of `wx_color()` is simply ignored and you need to use `wxGraphicsContext` for full transparency support. There are, however, a few exceptions: first, under macOS and GTK+ 3 colours with alpha channel are supported in all the normal wxDC-derived classes as they use `wxGraphicsContext` internally. Second, under all platforms `wxSVGFileDC` (not implemented in wx) also fully supports alpha channel. In both of these cases the instances of `wxPen` or `wxBrush` that are built from `wx_color()` use the colour's alpha values when stroking or filling.

Support for Transformation Matrix

On some platforms (currently under MSW, GTK+ 3, macOS) `wxDC` has support for applying an arbitrary affine transformation matrix to its coordinate system (since 3.1.1 this feature is also supported by `wxGCDC` in all ports). Call `CanUseTransformMatrix()` (not implemented in wx) to check if this support is available and then call `SetTransformMatrix()` (not implemented in wx) if it is. If the transformation matrix is not supported, `SetTransformMatrix()` (not implemented in wx) always simply returns `false` and doesn't do anything.

This feature is only available when `wxUSE_DC_TRANSFORM_MATRIX` build option is enabled.

See: **Overview dc**, `wxGraphicsContext`, `wxDCFontChanger` (not implemented in wx), `wxDCTextColourChanger` (not implemented in wx), `wxDCPenChanger` (not implemented in wx), `wxDCBrushChanger` (not implemented in wx), `wxDCClipper` (not implemented in wx)

wxWidgets docs: **wxDC**

## Data Types

```
wxDC() = wx:wx_object()
```

## Exports

```
blit(This, Dest, Size, Source, Src) -> boolean()
```

Types:

```
   This = wxDC()
   Dest = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Source = wxDC()
   Src = {X :: integer(), Y :: integer()}
```

```
blit(This, Dest, Size, Source, Src, Options :: [Option]) ->
        boolean()
```

Types:

```
   This = wxDC()
   Dest = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Source = wxDC()
   Src = {X :: integer(), Y :: integer()}
   Option =
       {rop, wx:wx_enum()} |
       {useMask, boolean()} |
       {srcPtMask, {X :: integer(), Y :: integer()}}
```

Copy from a source DC to this DC.

With this method you can specify the destination coordinates and the size of area to copy which will be the same for both the source and target DCs. If you need to apply scaling while copying, use `StretchBlit()` (not implemented in wx).

Notice that source DC coordinates `xsrc` and `ysrc` are interpreted using the current source DC coordinate system, i.e. the scale, origin position and axis directions are taken into account when transforming them to physical (pixel) coordinates.

Remark: There is partial support for `blit/6` in `wxPostScriptDC`, under X.

See: `StretchBlit()` (not implemented in wx), `wxMemoryDC`, `wxBitmap`, `wxMask`

```
calcBoundingBox(This, X, Y) -> ok
```

Types:

```
   This = wxDC()
   X = Y = integer()
```

Adds the specified point to the bounding box which can be retrieved with `minX/1`, `maxX/1` and `minY/1`, `maxY/1` functions.

See: `resetBoundingBox/1`

```
clear(This) -> ok
```

Types:

```
This = wxDC()
```

Clears the device context using the current background brush.

Note that `setBackground/2` method must be used to set the brush used by `clear/1`, the brush used for filling the shapes set by `setBrush/2` is ignored by it.

If no background brush was set, solid white brush is used to clear the device context.

```
crossHair(This, Pt) -> ok
```

Types:

```
This = wxDC()
Pt = {X :: integer(), Y :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
destroyClippingRegion(This) -> ok
```

Types:

```
This = wxDC()
```

Destroys the current clipping region so that none of the DC is clipped.

See: `setClippingRegion/3`

```
deviceToLogicalX(This, X) -> integer()
```

Types:

```
This = wxDC()
X = integer()
```

Convert `device` X coordinate to logical coordinate, using the current mapping mode, user scale factor, device origin and axis orientation.

```
deviceToLogicalXRel(This, X) -> integer()
```

Types:

```
This = wxDC()
X = integer()
```

Convert `device` X coordinate to relative logical coordinate, using the current mapping mode and user scale factor but ignoring the axis orientation.

Use this for converting a width, for example.

```
deviceToLogicalY(This, Y) -> integer()
```

Types:

```
This = wxDC()
Y = integer()
```

Converts `device` Y coordinate to logical coordinate, using the current mapping mode, user scale factor, device origin and axis orientation.

```
deviceToLogicalYRel(This, Y) -> integer()
```

Types:

```
This = wxDC()
Y = integer()
```

Convert `device` Y coordinate to relative logical coordinate, using the current mapping mode and user scale factor but ignoring the axis orientation.

Use this for converting a height, for example.

```
drawArc(This, PtStart, PtEnd, Centre) -> ok
```

Types:

```
This = wxDC()
PtStart = PtEnd = Centre = {X :: integer(), Y :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawBitmap(This, Bmp, Pt) -> ok
```

Types:

```
This = wxDC()
Bmp = wxBitmap:wxBitmap()
Pt = {X :: integer(), Y :: integer()}
```

```
drawBitmap(This, Bmp, Pt, Options :: [Option]) -> ok
```

Types:

```
This = wxDC()
Bmp = wxBitmap:wxBitmap()
Pt = {X :: integer(), Y :: integer()}
Option = {useMask, boolean()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawCheckMark(This, Rect) -> ok
```

Types:

```
This = wxDC()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawCircle(This, Pt, Radius) -> ok
```

Types:

```
This = wxDC()
Pt = {X :: integer(), Y :: integer()}
Radius = integer()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawEllipse(This, Rect) -> ok
```

Types:

```
This = wxDC()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawEllipse(This, Pt, Size) -> ok
```

Types:

```
This = wxDC()
Pt = {X :: integer(), Y :: integer()}
Size = {W :: integer(), H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawEllipticArc(This, Pt, Sz, Sa, Ea) -> ok
```

Types:

```
This = wxDC()
Pt = {X :: integer(), Y :: integer()}
Sz = {W :: integer(), H :: integer()}
Sa = Ea = number()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawIcon(This, Icon, Pt) -> ok
```

Types:

```
This = wxDC()
Icon = wxIcon:wxIcon()
Pt = {X :: integer(), Y :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawLabel(This, Text, Rect) -> ok
```

Types:

```
   This = wxDC()
   Text = unicode:chardata()
   Rect =
       {X :: integer(),
        Y :: integer(),
        W :: integer(),
        H :: integer()}

drawLabel(This, Text, Rect, Options :: [Option]) -> ok
```

Types:

```
   This = wxDC()
   Text = unicode:chardata()
   Rect =
       {X :: integer(),
        Y :: integer(),
        W :: integer(),
        H :: integer()}
   Option = {alignment, integer()} | {indexAccel, integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawLine(This, Pt1, Pt2) -> ok
```

Types:

```
   This = wxDC()
   Pt1 = Pt2 = {X :: integer(), Y :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawLines(This, Points) -> ok
```

Types:

```
   This = wxDC()
   Points = [{X :: integer(), Y :: integer()}]

drawLines(This, Points, Options :: [Option]) -> ok
```

Types:

```
   This = wxDC()
   Points = [{X :: integer(), Y :: integer()}]
   Option = {xoffset, integer()} | {yoffset, integer()}
```

Draws lines using an array of points of size n adding the optional offset coordinate.

The current pen is used for drawing the lines.

```
drawPolygon(This, Points) -> ok
```

Types:

```
   This = wxDC()
   Points = [{X :: integer(), Y :: integer()}]

drawPolygon(This, Points, Options :: [Option]) -> ok
```

Types:

```
   This = wxDC()
   Points = [{X :: integer(), Y :: integer()}]
   Option =
       {xoffset, integer()} |
       {yoffset, integer()} |
       {fillStyle, wx:wx_enum()}
```

Draws a filled polygon using an array of points of size n, adding the optional offset coordinate.

The first and last points are automatically closed.

The last argument specifies the fill rule: wxODDEVEN_RULE (the default) or wxWINDING_RULE.

The current pen is used for drawing the outline, and the current brush for filling the shape. Using a transparent brush suppresses filling.

```
drawPoint(This, Pt) -> ok
```

Types:

```
   This = wxDC()
   Pt = {X :: integer(), Y :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawRectangle(This, Rect) -> ok
```

Types:

```
   This = wxDC()
   Rect =
       {X :: integer(),
        Y :: integer(),
        W :: integer(),
        H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawRectangle(This, Pt, Sz) -> ok
```

Types:

```
   This = wxDC()
   Pt = {X :: integer(), Y :: integer()}
   Sz = {W :: integer(), H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawRotatedText(This, Text, Point, Angle) -> ok
```

Types:

```
This = wxDC()
Text = unicode:chardata()
Point = {X :: integer(), Y :: integer()}
Angle = number()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawRoundedRectangle(This, Rect, Radius) -> ok
```

Types:

```
This = wxDC()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
Radius = number()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawRoundedRectangle(This, Pt, Sz, Radius) -> ok
```

Types:

```
This = wxDC()
Pt = {X :: integer(), Y :: integer()}
Sz = {W :: integer(), H :: integer()}
Radius = number()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
drawText(This, Text, Pt) -> ok
```

Types:

```
This = wxDC()
Text = unicode:chardata()
Pt = {X :: integer(), Y :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
endDoc(This) -> ok
```

Types:

```
This = wxDC()
```

Ends a document (only relevant when outputting to a printer).

```
endPage(This) -> ok
```

Types:

```
This = wxDC()
```

Ends a document page (only relevant when outputting to a printer).

```
floodFill(This, Pt, Col) -> boolean()
```

Types:

```
This = wxDC()
Pt = {X :: integer(), Y :: integer()}
Col = wx:wx_colour()
```

```
floodFill(This, Pt, Col, Options :: [Option]) -> boolean()
```

Types:

```
This = wxDC()
Pt = {X :: integer(), Y :: integer()}
Col = wx:wx_colour()
Option = {style, wx:wx_enum()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
getBackground(This) -> wxBrush:wxBrush()
```

Types:

```
This = wxDC()
```

Gets the brush used for painting the background.

See: `setBackground/2`

```
getBackgroundMode(This) -> integer()
```

Types:

```
This = wxDC()
```

Returns the current background mode: `wxPENSTYLE_SOLID` or `wxPENSTYLE_TRANSPARENT`.

See: `setBackgroundMode/2`

```
getBrush(This) -> wxBrush:wxBrush()
```

Types:

```
This = wxDC()
```

Gets the current brush.

See: `setBrush/2`

```
getCharHeight(This) -> integer()
```

Types:

```
This = wxDC()
```

Gets the character height of the currently set font.

```
getCharWidth(This) -> integer()
```

Types:

```
This = wxDC()
```

Gets the average character width of the currently set font.

```
getClippingBox(This) -> Result
```

Types:

```
Result =
    {X :: integer(),
     Y :: integer(),
     Width :: integer(),
     Height :: integer()}
This = wxDC()
```

Gets the rectangle surrounding the current clipping region. If no clipping region is set this function returns the extent of the device context. @remarks Clipping region is given in logical coordinates. @param x If non-NULL, filled in with the logical horizontal coordinate of the top left corner of the clipping region if the function returns true or 0 otherwise. @param y If non-NULL, filled in with the logical vertical coordinate of the top left corner of the clipping region if the function returns true or 0 otherwise. @param width If non-NULL, filled in with the width of the clipping region if the function returns true or the device context width otherwise. @param height If non-NULL, filled in with the height of the clipping region if the function returns true or the device context height otherwise.

Return: true if there is a clipping region or false if there is no active clipping region (note that this return value is available only since wxWidgets 3.1.2, this function didn't return anything in the previous versions).

```
getFont(This) -> wxFont:wxFont()
```

Types:

```
This = wxDC()
```

Gets the current font.

Notice that even although each device context object has some default font after creation, this method would return a ?wxNullFont initially and only after calling `setFont/2` a valid font is returned.

```
getLayoutDirection(This) -> wx:wx_enum()
```

Types:

```
This = wxDC()
```

Gets the current layout direction of the device context.

On platforms where RTL layout is supported, the return value will either be `wxLayout_LeftToRight` or `wxLayout_RightToLeft`. If RTL layout is not supported, the return value will be `wxLayout_Default`.

See: `setLayoutDirection/2`

```
getLogicalFunction(This) -> wx:wx_enum()
```

Types:

```
This = wxDC()
```

Gets the current logical function.

See: `setLogicalFunction/2`

```
getMapMode(This) -> wx:wx_enum()
```

Types:

```
This = wxDC()
```

Gets the current mapping mode for the device context.

See: `setMapMode/2`

```
getMultiLineTextExtent(This, String) ->
                          {W :: integer(), H :: integer()}
```

Types:

```
This = wxDC()
String = unicode:chardata()
```

Gets the dimensions of the string using the currently selected font.

`string` is the text string to measure.

Return: The text extent as a {Width,Height} object.

Note: This function works with both single-line and multi-line strings.

See: `wxFont`, `setFont/2`, `getPartialTextExtents/2`, `getTextExtent/3`

```
getMultiLineTextExtent(This, String, Options :: [Option]) ->
                          {W :: integer(),
                           H :: integer(),
                           HeightLine :: integer()}
```

Types:

```
This = wxDC()
String = unicode:chardata()
Option = {font, wxFont:wxFont()}
```

Gets the dimensions of the string using the currently selected font.

`string` is the text string to measure, `heightLine`, if non NULL, is where to store the height of a single line.

The text extent is set in the given `w` and `h` pointers.

If the optional parameter `font` is specified and valid, then it is used for the text extent calculation, otherwise the currently selected font is used.

If `string` is empty, its horizontal extent is 0 but, for convenience when using this function for allocating enough space for a possibly multi-line string, its vertical extent is the same as the height of an empty line of text. Please note that this behaviour differs from that of `getTextExtent/3`.

Note: This function works with both single-line and multi-line strings.

See: `wxFont`, `setFont/2`, `getPartialTextExtents/2`, `getTextExtent/3`

```
getPartialTextExtents(This, Text) -> Result
```

Types:

```
Result = {Res :: boolean(), Widths :: [integer()]}
This = wxDC()
Text = unicode:chardata()
```

Fills the `widths` array with the widths from the beginning of `text` to the corresponding character of `text`.

The generic version simply builds a running total of the widths of each character using `getTextExtent/3`, however if the various platforms have a native API function that is faster or more accurate than the generic implementation then it should be used instead.

See: `getMultiLineTextExtent/3`, `getTextExtent/3`

```
getPen(This) -> wxPen:wxPen()
```

Types:

```
This = wxDC()
```

Gets the current pen.

See: `setPen/2`

```
getPixel(This, Pos) -> Result
```

Types:

```
Result = {Res :: boolean(), Colour :: wx:wx_colour4()}
This = wxDC()
Pos = {X :: integer(), Y :: integer()}
```

Gets in `colour` the colour at the specified location.

This method isn't available for `wxPostScriptDC` or `wxMetafileDC` (not implemented in wx) nor for any DC in wxOSX port and simply returns false there.

Note: Setting a pixel can be done using `drawPoint/2`.

Note: This method shouldn't be used with `wxPaintDC` as accessing the DC while drawing can result in unexpected results, notably in wxGTK.

```
getPPI(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxDC()
```

Returns the resolution of the device in pixels per inch.

```
getSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxDC()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
getSizeMM(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxDC()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
getTextBackground(This) -> wx:wx_colour4()
```

Types:

```
This = wxDC()
```

Gets the current text background colour.

See: `setTextBackground/2`

```
getTextExtent(This, String) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxDC()
String = unicode:chardata()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
getTextExtent(This, String, Options :: [Option]) -> Result
```

Types:

```
Result =
    {W :: integer(),
     H :: integer(),
     Descent :: integer(),
     ExternalLeading :: integer()}
This = wxDC()
String = unicode:chardata()
Option = {theFont, wxFont:wxFont()}
```

Gets the dimensions of the string using the currently selected font.

`string` is the text string to measure, `descent` is the dimension from the baseline of the font to the bottom of the descender, and `externalLeading` is any extra vertical space added to the font by the font designer (usually is zero).

The text extent is returned in `w` and `h` pointers or as a {Width,Height} object depending on which version of this function is used.

If the optional parameter `font` is specified and valid, then it is used for the text extent calculation. Otherwise the currently selected font is.

If `string` is empty, its extent is 0 in both directions, as expected.

Note: This function only works with single-line strings.

See: `wxFont`, `setFont/2`, `getPartialTextExtents/2`, `getMultiLineTextExtent/3`

```
getTextForeground(This) -> wx:wx_colour4()
```

Types:

```
This = wxDC()
```

Gets the current text foreground colour.

See: `setTextForeground/2`

```
getUserScale(This) -> {X :: number(), Y :: number()}
```

Types:

```
This = wxDC()
```

Gets the current user scale factor.

See: `setUserScale/3`

```
gradientFillConcentric(This, Rect, InitialColour, DestColour) ->
                          ok
```

Types:

```
This = wxDC()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
InitialColour = DestColour = wx:wx_colour()
```

Fill the area specified by rect with a radial gradient, starting from `initialColour` at the centre of the circle and fading to `destColour` on the circle outside.

The circle is placed at the centre of `rect`.

Note: Currently this function is very slow, don't use it for real-time drawing.

```
gradientFillConcentric(This, Rect, InitialColour, DestColour,
                       CircleCenter) ->
                          ok
```

Types:

```
This = wxDC()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
InitialColour = DestColour = wx:wx_colour()
CircleCenter = {X :: integer(), Y :: integer()}
```

Fill the area specified by rect with a radial gradient, starting from `initialColour` at the centre of the circle and fading to `destColour` on the circle outside.

`circleCenter` are the relative coordinates of centre of the circle in the specified `rect`.

Note: Currently this function is very slow, don't use it for real-time drawing.

```
gradientFillLinear(This, Rect, InitialColour, DestColour) -> ok
```

Types:

```
This = wxDC()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
```

```
        H :: integer()}
   InitialColour = DestColour = wx:wx_colour()

gradientFillLinear(This, Rect, InitialColour, DestColour,
                   Options :: [Option]) ->
                      ok
```

Types:

```
   This = wxDC()
   Rect =
       {X :: integer(),
        Y :: integer(),
        W :: integer(),
        H :: integer()}
   InitialColour = DestColour = wx:wx_colour()
   Option = {nDirection, wx:wx_enum()}
```

Fill the area specified by `rect` with a linear gradient, starting from `initialColour` and eventually fading to `destColour`.

The `nDirection` specifies the direction of the colour change, default is to use `initialColour` on the left part of the rectangle and `destColour` on the right one.

```
logicalToDeviceX(This, X) -> integer()
```

Types:

```
   This = wxDC()
   X = integer()
```

Converts logical X coordinate to device coordinate, using the current mapping mode, user scale factor, device origin and axis orientation.

```
logicalToDeviceXRel(This, X) -> integer()
```

Types:

```
   This = wxDC()
   X = integer()
```

Converts logical X coordinate to relative device coordinate, using the current mapping mode and user scale factor but ignoring the axis orientation.

Use this for converting a width, for example.

```
logicalToDeviceY(This, Y) -> integer()
```

Types:

```
   This = wxDC()
   Y = integer()
```

Converts logical Y coordinate to device coordinate, using the current mapping mode, user scale factor, device origin and axis orientation.

```
logicalToDeviceYRel(This, Y) -> integer()
```

Types:

```
This = wxDC()
Y = integer()
```

Converts logical Y coordinate to relative device coordinate, using the current mapping mode and user scale factor but ignoring the axis orientation.

Use this for converting a height, for example.

```
maxX(This) -> integer()
```

Types:

```
This = wxDC()
```

Gets the maximum horizontal extent used in drawing commands so far.

```
maxY(This) -> integer()
```

Types:

```
This = wxDC()
```

Gets the maximum vertical extent used in drawing commands so far.

```
minX(This) -> integer()
```

Types:

```
This = wxDC()
```

Gets the minimum horizontal extent used in drawing commands so far.

```
minY(This) -> integer()
```

Types:

```
This = wxDC()
```

Gets the minimum vertical extent used in drawing commands so far.

```
isOk(This) -> boolean()
```

Types:

```
This = wxDC()
```

Returns true if the DC is ok to use.

```
resetBoundingBox(This) -> ok
```

Types:

```
This = wxDC()
```

Resets the bounding box: after a call to this function, the bounding box doesn't contain anything.

See: `calcBoundingBox/3`

```
setAxisOrientation(This, XLeftRight, YBottomUp) -> ok
```

Types:

```
This = wxDC()
XLeftRight = YBottomUp = boolean()
```

Sets the x and y axis orientation (i.e. the direction from lowest to highest values on the axis).

The default orientation is x axis from left to right and y axis from top down.

```
setBackground(This, Brush) -> ok
```

Types:

```
This = wxDC()
Brush = wxBrush:wxBrush()
```

Sets the current background brush for the DC.

```
setBackgroundMode(This, Mode) -> ok
```

Types:

```
This = wxDC()
Mode = integer()
```

`mode` may be one of `wxPENSTYLE_SOLID` and `wxPENSTYLE_TRANSPARENT`.

This setting determines whether text will be drawn with a background colour or not.

```
setBrush(This, Brush) -> ok
```

Types:

```
This = wxDC()
Brush = wxBrush:wxBrush()
```

Sets the current brush for the DC.

If the argument is ?wxNullBrush (or another invalid brush; see `wxBrush:isOk/1`), the current brush is selected out of the device context (leaving `wxDC` without any valid brush), allowing the current brush to be destroyed safely.

See: `wxBrush`, `wxMemoryDC`, (for the interpretation of colours when drawing into a monochrome bitmap)

```
setClippingRegion(This, Rect) -> ok
```

Types:

```
This = wxDC()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setClippingRegion(This, Pt, Sz) -> ok
```

Types:

```
This = wxDC()
Pt = {X :: integer(), Y :: integer()}
Sz = {W :: integer(), H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setDeviceOrigin(This, X, Y) -> ok
```

Types:

```
This = wxDC()
X = Y = integer()
```

Sets the device origin (i.e. the origin in pixels after scaling has been applied).

This function may be useful in Windows printing operations for placing a graphic on a page.

```
setFont(This, Font) -> ok
```

Types:

```
This = wxDC()
Font = wxFont:wxFont()
```

Sets the current font for the DC.

If the argument is ?wxNullFont (or another invalid font; see `wxFont:isOk/1`), the current font is selected out of the device context (leaving `wxDC` without any valid font), allowing the current font to be destroyed safely.

See: `wxFont`

```
setLayoutDirection(This, Dir) -> ok
```

Types:

```
This = wxDC()
Dir = wx:wx_enum()
```

Sets the current layout direction for the device context.

See: `getLayoutDirection/1`

```
setLogicalFunction(This, Function) -> ok
```

Types:

```
This = wxDC()
Function = wx:wx_enum()
```

Sets the current logical function for the device context.

Note: This function is not fully supported in all ports, due to the limitations of the underlying drawing model. Notably, `wxINVERT` which was commonly used for drawing rubber bands or other moving outlines in the past, is not, and will not, be supported by wxGTK3 and wxMac. The suggested alternative is to draw temporarily objects normally and refresh the (affected part of the) window to remove them later.

It determines how a `source` pixel (from a pen or brush colour, or source device context if using `blit/6`) combines with a `destination` pixel in the current device context. Text drawing is not affected by this function.

See ?wxRasterOperationMode enumeration values for more info.

The default is `wxCOPY`, which simply draws with the current colour. The others combine the current colour and the background using a logical operation.

```
setMapMode(This, Mode) -> ok
```

Types:

```
This = wxDC()
Mode = wx:wx_enum()
```

The mapping mode of the device context defines the unit of measurement used to convert `logical` units to `device` units.

Note that in X, text drawing isn't handled consistently with the mapping mode; a font is always specified in point size. However, setting the user scale (see `setUserScale/3`) scales the text appropriately. In Windows, scalable TrueType fonts are always used; in X, results depend on availability of fonts, but usually a reasonable match is found.

The coordinate origin is always at the top left of the screen/printer.

Drawing to a Windows printer device context uses the current mapping mode, but mapping mode is currently ignored for PostScript output.

### setPalette(This, Palette) -> ok

Types:

```
This = wxDC()
Palette = wxPalette:wxPalette()
```

If this is a window DC or memory DC, assigns the given palette to the window or bitmap associated with the DC.

If the argument is ?wxNullPalette, the current palette is selected out of the device context, and the original palette restored.

See: `wxPalette`

### setPen(This, Pen) -> ok

Types:

```
This = wxDC()
Pen = wxPen:wxPen()
```

Sets the current pen for the DC.

If the argument is ?wxNullPen (or another invalid pen; see `wxPen:isOk/1`), the current pen is selected out of the device context (leaving `wxDC` without any valid pen), allowing the current pen to be destroyed safely.

See: `wxMemoryDC`, for the interpretation of colours when drawing into a monochrome bitmap

### setTextBackground(This, Colour) -> ok

Types:

```
This = wxDC()
Colour = wx:wx_colour()
```

Sets the current text background colour for the DC.

### setTextForeground(This, Colour) -> ok

Types:

```
This = wxDC()
Colour = wx:wx_colour()
```

Sets the current text foreground colour for the DC.

See: `wxMemoryDC`, for the interpretation of colours when drawing into a monochrome bitmap

```
setUserScale(This, XScale, YScale) -> ok
```

Types:

```
This = wxDC()
XScale = YScale = number()
```

Sets the user scaling factor, useful for applications which require 'zooming'.

```
startDoc(This, Message) -> boolean()
```

Types:

```
This = wxDC()
Message = unicode:chardata()
```

Starts a document (only relevant when outputting to a printer).

`message` is a message to show while printing.

```
startPage(This) -> ok
```

Types:

```
This = wxDC()
```

Starts a document page (only relevant when outputting to a printer).

# wxDCOverlay

Erlang module

Connects an overlay with a drawing DC.

See: `wxOverlay`, `wxDC`

wxWidgets docs: **wxDCOverlay**

## Data Types

```
wxDCOverlay() = wx:wx_object()
```

## Exports

```
new(Overlay, Dc) -> wxDCOverlay()
```

Types:

```
Overlay = wxOverlay:wxOverlay()
Dc = wxDC:wxDC()
```

Convenience wrapper that behaves the same using the entire area of the dc.

```
new(Overlay, Dc, X, Y, Width, Height) -> wxDCOverlay()
```

Types:

```
Overlay = wxOverlay:wxOverlay()
Dc = wxDC:wxDC()
X = Y = Width = Height = integer()
```

Connects this overlay to the corresponding drawing dc, if the overlay is not initialized yet this call will do so.

```
destroy(This :: wxDCOverlay()) -> ok
```

Removes the connection between the overlay and the dc.

```
clear(This) -> ok
```

Types:

```
This = wxDCOverlay()
```

Clears the layer, restoring the state at the last init.

# wxDataObject

Erlang module

A `wxDataObject` represents data that can be copied to or from the clipboard, or dragged and dropped. The important thing about `wxDataObject` is that this is a 'smart' piece of data unlike 'dumb' data containers such as memory buffers or files. Being 'smart' here means that the data object itself should know what data formats it supports and how to render itself in each of its supported formats.

A supported format, incidentally, is exactly the format in which the data can be requested from a data object or from which the data object may be set. In the general case, an object may support different formats on 'input' and 'output', i.e. it may be able to render itself in a given format but not be created from data on this format or vice versa. `wxDataObject` defines the `wxDataObject::Direction` (not implemented in wx) enumeration type which distinguishes between them.

See `wxDataFormat` (not implemented in wx) documentation for more about formats.

Not surprisingly, being 'smart' comes at a price of added complexity. This is reasonable for the situations when you really need to support multiple formats, but may be annoying if you only want to do something simple like cut and paste text.

To provide a solution for both cases, wxWidgets has two predefined classes which derive from `wxDataObject`: `wxDataObjectSimple` (not implemented in wx) and `wxDataObjectComposite` (not implemented in wx). `wxDataObjectSimple` (not implemented in wx) is the simplest `wxDataObject` possible and only holds data in a single format (such as HTML or text) and `wxDataObjectComposite` (not implemented in wx) is the simplest way to implement a `wxDataObject` that does support multiple formats because it achieves this by simply holding several `wxDataObjectSimple` (not implemented in wx) objects.

So, you have several solutions when you need a `wxDataObject` class (and you need one as soon as you want to transfer data via the clipboard or drag and drop):

Please note that the easiest way to use drag and drop and the clipboard with multiple formats is by using `wxDataObjectComposite` (not implemented in wx), but it is not the most efficient one as each `wxDataObjectSimple` (not implemented in wx) would contain the whole data in its respective formats. Now imagine that you want to paste 200 pages of text in your proprietary format, as well as Word, RTF, HTML, Unicode and plain text to the clipboard and even today's computers are in trouble. For this case, you will have to derive from `wxDataObject` directly and make it enumerate its formats and provide the data in the requested format on demand.

Note that neither the GTK+ data transfer mechanisms for clipboard and drag and drop, nor OLE data transfer, `copies` any data until another application actually requests the data. This is in contrast to the 'feel' offered to the user of a program who would normally think that the data resides in the clipboard after having pressed 'Copy' - in reality it is only declared to be `available`.

You may also derive your own data object classes from `wxCustomDataObject` (not implemented in wx) for user-defined types. The format of user-defined data is given as a mime-type string literal, such as "application/word" or "image/png". These strings are used as they are under Unix (so far only GTK+) to identify a format and are translated into their Windows equivalent under Win32 (using the OLE IDataObject for data exchange to and from the clipboard and for drag and drop). Note that the format string translation under Windows is not yet finished.

Each class derived directly from `wxDataObject` must override and implement all of its functions which are pure virtual in the base class. The data objects which only render their data or only set it (i.e. work in only one direction), should return 0 from `GetFormatCount()` (not implemented in wx).

See: **Overview dnd**, **Examples**, `wxFileDataObject`, `wxTextDataObject`, `wxBitmapDataObject`, `wxCustomDataObject` (not implemented in wx), `wxDropTarget` (not implemented in wx), `wxDropSource` (not implemented in wx), `wxTextDropTarget` (not implemented in wx), `wxFileDropTarget` (not implemented in wx)

wxWidgets docs: **wxDataObject**

## Data Types

```
wxDataObject() = wx:wx_object()
```

# wxDateEvent

Erlang module

This event class holds information about a date change and is used together with `wxDatePickerCtrl`. It also serves as a base class for `wxCalendarEvent`.

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxDateEvent**

## Data Types

```
wxDateEvent() = wx:wx_object()
wxDate() =
    #wxDate{type = wxDateEvent:wxDateEventType(),
            date = wx:wx_datetime()}
wxDateEventType() = date_changed
```

## Exports

```
getDate(This) -> wx:wx_datetime()
```

Types:

```
   This = wxDateEvent()
```

Returns the date.

# wxDatePickerCtrl

Erlang module

This control allows the user to select a date. Unlike `wxCalendarCtrl`, which is a relatively big control, `wxDatePickerCtrl` is implemented as a small window showing the currently selected date. The control can be edited using the keyboard, and can also display a popup window for more user-friendly date selection, depending on the styles used and the platform.

It is only available if `wxUSE_DATEPICKCTRL` is set to 1.

Styles

This class supports the following styles:

See: `wxTimePickerCtrl` (not implemented in wx), `wxCalendarCtrl`, `wxDateEvent`

This class is derived (and can use functions) from: `wxPickerBase` `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxDatePickerCtrl**

## Events

Event types emitted from this class: `date_changed`

## Data Types

```
wxDatePickerCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxDatePickerCtrl()
```

Default constructor.

```
new(Parent, Id) -> wxDatePickerCtrl()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxDatePickerCtrl()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Option =
    {date, wx:wx_datetime()} |
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {validator, wx:wx_object()}
```

Initializes the object and calls `Create()` (not implemented in wx) with all the parameters.

```
getRange(This, Dt1, Dt2) -> boolean()
```

Types:

```
This = wxDatePickerCtrl()
Dt1 = Dt2 = wx:wx_datetime()
```

If the control had been previously limited to a range of dates using `setRange/3`, returns the lower and upper bounds of this range.

If no range is set (or only one of the bounds is set), `dt1` and/or `dt2` are set to be invalid.

Notice that when using a native MSW implementation of this control the lower range is always set, even if `setRange/3` hadn't been called explicitly, as the native control only supports dates later than year 1601.

Return: false if no range limits are currently set, true if at least one bound is set.

```
getValue(This) -> wx:wx_datetime()
```

Types:

```
This = wxDatePickerCtrl()
```

Returns the currently entered date.

For a control with `wxDP_ALLOWNONE` style the returned value may be invalid if no date is entered, otherwise it is always valid.

```
setRange(This, Dt1, Dt2) -> ok
```

Types:

```
This = wxDatePickerCtrl()
Dt1 = Dt2 = wx:wx_datetime()
```

Sets the valid range for the date selection.

If `dt1` is valid, it becomes the earliest date (inclusive) accepted by the control. If `dt2` is valid, it becomes the latest possible date.

Notice that if the current value is not inside the new range, it will be adjusted to lie inside it, i.e. calling this method can change the control value, however no events are generated by it.

Remark: If the current value of the control is outside of the newly set range bounds, the behaviour is undefined.

```
setValue(This, Dt) -> ok
```

Types:

```
This = wxDatePickerCtrl()
Dt = wx:wx_datetime()
```

Changes the current value of the control.

The date should be valid unless the control was created with `wxDP_ALLOWNONE` style and included in the currently selected range, if any.

Calling this method does not result in a date change event.

```
destroy(This :: wxDatePickerCtrl()) -> ok
```

Destroys the object.

# wxDialog

Erlang module

A dialog box is a window with a title bar and sometimes a system menu, which can be moved around the screen. It can contain controls and other windows and is often used to allow the user to make some choice or to answer a question.

Dialogs can be made scrollable, automatically, for computers with low resolution screens: please see overview_dialog_autoscrolling for further details.

Dialogs usually contain either a single button allowing to close the dialog or two buttons, one accepting the changes and the other one discarding them (such button, if present, is automatically activated if the user presses the "Esc" key). By default, buttons with the standard wxID_OK and wxID_CANCEL identifiers behave as expected. Starting with wxWidgets 2.7 it is also possible to use a button with a different identifier instead, see `setAffirmativeId/2` and `SetEscapeId()` (not implemented in wx).

Also notice that the `createButtonSizer/2` should be used to create the buttons appropriate for the current platform and positioned correctly (including their order which is platform-dependent).

Modal and Modeless

There are two kinds of dialog, modal and modeless. A modal dialog blocks program flow and user input on other windows until it is dismissed, whereas a modeless dialog behaves more like a frame in that program flow continues, and input in other windows is still possible. To show a modal dialog you should use the `showModal/1` method while to show a dialog modelessly you simply use `show/2`, just as with frames.

Note that the modal dialog is one of the very few examples of wxWindow-derived objects which may be created on the stack and not on the heap. In other words, while most windows would be created like this:

You can achieve the same result with dialogs by using simpler code:

An application can define a `wxCloseEvent` handler for the dialog to respond to system close events.

Styles

This class supports the following styles:

See: **Overview dialog**, `wxFrame`, **Overview validator**

This class is derived (and can use functions) from: `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxDialog**

## Events

Event types emitted from this class: `close_window`, `init_dialog`

## Data Types

```
wxDialog() = wx:wx_object()
```

## Exports

```
new() -> wxDialog()
```

Default constructor.

```
new(Parent, Id, Title) -> wxDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()

new(Parent, Id, Title, Options :: [Option]) -> wxDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructor.

See: `create/5`

```
destroy(This :: wxDialog()) -> ok
```

Destructor.

Deletes any child windows before deleting the physical window.

See overview_windowdeletion for more info.

```
create(This, Parent, Id, Title) -> boolean()
```

Types:

```
   This = wxDialog()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()

create(This, Parent, Id, Title, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxDialog()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Used for two-step dialog box construction.

See: `new/4`

```
createButtonSizer(This, Flags) -> wxSizer:wxSizer()
```

Types:

```
This = wxDialog()
Flags = integer()
```

Creates a sizer with standard buttons.

`flags` is a bit list of the following flags: wxOK, wxCANCEL, wxYES, wxNO, wxAPPLY, wxCLOSE, wxHELP, wxNO_DEFAULT.

The sizer lays out the buttons in a manner appropriate to the platform.

This function uses `createStdDialogButtonSizer/2` internally for most platforms but doesn't create the sizer at all for the platforms with hardware buttons (such as smartphones) for which it sets up the hardware buttons appropriately and returns NULL, so don't forget to test that the return value is valid before using it.

```
createStdDialogButtonSizer(This, Flags) ->
                              wxStdDialogButtonSizer:wxStdDialogButtonSizer()
```

Types:

```
This = wxDialog()
Flags = integer()
```

Creates a `wxStdDialogButtonSizer` with standard buttons.

`flags` is a bit list of the following flags: wxOK, wxCANCEL, wxYES, wxNO, wxAPPLY, wxCLOSE, wxHELP, wxNO_DEFAULT.

The sizer lays out the buttons in a manner appropriate to the platform.

```
endModal(This, RetCode) -> ok
```

Types:

```
This = wxDialog()
RetCode = integer()
```

Ends a modal dialog, passing a value to be returned from the `showModal/1` invocation.

See: `showModal/1`, `getReturnCode/1`, `setReturnCode/2`

```
getAffirmativeId(This) -> integer()
```

Types:

```
This = wxDialog()
```

Gets the identifier of the button which works like standard OK button in this dialog.

See: `setAffirmativeId/2`

```
getReturnCode(This) -> integer()
```

Types:

```
This = wxDialog()
```

Gets the return code for this window.

Remark: A return code is normally associated with a modal dialog, where `showModal/1` returns a code to the application.

See: `setReturnCode/2`, `showModal/1`, `endModal/2`

```
isModal(This) -> boolean()
```

Types:

```
This = wxDialog()
```

Returns true if the dialog box is modal, false otherwise.

```
setAffirmativeId(This, Id) -> ok
```

Types:

```
This = wxDialog()
Id = integer()
```

Sets the identifier to be used as OK button.

When the button with this identifier is pressed, the dialog calls `wxWindow:validate/1` and `wxWindow:transferDataFromWindow/1` and, if they both return true, closes the dialog with the affirmative id return code.

Also, when the user presses a hardware OK button on the devices having one or the special OK button in the PocketPC title bar, an event with this id is generated.

By default, the affirmative id is wxID_OK.

See: `getAffirmativeId/1`, `SetEscapeId()` (not implemented in wx)

```
setReturnCode(This, RetCode) -> ok
```

Types:

```
This = wxDialog()
RetCode = integer()
```

Sets the return code for this window.

A return code is normally associated with a modal dialog, where `showModal/1` returns a code to the application. The function `endModal/2` calls `setReturnCode/2`.

See: `getReturnCode/1`, `showModal/1`, `endModal/2`

```
show(This) -> boolean()
```

Types:

```
This = wxDialog()
```

```
show(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxDialog()
Option = {show, boolean()}
```

Hides or shows the dialog.

The preferred way of dismissing a modal dialog is to use `endModal/2`.

```
showModal(This) -> integer()
```

Types:

```
This = wxDialog()
```

Shows an application-modal dialog.

Program flow does not return until the dialog has been dismissed with `endModal/2`.

Notice that it is possible to call `showModal/1` for a dialog which had been previously shown with `show/2`, this allows making an existing modeless dialog modal. However `showModal/1` can't be called twice without intervening `endModal/2` calls.

Note that this function creates a temporary event loop which takes precedence over the application's main event loop (see `wxEventLoopBase` (not implemented in wx)) and which is destroyed when the dialog is dismissed. This also results in a call to `wxApp::ProcessPendingEvents()` (not implemented in wx).

Return: The value set with `setReturnCode/2`.

See: `ShowWindowModal()` (not implemented in wx), `ShowWindowModalThenDo()` (not implemented in wx), `endModal/2`, `getReturnCode/1`, `setReturnCode/2`

# wxDirDialog

Erlang module

This class represents the directory chooser dialog.

Styles

This class supports the following styles:

Note: This flag cannot be used with the `wxDD_MULTIPLE` style.

Remark: MacOS 10.11+ does not display a title bar on the dialog. Use `setMessage/2` to change the string displayed to the user at the top of the dialog after creation. The `wxTopLevelWindow:setTitle/2` method is provided for compatibility with pre-10.11 MacOS versions that do still support displaying the title bar.

See: **Overview cmndlg**, `wxFileDialog`

This class is derived (and can use functions) from: `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxDirDialog**

## Data Types

```
wxDirDialog() = wx:wx_object()
```

## Exports

```
new(Parent) -> wxDirDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxDirDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
Option =
    {title, unicode:chardata()} |
    {defaultPath, unicode:chardata()} |
    {style, integer()} |
    {pos, {X :: integer(), Y :: integer()}} |
    {sz, {W :: integer(), H :: integer()}}
```

Constructor.

Use `wxDialog:showModal/1` to show the dialog.

```
destroy(This :: wxDirDialog()) -> ok
```

Destructor.

```
getPath(This) -> unicode:charlist()
```

Types:

```
This = wxDirDialog()
```

Returns the default or user-selected path.

Note: This function can't be used with dialogs which have the `wxDD_MULTIPLE` style, use `GetPaths()` (not implemented in wx) instead.

```
getMessage(This) -> unicode:charlist()
```

Types:

```
   This = wxDirDialog()
```

Returns the message that will be displayed on the dialog.

```
setMessage(This, Message) -> ok
```

Types:

```
   This = wxDirDialog()
   Message = unicode:chardata()
```

Sets the message that will be displayed on the dialog.

```
setPath(This, Path) -> ok
```

Types:

```
   This = wxDirDialog()
   Path = unicode:chardata()
```

Sets the default path.

# wxDirPickerCtrl

Erlang module

This control allows the user to select a directory. The generic implementation is a button which brings up a `wxDirDialog` when clicked. Native implementation may differ but this is usually a (small) widget which give access to the dir-chooser dialog. It is only available if `wxUSE_DIRPICKERCTRL` is set to 1 (the default).

Styles

This class supports the following styles:

See: `wxDirDialog`, `wxFileDirPickerEvent`

This class is derived (and can use functions) from: `wxPickerBase` `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxDirPickerCtrl**

## Events

Event types emitted from this class: `command_dirpicker_changed`

## Data Types

```
wxDirPickerCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxDirPickerCtrl()

new(Parent, Id) -> wxDirPickerCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()

new(Parent, Id, Options :: [Option]) -> wxDirPickerCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {path, unicode:chardata()} |
       {message, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Initializes the object and calls `create/4` with all the parameters.

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = wxDirPickerCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()

create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxDirPickerCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {path, unicode:chardata()} |
       {message, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Creates the widgets with the given parameters.

Return: true if the control was successfully created or false if creation failed.

```
getPath(This) -> unicode:charlist()
```

Types:

```
   This = wxDirPickerCtrl()
```

Returns the absolute path of the currently selected directory.

```
setPath(This, Dirname) -> ok
```

Types:

```
   This = wxDirPickerCtrl()
   Dirname = unicode:chardata()
```

Sets the absolute path of the currently selected directory.

If the control uses `wxDIRP_DIR_MUST_EXIST` and does not use `wxDIRP_USE_TEXTCTRL` style, the `dirname` must be a name of an existing directory and will be simply ignored by the native wxGTK implementation if this is not the case.

```
destroy(This :: wxDirPickerCtrl()) -> ok
```

Destroys the object.

# wxDisplay

Erlang module

Determines the sizes and locations of displays connected to the system.

wxWidgets docs: **wxDisplay**

## Data Types

```
wxDisplay() = wx:wx_object()
```

## Exports

```
new() -> wxDisplay()
```

Default constructor creating `wxDisplay` object representing the primary display.

```
new(Index) -> wxDisplay()
new(Window) -> wxDisplay()
```

Types:

```
Window = wxWindow:wxWindow()
```

Constructor creating the display object associated with the given window.

This is the most convenient way of finding the display on which the given window is shown while falling back to the default display if it is not shown at all or positioned outside of any display.

See: `getFromWindow/1`

Since: 3.1.2

```
destroy(This :: wxDisplay()) -> ok
```

Destructor.

```
isOk(This) -> boolean()
```

Types:

```
This = wxDisplay()
```

Returns true if the object was initialized successfully.

```
getClientArea(This) ->
                 {X :: integer(),
                  Y :: integer(),
                  W :: integer(),
                  H :: integer()}
```

Types:

```
This = wxDisplay()
```

Returns the client area of the display.

The client area is the part of the display available for the normal (non full screen) windows, usually it is the same as `getGeometry/1` but it could be less if there is a taskbar (or equivalent) on this display.

```
getGeometry(This) ->
               {X :: integer(),
                Y :: integer(),
                W :: integer(),
                H :: integer()}
```

Types:

```
This = wxDisplay()
```

Returns the bounding rectangle of the display whose index was passed to the constructor.

See: `getClientArea/1`, `wx_misc:displaySize/0`

```
getName(This) -> unicode:charlist()
```

Types:

```
This = wxDisplay()
```

Returns the display's name.

The returned value is currently an empty string under all platforms except MSW.

```
isPrimary(This) -> boolean()
```

Types:

```
This = wxDisplay()
```

Returns true if the display is the primary display.

The primary display is the one whose index is 0.

```
getCount() -> integer()
```

Returns the number of connected displays.

```
getFromPoint(Pt) -> integer()
```

Types:

```
Pt = {X :: integer(), Y :: integer()}
```

Returns the index of the display on which the given point lies, or `wxNOT_FOUND` if the point is not on any connected display.

```
getFromWindow(Win) -> integer()
```

Types:

```
Win = wxWindow:wxWindow()
```

Returns the index of the display on which the given window lies.

If the window is on more than one display it gets the display that overlaps the window the most.

Returns `wxNOT_FOUND` if the window is not on any connected display.

```
getPPI(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxDisplay()
```

Returns display resolution in pixels per inch.

Horizontal and vertical resolution are returned in `x` and `y` components of the {Width,Height} object respectively.

If the resolution information is not available, returns.

Since: 3.1.2

# wxDisplayChangedEvent

Erlang module

A display changed event is sent to top-level windows when the display resolution has changed.

This event is currently emitted under Windows only.

Only for:wxmsw

See: `wxDisplay`

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxDisplayChangedEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxDisplayChangedEventType` to subscribe to events of this type.

## Data Types

```
wxDisplayChangedEvent() = wx:wx_object()
wxDisplayChanged() =
    #wxDisplayChanged{type =
                          wxDisplayChangedEvent:wxDisplayChangedEventType()}
wxDisplayChangedEventType() = display_changed
```

# wxDropFilesEvent

Erlang module

This class is used for drop files events, that is, when files have been dropped onto the window.

The window must have previously been enabled for dropping by calling `wxWindow:dragAcceptFiles/2`.

Important note: this is a separate implementation to the more general drag and drop implementation documented in the overview_dnd. It uses the older, Windows message-based approach of dropping files.

Remark: Windows only until version 2.8.9, available on all platforms since 2.8.10.

See: **Overview events**, `wxWindow:dragAcceptFiles/2`

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxDropFilesEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxDropFilesEventType` to subscribe to events of this type.

## Data Types

```
wxDropFilesEvent() = wx:wx_object()
wxDropFiles() =
    #wxDropFiles{type = wxDropFilesEvent:wxDropFilesEventType(),
                 pos = {X :: integer(), Y :: integer()},
                 files = [unicode:chardata()]}
wxDropFilesEventType() = drop_files
```

## Exports

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxDropFilesEvent()
```

Returns the position at which the files were dropped.

Returns an array of filenames.

```
getNumberOfFiles(This) -> integer()
```

Types:

```
This = wxDropFilesEvent()
```

Returns the number of files dropped.

```
getFiles(This) -> [unicode:charlist()]
```

Types:

```
This = wxDropFilesEvent()
```

Returns an array of filenames.

# wxEraseEvent

Erlang module

An erase event is sent when a window's background needs to be repainted.

On some platforms, such as GTK+, this event is simulated (simply generated just before the paint event) and may cause flicker. It is therefore recommended that you set the text background colour explicitly in order to prevent flicker. The default background colour under GTK+ is grey.

To intercept this event, use the EVT_ERASE_BACKGROUND macro in an event table definition.

You must use the device context returned by `getDC/1` to draw on, don't create a `wxPaintDC` in the event handler.

See: **Overview events**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxEraseEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxEraseEventType` to subscribe to events of this type.

## Data Types

```
wxEraseEvent() = wx:wx_object()
wxErase() =
    #wxErase{type = wxEraseEvent:wxEraseEventType(),
             dc = wxDC:wxDC()}
wxEraseEventType() = erase_background
```

## Exports

```
getDC(This) -> wxDC:wxDC()
```

Types:

```
   This = wxEraseEvent()
```

Returns the device context associated with the erase event to draw on.

The returned pointer is never NULL.

# wxEvent

Erlang module

An event is a structure holding information about an event passed to a callback or member function.

`wxEvent` used to be a multipurpose event object, and is an abstract base class for other event classes (see below).

For more information about events, see the overview_events overview.

See: `wxCommandEvent`, `wxMouseEvent`

wxWidgets docs: **wxEvent**

## Data Types

```
wxEvent() = wx:wx_object()
```

## Exports

```
getId(This) -> integer()
```

Types:

```
This = wxEvent()
```

Returns the identifier associated with this event, such as a button command id.

```
getSkipped(This) -> boolean()
```

Types:

```
This = wxEvent()
```

Returns true if the event handler should be skipped, false otherwise.

```
getTimestamp(This) -> integer()
```

Types:

```
This = wxEvent()
```

Gets the timestamp for the event.

The timestamp is the time in milliseconds since some fixed moment (not necessarily the standard Unix Epoch, so only differences between the timestamps and not their absolute values usually make sense).

Warning: wxWidgets returns a non-NULL timestamp only for mouse and key events (see `wxMouseEvent` and `wxKeyEvent`).

```
isCommandEvent(This) -> boolean()
```

Types:

```
This = wxEvent()
```

Returns true if the event is or is derived from `wxCommandEvent` else it returns false.

Note: exists only for optimization purposes.

```
resumePropagation(This, PropagationLevel) -> ok
```

Types:

```
This = wxEvent()
PropagationLevel = integer()
```

Sets the propagation level to the given value (for example returned from an earlier call to `stopPropagation/1`).

```
shouldPropagate(This) -> boolean()
```

Types:

```
This = wxEvent()
```

Test if this event should be propagated or not, i.e. if the propagation level is currently greater than 0.

```
skip(This) -> ok
```

Types:

```
This = wxEvent()
```

```
skip(This, Options :: [Option]) -> ok
```

Types:

```
This = wxEvent()
Option = {skip, boolean()}
```

This method can be used inside an event handler to control whether further event handlers bound to this event will be called after the current one returns.

Without `skip/2` (or equivalently if Skip(false) is used), the event will not be processed any more. If Skip(true) is called, the event processing system continues searching for a further handler function for this event, even though it has been processed already in the current handler.

In general, it is recommended to skip all non-command events to allow the default handling to take place. The command events are, however, normally not skipped as usually a single command such as a button click or menu item selection must only be processed by one handler.

```
stopPropagation(This) -> integer()
```

Types:

```
This = wxEvent()
```

Stop the event from propagating to its parent window.

Returns the old propagation level value which may be later passed to `resumePropagation/2` to allow propagating the event again.

# wxEvtHandler

Erlang module

A class that can handle events from the windowing system. `wxWindow` is (and therefore all window classes are) derived from this class.

To get events from wxwidgets objects you subscribe to them by calling `connect/3`.

If the `callback` option is not supplied events are sent as messages.

These messages will be `#wx{}` where `EventRecord` is a record that depends on the `wxEventType`. The records are defined in: `wx/include/wx.hrl`.

If a callback was supplied to connect, the callback will be invoked (in another process) to handle the event. The callback should be of arity 2.

```
fun Callback (EventRecord::wx(), EventObject::wxObject()).
```

Note: The callback will be in executed in new process each time.

See: **Overview events**

wxWidgets docs: **wxEvtHandler**

## Data Types

```
wxEvtHandler() = wx:wx_object()
wxEventType() =
    wxActivateEvent:wxActivateEventType() |
    wxAuiManagerEvent:wxAuiManagerEventType() |
    wxAuiNotebookEvent:wxAuiNotebookEventType() |
    wxBookCtrlEvent:wxBookCtrlEventType() |
    wxCalendarEvent:wxCalendarEventType() |
    wxChildFocusEvent:wxChildFocusEventType() |
    wxClipboardTextEvent:wxClipboardTextEventType() |
    wxCloseEvent:wxCloseEventType() |
    wxColourPickerEvent:wxColourPickerEventType() |
    wxCommandEvent:wxCommandEventType() |
    wxContextMenuEvent:wxContextMenuEventType() |
    wxDateEvent:wxDateEventType() |
    wxDisplayChangedEvent:wxDisplayChangedEventType() |
    wxDropFilesEvent:wxDropFilesEventType() |
    wxEraseEvent:wxEraseEventType() |
    wxFileDirPickerEvent:wxFileDirPickerEventType() |
    wxFocusEvent:wxFocusEventType() |
    wxFontPickerEvent:wxFontPickerEventType() |
    wxGridEvent:wxGridEventType() |
    wxHelpEvent:wxHelpEventType() |
    wxHtmlLinkEvent:wxHtmlLinkEventType() |
    wxIconizeEvent:wxIconizeEventType() |
    wxIdleEvent:wxIdleEventType() |
    wxInitDialogEvent:wxInitDialogEventType() |
    wxJoystickEvent:wxJoystickEventType() |
    wxKeyEvent:wxKeyEventType() |
    wxListEvent:wxListEventType() |
```

```
    wxMaximizeEvent:wxMaximizeEventType() |
    wxMenuEvent:wxMenuEventType() |
    wxMouseCaptureChangedEvent:wxMouseCaptureChangedEventType() |
    wxMouseCaptureLostEvent:wxMouseCaptureLostEventType() |
    wxMouseEvent:wxMouseEventType() |
    wxMoveEvent:wxMoveEventType() |
    wxNavigationKeyEvent:wxNavigationKeyEventType() |
    wxPaintEvent:wxPaintEventType() |
    wxPaletteChangedEvent:wxPaletteChangedEventType() |
    wxQueryNewPaletteEvent:wxQueryNewPaletteEventType() |
    wxSashEvent:wxSashEventType() |
    wxScrollEvent:wxScrollEventType() |
    wxScrollWinEvent:wxScrollWinEventType() |
    wxSetCursorEvent:wxSetCursorEventType() |
    wxShowEvent:wxShowEventType() |
    wxSizeEvent:wxSizeEventType() |
    wxSpinEvent:wxSpinEventType() |
    wxSplitterEvent:wxSplitterEventType() |
    wxStyledTextEvent:wxStyledTextEventType() |
    wxSysColourChangedEvent:wxSysColourChangedEventType() |
    wxTaskBarIconEvent:wxTaskBarIconEventType() |
    wxTreeEvent:wxTreeEventType() |
    wxUpdateUIEvent:wxUpdateUIEventType() |
    wxWebViewEvent:wxWebViewEventType() |
    wxWindowCreateEvent:wxWindowCreateEventType() |
    wxWindowDestroyEvent:wxWindowDestroyEventType()
wx() =
    #wx{id = integer(),
        obj = wx:wx_object(),
        userData = term(),
        event = event()}
event() =
    wxActivateEvent:wxActivate() |
    wxAuiManagerEvent:wxAuiManager() |
    wxAuiNotebookEvent:wxAuiNotebook() |
    wxBookCtrlEvent:wxBookCtrl() |
    wxCalendarEvent:wxCalendar() |
    wxChildFocusEvent:wxChildFocus() |
    wxClipboardTextEvent:wxClipboardText() |
    wxCloseEvent:wxClose() |
    wxColourPickerEvent:wxColourPicker() |
    wxCommandEvent:wxCommand() |
    wxContextMenuEvent:wxContextMenu() |
    wxDateEvent:wxDate() |
    wxDisplayChangedEvent:wxDisplayChanged() |
    wxDropFilesEvent:wxDropFiles() |
    wxEraseEvent:wxErase() |
    wxFileDirPickerEvent:wxFileDirPicker() |
    wxFocusEvent:wxFocus() |
    wxFontPickerEvent:wxFontPicker() |
    wxGridEvent:wxGrid() |
    wxHelpEvent:wxHelp() |
```

```
    wxHtmlLinkEvent:wxHtmlLink() |
    wxIconizeEvent:wxIconize() |
    wxIdleEvent:wxIdle() |
    wxInitDialogEvent:wxInitDialog() |
    wxJoystickEvent:wxJoystick() |
    wxKeyEvent:wxKey() |
    wxListEvent:wxList() |
    wxMaximizeEvent:wxMaximize() |
    wxMenuEvent:wxMenu() |
    wxMouseCaptureChangedEvent:wxMouseCaptureChanged() |
    wxMouseCaptureLostEvent:wxMouseCaptureLost() |
    wxMouseEvent:wxMouse() |
    wxMoveEvent:wxMove() |
    wxNavigationKeyEvent:wxNavigationKey() |
    wxPaintEvent:wxPaint() |
    wxPaletteChangedEvent:wxPaletteChanged() |
    wxQueryNewPaletteEvent:wxQueryNewPalette() |
    wxSashEvent:wxSash() |
    wxScrollEvent:wxScroll() |
    wxScrollWinEvent:wxScrollWin() |
    wxSetCursorEvent:wxSetCursor() |
    wxShowEvent:wxShow() |
    wxSizeEvent:wxSize() |
    wxSpinEvent:wxSpin() |
    wxSplitterEvent:wxSplitter() |
    wxStyledTextEvent:wxStyledText() |
    wxSysColourChangedEvent:wxSysColourChanged() |
    wxTaskBarIconEvent:wxTaskBarIcon() |
    wxTreeEvent:wxTree() |
    wxUpdateUIEvent:wxUpdateUI() |
    wxWebViewEvent:wxWebView() |
    wxWindowCreateEvent:wxWindowCreate() |
    wxWindowDestroyEvent:wxWindowDestroy()
```

## Exports

```
connect(This :: wxEvtHandler(), EventType :: wxEventType()) -> ok

connect(This :: wxEvtHandler(),
        EventType :: wxEventType(),
        Options :: [Option]) ->
           ok
```

Types:

```
   Option =
       {id, integer()} |
       {lastId, integer()} |
       {skip, boolean()} |
       callback |
       {callback, function()} |
```

```
      {userData, term()}
```

This function subscribes to events.

Subscribes to events of type `EventType`, in the range `id`, `lastId`.

The events will be received as messages if no callback is supplied.

Options

id:`{id, integer()}` The identifier (or first of the identifier range) to be associated with this event handler. Default is ?wxID_ANY

lastid:`{lastId,integer()}` The second part of the identifier range. If used 'id' must be set as the starting identifier range. Default is ?wxID_ANY

skip:`{skip,boolean()}` If skip is true further event_handlers will be called. This is not used if the 'callback' option is used. Default is `false`.

callback:`{callback,function()}` Use a callback`fun(EventRecord::wx(),EventObject::wxObject())` to process the event. Default not specified i.e. a message will be delivered to the process calling this function.

userData:`{userData,term()}` An erlang term that will be sent with the event. Default: [].

```
disconnect(This :: wxEvtHandler()) -> boolean()

disconnect(This :: wxEvtHandler(), EventType :: wxEventType()) ->
              boolean()

disconnect(This :: wxEvtHandler(),
           EventType :: wxEventType(),
           Opts :: [Option]) ->
              boolean()
```

Types:

```
   Option =
       {id, integer()} | {lastId, integer()} | {callback, function()}
```

This function unsubscribes the process or callback fun from the event handler.

EventType may be the atom 'null' to match any eventtype. Notice that the options skip and userdata is not used to match the eventhandler.

# wxFileDataObject

Erlang module

`wxFileDataObject` is a specialization of `wxDataObject` for file names. The program works with it just as if it were a list of absolute file names, but internally it uses the same format as Explorer and other compatible programs under Windows or GNOME/KDE file manager under Unix which makes it possible to receive files from them using this class.

See: `wxDataObject`, `wxDataObjectSimple` (not implemented in wx), `wxTextDataObject`, `wxBitmapDataObject`, `wxDataObject`

This class is derived (and can use functions) from: `wxDataObject`

wxWidgets docs: **wxFileDataObject**

## Data Types

```
wxFileDataObject() = wx:wx_object()
```

## Exports

```
new() -> wxFileDataObject()
```

Constructor.

```
addFile(This, File) -> ok
```

Types:

```
   This = wxFileDataObject()
   File = unicode:chardata()
```

Adds a file to the file list represented by this data object (Windows only).

```
getFilenames(This) -> [unicode:charlist()]
```

Types:

```
   This = wxFileDataObject()
```

Returns the array of file names.

```
destroy(This :: wxFileDataObject()) -> ok
```

Destroys the object.

# wxFileDialog

Erlang module

This class represents the file chooser dialog.

The path and filename are distinct elements of a full file pathname. If path is ?wxEmptyString, the current directory will be used. If filename is ?wxEmptyString, no default filename will be supplied. The wildcard determines what files are displayed in the file selector, and file extension supplies a type extension for the required filename.

The typical usage for the open file dialog is:

The typical usage for the save file dialog is instead somewhat simpler:

Remark: All implementations of the `wxFileDialog` provide a wildcard filter. Typing a filename containing wildcards (*, ?) in the filename text item, and clicking on Ok, will result in only those files matching the pattern being displayed. The wildcard may be a specification for multiple types of file with a description for each, such as: It must be noted that wildcard support in the native Motif file dialog is quite limited: only one file type is supported, and it is displayed without the descriptive test; "BMP files (*.bmp)|*.bmp" is displayed as "*.bmp", and both "BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif" and "Image files|*.bmp;*.gif" are errors. On Mac macOS in the open file dialog the filter choice box is not shown by default. Instead all given wildcards are appplied at the same time: So in the above example all bmp, gif and png files are displayed. To enforce the display of the filter choice set the corresponding `wxSystemOptions` before calling the file open dialog: But in contrast to Windows and Unix, where the file type choice filters only the selected files, on Mac macOS even in this case the dialog shows all files matching all file types. The files which does not match the currently selected file type are greyed out and are not selectable.

Styles

This class supports the following styles:

See: **Overview cmndlg**, ?wxFileSelector()

This class is derived (and can use functions) from: `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxFileDialog**

## Data Types

```
wxFileDialog() = wx:wx_object()
```

## Exports

```
new(Parent) -> wxFileDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxFileDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {message, unicode:chardata()} |
       {defaultDir, unicode:chardata()} |
       {defaultFile, unicode:chardata()} |
       {wildCard, unicode:chardata()} |
       {style, integer()} |
```

```
       {pos, {X :: integer(), Y :: integer()}} |
       {sz, {W :: integer(), H :: integer()}}
```

Constructor.

Use `wxDialog:showModal/1` to show the dialog.

```
destroy(This :: wxFileDialog()) -> ok
```

Destructor.

```
getDirectory(This) -> unicode:charlist()
```

Types:

```
   This = wxFileDialog()
```

Returns the default directory.

```
getFilename(This) -> unicode:charlist()
```

Types:

```
   This = wxFileDialog()
```

Returns the default filename.

Note: This function can't be used with dialogs which have the `wxFD_MULTIPLE` style, use `getFilenames/1` instead.

```
getFilenames(This) -> [unicode:charlist()]
```

Types:

```
   This = wxFileDialog()
```

Fills the array `filenames` with the names of the files chosen.

This function should only be used with the dialogs which have `wxFD_MULTIPLE` style, use `getFilename/1` for the others.

Note that under Windows, if the user selects shortcuts, the filenames include paths, since the application cannot determine the full path of each referenced file by appending the directory containing the shortcuts to the filename.

```
getFilterIndex(This) -> integer()
```

Types:

```
   This = wxFileDialog()
```

Returns the index into the list of filters supplied, optionally, in the wildcard parameter.

Before the dialog is shown, this is the index which will be used when the dialog is first displayed.

After the dialog is shown, this is the index selected by the user.

```
getMessage(This) -> unicode:charlist()
```

Types:

```
   This = wxFileDialog()
```

Returns the message that will be displayed on the dialog.

```
getPath(This) -> unicode:charlist()
```

Types:

```
This = wxFileDialog()
```

Returns the full path (directory and filename) of the selected file.

Note: This function can't be used with dialogs which have the `wxFD_MULTIPLE` style, use `getPaths/1` instead.

```
getPaths(This) -> [unicode:charlist()]
```

Types:

```
This = wxFileDialog()
```

Fills the array `paths` with the full paths of the files chosen.

This function should only be used with the dialogs which have `wxFD_MULTIPLE` style, use `getPath/1` for the others.

```
getWildcard(This) -> unicode:charlist()
```

Types:

```
This = wxFileDialog()
```

Returns the file dialog wildcard.

```
setDirectory(This, Directory) -> ok
```

Types:

```
This = wxFileDialog()
Directory = unicode:chardata()
```

Sets the default directory.

```
setFilename(This, Setfilename) -> ok
```

Types:

```
This = wxFileDialog()
Setfilename = unicode:chardata()
```

Sets the default filename.

In wxGTK this will have little effect unless a default directory has previously been set.

```
setFilterIndex(This, FilterIndex) -> ok
```

Types:

```
This = wxFileDialog()
FilterIndex = integer()
```

Sets the default filter index, starting from zero.

```
setMessage(This, Message) -> ok
```

Types:

```
This = wxFileDialog()
Message = unicode:chardata()
```

Sets the message that will be displayed on the dialog.

```
setPath(This, Path) -> ok
```

Types:

```
   This = wxFileDialog()
   Path = unicode:chardata()
```

Sets the path (the combined directory and filename that will be returned when the dialog is dismissed).

```
setWildcard(This, WildCard) -> ok
```

Types:

```
   This = wxFileDialog()
   WildCard = unicode:chardata()
```

Sets the wildcard, which can contain multiple file types, for example: "BMP files (*.bmp)|*.bmp|GIF files (*.gif)|*.gif".

Note that the native Motif dialog has some limitations with respect to wildcards; see the Remarks section above.

# wxFileDirPickerEvent

Erlang module

This event class is used for the events generated by `wxFilePickerCtrl` and by `wxDirPickerCtrl`.

See: `wxFilePickerCtrl`, `wxDirPickerCtrl`

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxFileDirPickerEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxFileDirPickerEventType` to subscribe to events of this type.

## Data Types

```
wxFileDirPickerEvent() = wx:wx_object()
wxFileDirPicker() =
    #wxFileDirPicker{type =
                         wxFileDirPickerEvent:wxFileDirPickerEventType(),
                     path = unicode:chardata()}
wxFileDirPickerEventType() =
    command_filepicker_changed | command_dirpicker_changed
```

## Exports

```
getPath(This) -> unicode:charlist()
```

Types:

```
   This = wxFileDirPickerEvent()
```

Retrieve the absolute path of the file/directory the user has just selected.

# wxFilePickerCtrl

Erlang module

This control allows the user to select a file. The generic implementation is a button which brings up a `wxFileDialog` when clicked. Native implementation may differ but this is usually a (small) widget which give access to the file-chooser dialog. It is only available if `wxUSE_FILEPICKERCTRL` is set to 1 (the default).

Styles

This class supports the following styles:

See: `wxFileDialog`, `wxFileDirPickerEvent`

This class is derived (and can use functions) from: `wxPickerBase` `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxFilePickerCtrl**

## Events

Event types emitted from this class: `command_filepicker_changed`

## Data Types

```
wxFilePickerCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxFilePickerCtrl()
```

```
new(Parent, Id) -> wxFilePickerCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxFilePickerCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {path, unicode:chardata()} |
       {message, unicode:chardata()} |
       {wildcard, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Initializes the object and calls `create/4` with all the parameters.

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = wxFilePickerCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()

create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxFilePickerCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {path, unicode:chardata()} |
       {message, unicode:chardata()} |
       {wildcard, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Creates this widget with the given parameters.

Return: true if the control was successfully created or false if creation failed.

```
getPath(This) -> unicode:charlist()
```

Types:

```
   This = wxFilePickerCtrl()
```

Returns the absolute path of the currently selected file.

```
setPath(This, Filename) -> ok
```

Types:

```
   This = wxFilePickerCtrl()
   Filename = unicode:chardata()
```

Sets the absolute path of the currently selected file.

If the control uses `wxFLP_FILE_MUST_EXIST` and does not use `wxFLP_USE_TEXTCTRL` style, the `filename` must be a name of an existing file and will be simply ignored by the native wxGTK implementation if this is not the case (the generic implementation used under the other platforms accepts even invalid file names currently, but this is subject to change in the future, don't rely on being able to use non-existent paths with it).

```
destroy(This :: wxFilePickerCtrl()) -> ok
```

Destroys the object.

# wxFindReplaceData

Erlang module

`wxFindReplaceData` holds the data for `wxFindReplaceDialog`.

It is used to initialize the dialog with the default values and will keep the last values from the dialog when it is closed. It is also updated each time a `wxFindDialogEvent` (not implemented in wx) is generated so instead of using the `wxFindDialogEvent` (not implemented in wx) methods you can also directly query this object.

Note that all `SetXXX()` methods may only be called before showing the dialog and calling them has no effect later.

wxWidgets docs: **wxFindReplaceData**

## Data Types

```
wxFindReplaceData() = wx:wx_object()
```

## Exports

```
new() -> wxFindReplaceData()
```

```
new(Options :: [Option]) -> wxFindReplaceData()
```

Types:

```
Option = {flags, integer()}
```

Constructor initializes the flags to default value (0).

```
getFindString(This) -> unicode:charlist()
```

Types:

```
This = wxFindReplaceData()
```

Get the string to find.

```
getReplaceString(This) -> unicode:charlist()
```

Types:

```
This = wxFindReplaceData()
```

Get the replacement string.

```
getFlags(This) -> integer()
```

Types:

```
This = wxFindReplaceData()
```

Get the combination of `wxFindReplaceFlags` values.

```
setFlags(This, Flags) -> ok
```

Types:

```
This = wxFindReplaceData()
Flags = integer()
```

Set the flags to use to initialize the controls of the dialog.

```
setFindString(This, Str) -> ok
```

Types:

```
This = wxFindReplaceData()
Str = unicode:chardata()
```

Set the string to find (used as initial value by the dialog).

```
setReplaceString(This, Str) -> ok
```

Types:

```
This = wxFindReplaceData()
Str = unicode:chardata()
```

Set the replacement string (used as initial value by the dialog).

```
destroy(This :: wxFindReplaceData()) -> ok
```

Destroys the object.

# wxFindReplaceDialog

Erlang module

wxFindReplaceDialog is a standard modeless dialog which is used to allow the user to search for some text (and possibly replace it with something else).

The actual searching is supposed to be done in the owner window which is the parent of this dialog. Note that it means that unlike for the other standard dialogs this one must have a parent window. Also note that there is no way to use this dialog in a modal way; it is always, by design and implementation, modeless.

Please see the page_samples_dialogs sample for an example of using it.

This class is derived (and can use functions) from: wxDialog wxTopLevelWindow wxWindow wxEvtHandler

wxWidgets docs: **wxFindReplaceDialog**

## Data Types

```
wxFindReplaceDialog() = wx:wx_object()
```

## Exports

```
new() -> wxFindReplaceDialog()
```

```
new(Parent, Data, Title) -> wxFindReplaceDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
Data = wxFindReplaceData:wxFindReplaceData()
Title = unicode:chardata()
```

```
new(Parent, Data, Title, Options :: [Option]) ->
      wxFindReplaceDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
Data = wxFindReplaceData:wxFindReplaceData()
Title = unicode:chardata()
Option = {style, integer()}
```

After using default constructor create/5 must be called.

The parent and data parameters must be non-NULL.

```
destroy(This :: wxFindReplaceDialog()) -> ok
```

Destructor.

```
create(This, Parent, Data, Title) -> boolean()
```

Types:

```
   This = wxFindReplaceDialog()
   Parent = wxWindow:wxWindow()
   Data = wxFindReplaceData:wxFindReplaceData()
   Title = unicode:chardata()

create(This, Parent, Data, Title, Options :: [Option]) ->
          boolean()
```

Types:

```
   This = wxFindReplaceDialog()
   Parent = wxWindow:wxWindow()
   Data = wxFindReplaceData:wxFindReplaceData()
   Title = unicode:chardata()
   Option = {style, integer()}
```

Creates the dialog; use `wxWindow:show/2` to show it on screen.

The `parent` and `data` parameters must be non-NULL.

```
getData(This) -> wxFindReplaceData:wxFindReplaceData()
```

Types:

```
   This = wxFindReplaceDialog()
```

Get the `wxFindReplaceData` object used by this dialog.

# wxFlexGridSizer

Erlang module

A flex grid sizer is a sizer which lays out its children in a two-dimensional table with all table fields in one row having the same height and all fields in one column having the same width, but all rows or all columns are not necessarily the same height or width as in the `wxGridSizer`.

Since wxWidgets 2.5.0, `wxFlexGridSizer` can also size items equally in one direction but unequally ("flexibly") in the other. If the sizer is only flexible in one direction (this can be changed using `setFlexibleDirection/2`), it needs to be decided how the sizer should grow in the other ("non-flexible") direction in order to fill the available space. The `setNonFlexibleGrowMode/2` method serves this purpose.

See: `wxSizer`, **Overview sizer**

This class is derived (and can use functions) from: `wxGridSizer` `wxSizer`

wxWidgets docs: **wxFlexGridSizer**

## Data Types

```
wxFlexGridSizer() = wx:wx_object()
```

## Exports

```
new(Cols) -> wxFlexGridSizer()
```

Types:

```
   Cols = integer()
```

```
new(Cols, Options :: [Option]) -> wxFlexGridSizer()
```

Types:

```
   Cols = integer()
   Option = {gap, {W :: integer(), H :: integer()}}
```

```
new(Cols, Vgap, Hgap) -> wxFlexGridSizer()
new(Rows, Cols, Gap) -> wxFlexGridSizer()
```

Types:

```
   Rows = Cols = integer()
   Gap = {W :: integer(), H :: integer()}
```

```
new(Rows, Cols, Vgap, Hgap) -> wxFlexGridSizer()
```

Types:

```
   Rows = Cols = Vgap = Hgap = integer()
```

```
addGrowableCol(This, Idx) -> ok
```

Types:

```
   This = wxFlexGridSizer()
   Idx = integer()

addGrowableCol(This, Idx, Options :: [Option]) -> ok
```

Types:

```
   This = wxFlexGridSizer()
   Idx = integer()
   Option = {proportion, integer()}
```

Specifies that column `idx` (starting from zero) should be grown if there is extra space available to the sizer.

The `proportion` parameter has the same meaning as the stretch factor for the sizers (see `wxBoxSizer`) except that if all proportions are 0, then all columns are resized equally (instead of not being resized at all).

Notice that the column must not be already growable, if you need to change the proportion you must call `removeGrowableCol/2` first and then make it growable (with a different proportion) again. You can use `IsColGrowable()` (not implemented in wx) to check whether a column is already growable.

```
addGrowableRow(This, Idx) -> ok
```

Types:

```
   This = wxFlexGridSizer()
   Idx = integer()

addGrowableRow(This, Idx, Options :: [Option]) -> ok
```

Types:

```
   This = wxFlexGridSizer()
   Idx = integer()
   Option = {proportion, integer()}
```

Specifies that row idx (starting from zero) should be grown if there is extra space available to the sizer.

This is identical to `addGrowableCol/3` except that it works with rows and not columns.

```
getFlexibleDirection(This) -> integer()
```

Types:

```
   This = wxFlexGridSizer()
```

Returns a ?wxOrientation value that specifies whether the sizer flexibly resizes its columns, rows, or both (default).

Return: One of the following values:

See: `setFlexibleDirection/2`

```
getNonFlexibleGrowMode(This) -> wx:wx_enum()
```

Types:

```
   This = wxFlexGridSizer()
```

Returns the value that specifies how the sizer grows in the "non-flexible" direction if there is one.

The behaviour of the elements in the flexible direction (i.e. both rows and columns by default, or rows only if `getFlexibleDirection/1` is `wxVERTICAL` or columns only if it is `wxHORIZONTAL`) is always governed by their proportion as specified in the call to `addGrowableRow/3` or `addGrowableCol/3`. What happens in the other direction depends on the value of returned by this function as described below.

Return: One of the following values:

See: `setFlexibleDirection/2`, `setNonFlexibleGrowMode/2`

`removeGrowableCol(This, Idx) -> ok`

Types:

```
This = wxFlexGridSizer()
Idx = integer()
```

Specifies that the `idx` column index is no longer growable.

`removeGrowableRow(This, Idx) -> ok`

Types:

```
This = wxFlexGridSizer()
Idx = integer()
```

Specifies that the `idx` row index is no longer growable.

`setFlexibleDirection(This, Direction) -> ok`

Types:

```
This = wxFlexGridSizer()
Direction = integer()
```

Specifies whether the sizer should flexibly resize its columns, rows, or both.

Argument `direction` can be `wxVERTICAL`, `wxHORIZONTAL` or `wxBOTH` (which is the default value). Any other value is ignored.

See `getFlexibleDirection/1` for the explanation of these values. Note that this method does not trigger relayout.

`setNonFlexibleGrowMode(This, Mode) -> ok`

Types:

```
This = wxFlexGridSizer()
Mode = wx:wx_enum()
```

Specifies how the sizer should grow in the non-flexible direction if there is one (so `setFlexibleDirection/2` must have been called previously).

Argument `mode` can be one of those documented in `getNonFlexibleGrowMode/1`, please see there for their explanation. Note that this method does not trigger relayout.

`destroy(This :: wxFlexGridSizer()) -> ok`

Destroys the object.

# wxFocusEvent

Erlang module

A focus event is sent when a window's focus changes. The window losing focus receives a "kill focus" event while the window gaining it gets a "set focus" one.

Notice that the set focus event happens both when the user gives focus to the window (whether using the mouse or keyboard) and when it is done from the program itself using `wxWindow:setFocus/1`.

The focus event handlers should almost invariably call `wxEvent:skip/2` on their event argument to allow the default handling to take place. Failure to do this may result in incorrect behaviour of the native controls. Also note that wxEVT_KILL_FOCUS handler must not call `wxWindow:setFocus/1` as this, again, is not supported by all native controls. If you need to do this, consider using the `Delayed Action Mechanism` (not implemented in wx) described in `wxIdleEvent` documentation.

See: **Overview events**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxFocusEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxFocusEventType` to subscribe to events of this type.

## Data Types

```
wxFocusEvent() = wx:wx_object()
wxFocus() =
    #wxFocus{type = wxFocusEvent:wxFocusEventType(),
             win = wxWindow:wxWindow()}
wxFocusEventType() = set_focus | kill_focus
```

## Exports

```
getWindow(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxFocusEvent()
```

Returns the window associated with this event, that is the window which had the focus before for the `wxEVT_SET_FOCUS` event and the window which is going to receive focus for the `wxEVT_KILL_FOCUS` one.

Warning: the window pointer may be NULL!

# wxFont

Erlang module

A font is an object which determines the appearance of text.

Fonts are used for drawing text to a device context, and setting the appearance of a window's text, see `wxDC:setFont/2` and `wxWindow:setFont/2`.

The easiest way to create a custom font is to use `wxFontInfo` (not implemented in wx) object to specify the font attributes and then use `new/5` constructor. Alternatively, you could start with one of the pre-defined fonts or use `wxWindow:getFont/1` and modify the font, e.g. by increasing its size using `MakeLarger()` (not implemented in wx) or changing its weight using `MakeBold()` (not implemented in wx).

This class uses reference counting and copy-on-write internally so that assignments between two instances of this class are very cheap. You can therefore use actual objects instead of pointers without efficiency problems. If an instance of this class is changed it will create its own data internally so that other instances, which previously shared the data using the reference counting, are not affected.

You can retrieve the current system font settings with `wxSystemSettings`.

Predefined objects (include wx.hrl): ?wxNullFont, ?wxNORMAL_FONT, ?wxSMALL_FONT, ?wxITALIC_FONT, ?wxSWISS_FONT

See: **Overview font**, `wxDC:setFont/2`, `wxDC:drawText/3`, `wxDC:getTextExtent/3`, `wxFontDialog`, `wxSystemSettings`

wxWidgets docs: **wxFont**

## Data Types

```
wxFont() = wx:wx_object()
```

## Exports

```
new() -> wxFont()
```

Default ctor.

```
new(NativeInfoString) -> wxFont()
new(Font) -> wxFont()
```

Types:

```
Font = wxFont()
```

Copy constructor, uses reference counting.

```
new(PointSize, Family, Style, Weight) -> wxFont()
new(PixelSize, Family, Style, Weight) -> wxFont()
```

Types:

```
PixelSize = {W :: integer(), H :: integer()}
Family = Style = Weight = wx:wx_enum()

new(PointSize, Family, Style, Weight, Options :: [Option]) ->
```

```
       wxFont()
new(PixelSize, Family, Style, Weight, Options :: [Option]) ->
       wxFont()
```

Types:

```
   PixelSize = {W :: integer(), H :: integer()}
   Family = Style = Weight = wx:wx_enum()
   Option =
       {underline, boolean()} |
       {faceName, unicode:chardata()} |
       {encoding, wx:wx_enum()}
```

Creates a font object with the specified attributes and size in pixels.

Notice that the use of this constructor is often more verbose and less readable than the use of constructor from `wxFontInfo` (not implemented in wx), consider using that constructor instead.

Remark: If the desired font does not exist, the closest match will be chosen. Under Windows, only scalable TrueType fonts are used.

```
destroy(This :: wxFont()) -> ok
```

Destructor.

See reference-counted object destruction for more info.

Remark: Although all remaining fonts are deleted when the application exits, the application should try to clean up all fonts itself. This is because wxWidgets cannot know if a pointer to the font object is stored in an application data structure, and there is a risk of double deletion.

```
isFixedWidth(This) -> boolean()
```

Types:

```
   This = wxFont()
```

Returns true if the font is a fixed width (or monospaced) font, false if it is a proportional one or font is invalid.

Note that this function under some platforms is different from just testing for the font family being equal to `wxFONTFAMILY_TELETYPE` because native platform-specific functions are used for the check (resulting in a more accurate return value).

```
getDefaultEncoding() -> wx:wx_enum()
```

Returns the current application's default encoding.

See: **Overview fontencoding**, `setDefaultEncoding/1`

```
getFaceName(This) -> unicode:charlist()
```

Types:

```
   This = wxFont()
```

Returns the face name associated with the font, or the empty string if there is no face information.

See: `setFaceName/2`

```
getFamily(This) -> wx:wx_enum()
```

Types:

```
This = wxFont()
```

Gets the font family if possible.

As described in ?wxFontFamily docs the returned value acts as a rough, basic classification of the main font properties (look, spacing).

If the current font face name is not recognized by `wxFont` or by the underlying system, `wxFONTFAMILY_DEFAULT` is returned.

Note that currently this function is not very precise and so not particularly useful. Font families mostly make sense only for font creation, see `setFamily/2`.

See: `setFamily/2`

### `getNativeFontInfoDesc(This) -> unicode:charlist()`

Types:

```
This = wxFont()
```

Returns the platform-dependent string completely describing this font.

Returned string is always non-empty unless the font is invalid (in which case an assert is triggered).

Note that the returned string is not meant to be shown or edited by the user: a typical use of this function is for serializing in string-form a `wxFont` object.

See: `SetNativeFontInfo()` (not implemented in wx), `getNativeFontInfoUserDesc/1`

### `getNativeFontInfoUserDesc(This) -> unicode:charlist()`

Types:

```
This = wxFont()
```

Returns a user-friendly string for this font object.

Returned string is always non-empty unless the font is invalid (in which case an assert is triggered).

The string does not encode all `wxFont` infos under all platforms; e.g. under wxMSW the font family is not present in the returned string.

Some examples of the formats of returned strings (which are platform-dependent) are in `SetNativeFontInfoUserDesc()` (not implemented in wx).

See: `SetNativeFontInfoUserDesc()` (not implemented in wx), `getNativeFontInfoDesc/1`

### `getPointSize(This) -> integer()`

Types:

```
This = wxFont()
```

Gets the point size as an integer number.

This function is kept for compatibility reasons. New code should use `GetFractionalPointSize()` (not implemented in wx) and support fractional point sizes.

See: `setPointSize/2`

See: `GetFractionalPointSize()` (not implemented in wx)

### `getStyle(This) -> wx:wx_enum()`

Types:

```
This = wxFont()
```

Gets the font style.

See ?wxFontStyle for a list of valid styles.

See: `setStyle/2`

```
getUnderlined(This) -> boolean()
```

Types:

```
This = wxFont()
```

Returns true if the font is underlined, false otherwise.

See: `setUnderlined/2`

```
getWeight(This) -> wx:wx_enum()
```

Types:

```
This = wxFont()
```

Gets the font weight.

See ?wxFontWeight for a list of valid weight identifiers.

See: `setWeight/2`

```
ok(This) -> boolean()
```

Types:

```
This = wxFont()
```

See: `isOk/1`.

```
isOk(This) -> boolean()
```

Types:

```
This = wxFont()
```

Returns true if this object is a valid font, false otherwise.

```
setDefaultEncoding(Encoding) -> ok
```

Types:

```
Encoding = wx:wx_enum()
```

Sets the default font encoding.

See: **Overview fontencoding**, `getDefaultEncoding/0`

```
setFaceName(This, FaceName) -> boolean()
```

Types:

```
This = wxFont()
FaceName = unicode:chardata()
```

Sets the facename for the font.

Remark: To avoid portability problems, don't rely on a specific face, but specify the font family instead (see ? wxFontFamily and `setFamily/2`).

Return: true if the given face name exists; if the face name doesn't exist in the user's system then the font is invalidated (so that `isOk/1` will return false) and false is returned.

See: `getFaceName/1`, `setFamily/2`

```
setFamily(This, Family) -> ok
```

Types:

```
This = wxFont()
Family = wx:wx_enum()
```

Sets the font family.

As described in ?wxFontFamily docs the given `family` value acts as a rough, basic indication of the main font properties (look, spacing).

Note that changing the font family results in changing the font face name.

See: `getFamily/1`, `setFaceName/2`

```
setPointSize(This, PointSize) -> ok
```

Types:

```
This = wxFont()
PointSize = integer()
```

Sets the font size in points to an integer value.

This is a legacy version of the function only supporting integer point sizes. It can still be used, but to avoid unnecessarily restricting the font size in points to integer values, consider using the new (added in wxWidgets 3.1.2) `SetFractionalPointSize()` (not implemented in wx) function instead.

```
setStyle(This, Style) -> ok
```

Types:

```
This = wxFont()
Style = wx:wx_enum()
```

Sets the font style.

See: `getStyle/1`

```
setUnderlined(This, Underlined) -> ok
```

Types:

```
This = wxFont()
Underlined = boolean()
```

Sets underlining.

See: `getUnderlined/1`

```
setWeight(This, Weight) -> ok
```

Types:

```
This = wxFont()
Weight = wx:wx_enum()
```

Sets the font weight.

See: `getWeight/1`

# wxFontData

Erlang module

This class holds a variety of information related to font dialogs.

See: **Overview cmndlg**, `wxFont`, `wxFontDialog`

wxWidgets docs: **wxFontData**

## Data Types

```
wxFontData() = wx:wx_object()
```

## Exports

```
new() -> wxFontData()
```

Constructor.

Initializes `fontColour` to black, `showHelp` to false, `allowSymbols` to true, `enableEffects` to true, `minSize` to 0 and `maxSize` to 0.

```
new(Data) -> wxFontData()
```

Types:

```
Data = wxFontData()
```

Copy Constructor.

```
enableEffects(This, Enable) -> ok
```

Types:

```
This = wxFontData()
Enable = boolean()
```

Enables or disables "effects" under Windows or generic only.

This refers to the controls for manipulating colour, strikeout and underline properties.

The default value is true.

```
getAllowSymbols(This) -> boolean()
```

Types:

```
This = wxFontData()
```

Under Windows, returns a flag determining whether symbol fonts can be selected.

Has no effect on other platforms.

The default value is true.

```
getColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxFontData()
```

Gets the colour associated with the font dialog.

The default value is black.

```
getChosenFont(This) -> wxFont:wxFont()
```

Types:

```
   This = wxFontData()
```

Gets the font chosen by the user if the user pressed OK (`wxFontDialog::ShowModal()` (not implemented in wx) returned wxID_OK).

```
getEnableEffects(This) -> boolean()
```

Types:

```
   This = wxFontData()
```

Determines whether "effects" are enabled under Windows.

This refers to the controls for manipulating colour, strikeout and underline properties.

The default value is true.

```
getInitialFont(This) -> wxFont:wxFont()
```

Types:

```
   This = wxFontData()
```

Gets the font that will be initially used by the font dialog.

This should have previously been set by the application.

```
getShowHelp(This) -> boolean()
```

Types:

```
   This = wxFontData()
```

Returns true if the Help button will be shown (Windows only).

The default value is false.

```
setAllowSymbols(This, AllowSymbols) -> ok
```

Types:

```
   This = wxFontData()
   AllowSymbols = boolean()
```

Under Windows, determines whether symbol fonts can be selected.

Has no effect on other platforms.

The default value is true.

```
setChosenFont(This, Font) -> ok
```

Types:

```
   This = wxFontData()
   Font = wxFont:wxFont()
```

Sets the font that will be returned to the user (for internal use only).

```
setColour(This, Colour) -> ok
```

Types:

```
   This = wxFontData()
   Colour = wx:wx_colour()
```

Sets the colour that will be used for the font foreground colour.

The default colour is black.

```
setInitialFont(This, Font) -> ok
```

Types:

```
   This = wxFontData()
   Font = wxFont:wxFont()
```

Sets the font that will be initially used by the font dialog.

```
setRange(This, Min, Max) -> ok
```

Types:

```
   This = wxFontData()
   Min = Max = integer()
```

Sets the valid range for the font point size (Windows only).

The default is 0, 0 (unrestricted range).

```
setShowHelp(This, ShowHelp) -> ok
```

Types:

```
   This = wxFontData()
   ShowHelp = boolean()
```

Determines whether the Help button will be displayed in the font dialog (Windows only).

The default value is false.

```
destroy(This :: wxFontData()) -> ok
```

Destroys the object.

# wxFontDialog

Erlang module

This class represents the font chooser dialog.

See: **Overview cmndlg**, `wxFontData`, ?wxGetFontFromUser()

This class is derived (and can use functions) from: `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxFontDialog**

## Data Types

```
wxFontDialog() = wx:wx_object()
```

## Exports

```
new() -> wxFontDialog()
```

Default ctor.

`create/3` must be called before the dialog can be shown.

```
new(Parent, Data) -> wxFontDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
Data = wxFontData:wxFontData()
```

Constructor.

Pass a parent window, and the `wxFontData` object to be used to initialize the dialog controls.

```
create(This, Parent, Data) -> boolean()
```

Types:

```
This = wxFontDialog()
Parent = wxWindow:wxWindow()
Data = wxFontData:wxFontData()
```

Creates the dialog if the `wxFontDialog` object had been initialized using the default constructor.

Return: true on success and false if an error occurred.

```
getFontData(This) -> wxFontData:wxFontData()
```

Types:

```
This = wxFontDialog()
```

Returns the `wxFontData` associated with the font dialog.

```
destroy(This :: wxFontDialog()) -> ok
```

Destroys the object.

# wxFontPickerCtrl

Erlang module

This control allows the user to select a font. The generic implementation is a button which brings up a `wxFontDialog` when clicked. Native implementation may differ but this is usually a (small) widget which give access to the font-chooser dialog. It is only available if `wxUSE_FONTPICKERCTRL` is set to 1 (the default).

Styles

This class supports the following styles:

See: `wxFontDialog`, `wxFontPickerEvent`

This class is derived (and can use functions) from: `wxPickerBase` `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxFontPickerCtrl**

## Events

Event types emitted from this class: `command_fontpicker_changed`

## Data Types

```
wxFontPickerCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxFontPickerCtrl()
```

```
new(Parent, Id) -> wxFontPickerCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxFontPickerCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {initial, wxFont:wxFont()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Initializes the object and calls `create/4` with all the parameters.

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = wxFontPickerCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()

create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxFontPickerCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {initial, wxFont:wxFont()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Creates this widget with given parameters.

Return: true if the control was successfully created or false if creation failed.

```
getSelectedFont(This) -> wxFont:wxFont()
```

Types:

```
   This = wxFontPickerCtrl()
```

Returns the currently selected font.

Note that this function is completely different from `wxWindow:getFont/1`.

```
setSelectedFont(This, Font) -> ok
```

Types:

```
   This = wxFontPickerCtrl()
   Font = wxFont:wxFont()
```

Sets the currently selected font.

Note that this function is completely different from `wxWindow:setFont/2`.

```
getMaxPointSize(This) -> integer()
```

Types:

```
   This = wxFontPickerCtrl()
```

Returns the maximum point size value allowed for the user-chosen font.

```
setMaxPointSize(This, Max) -> ok
```

Types:

```
   This = wxFontPickerCtrl()
   Max = integer()
```

Sets the maximum point size value allowed for the user-chosen font.

The default value is 100. Note that big fonts can require a lot of memory and CPU time both for creation and for rendering; thus, specially because the user has the option to specify the fontsize through a text control (see

wxFNTP_USE_TEXTCTRL), it's a good idea to put a limit to the maximum font size when huge fonts do not make much sense.

```
destroy(This :: wxFontPickerCtrl()) -> ok
```

Destroys the object.

# wxFontPickerEvent

Erlang module

This event class is used for the events generated by `wxFontPickerCtrl`.

See: `wxFontPickerCtrl`

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxFontPickerEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxFontPickerEventType` to subscribe to events of this type.

## Data Types

```
wxFontPickerEvent() = wx:wx_object()
wxFontPicker() =
    #wxFontPicker{type =
                      wxFontPickerEvent:wxFontPickerEventType(),
                  font = wxFont:wxFont()}
wxFontPickerEventType() = command_fontpicker_changed
```

## Exports

```
getFont(This) -> wxFont:wxFont()
```

Types:

```
   This = wxFontPickerEvent()
```

Retrieve the font the user has just selected.

# wxFrame

Erlang module

A frame is a window whose size and position can (usually) be changed by the user.

It usually has thick borders and a title bar, and can optionally contain a menu bar, toolbar and status bar. A frame can contain any window that is not a frame or dialog.

A frame that has a status bar and toolbar, created via the `createStatusBar/2` and `createToolBar/2` functions, manages these windows and adjusts the value returned by `wxWindow:getClientSize/1` to reflect the remaining size available to application windows.

Remark: An application should normally define an `wxCloseEvent` handler for the frame to respond to system close events, for example so that related data and subwindows can be cleaned up.

Default event processing

`wxFrame` processes the following events:

Styles

This class supports the following styles:

See also the overview_windowstyles.

Extra Styles

This class supports the following extra styles:

See: `wxMDIParentFrame`, `wxMDIChildFrame`, `wxMiniFrame`, `wxDialog`

This class is derived (and can use functions) from: `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxFrame**

## Events

Event types emitted from this class: `close_window`, `iconize`, `menu_open`, `menu_close`, `menu_highlight`

## Data Types

```
wxFrame() = wx:wx_object()
```

## Exports

```
new() -> wxFrame()
```

Default constructor.

```
new(Parent, Id, Title) -> wxFrame()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()

new(Parent, Id, Title, Options :: [Option]) -> wxFrame()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructor, creating the window.

Remark: For Motif, MWM (the Motif Window Manager) should be running for any window styles to work (otherwise all styles take effect).

See: `create/5`

```
destroy(This :: wxFrame()) -> ok
```

Destructor.

Destroys all child windows and menu bar if present.

See overview_windowdeletion for more info.

```
create(This, Parent, Id, Title) -> boolean()
```

Types:

```
   This = wxFrame()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()

create(This, Parent, Id, Title, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxFrame()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Used in two-step frame construction.

See `new/4` for further details.

```
createStatusBar(This) -> wxStatusBar:wxStatusBar()
```

Types:

```
   This = wxFrame()
```

```
createStatusBar(This, Options :: [Option]) ->
                  wxStatusBar:wxStatusBar()
```

Types:

```
   This = wxFrame()
   Option =
       {number, integer()} | {style, integer()} | {id, integer()}
```

Creates a status bar at the bottom of the frame.

Return: A pointer to the status bar if it was created successfully, NULL otherwise.

Remark: The width of the status bar is the whole width of the frame (adjusted automatically when resizing), and the height and text size are chosen by the host windowing system.

See: `setStatusText/3`, `OnCreateStatusBar()` (not implemented in wx), `getStatusBar/1`

```
createToolBar(This) -> wxToolBar:wxToolBar()
```

Types:

```
   This = wxFrame()
```

```
createToolBar(This, Options :: [Option]) -> wxToolBar:wxToolBar()
```

Types:

```
   This = wxFrame()
   Option = {style, integer()} | {id, integer()}
```

Creates a toolbar at the top or left of the frame.

Return: A pointer to the toolbar if it was created successfully, NULL otherwise.

Remark: By default, the toolbar is an instance of `wxToolBar`. To use a different class, override `OnCreateToolBar()` (not implemented in wx). When a toolbar has been created with this function, or made known to the frame with `setToolBar/2`, the frame will manage the toolbar position and adjust the return value from `wxWindow:getClientSize/1` to reflect the available space for application windows. Under Pocket PC, you should always use this function for creating the toolbar to be managed by the frame, so that wxWidgets can use a combined menubar and toolbar. Where you manage your own toolbars, create a `wxToolBar` as usual.

See: `createStatusBar/2`, `OnCreateToolBar()` (not implemented in wx), `setToolBar/2`, `getToolBar/1`

```
getClientAreaOrigin(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
   This = wxFrame()
```

Returns the origin of the frame client area (in client coordinates).

It may be different from (0, 0) if the frame has a toolbar.

```
getMenuBar(This) -> wxMenuBar:wxMenuBar()
```

Types:

```
This = wxFrame()
```

Returns a pointer to the menubar currently associated with the frame (if any).

See: `setMenuBar/2`, `wxMenuBar`, `wxMenu`

```
getStatusBar(This) -> wxStatusBar:wxStatusBar()
```

Types:

```
This = wxFrame()
```

Returns a pointer to the status bar currently associated with the frame (if any).

See: `createStatusBar/2`, `wxStatusBar`

```
getStatusBarPane(This) -> integer()
```

Types:

```
This = wxFrame()
```

Returns the status bar pane used to display menu and toolbar help.

See: `setStatusBarPane/2`

```
getToolBar(This) -> wxToolBar:wxToolBar()
```

Types:

```
This = wxFrame()
```

Returns a pointer to the toolbar currently associated with the frame (if any).

See: `createToolBar/2`, `wxToolBar`, `setToolBar/2`

```
processCommand(This, Id) -> boolean()
```

Types:

```
This = wxFrame()
Id = integer()
```

Simulate a menu command.

```
sendSizeEvent(This) -> ok
```

Types:

```
This = wxFrame()
```

```
sendSizeEvent(This, Options :: [Option]) -> ok
```

Types:

```
This = wxFrame()
Option = {flags, integer()}
```

This function sends a dummy `wxSizeEvent` to the window allowing it to re-layout its children positions.

It is sometimes useful to call this function after adding or deleting a children after the frame creation or if a child size changes. Note that if the frame is using either sizers or constraints for the children layout, it is enough to call `wxWindow:layout/1` directly and this function should not be used in this case.

If `flags` includes `wxSEND_EVENT_POST` value, this function posts the event, i.e. schedules it for later processing, instead of dispatching it directly. You can also use `PostSizeEvent()` (not implemented in wx) as a more readable equivalent of calling this function with this flag.

```
setMenuBar(This, MenuBar) -> ok
```

Types:

```
This = wxFrame()
MenuBar = wxMenuBar:wxMenuBar()
```

Tells the frame to show the given menu bar.

Remark: If the frame is destroyed, the menu bar and its menus will be destroyed also, so do not delete the menu bar explicitly (except by resetting the frame's menu bar to another frame or NULL). Under Windows, a size event is generated, so be sure to initialize data members properly before calling `setMenuBar/2`. Note that on some platforms, it is not possible to call this function twice for the same frame object.

See: `getMenuBar/1`, `wxMenuBar`, `wxMenu`

```
setStatusBar(This, StatusBar) -> ok
```

Types:

```
This = wxFrame()
StatusBar = wxStatusBar:wxStatusBar()
```

Associates a status bar with the frame.

If `statusBar` is NULL, then the status bar, if present, is detached from the frame, but `not` deleted.

See: `createStatusBar/2`, `wxStatusBar`, `getStatusBar/1`

```
setStatusBarPane(This, N) -> ok
```

Types:

```
This = wxFrame()
N = integer()
```

Set the status bar pane used to display menu and toolbar help.

Using -1 disables help display.

```
setStatusText(This, Text) -> ok
```

Types:

```
This = wxFrame()
Text = unicode:chardata()
```

```
setStatusText(This, Text, Options :: [Option]) -> ok
```

Types:

```
This = wxFrame()
Text = unicode:chardata()
Option = {number, integer()}
```

Sets the status bar text and updates the status bar display.

This is a simple wrapper for `wxStatusBar:setStatusText/3` which doesn't do anything if the frame has no status bar, i.e. `getStatusBar/1` returns NULL.

Remark: Use an empty string to clear the status bar.

See: `createStatusBar/2`, `wxStatusBar`

```
setStatusWidths(This, Widths_field) -> ok
```

Types:

```
This = wxFrame()
Widths_field = [integer()]
```

Sets the widths of the fields in the status bar.

Remark: The widths of the variable fields are calculated from the total width of all fields, minus the sum of widths of the non-variable fields, divided by the number of variable fields.

```
setToolBar(This, ToolBar) -> ok
```

Types:

```
This = wxFrame()
ToolBar = wxToolBar:wxToolBar()
```

Associates a toolbar with the frame.

# wxGBSizerItem

Erlang module

The `wxGBSizerItem` class is used by the `wxGridBagSizer` for tracking the items in the sizer. It adds grid position and spanning information to the normal `wxSizerItem` by adding `wxGBPosition` (not implemented in wx) and `wxGBSpan` (not implemented in wx) attributes. Most of the time you will not need to use a `wxGBSizerItem` directly in your code, but there are a couple of cases where it is handy.

This class is derived (and can use functions) from: `wxSizerItem`

wxWidgets docs: **wxGBSizerItem**

## Data Types

```
wxGBSizerItem() = wx:wx_object()
```

# wxGCDC

Erlang module

wxGCDC is a device context that draws on a wxGraphicsContext.

wxGCDC does its best to implement wxDC API, but the following features are not (fully) implemented because wxGraphicsContext doesn't support them:

See: wxDC, wxGraphicsContext

This class is derived (and can use functions) from: wxDC

wxWidgets docs: **wxGCDC**

## Data Types

```
wxGCDC() = wx:wx_object()
```

## Exports

```
new() -> wxGCDC()
```

```
new(WindowDC) -> wxGCDC()
```

Types:

```
WindowDC =
    wxWindowDC:wxWindowDC() |
    wxMemoryDC:wxMemoryDC() |
    wxGraphicsContext:wxGraphicsContext()
```

Constructs a wxGCDC from a wxWindowDC.

```
destroy(This :: wxGCDC()) -> ok
```

```
getGraphicsContext(This) -> wxGraphicsContext:wxGraphicsContext()
```

Types:

```
This = wxGCDC()
```

Retrieves associated wxGraphicsContext.

```
setGraphicsContext(This, Context) -> ok
```

Types:

```
This = wxGCDC()
Context = wxGraphicsContext:wxGraphicsContext()
```

Set the graphics context to be used for this wxGCDC.

Note that this object takes ownership of context and will delete it when it is destroyed or when setGraphicsContext/2 is called again.

Also, unlike the constructor taking wxGraphicsContext, this method will reapply the current font, pen and brush, so that this object continues to use them, if they had been changed before (which is never the case when constructing wxGCDC directly from wxGraphicsContext).

# wxGLCanvas

Erlang module

wxGLCanvas is a class for displaying OpenGL graphics. It is always used in conjunction with wxGLContext as the context can only be made current (i.e. active for the OpenGL commands) when it is associated to a wxGLCanvas.

More precisely, you first need to create a wxGLCanvas window and then create an instance of a wxGLContext that is initialized with this wxGLCanvas and then later use either setCurrent/2 with the instance of the wxGLContext or wxGLContext:setCurrent/2 with the instance of the wxGLCanvas (which might be not the same as was used for the creation of the context) to bind the OpenGL state that is represented by the rendering context to the canvas, and then finally call swapBuffers/1 to swap the buffers of the OpenGL canvas and thus show your current output.

Please note that wxGLContext always uses physical pixels, even on the platforms where wxWindow uses logical pixels, affected by the coordinate scaling, on high DPI displays. Thus, if you want to set the OpenGL view port to the size of entire window, you must multiply the result returned by wxWindow:getClientSize/1 by wxWindow:getContentScaleFactor/1 before passing it to glViewport(). Same considerations apply to other OpenGL functions and other coordinates, notably those retrieved from wxMouseEvent in the event handlers.

Notice that versions of wxWidgets previous to 2.9 used to implicitly create a wxGLContext inside wxGLCanvas itself. This is still supported in the current version but is deprecated now and will be removed in the future, please update your code to create the rendering contexts explicitly.

To set up the attributes for the canvas (number of bits for the depth buffer, number of bits for the stencil buffer and so on) you pass them in the constructor using a wxGLAttributes (not implemented in wx) instance. You can still use the way before 3.1.0 (setting up the correct values of the attribList parameter) but it's discouraged.

Note: On those platforms which use a configure script (e.g. Linux and macOS) OpenGL support is automatically enabled if the relative headers and libraries are found. To switch it on under the other platforms (e.g. Windows), you need to edit the setup.h file and set wxUSE_GLCANVAS to 1 and then also pass USE_OPENGL=1 to the make utility. You may also need to add opengl32.lib (and glu32.lib for old OpenGL versions) to the list of the libraries your program is linked with.

See: wxGLContext, wxGLAttributes (not implemented in wx), wxGLContextAttrs (not implemented in wx)

This class is derived (and can use functions) from: wxWindow wxEvtHandler

wxWidgets docs: **wxGLCanvas**

## Data Types

```
wxGLCanvas() = wx:wx_object()
```

## Exports

```
new(Parent) -> wxGLCanvas()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxGLCanvas()
```

Types:

```
Parent = wxWindow:wxWindow()
Option =
    {id, integer()} |
    {attribList, [integer()]} |
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {name, unicode:chardata()} |
    {palette, wxPalette:wxPalette()}
```

This constructor is still available only for compatibility reasons.

Please use the constructor with `wxGLAttributes` (not implemented in wx) instead.

If `attribList` is not specified, `wxGLAttributes::PlatformDefaults()` (not implemented in wx) is used, plus some other attributes (see below).

### setCurrent(This, Context) -> boolean()

Types:

```
This = wxGLCanvas()
Context = wxGLContext:wxGLContext()
```

Makes the OpenGL state that is represented by the OpenGL rendering context `context` current, i.e.

it will be used by all subsequent OpenGL calls.

This is equivalent to `wxGLContext:setCurrent/2` called with this window as parameter.

Note: This function may only be called when the window is shown on screen, in particular it can't usually be called from the constructor as the window isn't yet shown at this moment.

Return: false if an error occurred.

### createSurface(This) -> boolean()

Types:

```
This = wxGLCanvas()
```

### isDisplaySupported(AttribList) -> boolean()

Types:

```
AttribList = [integer()]
```

Determines if a canvas having the specified attributes is available.

This only applies for visual attributes, not rendering context attributes. Please, use the new form of this method, using `wxGLAttributes` (not implemented in wx).

Return: true if attributes are supported.

### swapBuffers(This) -> boolean()

Types:

```
This = wxGLCanvas()
```

Swaps the double-buffer of this window, making the back-buffer the front-buffer and vice versa, so that the output of the previous OpenGL commands is displayed on the window.

Return: false if an error occurred.

```
destroy(This :: wxGLCanvas()) -> ok
```

Destroys the object.

# wxGLContext

Erlang module

An instance of a `wxGLContext` represents the state of an OpenGL state machine and the connection between OpenGL and the system.

The OpenGL state includes everything that can be set with the OpenGL API: colors, rendering variables, buffer data ids, texture objects, etc. It is possible to have multiple rendering contexts share buffer data and textures. This feature is specially useful when the application use multiple threads for updating data into the memory of the graphics card.

Whether one only rendering context is used with or bound to multiple output windows or if each window has its own bound context is a developer decision. It is important to take into account that GPU makers may set different pointers to the same OGL function for different contexts. The way these pointers are retrieved from the OGL driver should be used again for each new context.

Binding (making current) a rendering context with another instance of a `wxGLCanvas` however works only if the both `wxGLCanvas` instances were created with the same attributes.

OpenGL version 3 introduced a new type of specification profile, the modern core profile. The old compatibility profile maintains all legacy features. Since wxWidgets 3.1.0 you can choose the type of context and even ask for a specified OGL version number. However, its advised to use only core profile as the compatibility profile may run a bit slower.

OpenGL core profile specification defines several flags at context creation that determine not only the type of context but also some features. Some of these flags can be set in the list of attributes used at `wxGLCanvas` ctor. But since wxWidgets 3.1.0 it is strongly encouraged to use the new mechanism: setting the context attributes with a `wxGLContextAttrs` (not implemented in wx) object and the canvas attributes with a `wxGLAttributes` (not implemented in wx) object.

The best way of knowing if your OpenGL environment supports a specific type of context is creating a `wxGLContext` instance and checking `isOK/1`. If it returns false, then simply delete that instance and create a new one with other attributes.

wxHAS_OPENGL_ES is defined on platforms that only have this implementation available (e.g. the iPhone) and don't support the full specification.

See: `wxGLCanvas`, `wxGLContextAttrs` (not implemented in wx), `wxGLAttributes` (not implemented in wx)

wxWidgets docs: **wxGLContext**

## Data Types

```
wxGLContext() = wx:wx_object()
```

## Exports

```
new(Win) -> wxGLContext()
```

Types:

```
   Win = wxGLCanvas:wxGLCanvas()
```

```
new(Win, Options :: [Option]) -> wxGLContext()
```

Types:

```
Win = wxGLCanvas:wxGLCanvas()
Option = {other, wxGLContext()}
```

Constructor.

```
setCurrent(This, Win) -> boolean()
```

Types:

```
This = wxGLContext()
Win = wxGLCanvas:wxGLCanvas()
```

Makes the OpenGL state that is represented by this rendering context current with the `wxGLCanvas` `win`.

Note: `win` can be a different `wxGLCanvas` window than the one that was passed to the constructor of this rendering context. If `RC` is an object of type `wxGLContext`, the statements `"RC.SetCurrent(win);"` and `"win.SetCurrent(RC);"` are equivalent, see `wxGLCanvas:setCurrent/2`.

```
isOK(This) -> boolean()
```

Types:

```
This = wxGLContext()
```

Checks if the underlying OpenGL rendering context was correctly created by the system with the requested attributes.

If this function returns false then the `wxGLContext` object is useless and should be deleted and recreated with different attributes.

Since: 3.1.0

```
destroy(This :: wxGLContext()) -> ok
```

Destroys the object.

# wxGauge

Erlang module

A gauge is a horizontal or vertical bar which shows a quantity (often time).

`wxGauge` supports two working modes: determinate and indeterminate progress.

The first is the usual working mode (see `setValue/2` and `setRange/2`) while the second can be used when the program is doing some processing but you don't know how much progress is being done. In this case, you can periodically call the `pulse/1` function to make the progress bar switch to indeterminate mode (graphically it's usually a set of blocks which move or bounce in the bar control).

`wxGauge` supports dynamic switch between these two work modes.

There are no user commands for the gauge.

Styles

This class supports the following styles:

See: `wxSlider`, `wxScrollBar`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxGauge**

## Data Types

```
wxGauge() = wx:wx_object()
```

## Exports

```
new() -> wxGauge()
```

Default constructor.

```
new(Parent, Id, Range) -> wxGauge()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = Range = integer()
```

```
new(Parent, Id, Range, Options :: [Option]) -> wxGauge()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = Range = integer()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {validator, wx:wx_object()}
```

Constructor, creating and showing a gauge.

See: `create/5`

```
destroy(This :: wxGauge()) -> ok
```

Destructor, destroying the gauge.

```
create(This, Parent, Id, Range) -> boolean()
```

Types:

```
   This = wxGauge()
   Parent = wxWindow:wxWindow()
   Id = Range = integer()
```

```
create(This, Parent, Id, Range, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxGauge()
   Parent = wxWindow:wxWindow()
   Id = Range = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Creates the gauge for two-step construction.

See `new/4` for further details.

```
getRange(This) -> integer()
```

Types:

```
   This = wxGauge()
```

Returns the maximum position of the gauge.

See: `setRange/2`

```
getValue(This) -> integer()
```

Types:

```
   This = wxGauge()
```

Returns the current position of the gauge.

See: `setValue/2`

```
isVertical(This) -> boolean()
```

Types:

```
   This = wxGauge()
```

Returns true if the gauge is vertical (has `wxGA_VERTICAL` style) and false otherwise.

```
setRange(This, Range) -> ok
```

Types:

```
This = wxGauge()
Range = integer()
```

Sets the range (maximum value) of the gauge.

This function makes the gauge switch to determinate mode, if it's not already.

When the gauge is in indeterminate mode, under wxMSW the gauge repeatedly goes from zero to `range` and back; under other ports when in indeterminate mode, the `range` setting is ignored.

See: `getRange/1`

```
setValue(This, Pos) -> ok
```

Types:

```
This = wxGauge()
Pos = integer()
```

Sets the position of the gauge.

The `pos` must be between 0 and the gauge range as returned by `getRange/1`, inclusive.

This function makes the gauge switch to determinate mode, if it was in indeterminate mode before.

See: `getValue/1`

```
pulse(This) -> ok
```

Types:

```
This = wxGauge()
```

Switch the gauge to indeterminate mode (if required) and makes the gauge move a bit to indicate the user that some progress has been made.

Note: After calling this function the value returned by `getValue/1` is undefined and thus you need to explicitly call `setValue/2` if you want to restore the determinate mode.

# wxGenericDirCtrl

Erlang module

This control can be used to place a directory listing (with optional files) on an arbitrary window.

The control contains a `wxTreeCtrl` window representing the directory hierarchy, and optionally, a `wxChoice` window containing a list of filters.

Styles

This class supports the following styles:

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxGenericDirCtrl**

## Events

Event types emitted from this class: `dirctrl_selectionchanged`, `dirctrl_fileactivated`

## Data Types

```
wxGenericDirCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxGenericDirCtrl()
```

Default constructor.

```
new(Parent) -> wxGenericDirCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxGenericDirCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {dir, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {filter, unicode:chardata()} |
       {defaultFilter, integer()}
```

Main constructor.

```
destroy(This :: wxGenericDirCtrl()) -> ok
```

Destructor.

```
create(This, Parent) -> boolean()
```
Types:
```
   This = wxGenericDirCtrl()
   Parent = wxWindow:wxWindow()
```

```
create(This, Parent, Options :: [Option]) -> boolean()
```
Types:
```
   This = wxGenericDirCtrl()
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {dir, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {filter, unicode:chardata()} |
       {defaultFilter, integer()}
```
Create function for two-step construction.

See `new/2` for details.

```
init(This) -> ok
```
Types:
```
   This = wxGenericDirCtrl()
```
Initializes variables.

```
collapseTree(This) -> ok
```
Types:
```
   This = wxGenericDirCtrl()
```
Collapses the entire tree.

```
expandPath(This, Path) -> boolean()
```
Types:
```
   This = wxGenericDirCtrl()
   Path = unicode:chardata()
```
Tries to expand as much of the given `path` as possible, so that the filename or directory is visible in the tree control.

```
getDefaultPath(This) -> unicode:charlist()
```
Types:
```
   This = wxGenericDirCtrl()
```
Gets the default path.

```
getPath(This) -> unicode:charlist()
```
Types:

```
This = wxGenericDirCtrl()
```

Gets the currently-selected directory or filename.

```
getPath(This, ItemId) -> unicode:charlist()
```

Types:

```
This = wxGenericDirCtrl()
ItemId = integer()
```

Gets the path corresponding to the given tree control item.

Since: 2.9.5

```
getFilePath(This) -> unicode:charlist()
```

Types:

```
This = wxGenericDirCtrl()
```

Gets selected filename path only (else empty string).

This function doesn't count a directory as a selection.

```
getFilter(This) -> unicode:charlist()
```

Types:

```
This = wxGenericDirCtrl()
```

Returns the filter string.

```
getFilterIndex(This) -> integer()
```

Types:

```
This = wxGenericDirCtrl()
```

Returns the current filter index (zero-based).

```
getRootId(This) -> integer()
```

Types:

```
This = wxGenericDirCtrl()
```

Returns the root id for the tree control.

```
getTreeCtrl(This) -> wxTreeCtrl:wxTreeCtrl()
```

Types:

```
This = wxGenericDirCtrl()
```

Returns a pointer to the tree control.

```
reCreateTree(This) -> ok
```

Types:

```
This = wxGenericDirCtrl()
```

Collapse and expand the tree, thus re-creating it from scratch.

May be used to update the displayed directory content.

```
setDefaultPath(This, Path) -> ok
```

Types:

```
   This = wxGenericDirCtrl()
   Path = unicode:chardata()
```

Sets the default path.

```
setFilter(This, Filter) -> ok
```

Types:

```
   This = wxGenericDirCtrl()
   Filter = unicode:chardata()
```

Sets the filter string.

```
setFilterIndex(This, N) -> ok
```

Types:

```
   This = wxGenericDirCtrl()
   N = integer()
```

Sets the current filter index (zero-based).

```
setPath(This, Path) -> ok
```

Types:

```
   This = wxGenericDirCtrl()
   Path = unicode:chardata()
```

Sets the current path.

# wxGraphicsBrush

Erlang module

A `wxGraphicsBrush` is a native representation of a brush. The contents are specific and private to the respective renderer. Instances are ref counted and can therefore be assigned as usual. The only way to get a valid instance is via `wxGraphicsContext:createBrush/2` or `wxGraphicsRenderer:createBrush/2`.

This class is derived (and can use functions) from: `wxGraphicsObject`

wxWidgets docs: **wxGraphicsBrush**

## Data Types

```
wxGraphicsBrush() = wx:wx_object()
```

# wxGraphicsContext

Erlang module

A `wxGraphicsContext` instance is the object that is drawn upon. It is created by a renderer using `wxGraphicsRenderer:createContext/2`. This can be either directly using a renderer instance, or indirectly using the static convenience `create/1` functions of `wxGraphicsContext` that always delegate the task to the default renderer.

Remark: For some renderers (like Direct2D or Cairo) processing of drawing operations may be deferred (Direct2D render target normally builds up a batch of rendering commands but defers processing of these commands, Cairo operates on a separate surface) so to make drawing results visible you need to update the content of the context by calling `wxGraphicsContext::Flush()` (not implemented in wx) or by destroying the context.

See: `wxGraphicsRenderer:createContext/2`, `wxGCDC`, `wxDC`

This class is derived (and can use functions) from: `wxGraphicsObject`

wxWidgets docs: **wxGraphicsContext**

## Data Types

```
wxGraphicsContext() = wx:wx_object()
```

## Exports

```
destroy(This :: wxGraphicsContext()) -> ok
```

Creates a `wxGraphicsContext` from a `wxWindow`.

See: `wxGraphicsRenderer:createContext/2`

```
create() -> wxGraphicsContext()
```

Create a lightweight context that can be used only for measuring text.

```
create(WindowDC) -> wxGraphicsContext()
```

Types:

```
WindowDC =
    wxWindowDC:wxWindowDC() |
    wxWindow:wxWindow() |
    wxMemoryDC:wxMemoryDC() |
    wxImage:wxImage()
```

Creates a `wxGraphicsContext` from a `wxWindowDC`.

See: `wxGraphicsRenderer:createContext/2`

```
createPen(This, Pen) -> wxGraphicsPen:wxGraphicsPen()
```

Types:

```
This = wxGraphicsContext()
Pen = wxPen:wxPen()
```

Creates a native pen from a `wxPen`.

Prefer to use the overload taking wxGraphicsPenInfo (not implemented in wx) unless you already have a wxPen as constructing one only to pass it to this method is wasteful.

```
createBrush(This, Brush) -> wxGraphicsBrush:wxGraphicsBrush()
```

Types:

```
This = wxGraphicsContext()
Brush = wxBrush:wxBrush()
```

Creates a native brush from a wxBrush.

```
createRadialGradientBrush(This, StartX, StartY, EndX, EndY,
                          Radius, Stops) ->
                             wxGraphicsBrush:wxGraphicsBrush()
```

Types:

```
This = wxGraphicsContext()
StartX = StartY = EndX = EndY = Radius = number()
Stops = wxGraphicsGradientStops:wxGraphicsGradientStops()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
createRadialGradientBrush(This, StartX, StartY, EndX, EndY,
                          Radius, OColor, CColor) ->
                             wxGraphicsBrush:wxGraphicsBrush()
```

Types:

```
This = wxGraphicsContext()
StartX = StartY = EndX = EndY = Radius = number()
OColor = CColor = wx:wx_colour()
```

```
Creates a native brush with a radial gradient. The brush originates at (@a
startX, @a startY) and ends on a circle around (@a endX, @a endY) with the given
@a radius. The gradient may be specified either by its start and end colours @a
oColor and @a cColor or by a full set of gradient @a stops. The version taking
wxGraphicsGradientStops is new in wxWidgets 2.9.1.
```

The ability to apply a transformation matrix to the gradient was added in 3.1.3

```
createLinearGradientBrush(This, X1, Y1, X2, Y2, Stops) ->
                             wxGraphicsBrush:wxGraphicsBrush()
```

Types:

```
This = wxGraphicsContext()
X1 = Y1 = X2 = Y2 = number()
Stops = wxGraphicsGradientStops:wxGraphicsGradientStops()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
createLinearGradientBrush(This, X1, Y1, X2, Y2, C1, C2) ->
```

wxGraphicsBrush:wxGraphicsBrush()

Types:

```
This = wxGraphicsContext()
X1 = Y1 = X2 = Y2 = number()
C1 = C2 = wx:wx_colour()
```

Creates a native brush with a linear gradient. The brush starts at (@a x1, @a y1) and ends at (@a x2, @a y2). Either just the start and end gradient colours (@a c1 and @a c2) or full set of gradient @a stops can be specified. The version taking wxGraphicsGradientStops is new in wxWidgets 2.9.1.

The matrix parameter was added in wxWidgets 3.1.3

```
createFont(This, Font) -> wxGraphicsFont:wxGraphicsFont()
```

Types:

```
This = wxGraphicsContext()
Font = wxFont:wxFont()
```

```
createFont(This, SizeInPixels, Facename) ->
              wxGraphicsFont:wxGraphicsFont()
createFont(This, Font, Facename :: [Option]) ->
              wxGraphicsFont:wxGraphicsFont()
```

Types:

```
This = wxGraphicsContext()
Font = wxFont:wxFont()
Option = {col, wx:wx_colour()}
```

Creates a native graphics font from a wxFont and a text colour.

Remark: For Direct2D graphics fonts can be created from TrueType fonts only.

```
createFont(This, SizeInPixels, Facename, Options :: [Option]) ->
              wxGraphicsFont:wxGraphicsFont()
```

Types:

```
This = wxGraphicsContext()
SizeInPixels = number()
Facename = unicode:chardata()
Option = {flags, integer()} | {col, wx:wx_colour()}
```

Creates a font object with the specified attributes.

The use of overload taking wxFont is preferred, see wxGraphicsRenderer:createFont/4 for more details.

Remark: For Direct2D graphics fonts can be created from TrueType fonts only.

Since: 2.9.3

```
createMatrix(This) -> wxGraphicsMatrix:wxGraphicsMatrix()
```

Types:

```
   This = wxGraphicsContext()

createMatrix(This, Options :: [Option]) ->
                wxGraphicsMatrix:wxGraphicsMatrix()
```

Types:

```
   This = wxGraphicsContext()
   Option =
       {a, number()} |
       {b, number()} |
       {c, number()} |
       {d, number()} |
       {tx, number()} |
       {ty, number()}
```

Creates a native affine transformation matrix from the passed in values.

The default parameters result in an identity matrix.

```
createPath(This) -> wxGraphicsPath:wxGraphicsPath()
```

Types:

```
   This = wxGraphicsContext()
```

Creates a native graphics path which is initially empty.

```
clip(This, Region) -> ok
```

Types:

```
   This = wxGraphicsContext()
   Region = wxRegion:wxRegion()
```

Sets the clipping region to the intersection of the given region and the previously set clipping region.

The clipping region is an area to which drawing is restricted.

Remark:

```
clip(This, X, Y, W, H) -> ok
```

Types:

```
   This = wxGraphicsContext()
   X = Y = W = H = number()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
resetClip(This) -> ok
```

Types:

```
   This = wxGraphicsContext()
```

Resets the clipping to original shape.

```
drawBitmap(This, Bmp, X, Y, W, H) -> ok
```

Types:

```
This = wxGraphicsContext()
Bmp = wxBitmap:wxBitmap()
X = Y = W = H = number()
```

Draws the bitmap.

In case of a mono bitmap, this is treated as a mask and the current brushed is used for filling.

```
drawEllipse(This, X, Y, W, H) -> ok
```

Types:

```
This = wxGraphicsContext()
X = Y = W = H = number()
```

Draws an ellipse.

```
drawIcon(This, Icon, X, Y, W, H) -> ok
```

Types:

```
This = wxGraphicsContext()
Icon = wxIcon:wxIcon()
X = Y = W = H = number()
```

Draws the icon.

```
drawLines(This, Points) -> ok
```

Types:

```
This = wxGraphicsContext()
Points = [{X :: float(), Y :: float()}]
```

```
drawLines(This, Points, Options :: [Option]) -> ok
```

Types:

```
This = wxGraphicsContext()
Points = [{X :: float(), Y :: float()}]
Option = {fillStyle, wx:wx_enum()}
```

Draws a polygon.

```
drawPath(This, Path) -> ok
```

Types:

```
This = wxGraphicsContext()
Path = wxGraphicsPath:wxGraphicsPath()
```

```
drawPath(This, Path, Options :: [Option]) -> ok
```

Types:

```
This = wxGraphicsContext()
Path = wxGraphicsPath:wxGraphicsPath()
Option = {fillStyle, wx:wx_enum()}
```

Draws the path by first filling and then stroking.

```
drawRectangle(This, X, Y, W, H) -> ok
```

Types:

```
This = wxGraphicsContext()
X = Y = W = H = number()
```

Draws a rectangle.

```
drawRoundedRectangle(This, X, Y, W, H, Radius) -> ok
```

Types:

```
This = wxGraphicsContext()
X = Y = W = H = Radius = number()
```

Draws a rounded rectangle.

```
drawText(This, Str, X, Y) -> ok
```

Types:

```
This = wxGraphicsContext()
Str = unicode:chardata()
X = Y = number()
```

Draws text at the defined position.

```
drawText(This, Str, X, Y, Angle) -> ok
drawText(This, Str, X, Y, BackgroundBrush) -> ok
```

Types:

```
This = wxGraphicsContext()
Str = unicode:chardata()
X = Y = number()
BackgroundBrush = wxGraphicsBrush:wxGraphicsBrush()
```

Draws text at the defined position.

```
drawText(This, Str, X, Y, Angle, BackgroundBrush) -> ok
```

Types:

```
This = wxGraphicsContext()
Str = unicode:chardata()
X = Y = Angle = number()
BackgroundBrush = wxGraphicsBrush:wxGraphicsBrush()
```

Draws text at the defined position.

```
fillPath(This, Path) -> ok
```
Types:
```
   This = wxGraphicsContext()
   Path = wxGraphicsPath:wxGraphicsPath()
```

```
fillPath(This, Path, Options :: [Option]) -> ok
```
Types:
```
   This = wxGraphicsContext()
   Path = wxGraphicsPath:wxGraphicsPath()
   Option = {fillStyle, wx:wx_enum()}
```
Fills the path with the current brush.

```
strokePath(This, Path) -> ok
```
Types:
```
   This = wxGraphicsContext()
   Path = wxGraphicsPath:wxGraphicsPath()
```
Strokes along a path with the current pen.

```
getPartialTextExtents(This, Text) -> [number()]
```
Types:
```
   This = wxGraphicsContext()
   Text = unicode:chardata()
```
Fills the `widths` array with the widths from the beginning of `text` to the corresponding character of `text`.

```
getTextExtent(This, Text) -> Result
```
Types:
```
   Result =
       {Width :: number(),
        Height :: number(),
        Descent :: number(),
        ExternalLeading :: number()}
   This = wxGraphicsContext()
   Text = unicode:chardata()
```
Gets the dimensions of the string using the currently selected font.

```
rotate(This, Angle) -> ok
```
Types:
```
   This = wxGraphicsContext()
   Angle = number()
```
Rotates the current transformation matrix (in radians).

```
scale(This, XScale, YScale) -> ok
```
Types:

```
   This = wxGraphicsContext()
   XScale = YScale = number()
```

Scales the current transformation matrix.

```
translate(This, Dx, Dy) -> ok
```

Types:

```
   This = wxGraphicsContext()
   Dx = Dy = number()
```

Translates the current transformation matrix.

```
getTransform(This) -> wxGraphicsMatrix:wxGraphicsMatrix()
```

Types:

```
   This = wxGraphicsContext()
```

Gets the current transformation matrix of this context.

```
setTransform(This, Matrix) -> ok
```

Types:

```
   This = wxGraphicsContext()
   Matrix = wxGraphicsMatrix:wxGraphicsMatrix()
```

Sets the current transformation matrix of this context.

```
concatTransform(This, Matrix) -> ok
```

Types:

```
   This = wxGraphicsContext()
   Matrix = wxGraphicsMatrix:wxGraphicsMatrix()
```

Concatenates the passed in transform with the current transform of this context.

```
setBrush(This, Brush) -> ok
```

Types:

```
   This = wxGraphicsContext()
   Brush = wxGraphicsBrush:wxGraphicsBrush() | wxBrush:wxBrush()
```

Sets the brush for filling paths.

```
setFont(This, Font) -> ok
```

Types:

```
   This = wxGraphicsContext()
   Font = wxGraphicsFont:wxGraphicsFont()
```

Sets the font for drawing text.

```
setFont(This, Font, Colour) -> ok
```

Types:

```
This = wxGraphicsContext()
Font = wxFont:wxFont()
Colour = wx:wx_colour()
```

Sets the font for drawing text.

Remark: For Direct2D only TrueType fonts can be used.

```
setPen(This, Pen) -> ok
```

Types:

```
This = wxGraphicsContext()
Pen = wxPen:wxPen() | wxGraphicsPen:wxGraphicsPen()
```

Sets the pen used for stroking.

```
strokeLine(This, X1, Y1, X2, Y2) -> ok
```

Types:

```
This = wxGraphicsContext()
X1 = Y1 = X2 = Y2 = number()
```

Strokes a single line.

```
strokeLines(This, Points) -> ok
```

Types:

```
This = wxGraphicsContext()
Points = [{X :: float(), Y :: float()}]
```

Stroke lines connecting all the points.

Unlike the other overload of this function, this method draws a single polyline and not a number of disconnected lines.

# wxGraphicsFont

Erlang module

A `wxGraphicsFont` is a native representation of a font. The contents are specific and private to the respective renderer. Instances are ref counted and can therefore be assigned as usual. The only way to get a valid instance is via `wxGraphicsContext:createFont/4` or `wxGraphicsRenderer:createFont/4`.

This class is derived (and can use functions) from: `wxGraphicsObject`

wxWidgets docs: **wxGraphicsFont**

## Data Types

```
wxGraphicsFont() = wx:wx_object()
```

# wxGraphicsGradientStops

Erlang module

The stops are maintained in order of position. If two or more stops are added with the same position then the one(s) added later come later. This can be useful for producing discontinuities in the colour gradient.

Notice that this class is write-once, you can't modify the stops once they had been added.

Since: 2.9.1

wxWidgets docs: **wxGraphicsGradientStops**

## Data Types

```
wxGraphicsGradientStops() = wx:wx_object()
```

## Exports

```
new() -> wxGraphicsGradientStops()
```

```
new(Options :: [Option]) -> wxGraphicsGradientStops()
```

Types:

```
Option = {startCol, wx:wx_colour()} | {endCol, wx:wx_colour()}
```

Initializes the gradient stops with the given boundary colours.

Creates a `wxGraphicsGradientStops` instance with start colour given by `startCol` and end colour given by `endCol`.

```
item(This, N) -> {wx:wx_colour4(), float()}
```

Types:

```
This = wxGraphicsGradientStops()
N = integer()
```

Returns the stop at the given index.

```
getCount(This) -> integer()
```

Types:

```
This = wxGraphicsGradientStops()
```

Returns the number of stops.

```
setStartColour(This, Col) -> ok
```

Types:

```
This = wxGraphicsGradientStops()
Col = wx:wx_colour()
```

Set the start colour to `col`.

```
getStartColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxGraphicsGradientStops()
```

Returns the start colour.

```
setEndColour(This, Col) -> ok
```

Types:

```
This = wxGraphicsGradientStops()
Col = wx:wx_colour()
```

Set the end colour to `col`.

```
getEndColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxGraphicsGradientStops()
```

Returns the end colour.

```
add(This, Col, Pos) -> ok
```

Types:

```
This = wxGraphicsGradientStops()
Col = wx:wx_colour()
Pos = number()
```

Add a new stop.

```
destroy(This :: wxGraphicsGradientStops()) -> ok
```

Destroys the object.

# wxGraphicsMatrix

Erlang module

A `wxGraphicsMatrix` is a native representation of an affine matrix. The contents are specific and private to the respective renderer. Instances are ref counted and can therefore be assigned as usual. The only way to get a valid instance is via `wxGraphicsContext:createMatrix/2` or `wxGraphicsRenderer:createMatrix/2`.

This class is derived (and can use functions) from: `wxGraphicsObject`

wxWidgets docs: **wxGraphicsMatrix**

## Data Types

```
wxGraphicsMatrix() = wx:wx_object()
```

## Exports

```
concat(This, T) -> ok
```

Types:

```
This = T = wxGraphicsMatrix()
```

Concatenates the matrix passed with the current matrix.

The effect of the resulting transformation is to first apply the transformation in `t` to the coordinates and then apply the transformation in the current matrix to the coordinates.

```
get(This) -> Result
```

Types:

```
Result =
    {A :: number(),
     B :: number(),
     C :: number(),
     D :: number(),
     Tx :: number(),
     Ty :: number()}
This = wxGraphicsMatrix()
```

Returns the component values of the matrix via the argument pointers.

```
invert(This) -> ok
```

Types:

```
This = wxGraphicsMatrix()
```

Inverts the matrix.

```
isEqual(This, T) -> boolean()
```

Types:

```
This = T = wxGraphicsMatrix()
```

Returns true if the elements of the transformation matrix are equal.

```
isIdentity(This) -> boolean()
```

Types:

```
   This = wxGraphicsMatrix()
```

Return true if this is the identity matrix.

```
rotate(This, Angle) -> ok
```

Types:

```
   This = wxGraphicsMatrix()
   Angle = number()
```

Rotates this matrix clockwise (in radians).

```
scale(This, XScale, YScale) -> ok
```

Types:

```
   This = wxGraphicsMatrix()
   XScale = YScale = number()
```

Scales this matrix.

```
translate(This, Dx, Dy) -> ok
```

Types:

```
   This = wxGraphicsMatrix()
   Dx = Dy = number()
```

Translates this matrix.

```
set(This) -> ok
```

Types:

```
   This = wxGraphicsMatrix()
```

```
set(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxGraphicsMatrix()
   Option =
       {a, number()} |
       {b, number()} |
       {c, number()} |
       {d, number()} |
       {tx, number()} |
       {ty, number()}
```

Sets the matrix to the respective values (default values are the identity matrix).

```
transformPoint(This) -> {X :: number(), Y :: number()}
```

Types:

```
   This = wxGraphicsMatrix()
```

Applies this matrix to a point.

```
transformDistance(This) -> {Dx :: number(), Dy :: number()}
```

Types:

```
This = wxGraphicsMatrix()
```

Applies this matrix to a distance (ie.

performs all transforms except translations).

# wxGraphicsObject

Erlang module

This class is the superclass of native graphics objects like pens etc. It allows reference counting. Not instantiated by user code.

See: `wxGraphicsBrush`, `wxGraphicsPen`, `wxGraphicsMatrix`, `wxGraphicsPath`

wxWidgets docs: **wxGraphicsObject**

## Data Types

```
wxGraphicsObject() = wx:wx_object()
```

## Exports

```
destroy(This :: wxGraphicsObject()) -> ok
```

```
getRenderer(This) -> wxGraphicsRenderer:wxGraphicsRenderer()
```

Types:

```
This = wxGraphicsObject()
```

Returns the renderer that was used to create this instance, or NULL if it has not been initialized yet.

```
isNull(This) -> boolean()
```

Types:

```
This = wxGraphicsObject()
```

Return: false if this object is valid, otherwise returns true.

# wxGraphicsPath

Erlang module

A `wxGraphicsPath` is a native representation of a geometric path. The contents are specific and private to the respective renderer. Instances are reference counted and can therefore be assigned as usual. The only way to get a valid instance is by using `wxGraphicsContext:createPath/1` or `wxGraphicsRenderer:createPath/1`.

This class is derived (and can use functions) from: `wxGraphicsObject`

wxWidgets docs: **wxGraphicsPath**

## Data Types

```
wxGraphicsPath() = wx:wx_object()
```

## Exports

```
moveToPoint(This, P) -> ok
```

Types:

```
This = wxGraphicsPath()
P = {X :: float(), Y :: float()}
```

Begins a new subpath at p.

```
moveToPoint(This, X, Y) -> ok
```

Types:

```
This = wxGraphicsPath()
X = Y = number()
```

Begins a new subpath at (x,y).

```
addArc(This, C, R, StartAngle, EndAngle, Clockwise) -> ok
```

Types:

```
This = wxGraphicsPath()
C = {X :: float(), Y :: float()}
R = StartAngle = EndAngle = number()
Clockwise = boolean()
```

```
addArc(This, X, Y, R, StartAngle, EndAngle, Clockwise) -> ok
```

Types:

```
This = wxGraphicsPath()
X = Y = R = StartAngle = EndAngle = number()
Clockwise = boolean()
```

Adds an arc of a circle.

The circle is defined by the coordinates of its centre (x, y) or c and its radius r. The arc goes from the starting angle startAngle to endAngle either clockwise or counter-clockwise depending on the value of clockwise argument.

The angles are measured in radians but, contrary to the usual mathematical convention, are always clockwise from the horizontal axis.

If for clockwise arc endAngle is less than startAngle it will be progressively increased by 2*pi until it is greater than startAngle. If for counter-clockwise arc endAngle is greater than startAngle it will be progressively decreased by 2*pi until it is less than startAngle.

If there is a current point set, an initial line segment will be added to the path to connect the current point to the beginning of the arc.

```
addArcToPoint(This, X1, Y1, X2, Y2, R) -> ok
```

Types:

```
This = wxGraphicsPath()
X1 = Y1 = X2 = Y2 = R = number()
```

Adds an arc (of a circle with radius r) that is tangent to the line connecting current point and (x1, y1) and to the line connecting (x1, y1) and (x2, y2).

If the current point and the starting point of the arc are different, a straight line connecting these points is also appended. If there is no current point before the call to addArcToPoint/6 this function will behave as if preceded by a call to MoveToPoint(0, 0). After this call the current point will be at the ending point of the arc.

```
addCircle(This, X, Y, R) -> ok
```

Types:

```
This = wxGraphicsPath()
X = Y = R = number()
```

Appends a circle around (x,y) with radius r as a new closed subpath.

After this call the current point will be at (x+r, y).

```
addCurveToPoint(This, C1, C2, E) -> ok
```

Types:

```
This = wxGraphicsPath()
C1 = C2 = E = {X :: float(), Y :: float()}
```

Adds a cubic bezier curve from the current point, using two control points and an end point.

If there is no current point before the call to addCurveToPoint/7 this function will behave as if preceded by a call to MoveToPoint(c1).

```
addCurveToPoint(This, Cx1, Cy1, Cx2, Cy2, X, Y) -> ok
```

Types:

```
This = wxGraphicsPath()
Cx1 = Cy1 = Cx2 = Cy2 = X = Y = number()
```

Adds a cubic bezier curve from the current point, using two control points and an end point.

If there is no current point before the call to `addCurveToPoint/7` this function will behave as if preceded by a call to MoveToPoint(`cx1`, `cy1`).

`addEllipse(This, X, Y, W, H) -> ok`

Types:

```
This = wxGraphicsPath()
X = Y = W = H = number()
```

Appends an ellipse fitting into the passed in rectangle as a new closed subpath.

After this call the current point will be at (x+`w`, y+`h/2`).

`addLineToPoint(This, P) -> ok`

Types:

```
This = wxGraphicsPath()
P = {X :: float(), Y :: float()}
```

Adds a straight line from the current point to `p`.

If current point is not yet set before the call to `addLineToPoint/3` this function will behave as `moveToPoint/3`.

`addLineToPoint(This, X, Y) -> ok`

Types:

```
This = wxGraphicsPath()
X = Y = number()
```

Adds a straight line from the current point to (`x`,`y`).

If current point is not yet set before the call to `addLineToPoint/3` this function will behave as `moveToPoint/3`.

`addPath(This, Path) -> ok`

Types:

```
This = Path = wxGraphicsPath()
```

Adds another path onto the current path.

After this call the current point will be at the added path's current point. For Direct2D the path being appended shouldn't contain a started non-empty subpath when this function is called.

`addQuadCurveToPoint(This, Cx, Cy, X, Y) -> ok`

Types:

```
This = wxGraphicsPath()
Cx = Cy = X = Y = number()
```

Adds a quadratic bezier curve from the current point, using a control point and an end point.

If there is no current point before the call to `addQuadCurveToPoint/5` this function will behave as if preceded by a call to MoveToPoint(`cx`, `cy`).

`addRectangle(This, X, Y, W, H) -> ok`

Types:

```
This = wxGraphicsPath()
X = Y = W = H = number()
```

Appends a rectangle as a new closed subpath.

After this call the current point will be at (x, y).

```
addRoundedRectangle(This, X, Y, W, H, Radius) -> ok
```

Types:

```
This = wxGraphicsPath()
X = Y = W = H = Radius = number()
```

Appends a rounded rectangle as a new closed subpath.

If `radius` equals 0 this function will behave as `addRectangle/5`, otherwise after this call the current point will be at (x+w, y+h/2).

```
closeSubpath(This) -> ok
```

Types:

```
This = wxGraphicsPath()
```

Closes the current sub-path.

After this call the current point will be at the joined endpoint of the sub-path.

```
contains(This, C) -> boolean()
```

Types:

```
This = wxGraphicsPath()
C = {X :: float(), Y :: float()}
```

```
contains(This, X, Y) -> boolean()
contains(This, C, Y :: [Option]) -> boolean()
```

Types:

```
This = wxGraphicsPath()
C = {X :: float(), Y :: float()}
Option = {fillStyle, wx:wx_enum()}
```

Return: true if the point is within the path.

```
contains(This, X, Y, Options :: [Option]) -> boolean()
```

Types:

```
This = wxGraphicsPath()
X = Y = number()
Option = {fillStyle, wx:wx_enum()}
```

Return: true if the point is within the path.

```
getBox(This) ->
          {X :: float(), Y :: float(), W :: float(), H :: float()}
```

Types:

```
This = wxGraphicsPath()
```

Gets the bounding box enclosing all points (possibly including control points).

```
getCurrentPoint(This) -> {X :: float(), Y :: float()}
```

Types:

```
This = wxGraphicsPath()
```

Gets the last point of the current path, (0,0) if not yet set.

```
transform(This, Matrix) -> ok
```

Types:

```
This = wxGraphicsPath()
Matrix = wxGraphicsMatrix:wxGraphicsMatrix()
```

Transforms each point of this path by the matrix.

For Direct2D the current path shouldn't contain a started non-empty subpath when this function is called.

# wxGraphicsPen

Erlang module

A `wxGraphicsPen` is a native representation of a pen. The contents are specific and private to the respective renderer. Instances are ref counted and can therefore be assigned as usual. The only way to get a valid instance is via `wxGraphicsContext:createPen/2` or `wxGraphicsRenderer::CreatePen()` (not implemented in wx).

This class is derived (and can use functions) from: `wxGraphicsObject`

wxWidgets docs: **wxGraphicsPen**

## Data Types

```
wxGraphicsPen() = wx:wx_object()
```

# wxGraphicsRenderer

Erlang module

A `wxGraphicsRenderer` is the instance corresponding to the rendering engine used. There may be multiple instances on a system, if there are different rendering engines present, but there is always only one instance per engine. This instance is pointed back to by all objects created by it (`wxGraphicsContext`, `wxGraphicsPath` etc.) and can be retrieved through their `wxGraphicsObject:getRenderer/1` method. Therefore you can create an additional instance of a path etc. by calling `wxGraphicsObject:getRenderer/1` and then using the appropriate CreateXXX() function of that renderer.

wxWidgets docs: **wxGraphicsRenderer**

## Data Types

```
wxGraphicsRenderer() = wx:wx_object()
```

## Exports

```
getDefaultRenderer() -> wxGraphicsRenderer()
```

Returns the default renderer on this platform.

On macOS this is the Core Graphics (a.k.a. Quartz 2D) renderer, on MSW the GDIPlus renderer, and on GTK we currently default to the Cairo renderer.

```
createContext(This, WindowDC) ->
                 wxGraphicsContext:wxGraphicsContext()
```

Types:

```
   This = wxGraphicsRenderer()
   WindowDC =
       wxWindowDC:wxWindowDC() |
       wxWindow:wxWindow() |
       wxMemoryDC:wxMemoryDC()
```

Creates a `wxGraphicsContext` from a `wxWindowDC`.

```
createBrush(This, Brush) -> wxGraphicsBrush:wxGraphicsBrush()
```

Types:

```
   This = wxGraphicsRenderer()
   Brush = wxBrush:wxBrush()
```

Creates a native brush from a `wxBrush`.

```
createLinearGradientBrush(This, X1, Y1, X2, Y2, Stops) ->
                             wxGraphicsBrush:wxGraphicsBrush()
```

Types:

```
   This = wxGraphicsRenderer()
   X1 = Y1 = X2 = Y2 = number()
   Stops = wxGraphicsGradientStops:wxGraphicsGradientStops()
```

Creates a native brush with a linear gradient.

Stops support is new since wxWidgets 2.9.1, previously only the start and end colours could be specified.

The ability to apply a transformation matrix to the gradient was added in 3.1.3

```
createRadialGradientBrush(This, StartX, StartY, EndX, EndY,
                          Radius, Stops) ->
                             wxGraphicsBrush:wxGraphicsBrush()
```

Types:

```
   This = wxGraphicsRenderer()
   StartX = StartY = EndX = EndY = Radius = number()
   Stops = wxGraphicsGradientStops:wxGraphicsGradientStops()
```

Creates a native brush with a radial gradient.

Stops support is new since wxWidgets 2.9.1, previously only the start and end colours could be specified.

The ability to apply a transformation matrix to the gradient was added in 3.1.3

```
createFont(This, Font) -> wxGraphicsFont:wxGraphicsFont()
```

Types:

```
   This = wxGraphicsRenderer()
   Font = wxFont:wxFont()

createFont(This, SizeInPixels, Facename) ->
              wxGraphicsFont:wxGraphicsFont()
createFont(This, Font, Facename :: [Option]) ->
              wxGraphicsFont:wxGraphicsFont()
```

Types:

```
   This = wxGraphicsRenderer()
   Font = wxFont:wxFont()
   Option = {col, wx:wx_colour()}
```

Creates a native graphics font from a `wxFont` and a text colour.

```
createFont(This, SizeInPixels, Facename, Options :: [Option]) ->
              wxGraphicsFont:wxGraphicsFont()
```

Types:

```
   This = wxGraphicsRenderer()
   SizeInPixels = number()
   Facename = unicode:chardata()
   Option = {flags, integer()} | {col, wx:wx_colour()}
```

Creates a graphics font with the given characteristics.

If possible, the `createFont/4` overload taking `wxFont` should be used instead. The main advantage of this overload is that it can be used without X server connection under Unix when using Cairo.

Since: 2.9.3

```
createMatrix(This) -> wxGraphicsMatrix:wxGraphicsMatrix()
```

Types:

```
   This = wxGraphicsRenderer()
```

```
createMatrix(This, Options :: [Option]) ->
                wxGraphicsMatrix:wxGraphicsMatrix()
```

Types:

```
   This = wxGraphicsRenderer()
   Option =
       {a, number()} |
       {b, number()} |
       {c, number()} |
       {d, number()} |
       {tx, number()} |
       {ty, number()}
```

Creates a native affine transformation matrix from the passed in values.

The defaults result in an identity matrix.

```
createPath(This) -> wxGraphicsPath:wxGraphicsPath()
```

Types:

```
   This = wxGraphicsRenderer()
```

Creates a native graphics path which is initially empty.

# wxGrid

Erlang module

wxGrid and its related classes are used for displaying and editing tabular data. They provide a rich set of features for display, editing, and interacting with a variety of data sources. For simple applications, and to help you get started, wxGrid is the only class you need to refer to directly. It will set up default instances of the other classes and manage them for you. For more complex applications you can derive your own classes for custom grid views, grid data tables, cell editors and renderers. The overview_grid has examples of simple and more complex applications, explains the relationship between the various grid classes and has a summary of the keyboard shortcuts and mouse functions provided by wxGrid.

A wxGridTableBase (not implemented in wx) class holds the actual data to be displayed by a wxGrid class. One or more wxGrid classes may act as a view for one table class. The default table class is called wxGridStringTable (not implemented in wx) and holds an array of strings. An instance of such a class is created by createGrid/4.

wxGridCellRenderer is the abstract base class for rendering contents in a cell. The following renderers are predefined:

The look of a cell can be further defined using wxGridCellAttr. An object of this type may be returned by wxGridTableBase::GetAttr() (not implemented in wx).

wxGridCellEditor is the abstract base class for editing the value of a cell. The following editors are predefined:

Please see wxGridEvent, wxGridSizeEvent (not implemented in wx), wxGridRangeSelectEvent (not implemented in wx), and wxGridEditorCreatedEvent (not implemented in wx) for the documentation of all event types you can use with wxGrid.

See: **Overview grid**, wxGridUpdateLocker (not implemented in wx)

This class is derived (and can use functions) from: wxScrolledWindow wxPanel wxWindow wxEvtHandler

wxWidgets docs: **wxGrid**

## Data Types

```
wxGrid() = wx:wx_object()
```

## Exports

```
new() -> wxGrid()
```

Default constructor.

You must call Create() (not implemented in wx) to really create the grid window and also call createGrid/4 or SetTable() (not implemented in wx) or AssignTable() (not implemented in wx) to initialize its contents.

```
new(Parent, Id) -> wxGrid()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxGrid()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Constructor creating the grid window.

You must call either `createGrid/4` or `SetTable()` (not implemented in wx) or `AssignTable()` (not implemented in wx) to initialize the grid contents before using it.

```
destroy(This :: wxGrid()) -> ok
```

Destructor.

This will also destroy the associated grid table unless you passed a table object to the grid and specified that the grid should not take ownership of the table (see `SetTable()` (not implemented in wx)).

```
appendCols(This) -> boolean()
```

Types:

```
This = wxGrid()
```

```
appendCols(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxGrid()
Option = {numCols, integer()} | {updateLabels, boolean()}
```

Appends one or more new columns to the right of the grid.

The `updateLabels` argument is not used at present. If you are using a derived grid table class you will need to override `wxGridTableBase::AppendCols()` (not implemented in wx). See `insertCols/2` for further information.

Return: true on success or false if appending columns failed.

```
appendRows(This) -> boolean()
```

Types:

```
This = wxGrid()
```

```
appendRows(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxGrid()
Option = {numRows, integer()} | {updateLabels, boolean()}
```

Appends one or more new rows to the bottom of the grid.

The `updateLabels` argument is not used at present. If you are using a derived grid table class you will need to override `wxGridTableBase::AppendRows()` (not implemented in wx). See `insertRows/2` for further information.

Return: true on success or false if appending rows failed.

```
autoSize(This) -> ok
```
Types:
```
   This = wxGrid()
```
Automatically sets the height and width of all rows and columns to fit their contents.

```
autoSizeColumn(This, Col) -> ok
```
Types:
```
   This = wxGrid()
   Col = integer()
```

```
autoSizeColumn(This, Col, Options :: [Option]) -> ok
```
Types:
```
   This = wxGrid()
   Col = integer()
   Option = {setAsMin, boolean()}
```
Automatically sizes the column to fit its contents.

If `setAsMin` is true the calculated width will also be set as the minimal width for the column.

```
autoSizeColumns(This) -> ok
```
Types:
```
   This = wxGrid()
```

```
autoSizeColumns(This, Options :: [Option]) -> ok
```
Types:
```
   This = wxGrid()
   Option = {setAsMin, boolean()}
```
Automatically sizes all columns to fit their contents.

If `setAsMin` is true the calculated widths will also be set as the minimal widths for the columns.

```
autoSizeRow(This, Row) -> ok
```
Types:
```
   This = wxGrid()
   Row = integer()
```

```
autoSizeRow(This, Row, Options :: [Option]) -> ok
```
Types:
```
   This = wxGrid()
   Row = integer()
   Option = {setAsMin, boolean()}
```
Automatically sizes the row to fit its contents.

If `setAsMin` is true the calculated height will also be set as the minimal height for the row.

```
autoSizeRows(This) -> ok
```

Types:

```
This = wxGrid()
```

```
autoSizeRows(This, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
Option = {setAsMin, boolean()}
```

Automatically sizes all rows to fit their contents.

If `setAsMin` is true the calculated heights will also be set as the minimal heights for the rows.

```
beginBatch(This) -> ok
```

Types:

```
This = wxGrid()
```

Increments the grid's batch count.

When the count is greater than zero repainting of the grid is suppressed. Each call to BeginBatch must be matched by a later call to `endBatch/1`. Code that does a lot of grid modification can be enclosed between `beginBatch/1` and `endBatch/1` calls to avoid screen flicker. The final `endBatch/1` call will cause the grid to be repainted.

Notice that you should use `wxGridUpdateLocker` (not implemented in wx) which ensures that there is always a matching `endBatch/1` call for this `beginBatch/1` if possible instead of calling this method directly.

```
blockToDeviceRect(This, TopLeft, BottomRight) ->
                     {X :: integer(),
                      Y :: integer(),
                      W :: integer(),
                      H :: integer()}
```

Types:

```
This = wxGrid()
TopLeft = BottomRight = {R :: integer(), C :: integer()}
```

Convert grid cell coordinates to grid window pixel coordinates.

This function returns the rectangle that encloses the block of cells limited by `topLeft` and `bottomRight` cell in device coords and clipped to the client size of the grid window.

Since: 3.1.3 Parameter `gridWindow` has been added.

See: `cellToRect/3`

```
canDragCell(This) -> boolean()
```

Types:

```
This = wxGrid()
```

Return true if the dragging of cells is enabled or false otherwise.

```
canDragColMove(This) -> boolean()
```

Types:

```
This = wxGrid()
```

Returns true if columns can be moved by dragging with the mouse.

Columns can be moved by dragging on their labels.

```
canDragGridRowEdges(This) -> boolean()
```

Types:

```
This = wxGrid()
```

Return true if row edges inside the grid can be dragged to resize the rows.

See: `canDragGridSize/1`, `canDragRowSize/2`

Since: 3.1.4

```
canDragColSize(This, Col) -> boolean()
```

Types:

```
This = wxGrid()
Col = integer()
```

Returns true if the given column can be resized by dragging with the mouse.

This function returns true if resizing the columns interactively is globally enabled, i.e. if `disableDragColSize/1` hadn't been called, and if this column wasn't explicitly marked as non-resizable with `DisableColResize()` (not implemented in wx).

```
canDragRowSize(This, Row) -> boolean()
```

Types:

```
This = wxGrid()
Row = integer()
```

Returns true if the given row can be resized by dragging with the mouse.

This is the same as `canDragColSize/2` but for rows.

```
canDragGridSize(This) -> boolean()
```

Types:

```
This = wxGrid()
```

Return true if the dragging of grid lines to resize rows and columns is enabled or false otherwise.

```
canEnableCellControl(This) -> boolean()
```

Types:

```
This = wxGrid()
```

Returns true if the in-place edit control for the current grid cell can be used and false otherwise.

This function always returns false for the read-only cells.

```
cellToRect(This, Coords) ->
              {X :: integer(),
               Y :: integer(),
               W :: integer(),
```

```
               H :: integer()}
```

Types:

```
   This = wxGrid()
   Coords = {R :: integer(), C :: integer()}
```

Return the rectangle corresponding to the grid cell's size and position in logical coordinates.

See: `blockToDeviceRect/3`

```
cellToRect(This, Row, Col) ->
              {X :: integer(),
               Y :: integer(),
               W :: integer(),
               H :: integer()}
```

Types:

```
   This = wxGrid()
   Row = Col = integer()
```

Return the rectangle corresponding to the grid cell's size and position in logical coordinates.

See: `blockToDeviceRect/3`

```
clearGrid(This) -> ok
```

Types:

```
   This = wxGrid()
```

Clears all data in the underlying grid table and repaints the grid.

The table is not deleted by this function. If you are using a derived table class then you need to override `wxGridTableBase::Clear()` (not implemented in wx) for this function to have any effect.

```
clearSelection(This) -> ok
```

Types:

```
   This = wxGrid()
```

Deselects all cells that are currently selected.

```
createGrid(This, NumRows, NumCols) -> boolean()
```

Types:

```
   This = wxGrid()
   NumRows = NumCols = integer()
```

```
createGrid(This, NumRows, NumCols, Options :: [Option]) ->
              boolean()
```

Types:

```
   This = wxGrid()
   NumRows = NumCols = integer()
   Option = {selmode, wx:wx_enum()}
```

Creates a grid with the specified initial number of rows and columns.

Call this directly after the grid constructor. When you use this function `wxGrid` will create and manage a simple table of string values for you. All of the grid data will be stored in memory.

For applications with more complex data types or relationships, or for dealing with very large datasets, you should derive your own grid table class and pass a table object to the grid with `SetTable()` (not implemented in wx) or `AssignTable()` (not implemented in wx).

```
deleteCols(This) -> boolean()
```

Types:

```
   This = wxGrid()
```

```
deleteCols(This, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxGrid()
   Option =
       {pos, integer()} |
       {numCols, integer()} |
       {updateLabels, boolean()}
```

Deletes one or more columns from a grid starting at the specified position.

The `updateLabels` argument is not used at present. If you are using a derived grid table class you will need to override `wxGridTableBase::DeleteCols()` (not implemented in wx). See `insertCols/2` for further information.

Return: true on success or false if deleting columns failed.

```
deleteRows(This) -> boolean()
```

Types:

```
   This = wxGrid()
```

```
deleteRows(This, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxGrid()
   Option =
       {pos, integer()} |
       {numRows, integer()} |
       {updateLabels, boolean()}
```

Deletes one or more rows from a grid starting at the specified position.

The `updateLabels` argument is not used at present. If you are using a derived grid table class you will need to override `wxGridTableBase::DeleteRows()` (not implemented in wx). See `insertRows/2` for further information.

Return: true on success or false if deleting rows failed.

```
disableCellEditControl(This) -> ok
```

Types:

```
   This = wxGrid()
```

Disables in-place editing of grid cells.

Equivalent to calling EnableCellEditControl(false).

```
disableDragColSize(This) -> ok
```

Types:

```
This = wxGrid()
```

Disables column sizing by dragging with the mouse.

Equivalent to passing false to `enableDragColSize/2`.

```
disableDragGridSize(This) -> ok
```

Types:

```
This = wxGrid()
```

Disable mouse dragging of grid lines to resize rows and columns.

Equivalent to passing false to `enableDragGridSize/2`

```
disableDragRowSize(This) -> ok
```

Types:

```
This = wxGrid()
```

Disables row sizing by dragging with the mouse.

Equivalent to passing false to `enableDragRowSize/2`.

```
enableCellEditControl(This) -> ok
```

Types:

```
This = wxGrid()
```

```
enableCellEditControl(This, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
Option = {enable, boolean()}
```

Enables or disables in-place editing of grid cell data.

Enabling in-place editing generates `wxEVT_GRID_EDITOR_SHOWN` and, if it isn't vetoed by the application, shows the in-place editor which allows the user to change the cell value.

Disabling in-place editing does nothing if the in-place editor isn't currently shown, otherwise the `wxEVT_GRID_EDITOR_HIDDEN` event is generated but, unlike the "shown" event, it can't be vetoed and the in-place editor is dismissed unconditionally.

Note that it is an error to call this function if the current cell is read-only, use `canEnableCellControl/1` to check for this precondition.

```
enableDragColSize(This) -> ok
```

Types:

```
This = wxGrid()
```

```
enableDragColSize(This, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
Option = {enable, boolean()}
```

Enables or disables column sizing by dragging with the mouse.

See: `DisableColResize()` (not implemented in wx)

```
enableDragGridSize(This) -> ok
```

Types:

```
This = wxGrid()
```

```
enableDragGridSize(This, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
Option = {enable, boolean()}
```

Enables or disables row and column resizing by dragging gridlines with the mouse.

```
enableDragRowSize(This) -> ok
```

Types:

```
This = wxGrid()
```

```
enableDragRowSize(This, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
Option = {enable, boolean()}
```

Enables or disables row sizing by dragging with the mouse.

See: `DisableRowResize()` (not implemented in wx)

```
enableEditing(This, Edit) -> ok
```

Types:

```
This = wxGrid()
Edit = boolean()
```

Makes the grid globally editable or read-only.

If the edit argument is false this function sets the whole grid as read-only. If the argument is true the grid is set to the default state where cells may be editable. In the default state you can set single grid cells and whole rows and columns to be editable or read-only via `wxGridCellAttr:setReadOnly/2`. For single cells you can also use the shortcut function `setReadOnly/4`.

For more information about controlling grid cell attributes see the `wxGridCellAttr` class and the overview_grid.

```
enableGridLines(This) -> ok
```

Types:

```
This = wxGrid()
```

```
enableGridLines(This, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
Option = {enable, boolean()}
```

Turns the drawing of grid lines on or off.

```
endBatch(This) -> ok
```

Types:

```
This = wxGrid()
```

Decrements the grid's batch count.

When the count is greater than zero repainting of the grid is suppressed. Each previous call to `beginBatch/1` must be matched by a later call to `endBatch/1`. Code that does a lot of grid modification can be enclosed between `beginBatch/1` and `endBatch/1` calls to avoid screen flicker. The final `endBatch/1` will cause the grid to be repainted.

See: `wxGridUpdateLocker` (not implemented in wx)

```
fit(This) -> ok
```

Types:

```
This = wxGrid()
```

Overridden `wxWindow` method.

```
forceRefresh(This) -> ok
```

Types:

```
This = wxGrid()
```

Causes immediate repainting of the grid.

Use this instead of the usual `wxWindow:refresh/2`.

```
getBatchCount(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the number of times that `beginBatch/1` has been called without (yet) matching calls to `endBatch/1`.

While the grid's batch count is greater than zero the display will not be updated.

```
getCellAlignment(This, Row, Col) ->
                    {Horiz :: integer(), Vert :: integer()}
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Sets the arguments to the horizontal and vertical text alignment values for the grid cell at the specified location.

Horizontal alignment will be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`.

Vertical alignment will be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

```
getCellBackgroundColour(This, Row, Col) -> wx:wx_colour4()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns the background colour of the cell at the specified location.

```
getCellEditor(This, Row, Col) ->
                wxGridCellEditor:wxGridCellEditor()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns a pointer to the editor for the cell at the specified location.

See `wxGridCellEditor` and the overview_grid for more information about cell editors and renderers.

The caller must call DecRef() on the returned pointer.

```
getCellFont(This, Row, Col) -> wxFont:wxFont()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns the font for text in the grid cell at the specified location.

```
getCellRenderer(This, Row, Col) ->
                  wxGridCellRenderer:wxGridCellRenderer()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns a pointer to the renderer for the grid cell at the specified location.

See `wxGridCellRenderer` and the overview_grid for more information about cell editors and renderers.

The caller must call DecRef() on the returned pointer.

```
getCellTextColour(This, Row, Col) -> wx:wx_colour4()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns the text colour for the grid cell at the specified location.

```
getCellValue(This, Coords) -> unicode:charlist()
```

Types:

```
This = wxGrid()
Coords = {R :: integer(), C :: integer()}
```

Returns the string contained in the cell at the specified location.

For simple applications where a grid object automatically uses a default grid table of string values you use this function together with `setCellValue/4` to access cell values. For more complex applications where you have derived your

own grid table class that contains various data types (e.g. numeric, boolean or user-defined custom types) then you only use this function for those cells that contain string values.

See `wxGridTableBase::CanGetValueAs()` (not implemented in wx) and the overview_grid for more information.

```
getCellValue(This, Row, Col) -> unicode:charlist()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns the string contained in the cell at the specified location.

For simple applications where a grid object automatically uses a default grid table of string values you use this function together with `setCellValue/4` to access cell values. For more complex applications where you have derived your own grid table class that contains various data types (e.g. numeric, boolean or user-defined custom types) then you only use this function for those cells that contain string values.

See `wxGridTableBase::CanGetValueAs()` (not implemented in wx) and the overview_grid for more information.

```
getColLabelAlignment(This) ->
                        {Horiz :: integer(), Vert :: integer()}
```

Types:

```
This = wxGrid()
```

Sets the arguments to the current column label alignment values.

Horizontal alignment will be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`.

Vertical alignment will be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

```
getColLabelSize(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the current height of the column labels.

```
getColLabelValue(This, Col) -> unicode:charlist()
```

Types:

```
This = wxGrid()
Col = integer()
```

Returns the specified column label.

The default grid table class provides column labels of the form A,B...Z,AA,AB...ZZ,AAA... If you are using a custom grid table you can override `wxGridTableBase::GetColLabelValue()` (not implemented in wx) to provide your own labels.

```
getColMinimalAcceptableWidth(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the minimal width to which a column may be resized.

Use `setColMinimalAcceptableWidth/2` to change this value globally or `setColMinimalWidth/3` to do it for individual columns.

See: `getRowMinimalAcceptableHeight/1`

```
getDefaultCellAlignment(This) ->
                            {Horiz :: integer(), Vert :: integer()}
```

Types:

```
This = wxGrid()
```

Returns the default cell alignment.

Horizontal alignment will be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`.

Vertical alignment will be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

See: `setDefaultCellAlignment/3`

```
getDefaultCellBackgroundColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxGrid()
```

Returns the current default background colour for grid cells.

```
getDefaultCellFont(This) -> wxFont:wxFont()
```

Types:

```
This = wxGrid()
```

Returns the current default font for grid cell text.

```
getDefaultCellTextColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxGrid()
```

Returns the current default colour for grid cell text.

```
getDefaultColLabelSize(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the default height for column labels.

```
getDefaultColSize(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the current default width for grid columns.

```
getDefaultEditor(This) -> wxGridCellEditor:wxGridCellEditor()
```

Types:

```
This = wxGrid()
```

Returns a pointer to the current default grid cell editor.

See `wxGridCellEditor` and the overview_grid for more information about cell editors and renderers.

```
getDefaultEditorForCell(This, C) ->
                           wxGridCellEditor:wxGridCellEditor()
```

Types:

```
This = wxGrid()
C = {R :: integer(), C :: integer()}
```

Returns the default editor for the specified cell.

The base class version returns the editor appropriate for the current cell type but this method may be overridden in the derived classes to use custom editors for some cells by default.

Notice that the same may be achieved in a usually simpler way by associating a custom editor with the given cell or cells.

The caller must call DecRef() on the returned pointer.

```
getDefaultEditorForCell(This, Row, Col) ->
                           wxGridCellEditor:wxGridCellEditor()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns the default editor for the specified cell.

The base class version returns the editor appropriate for the current cell type but this method may be overridden in the derived classes to use custom editors for some cells by default.

Notice that the same may be achieved in a usually simpler way by associating a custom editor with the given cell or cells.

The caller must call DecRef() on the returned pointer.

```
getDefaultEditorForType(This, TypeName) ->
                           wxGridCellEditor:wxGridCellEditor()
```

Types:

```
This = wxGrid()
TypeName = unicode:chardata()
```

Returns the default editor for the cells containing values of the given type.

The base class version returns the editor which was associated with the specified `typeName` when it was registered `registerDataType/4` but this function may be overridden to return something different. This allows overriding an editor used for one of the standard types.

The caller must call DecRef() on the returned pointer.

```
getDefaultRenderer(This) ->
                       wxGridCellRenderer:wxGridCellRenderer()
```

Types:

```
This = wxGrid()
```

Returns a pointer to the current default grid cell renderer.

See `wxGridCellRenderer` and the overview_grid for more information about cell editors and renderers.

The caller must call DecRef() on the returned pointer.

```
getDefaultRendererForCell(This, Row, Col) ->
                             wxGridCellRenderer:wxGridCellRenderer()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns the default renderer for the given cell.

The base class version returns the renderer appropriate for the current cell type but this method may be overridden in the derived classes to use custom renderers for some cells by default.

The caller must call DecRef() on the returned pointer.

```
getDefaultRendererForType(This, TypeName) ->
                             wxGridCellRenderer:wxGridCellRenderer()
```

Types:

```
This = wxGrid()
TypeName = unicode:chardata()
```

Returns the default renderer for the cell containing values of the given type.

See: `getDefaultEditorForType/2`

```
getDefaultRowLabelSize(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the default width for the row labels.

```
getDefaultRowSize(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the current default height for grid rows.

```
getGridCursorCol(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the current grid cell column position.

See: `GetGridCursorCoords()` (not implemented in wx)

```
getGridCursorRow(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the current grid cell row position.

See: `GetGridCursorCoords()` (not implemented in wx)

```
getGridLineColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxGrid()
```

Returns the colour used for grid lines.

See: `GetDefaultGridLinePen()` (not implemented in wx)

```
gridLinesEnabled(This) -> boolean()
```

Types:

```
This = wxGrid()
```

Returns true if drawing of grid lines is turned on, false otherwise.

```
getLabelBackgroundColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxGrid()
```

Returns the colour used for the background of row and column labels.

```
getLabelFont(This) -> wxFont:wxFont()
```

Types:

```
This = wxGrid()
```

Returns the font used for row and column labels.

```
getLabelTextColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxGrid()
```

Returns the colour used for row and column label text.

```
getNumberCols(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the total number of grid columns.

This is the same as the number of columns in the underlying grid table.

```
getNumberRows(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the total number of grid rows.

This is the same as the number of rows in the underlying grid table.

```
getOrCreateCellAttr(This, Row, Col) ->
                       wxGridCellAttr:wxGridCellAttr()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns the attribute for the given cell creating one if necessary.

If the cell already has an attribute, it is returned. Otherwise a new attribute is created, associated with the cell and returned. In any case the caller must call DecRef() on the returned pointer.

Prefer to use `GetOrCreateCellAttrPtr()` (not implemented in wx) to avoid the need to call DecRef() on the returned pointer.

This function may only be called if `CanHaveAttributes()` (not implemented in wx) returns true.

```
getRowMinimalAcceptableHeight(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the minimal size to which rows can be resized.

Use `setRowMinimalAcceptableHeight/2` to change this value globally or `setRowMinimalHeight/3` to do it for individual cells.

See: `getColMinimalAcceptableWidth/1`

```
getRowLabelAlignment(This) ->
                        {Horiz :: integer(), Vert :: integer()}
```

Types:

```
This = wxGrid()
```

Returns the alignment used for row labels.

Horizontal alignment will be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`.

Vertical alignment will be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

```
getRowLabelSize(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the current width of the row labels.

```
getRowLabelValue(This, Row) -> unicode:charlist()
```

Types:

```
This = wxGrid()
Row = integer()
```

Returns the specified row label.

The default grid table class provides numeric row labels. If you are using a custom grid table you can override `wxGridTableBase::GetRowLabelValue()` (not implemented in wx) to provide your own labels.

```
getRowSize(This, Row) -> integer()
```

Types:

```
This = wxGrid()
Row = integer()
```

Returns the height of the specified row.

```
getScrollLineX(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the number of pixels per horizontal scroll increment.

The default is 15.

See: `getScrollLineY/1`, `setScrollLineX/2`, `setScrollLineY/2`

```
getScrollLineY(This) -> integer()
```

Types:

```
This = wxGrid()
```

Returns the number of pixels per vertical scroll increment.

The default is 15.

See: `getScrollLineX/1`, `setScrollLineX/2`, `setScrollLineY/2`

```
getSelectedCells(This) -> [{R :: integer(), C :: integer()}]
```

Types:

```
This = wxGrid()
```

Returns an array of individually selected cells.

Notice that this array does `not` contain all the selected cells in general as it doesn't include the cells selected as part of column, row or block selection. You must use this method, `getSelectedCols/1`, `getSelectedRows/1` and `getSelectionBlockTopLeft/1` and `getSelectionBlockBottomRight/1` methods to obtain the entire selection in general.

Please notice this behaviour is by design and is needed in order to support grids of arbitrary size (when an entire column is selected in a grid with a million of columns, we don't want to create an array with a million of entries in this function, instead it returns an empty array and `getSelectedCols/1` returns an array containing one element).

The function can be slow for the big grids, use `GetSelectedBlocks()` (not implemented in wx) in the new code.

```
getSelectedCols(This) -> [integer()]
```

Types:

```
This = wxGrid()
```

Returns an array of selected columns.

Please notice that this method alone is not sufficient to find all the selected columns as it contains only the columns which were individually selected but not those being part of the block selection or being selected in virtue of all of their cells being selected individually, please see `getSelectedCells/1` for more details.

The function can be slow for the big grids, use `GetSelectedBlocks()` (not implemented in wx) in the new code.

```
getSelectedRows(This) -> [integer()]
```

Types:

```
This = wxGrid()
```

Returns an array of selected rows.

Please notice that this method alone is not sufficient to find all the selected rows as it contains only the rows which were individually selected but not those being part of the block selection or being selected in virtue of all of their cells being selected individually, please see `getSelectedCells/1` for more details.

The function can be slow for the big grids, use `GetSelectedBlocks()` (not implemented in wx) in the new code.

```
getSelectionBackground(This) -> wx:wx_colour4()
```

Types:

```
This = wxGrid()
```

Returns the colour used for drawing the selection background.

```
getSelectionBlockTopLeft(This) ->
                            [{R :: integer(), C :: integer()}]
```

Types:

```
This = wxGrid()
```

Returns an array of the top left corners of blocks of selected cells.

Please see `getSelectedCells/1` for more information about the selection representation in `wxGrid`.

The function can be slow for the big grids, use `GetSelectedBlocks()` (not implemented in wx) in the new code.

See: `getSelectionBlockBottomRight/1`

```
getSelectionBlockBottomRight(This) ->
                                [{R :: integer(), C :: integer()}]
```

Types:

```
This = wxGrid()
```

Returns an array of the bottom right corners of blocks of selected cells.

Please see `getSelectedCells/1` for more information about the selection representation in `wxGrid`.

The function can be slow for the big grids, use `GetSelectedBlocks()` (not implemented in wx) in the new code.

See: `getSelectionBlockTopLeft/1`

```
getSelectionForeground(This) -> wx:wx_colour4()
```

Types:

```
This = wxGrid()
```

Returns the colour used for drawing the selection foreground.

```
getGridWindow(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxGrid()
```

Return the main grid window containing the grid cells.

This window is always shown.

```
getGridRowLabelWindow(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxGrid()
```

Return the row labels window.

This window is not shown if the row labels were hidden using `HideRowLabels()` (not implemented in wx).

```
getGridColLabelWindow(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxGrid()
```

Return the column labels window.

This window is not shown if the columns labels were hidden using `HideColLabels()` (not implemented in wx).

Depending on whether `UseNativeColHeader()` (not implemented in wx) was called or not this can be either a `wxHeaderCtrl` (not implemented in wx) or a plain `wxWindow`. This function returns a valid window pointer in either case but in the former case you can also use `GetGridColHeader()` (not implemented in wx) to access it if you need wxHeaderCtrl-specific functionality.

```
getGridCornerLabelWindow(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxGrid()
```

Return the window in the top left grid corner.

This window is shown only of both columns and row labels are shown and normally doesn't contain anything. Clicking on it is handled by `wxGrid` however and can be used to select the entire grid.

```
hideCellEditControl(This) -> ok
```

Types:

```
This = wxGrid()
```

Hides the in-place cell edit control.

```
insertCols(This) -> boolean()
```

Types:

```
This = wxGrid()
```

```
insertCols(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxGrid()
Option =
    {pos, integer()} |
    {numCols, integer()} |
    {updateLabels, boolean()}
```

Inserts one or more new columns into a grid with the first new column at the specified position.

Notice that inserting the columns in the grid requires grid table cooperation: when this method is called, grid object begins by requesting the underlying grid table to insert new columns. If this is successful the table notifies the grid and the grid updates the display. For a default grid (one where you have called `createGrid/4`) this process is automatic. If you are using a custom grid table (specified with `SetTable()` (not implemented in wx) or `AssignTable()`

(not implemented in wx)) then you must override `wxGridTableBase::InsertCols()` (not implemented in wx) in your derived table class.

Return: true if the columns were successfully inserted, false if an error occurred (most likely the table couldn't be updated).

```
insertRows(This) -> boolean()
```

Types:

```
   This = wxGrid()
```

```
insertRows(This, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxGrid()
   Option =
       {pos, integer()} |
       {numRows, integer()} |
       {updateLabels, boolean()}
```

Inserts one or more new rows into a grid with the first new row at the specified position.

Notice that you must implement `wxGridTableBase::InsertRows()` (not implemented in wx) if you use a grid with a custom table, please see `insertCols/2` for more information.

Return: true if the rows were successfully inserted, false if an error occurred (most likely the table couldn't be updated).

```
isCellEditControlEnabled(This) -> boolean()
```

Types:

```
   This = wxGrid()
```

Returns true if the in-place edit control is currently enabled.

```
isCurrentCellReadOnly(This) -> boolean()
```

Types:

```
   This = wxGrid()
```

Returns true if the current cell is read-only.

See: `setReadOnly/4`, `isReadOnly/3`

```
isEditable(This) -> boolean()
```

Types:

```
   This = wxGrid()
```

Returns false if the whole grid has been set as read-only or true otherwise.

See `enableEditing/2` for more information about controlling the editing status of grid cells.

```
isInSelection(This, Coords) -> boolean()
```

Types:

```
   This = wxGrid()
   Coords = {R :: integer(), C :: integer()}
```

Returns true if the given cell is selected.

```
isInSelection(This, Row, Col) -> boolean()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns true if the given cell is selected.

```
isReadOnly(This, Row, Col) -> boolean()
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Returns true if the cell at the specified location can't be edited.

See: `setReadOnly/4`, `isCurrentCellReadOnly/1`

```
isSelection(This) -> boolean()
```

Types:

```
This = wxGrid()
```

Returns true if there are currently any selected cells, rows, columns or blocks.

```
isVisible(This, Coords) -> boolean()
```

Types:

```
This = wxGrid()
Coords = {R :: integer(), C :: integer()}
```

```
isVisible(This, Row, Col) -> boolean()
isVisible(This, Coords, Col :: [Option]) -> boolean()
```

Types:

```
This = wxGrid()
Coords = {R :: integer(), C :: integer()}
Option = {wholeCellVisible, boolean()}
```

Returns true if a cell is either entirely or at least partially visible in the grid window.

By default, the cell must be entirely visible for this function to return true but if `wholeCellVisible` is false, the function returns true even if the cell is only partially visible.

```
isVisible(This, Row, Col, Options :: [Option]) -> boolean()
```

Types:

```
This = wxGrid()
Row = Col = integer()
Option = {wholeCellVisible, boolean()}
```

Returns true if a cell is either entirely or at least partially visible in the grid window.

By default, the cell must be entirely visible for this function to return true but if `wholeCellVisible` is false, the function returns true even if the cell is only partially visible.

```
makeCellVisible(This, Coords) -> ok
```

Types:

```
   This = wxGrid()
   Coords = {R :: integer(), C :: integer()}
```

Brings the specified cell into the visible grid cell area with minimal scrolling.

Does nothing if the cell is already visible.

```
makeCellVisible(This, Row, Col) -> ok
```

Types:

```
   This = wxGrid()
   Row = Col = integer()
```

Brings the specified cell into the visible grid cell area with minimal scrolling.

Does nothing if the cell is already visible.

```
moveCursorDown(This, ExpandSelection) -> boolean()
```

Types:

```
   This = wxGrid()
   ExpandSelection = boolean()
```

Moves the grid cursor down by one row.

If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

```
moveCursorLeft(This, ExpandSelection) -> boolean()
```

Types:

```
   This = wxGrid()
   ExpandSelection = boolean()
```

Moves the grid cursor left by one column.

If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

```
moveCursorRight(This, ExpandSelection) -> boolean()
```

Types:

```
   This = wxGrid()
   ExpandSelection = boolean()
```

Moves the grid cursor right by one column.

If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

```
moveCursorUp(This, ExpandSelection) -> boolean()
```

Types:

```
   This = wxGrid()
   ExpandSelection = boolean()
```

Moves the grid cursor up by one row.

If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

```
moveCursorDownBlock(This, ExpandSelection) -> boolean()
```

Types:

```
This = wxGrid()
ExpandSelection = boolean()
```

Moves the grid cursor down in the current column such that it skips to the beginning or end of a block of non-empty cells.

If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

```
moveCursorLeftBlock(This, ExpandSelection) -> boolean()
```

Types:

```
This = wxGrid()
ExpandSelection = boolean()
```

Moves the grid cursor left in the current row such that it skips to the beginning or end of a block of non-empty cells.

If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

```
moveCursorRightBlock(This, ExpandSelection) -> boolean()
```

Types:

```
This = wxGrid()
ExpandSelection = boolean()
```

Moves the grid cursor right in the current row such that it skips to the beginning or end of a block of non-empty cells.

If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

```
moveCursorUpBlock(This, ExpandSelection) -> boolean()
```

Types:

```
This = wxGrid()
ExpandSelection = boolean()
```

Moves the grid cursor up in the current column such that it skips to the beginning or end of a block of non-empty cells.

If a block of cells was previously selected it will expand if the argument is true or be cleared if the argument is false.

```
movePageDown(This) -> boolean()
```

Types:

```
This = wxGrid()
```

Moves the grid cursor down by some number of rows so that the previous bottom visible row becomes the top visible row.

```
movePageUp(This) -> boolean()
```

Types:

```
This = wxGrid()
```

Moves the grid cursor up by some number of rows so that the previous top visible row becomes the bottom visible row.

```
registerDataType(This, TypeName, Renderer, Editor) -> ok
```

Types:

```
This = wxGrid()
TypeName = unicode:chardata()
Renderer = wxGridCellRenderer:wxGridCellRenderer()
Editor = wxGridCellEditor:wxGridCellEditor()
```

Register a new data type.

The data types allow to naturally associate specific renderers and editors to the cells containing values of the given type. For example, the grid automatically registers a data type with the name `wxGRID_VALUE_STRING` which uses `wxGridCellStringRenderer` and `wxGridCellTextEditor` as its renderer and editor respectively - this is the data type used by all the cells of the default `wxGridStringTable` (not implemented in wx), so this renderer and editor are used by default for all grid cells.

However if a custom table returns `wxGRID_VALUE_BOOL` from its `wxGridTableBase::GetTypeName()` (not implemented in wx) method, then `wxGridCellBoolRenderer` and `wxGridCellBoolEditor` are used for it because the grid also registers a boolean data type with this name.

And as this mechanism is completely generic, you may register your own data types using your own custom renderers and editors. Just remember that the table must identify a cell as being of the given type for them to be used for this cell.

```
saveEditControlValue(This) -> ok
```

Types:

```
This = wxGrid()
```

Sets the value of the current grid cell to the current in-place edit control value.

This is called automatically when the grid cursor moves from the current cell to a new cell. It is also a good idea to call this function when closing a grid since any edits to the final cell location will not be saved otherwise.

```
selectAll(This) -> ok
```

Types:

```
This = wxGrid()
```

Selects all cells in the grid.

```
selectBlock(This, TopLeft, BottomRight) -> ok
```

Types:

```
This = wxGrid()
TopLeft = BottomRight = {R :: integer(), C :: integer()}
```

```
selectBlock(This, TopLeft, BottomRight, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
TopLeft = BottomRight = {R :: integer(), C :: integer()}
Option = {addToSelected, boolean()}
```

Selects a rectangular block of cells.

If `addToSelected` is false then any existing selection will be deselected; if true the column will be added to the existing selection.

```
selectBlock(This, TopRow, LeftCol, BottomRow, RightCol) -> ok
```

Types:

```
   This = wxGrid()
   TopRow = LeftCol = BottomRow = RightCol = integer()

selectBlock(This, TopRow, LeftCol, BottomRow, RightCol,
            Options :: [Option]) ->
               ok
```

Types:

```
   This = wxGrid()
   TopRow = LeftCol = BottomRow = RightCol = integer()
   Option = {addToSelected, boolean()}
```

Selects a rectangular block of cells.

If `addToSelected` is false then any existing selection will be deselected; if true the column will be added to the existing selection.

```
selectCol(This, Col) -> ok
```

Types:

```
   This = wxGrid()
   Col = integer()

selectCol(This, Col, Options :: [Option]) -> ok
```

Types:

```
   This = wxGrid()
   Col = integer()
   Option = {addToSelected, boolean()}
```

Selects the specified column.

If `addToSelected` is false then any existing selection will be deselected; if true the column will be added to the existing selection.

This method won't select anything if the current selection mode is wxGridSelectRows.

```
selectRow(This, Row) -> ok
```

Types:

```
   This = wxGrid()
   Row = integer()

selectRow(This, Row, Options :: [Option]) -> ok
```

Types:

```
   This = wxGrid()
   Row = integer()
   Option = {addToSelected, boolean()}
```

Selects the specified row.

If `addToSelected` is false then any existing selection will be deselected; if true the row will be added to the existing selection.

This method won't select anything if the current selection mode is wxGridSelectColumns.

```
setCellAlignment(This, Row, Col, Horiz, Vert) -> ok
```

Types:

```
This = wxGrid()
Row = Col = Horiz = Vert = integer()
```

Sets the horizontal and vertical alignment for grid cell text at the specified location.

Horizontal alignment should be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`.

Vertical alignment should be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

```
setCellBackgroundColour(This, Row, Col, Colour) -> ok
```

Types:

```
This = wxGrid()
Row = Col = integer()
Colour = wx:wx_colour()
```

Set the background colour for the given cell or all cells by default.

```
setCellEditor(This, Row, Col, Editor) -> ok
```

Types:

```
This = wxGrid()
Row = Col = integer()
Editor = wxGridCellEditor:wxGridCellEditor()
```

Sets the editor for the grid cell at the specified location.

The grid will take ownership of the pointer.

See `wxGridCellEditor` and the overview_grid for more information about cell editors and renderers.

```
setCellFont(This, Row, Col, Font) -> ok
```

Types:

```
This = wxGrid()
Row = Col = integer()
Font = wxFont:wxFont()
```

Sets the font for text in the grid cell at the specified location.

```
setCellRenderer(This, Row, Col, Renderer) -> ok
```

Types:

```
This = wxGrid()
Row = Col = integer()
Renderer = wxGridCellRenderer:wxGridCellRenderer()
```

Sets the renderer for the grid cell at the specified location.

The grid will take ownership of the pointer.

See `wxGridCellRenderer` and the overview_grid for more information about cell editors and renderers.

```
setCellTextColour(This, Row, Col, Colour) -> ok
```

Types:

```
This = wxGrid()
Row = Col = integer()
Colour = wx:wx_colour()
```

Sets the text colour for the given cell.

```
setCellValue(This, Coords, S) -> ok
```

Types:

```
This = wxGrid()
Coords = {R :: integer(), C :: integer()}
S = unicode:chardata()
```

Sets the string value for the cell at the specified location.

For simple applications where a grid object automatically uses a default grid table of string values you use this function together with `getCellValue/3` to access cell values. For more complex applications where you have derived your own grid table class that contains various data types (e.g. numeric, boolean or user-defined custom types) then you only use this function for those cells that contain string values.

See `wxGridTableBase::CanSetValueAs()` (not implemented in wx) and the overview_grid for more information.

```
setCellValue(This, Row, Col, S) -> ok
```

Types:

```
This = wxGrid()
Row = Col = integer()
S = unicode:chardata()
```

Sets the string value for the cell at the specified location.

For simple applications where a grid object automatically uses a default grid table of string values you use this function together with `getCellValue/3` to access cell values. For more complex applications where you have derived your own grid table class that contains various data types (e.g. numeric, boolean or user-defined custom types) then you only use this function for those cells that contain string values.

See `wxGridTableBase::CanSetValueAs()` (not implemented in wx) and the overview_grid for more information.

```
setColAttr(This, Col, Attr) -> ok
```

Types:

```
This = wxGrid()
Col = integer()
Attr = wxGridCellAttr:wxGridCellAttr()
```

Sets the cell attributes for all cells in the specified column.

For more information about controlling grid cell attributes see the `wxGridCellAttr` cell attribute class and the overview_grid.

```
setColFormatBool(This, Col) -> ok
```

Types:

```
This = wxGrid()
Col = integer()
```

Sets the specified column to display boolean values.

See: `setColFormatCustom/3`

```
setColFormatNumber(This, Col) -> ok
```

Types:

```
This = wxGrid()
Col = integer()
```

Sets the specified column to display integer values.

See: `setColFormatCustom/3`

```
setColFormatFloat(This, Col) -> ok
```

Types:

```
This = wxGrid()
Col = integer()
```

```
setColFormatFloat(This, Col, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
Col = integer()
Option = {width, integer()} | {precision, integer()}
```

Sets the specified column to display floating point values with the given width and precision.

See: `setColFormatCustom/3`

```
setColFormatCustom(This, Col, TypeName) -> ok
```

Types:

```
This = wxGrid()
Col = integer()
TypeName = unicode:chardata()
```

Sets the specified column to display data in a custom format.

This method provides an alternative to defining a custom grid table which would return `typeName` from its GetTypeName() method for the cells in this column: while it doesn't really change the type of the cells in this column, it does associate the renderer and editor used for the cells of the specified type with them.

See the overview_grid for more information on working with custom data types.

```
setColLabelAlignment(This, Horiz, Vert) -> ok
```

Types:

```
This = wxGrid()
Horiz = Vert = integer()
```

Sets the horizontal and vertical alignment of column label text.

Horizontal alignment should be one of wxALIGN_LEFT, wxALIGN_CENTRE or wxALIGN_RIGHT. Vertical alignment should be one of wxALIGN_TOP, wxALIGN_CENTRE or wxALIGN_BOTTOM.

```
setColLabelSize(This, Height) -> ok
```

Types:

```
This = wxGrid()
Height = integer()
```

Sets the height of the column labels.

If height equals to wxGRID_AUTOSIZE then height is calculated automatically so that no label is truncated. Note that this could be slow for a large table.

```
setColLabelValue(This, Col, Value) -> ok
```

Types:

```
This = wxGrid()
Col = integer()
Value = unicode:chardata()
```

Set the value for the given column label.

If you are using a custom grid table you must override wxGridTableBase::SetColLabelValue() (not implemented in wx) for this to have any effect.

```
setColMinimalWidth(This, Col, Width) -> ok
```

Types:

```
This = wxGrid()
Col = Width = integer()
```

Sets the minimal width for the specified column col.

It is usually best to call this method during grid creation as calling it later will not resize the column to the given minimal width even if it is currently narrower than it.

width must be greater than the minimal acceptable column width as returned by getColMinimalAcceptableWidth/1.

```
setColMinimalAcceptableWidth(This, Width) -> ok
```

Types:

```
This = wxGrid()
Width = integer()
```

Sets the minimal width to which the user can resize columns.

See: getColMinimalAcceptableWidth/1

```
setColSize(This, Col, Width) -> ok
```

Types:

```
This = wxGrid()
Col = Width = integer()
```

Sets the width of the specified column.

```
setDefaultCellAlignment(This, Horiz, Vert) -> ok
```

Types:

```
This = wxGrid()
Horiz = Vert = integer()
```

Sets the default horizontal and vertical alignment for grid cell text.

Horizontal alignment should be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`. Vertical alignment should be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

```
setDefaultCellBackgroundColour(This, Colour) -> ok
```

Types:

```
This = wxGrid()
Colour = wx:wx_colour()
```

Sets the default background colour for grid cells.

```
setDefaultCellFont(This, Font) -> ok
```

Types:

```
This = wxGrid()
Font = wxFont:wxFont()
```

Sets the default font to be used for grid cell text.

```
setDefaultCellTextColour(This, Colour) -> ok
```

Types:

```
This = wxGrid()
Colour = wx:wx_colour()
```

Sets the current default colour for grid cell text.

```
setDefaultEditor(This, Editor) -> ok
```

Types:

```
This = wxGrid()
Editor = wxGridCellEditor:wxGridCellEditor()
```

Sets the default editor for grid cells.

The grid will take ownership of the pointer.

See `wxGridCellEditor` and the overview_grid for more information about cell editors and renderers.

```
setDefaultRenderer(This, Renderer) -> ok
```

Types:

```
This = wxGrid()
Renderer = wxGridCellRenderer:wxGridCellRenderer()
```

Sets the default renderer for grid cells.

The grid will take ownership of the pointer.

See `wxGridCellRenderer` and the overview_grid for more information about cell editors and renderers.

```
setDefaultColSize(This, Width) -> ok
```

Types:

```
This = wxGrid()
Width = integer()
```

```
setDefaultColSize(This, Width, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
Width = integer()
Option = {resizeExistingCols, boolean()}
```

Sets the default width for columns in the grid.

This will only affect columns subsequently added to the grid unless `resizeExistingCols` is true.

If `width` is less than `getColMinimalAcceptableWidth/1`, then the minimal acceptable width is used instead of it.

```
setDefaultRowSize(This, Height) -> ok
```

Types:

```
This = wxGrid()
Height = integer()
```

```
setDefaultRowSize(This, Height, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
Height = integer()
Option = {resizeExistingRows, boolean()}
```

Sets the default height for rows in the grid.

This will only affect rows subsequently added to the grid unless `resizeExistingRows` is true.

If `height` is less than `getRowMinimalAcceptableHeight/1`, then the minimal acceptable height is used instead of it.

```
setGridCursor(This, Coords) -> ok
```

Types:

```
This = wxGrid()
Coords = {R :: integer(), C :: integer()}
```

Set the grid cursor to the specified cell.

The grid cursor indicates the current cell and can be moved by the user using the arrow keys or the mouse.

Calling this function generates a `wxEVT_GRID_SELECT_CELL` event and if the event handler vetoes this event, the cursor is not moved.

This function doesn't make the target call visible, use `GoToCell()` (not implemented in wx) to do this.

```
setGridCursor(This, Row, Col) -> ok
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

Set the grid cursor to the specified cell.

The grid cursor indicates the current cell and can be moved by the user using the arrow keys or the mouse.

Calling this function generates a `wxEVT_GRID_SELECT_CELL` event and if the event handler vetoes this event, the cursor is not moved.

This function doesn't make the target call visible, use `GoToCell()` (not implemented in wx) to do this.

```
setGridLineColour(This, Colour) -> ok
```

Types:

```
This = wxGrid()
Colour = wx:wx_colour()
```

Sets the colour used to draw grid lines.

```
setLabelBackgroundColour(This, Colour) -> ok
```

Types:

```
This = wxGrid()
Colour = wx:wx_colour()
```

Sets the background colour for row and column labels.

```
setLabelFont(This, Font) -> ok
```

Types:

```
This = wxGrid()
Font = wxFont:wxFont()
```

Sets the font for row and column labels.

```
setLabelTextColour(This, Colour) -> ok
```

Types:

```
This = wxGrid()
Colour = wx:wx_colour()
```

Sets the colour for row and column label text.

```
setMargins(This, ExtraWidth, ExtraHeight) -> ok
```

Types:

```
This = wxGrid()
ExtraWidth = ExtraHeight = integer()
```

Sets the extra margins used around the grid area.

A grid may occupy more space than needed for its data display and this function allows setting how big this extra space is

```
setReadOnly(This, Row, Col) -> ok
```

Types:

```
This = wxGrid()
Row = Col = integer()
```

```
setReadOnly(This, Row, Col, Options :: [Option]) -> ok
```

Types:

```
This = wxGrid()
Row = Col = integer()
Option = {isReadOnly, boolean()}
```

Makes the cell at the specified location read-only or editable.

See: `isReadOnly/3`

```
setRowAttr(This, Row, Attr) -> ok
```

Types:

```
This = wxGrid()
Row = integer()
Attr = wxGridCellAttr:wxGridCellAttr()
```

Sets the cell attributes for all cells in the specified row.

The grid takes ownership of the attribute pointer.

See the `wxGridCellAttr` class for more information about controlling cell attributes.

```
setRowLabelAlignment(This, Horiz, Vert) -> ok
```

Types:

```
This = wxGrid()
Horiz = Vert = integer()
```

Sets the horizontal and vertical alignment of row label text.

Horizontal alignment should be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT`. Vertical alignment should be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

```
setRowLabelSize(This, Width) -> ok
```

Types:

```
This = wxGrid()
Width = integer()
```

Sets the width of the row labels.

If `width` equals `wxGRID_AUTOSIZE` then width is calculated automatically so that no label is truncated. Note that this could be slow for a large table.

```
setRowLabelValue(This, Row, Value) -> ok
```

Types:

```
This = wxGrid()
Row = integer()
Value = unicode:chardata()
```

Sets the value for the given row label.

If you are using a derived grid table you must override `wxGridTableBase::SetRowLabelValue()` (not implemented in wx) for this to have any effect.

`setRowMinimalHeight(This, Row, Height) -> ok`

Types:

```
This = wxGrid()
Row = Height = integer()
```

Sets the minimal `height` for the specified `row`.

See `setColMinimalWidth/3` for more information.

`setRowMinimalAcceptableHeight(This, Height) -> ok`

Types:

```
This = wxGrid()
Height = integer()
```

Sets the minimal row `height` used by default.

See `setColMinimalAcceptableWidth/2` for more information.

`setRowSize(This, Row, Height) -> ok`

Types:

```
This = wxGrid()
Row = Height = integer()
```

Sets the height of the specified row.

See `setColSize/3` for more information.

`setScrollLineX(This, X) -> ok`

Types:

```
This = wxGrid()
X = integer()
```

Sets the number of pixels per horizontal scroll increment.

The default is 15.

See: `getScrollLineX/1`, `getScrollLineY/1`, `setScrollLineY/2`

`setScrollLineY(This, Y) -> ok`

Types:

```
This = wxGrid()
Y = integer()
```

Sets the number of pixels per vertical scroll increment.

The default is 15.

See: `getScrollLineX/1`, `getScrollLineY/1`, `setScrollLineX/2`

`setSelectionBackground(This, C) -> ok`

Types:

```
This = wxGrid()
C = wx:wx_colour()
```

Set the colour to be used for drawing the selection background.

`setSelectionForeground(This, C) -> ok`

Types:

```
This = wxGrid()
C = wx:wx_colour()
```

Set the colour to be used for drawing the selection foreground.

`setSelectionMode(This, Selmode) -> ok`

Types:

```
This = wxGrid()
Selmode = wx:wx_enum()
```

Set the selection behaviour of the grid.

The existing selection is converted to conform to the new mode if possible and discarded otherwise (e.g. any individual selected cells are deselected if the new mode allows only the selection of the entire rows or columns).

`showCellEditControl(This) -> ok`

Types:

```
This = wxGrid()
```

Displays the active in-place cell edit control for the current cell after it was hidden.

This method should only be called after calling `hideCellEditControl/1`, to start editing the current grid cell use `enableCellEditControl/2` instead.

`xToCol(This, X) -> integer()`

Types:

```
This = wxGrid()
X = integer()
```

`xToCol(This, X, Options :: [Option]) -> integer()`

Types:

```
This = wxGrid()
X = integer()
Option = {clipToMinMax, boolean()}
```

Returns the column at the given pixel position depending on the window.

Return: The column index or `wxNOT_FOUND`.

`xToEdgeOfCol(This, X) -> integer()`

Types:

```
This = wxGrid()
X = integer()
```

Returns the column whose right hand edge is close to the given logical `x` position.

If no column edge is near to this position wxNOT_FOUND is returned.

```
yToEdgeOfRow(This, Y) -> integer()
```

Types:

```
This = wxGrid()
Y = integer()
```

Returns the row whose bottom edge is close to the given logical y position.

If no row edge is near to this position wxNOT_FOUND is returned.

```
yToRow(This, Y) -> integer()
```

Types:

```
This = wxGrid()
Y = integer()
```

```
yToRow(This, Y, Options :: [Option]) -> integer()
```

Types:

```
This = wxGrid()
Y = integer()
Option = {clipToMinMax, boolean()}
```

Returns the grid row that corresponds to the logical y coordinate.

The parameter gridWindow is new since wxWidgets 3.1.3. If it is specified, i.e. non-NULL, only the cells of this window are considered, i.e. the function returns wxNOT_FOUND if y is out of bounds.

If gridWindow is NULL, the function returns wxNOT_FOUND only if there is no row at all at the y position.

# wxGridBagSizer

Erlang module

A `wxSizer` that can lay out items in a virtual grid like a `wxFlexGridSizer` but in this case explicit positioning of the items is allowed using `wxGBPosition` (not implemented in wx), and items can optionally span more than one row and/or column using `wxGBSpan` (not implemented in wx).

This class is derived (and can use functions) from: `wxFlexGridSizer` `wxGridSizer` `wxSizer`

wxWidgets docs: **wxGridBagSizer**

## Data Types

```
wxGridBagSizer() = wx:wx_object()
```

## Exports

```
new() -> wxGridBagSizer()
```

```
new(Options :: [Option]) -> wxGridBagSizer()
```

Types:

```
Option = {vgap, integer()} | {hgap, integer()}
```

Constructor, with optional parameters to specify the gap between the rows and columns.

```
add(This, Item) -> wxSizerItem:wxSizerItem()
```

Types:

```
This = wxGridBagSizer()
Item = wxGBSizerItem:wxGBSizerItem()
```

```
add(This, Window, Pos) -> wxSizerItem:wxSizerItem()
```

Types:

```
This = wxGridBagSizer()
Window = wxWindow:wxWindow() | wxSizer:wxSizer()
Pos = {R :: integer(), C :: integer()}
```

```
add(This, Width, Height, Pos) -> wxSizerItem:wxSizerItem()
add(This, Window, Pos, Pos :: [Option]) ->
       wxSizerItem:wxSizerItem()
```

Types:

```
This = wxGridBagSizer()
Window = wxWindow:wxWindow() | wxSizer:wxSizer()
Pos = {R :: integer(), C :: integer()}
Option =
    {span, {RS :: integer(), CS :: integer()}} |
    {flag, integer()} |
    {border, integer()} |
```

```
      {userData, wx:wx_object()}
```

Adds the given item to the given position.

Return: A valid pointer if the item was successfully placed at the given position, or NULL if something was already there.

```
add(This, Width, Height, Pos, Options :: [Option]) ->
      wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxGridBagSizer()
   Width = Height = integer()
   Pos = {R :: integer(), C :: integer()}
   Option =
       {span, {RS :: integer(), CS :: integer()}} |
       {flag, integer()} |
       {border, integer()} |
       {userData, wx:wx_object()}
```

Adds a spacer to the given position.

`width` and `height` specify the dimension of the spacer to be added.

Return: A valid pointer if the spacer was successfully placed at the given position, or NULL if something was already there.

```
calcMin(This) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxGridBagSizer()
```

Called when the managed size of the sizer is needed or when layout needs done.

```
checkForIntersection(This, Item) -> boolean()
```

Types:

```
   This = wxGridBagSizer()
   Item = wxGBSizerItem:wxGBSizerItem()
```

```
checkForIntersection(This, Pos, Span) -> boolean()
checkForIntersection(This, Item, Span :: [Option]) -> boolean()
```

Types:

```
   This = wxGridBagSizer()
   Item = wxGBSizerItem:wxGBSizerItem()
   Option = {excludeItem, wxGBSizerItem:wxGBSizerItem()}
```

Look at all items and see if any intersect (or would overlap) the given item.

Returns true if so, false if there would be no overlap. If an `excludeItem` is given then it will not be checked for intersection, for example it may be the item we are checking the position of.

```
checkForIntersection(This, Pos, Span, Options :: [Option]) ->
                        boolean()
```

Types:

```
   This = wxGridBagSizer()
   Pos = {R :: integer(), C :: integer()}
   Span = {RS :: integer(), CS :: integer()}
   Option = {excludeItem, wxGBSizerItem:wxGBSizerItem()}
```

```
findItem(This, Window) -> wxGBSizerItem:wxGBSizerItem()
```

Types:

```
   This = wxGridBagSizer()
   Window = wxWindow:wxWindow() | wxSizer:wxSizer()
```

Find the sizer item for the given window or subsizer, returns NULL if not found.

(non-recursive)

```
findItemAtPoint(This, Pt) -> wxGBSizerItem:wxGBSizerItem()
```

Types:

```
   This = wxGridBagSizer()
   Pt = {X :: integer(), Y :: integer()}
```

Return the sizer item located at the point given in pt, or NULL if there is no item at that point.

The (x,y) coordinates in `pt` correspond to the client coordinates of the window using the sizer for layout. (non-recursive)

```
findItemAtPosition(This, Pos) -> wxGBSizerItem:wxGBSizerItem()
```

Types:

```
   This = wxGridBagSizer()
   Pos = {R :: integer(), C :: integer()}
```

Return the sizer item for the given grid cell, or NULL if there is no item at that position.

(non-recursive)

```
findItemWithData(This, UserData) -> wxGBSizerItem:wxGBSizerItem()
```

Types:

```
   This = wxGridBagSizer()
   UserData = wx:wx_object()
```

Return the sizer item that has a matching user data (it only compares pointer values) or NULL if not found.

(non-recursive)

```
getCellSize(This, Row, Col) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxGridBagSizer()
   Row = Col = integer()
```

Get the size of the specified cell, including hgap and vgap.

Only valid after window layout has been performed.

```
getEmptyCellSize(This) -> {W :: integer(), H :: integer()}
```
Types:
```
   This = wxGridBagSizer()
```
Get the size used for cells in the grid with no item.

```
getItemPosition(This, Window) -> {R :: integer(), C :: integer()}
getItemPosition(This, Index) -> {R :: integer(), C :: integer()}
```
Types:
```
   This = wxGridBagSizer()
   Index = integer()
```

```
getItemSpan(This, Window) -> {RS :: integer(), CS :: integer()}
getItemSpan(This, Index) -> {RS :: integer(), CS :: integer()}
```
Types:
```
   This = wxGridBagSizer()
   Index = integer()
```

```
setEmptyCellSize(This, Sz) -> ok
```
Types:
```
   This = wxGridBagSizer()
   Sz = {W :: integer(), H :: integer()}
```
Set the size used for cells in the grid with no item.

```
setItemPosition(This, Window, Pos) -> boolean()
setItemPosition(This, Index, Pos) -> boolean()
```
Types:
```
   This = wxGridBagSizer()
   Index = integer()
   Pos = {R :: integer(), C :: integer()}
```

```
setItemSpan(This, Window, Span) -> boolean()
setItemSpan(This, Index, Span) -> boolean()
```
Types:
```
   This = wxGridBagSizer()
   Index = integer()
   Span = {RS :: integer(), CS :: integer()}
```

```
destroy(This :: wxGridBagSizer()) -> ok
```
Destroys the object.

# wxGridCellAttr

Erlang module

This class can be used to alter the cells' appearance in the grid by changing their attributes from the defaults. An object of this class may be returned by `wxGridTableBase::GetAttr()` (not implemented in wx).

Note that objects of this class are reference-counted and it's recommended to use wxGridCellAttrPtr smart pointer class when working with them to avoid memory leaks.

wxWidgets docs: **wxGridCellAttr**

## Data Types

```
wxGridCellAttr() = wx:wx_object()
```

## Exports

```
setTextColour(This, ColText) -> ok
```

Types:

```
This = wxGridCellAttr()
ColText = wx:wx_colour()
```

Sets the text colour.

```
setBackgroundColour(This, ColBack) -> ok
```

Types:

```
This = wxGridCellAttr()
ColBack = wx:wx_colour()
```

Sets the background colour.

```
setFont(This, Font) -> ok
```

Types:

```
This = wxGridCellAttr()
Font = wxFont:wxFont()
```

Sets the font.

```
setAlignment(This, HAlign, VAlign) -> ok
```

Types:

```
This = wxGridCellAttr()
HAlign = VAlign = integer()
```

Sets the alignment.

`hAlign` can be one of `wxALIGN_LEFT`, `wxALIGN_CENTRE` or `wxALIGN_RIGHT` and `vAlign` can be one of `wxALIGN_TOP`, `wxALIGN_CENTRE` or `wxALIGN_BOTTOM`.

```
setReadOnly(This) -> ok
```

Types:

```
   This = wxGridCellAttr()
```

```
setReadOnly(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxGridCellAttr()
   Option = {isReadOnly, boolean()}
```

Sets the cell as read-only.

```
setRenderer(This, Renderer) -> ok
```

Types:

```
   This = wxGridCellAttr()
   Renderer = wxGridCellRenderer:wxGridCellRenderer()
```

Sets the renderer to be used for cells with this attribute.

Takes ownership of the pointer.

```
setEditor(This, Editor) -> ok
```

Types:

```
   This = wxGridCellAttr()
   Editor = wxGridCellEditor:wxGridCellEditor()
```

Sets the editor to be used with the cells with this attribute.

```
hasTextColour(This) -> boolean()
```

Types:

```
   This = wxGridCellAttr()
```

Returns true if this attribute has a valid text colour set.

```
hasBackgroundColour(This) -> boolean()
```

Types:

```
   This = wxGridCellAttr()
```

Returns true if this attribute has a valid background colour set.

```
hasFont(This) -> boolean()
```

Types:

```
   This = wxGridCellAttr()
```

Returns true if this attribute has a valid font set.

```
hasAlignment(This) -> boolean()
```

Types:

```
   This = wxGridCellAttr()
```

Returns true if this attribute has a valid alignment set.

```
hasRenderer(This) -> boolean()
```

Types:

```
This = wxGridCellAttr()
```

Returns true if this attribute has a valid cell renderer set.

```
hasEditor(This) -> boolean()
```

Types:

```
This = wxGridCellAttr()
```

Returns true if this attribute has a valid cell editor set.

```
getTextColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxGridCellAttr()
```

Returns the text colour.

```
getBackgroundColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxGridCellAttr()
```

Returns the background colour.

```
getFont(This) -> wxFont:wxFont()
```

Types:

```
This = wxGridCellAttr()
```

Returns the font.

```
getAlignment(This) -> {HAlign :: integer(), VAlign :: integer()}
```

Types:

```
This = wxGridCellAttr()
```

Get the alignment to use for the cell with the given attribute.

If this attribute doesn't specify any alignment, the default attribute alignment is used (which can be changed using `wxGrid:setDefaultCellAlignment/3` but is left and top by default).

Notice that `hAlign` and `vAlign` values are always overwritten by this function, use `GetNonDefaultAlignment()` (not implemented in wx) if this is not desirable.

```
getRenderer(This, Grid, Row, Col) ->
               wxGridCellRenderer:wxGridCellRenderer()
```

Types:

```
This = wxGridCellAttr()
Grid = wxGrid:wxGrid()
Row = Col = integer()
```

Returns the cell renderer.

The caller is responsible for calling `DecRef()` (not implemented in wx) on the returned pointer, use `GetRendererPtr()` (not implemented in wx) to do it automatically.

```
getEditor(This, Grid, Row, Col) ->
```

```
wxGridCellEditor:wxGridCellEditor()
```

Types:

```
This = wxGridCellAttr()
Grid = wxGrid:wxGrid()
Row = Col = integer()
```

Returns the cell editor.

The caller is responsible for calling `DecRef()` (not implemented in wx) on the returned pointer, use `GetEditorPtr()` (not implemented in wx) to do it automatically.

```
isReadOnly(This) -> boolean()
```

Types:

```
This = wxGridCellAttr()
```

Returns true if this cell is set as read-only.

```
setDefAttr(This, DefAttr) -> ok
```

Types:

```
This = DefAttr = wxGridCellAttr()
```

# wxGridCellBoolEditor

Erlang module

Grid cell editor for boolean data.

See: `wxGridCellEditor`, `wxGridCellAutoWrapStringEditor` (not implemented in wx), `wxGridCellChoiceEditor`, `wxGridCellEnumEditor` (not implemented in wx), `wxGridCellFloatEditor`, `wxGridCellNumberEditor`, `wxGridCellTextEditor`, `wxGridCellDateEditor` (not implemented in wx)

This class is derived (and can use functions) from: `wxGridCellEditor`

wxWidgets docs: **wxGridCellBoolEditor**

## Data Types

```
wxGridCellBoolEditor() = wx:wx_object()
```

## Exports

```
new() -> wxGridCellBoolEditor()
```

Default constructor.

```
isTrueValue(Value) -> boolean()
```

Types:

```
Value = unicode:chardata()
```

Returns true if the given `value` is equal to the string representation of the truth value we currently use (see `useStringValues/1`).

```
useStringValues() -> ok

useStringValues(Options :: [Option]) -> ok
```

Types:

```
Option =
    {valueTrue, unicode:chardata()} |
    {valueFalse, unicode:chardata()}
```

This method allows you to customize the values returned by `wxGridCellNumberEditor:getValue/1` for the cell using this editor.

By default, the default values of the arguments are used, i.e. `"1"` is returned if the cell is checked and an empty string otherwise.

```
destroy(This :: wxGridCellBoolEditor()) -> ok
```

Destroys the object.

# wxGridCellBoolRenderer

Erlang module

This class may be used to format boolean data in a cell.

See: `wxGridCellRenderer`, `wxGridCellAutoWrapStringRenderer` (not implemented in wx), `wxGridCellDateTimeRenderer` (not implemented in wx), `wxGridCellEnumRenderer` (not implemented in wx), `wxGridCellFloatRenderer`, `wxGridCellNumberRenderer`, `wxGridCellStringRenderer`

This class is derived (and can use functions) from: `wxGridCellRenderer`

wxWidgets docs: **wxGridCellBoolRenderer**

## Data Types

```
wxGridCellBoolRenderer() = wx:wx_object()
```

## Exports

```
new() -> wxGridCellBoolRenderer()
```

```
destroy(This :: wxGridCellBoolRenderer()) -> ok
```

Destroys the object.

# wxGridCellChoiceEditor

Erlang module

Grid cell editor for string data providing the user a choice from a list of strings.

See: `wxGridCellEditor`, `wxGridCellAutoWrapStringEditor` (not implemented in wx), `wxGridCellBoolEditor`, `wxGridCellEnumEditor` (not implemented in wx), `wxGridCellFloatEditor`, `wxGridCellNumberEditor`, `wxGridCellTextEditor`, `wxGridCellDateEditor` (not implemented in wx)

This class is derived (and can use functions) from: `wxGridCellEditor`

wxWidgets docs: **wxGridCellChoiceEditor**

## Data Types

```
wxGridCellChoiceEditor() = wx:wx_object()
```

## Exports

```
new(Choices) -> wxGridCellChoiceEditor()
```

Types:

```
Choices = [unicode:chardata()]
```

```
new(Choices, Options :: [Option]) -> wxGridCellChoiceEditor()
```

Types:

```
Choices = [unicode:chardata()]
Option = {allowOthers, boolean()}
```

Choice cell renderer ctor.

```
setParameters(This, Params) -> ok
```

Types:

```
This = wxGridCellChoiceEditor()
Params = unicode:chardata()
```

Parameters string format is "item1[,item2[...,itemN]]".

This method can be called before the editor is used for the first time, or later, in which case it replaces the previously specified strings with the new ones.

```
destroy(This :: wxGridCellChoiceEditor()) -> ok
```

Destroys the object.

# wxGridCellEditor

Erlang module

This class is responsible for providing and manipulating the in-place edit controls for the grid. Instances of `wxGridCellEditor` (actually, instances of derived classes since it is an abstract class) can be associated with the cell attributes for individual cells, rows, columns, or even for the entire grid.

Normally `wxGridCellEditor` shows some UI control allowing the user to edit the cell, but starting with wxWidgets 3.1.4 it's also possible to define "activatable" cell editors, that change the value of the cell directly when it's activated (typically by pressing Space key or clicking on it), see `TryActivate()` (not implemented in wx) method. Note that when implementing an editor which is always activatable, i.e. never shows any in-place editor, it is more convenient to derive its class from `wxGridCellActivatableEditor` (not implemented in wx) than from `wxGridCellEditor` itself.

See: `wxGridCellAutoWrapStringEditor` (not implemented in wx), `wxGridCellBoolEditor`, `wxGridCellChoiceEditor`, `wxGridCellEnumEditor` (not implemented in wx), `wxGridCellFloatEditor`, `wxGridCellNumberEditor`, `wxGridCellTextEditor`, `wxGridCellDateEditor` (not implemented in wx)

wxWidgets docs: **wxGridCellEditor**

## Data Types

```
wxGridCellEditor() = wx:wx_object()
```

## Exports

```
create(This, Parent, Id, EvtHandler) -> ok
```

Types:

```
This = wxGridCellEditor()
Parent = wxWindow:wxWindow()
Id = integer()
EvtHandler = wxEvtHandler:wxEvtHandler()
```

Creates the actual edit control.

```
isCreated(This) -> boolean()
```

Types:

```
This = wxGridCellEditor()
```

Returns true if the edit control has been created.

```
setSize(This, Rect) -> ok
```

Types:

```
This = wxGridCellEditor()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
```

```
        H :: integer()}
```

Size and position the edit control.

```
show(This, Show) -> ok
```

Types:

```
   This = wxGridCellEditor()
   Show = boolean()
```

```
show(This, Show, Options :: [Option]) -> ok
```

Types:

```
   This = wxGridCellEditor()
   Show = boolean()
   Option = {attr, wxGridCellAttr:wxGridCellAttr()}
```

Show or hide the edit control, use the specified attributes to set colours/fonts for it.

```
reset(This) -> ok
```

Types:

```
   This = wxGridCellEditor()
```

Reset the value in the control back to its starting value.

```
startingKey(This, Event) -> ok
```

Types:

```
   This = wxGridCellEditor()
   Event = wxKeyEvent:wxKeyEvent()
```

If the editor is enabled by pressing keys on the grid, this will be called to let the editor do something about that first key if desired.

```
startingClick(This) -> ok
```

Types:

```
   This = wxGridCellEditor()
```

If the editor is enabled by clicking on the cell, this method will be called.

```
handleReturn(This, Event) -> ok
```

Types:

```
   This = wxGridCellEditor()
   Event = wxKeyEvent:wxKeyEvent()
```

Some types of controls on some platforms may need some help with the Return key.

# wxGridCellFloatEditor

Erlang module

The editor for floating point numbers data.

See: `wxGridCellEditor`, `wxGridCellAutoWrapStringEditor` (not implemented in wx), `wxGridCellBoolEditor`, `wxGridCellChoiceEditor`, `wxGridCellEnumEditor` (not implemented in wx), `wxGridCellNumberEditor`, `wxGridCellTextEditor`, `wxGridCellDateEditor` (not implemented in wx)

This class is derived (and can use functions) from: `wxGridCellEditor`

wxWidgets docs: **wxGridCellFloatEditor**

## Data Types

```
wxGridCellFloatEditor() = wx:wx_object()
```

## Exports

```
new() -> wxGridCellFloatEditor()

new(Options :: [Option]) -> wxGridCellFloatEditor()
```

Types:

```
   Option =
       {width, integer()} |
       {precision, integer()} |
       {format, integer()}
```

Float cell editor ctor.

```
setParameters(This, Params) -> ok
```

Types:

```
   This = wxGridCellFloatEditor()
   Params = unicode:chardata()
```

The parameters string format is "width[,precision[,format]]" where `format` should be chosen between f|e|g|E|G (f is used by default)

```
destroy(This :: wxGridCellFloatEditor()) -> ok
```

Destroys the object.

# wxGridCellFloatRenderer

Erlang module

This class may be used to format floating point data in a cell.

See: `wxGridCellRenderer`, `wxGridCellAutoWrapStringRenderer` (not implemented in wx), `wxGridCellBoolRenderer`, `wxGridCellDateTimeRenderer` (not implemented in wx), `wxGridCellEnumRenderer` (not implemented in wx), `wxGridCellNumberRenderer`, `wxGridCellStringRenderer`

This class is derived (and can use functions) from: `wxGridCellStringRenderer` `wxGridCellRenderer`

wxWidgets docs: **wxGridCellFloatRenderer**

## Data Types

```
wxGridCellFloatRenderer() = wx:wx_object()
```

## Exports

```
new() -> wxGridCellFloatRenderer()

new(Options :: [Option]) -> wxGridCellFloatRenderer()
```

Types:

```
   Option =
       {width, integer()} |
       {precision, integer()} |
       {format, integer()}
```

Float cell renderer ctor.

```
getPrecision(This) -> integer()
```

Types:

```
   This = wxGridCellFloatRenderer()
```

Returns the precision.

```
getWidth(This) -> integer()
```

Types:

```
   This = wxGridCellFloatRenderer()
```

Returns the width.

```
setParameters(This, Params) -> ok
```

Types:

```
   This = wxGridCellFloatRenderer()
   Params = unicode:chardata()
```

The parameters string format is "width[,precision[,format]]" where `format` should be chosen between f|e|g|E|G (f is used by default)

```
setPrecision(This, Precision) -> ok
```

Types:

```
   This = wxGridCellFloatRenderer()
   Precision = integer()
```

Sets the precision.

```
setWidth(This, Width) -> ok
```

Types:

```
   This = wxGridCellFloatRenderer()
   Width = integer()
```

Sets the width.

```
destroy(This :: wxGridCellFloatRenderer()) -> ok
```

Destroys the object.

# wxGridCellNumberEditor

Erlang module

Grid cell editor for numeric integer data.

See: `wxGridCellEditor`, `wxGridCellAutoWrapStringEditor` (not implemented in wx), `wxGridCellBoolEditor`, `wxGridCellChoiceEditor`, `wxGridCellEnumEditor` (not implemented in wx), `wxGridCellFloatEditor`, `wxGridCellTextEditor`, `wxGridCellDateEditor` (not implemented in wx)

This class is derived (and can use functions) from: `wxGridCellTextEditor` `wxGridCellEditor`

wxWidgets docs: **wxGridCellNumberEditor**

## Data Types

```
wxGridCellNumberEditor() = wx:wx_object()
```

## Exports

```
new() -> wxGridCellNumberEditor()
```

```
new(Options :: [Option]) -> wxGridCellNumberEditor()
```

Types:

```
Option = {min, integer()} | {max, integer()}
```

Allows you to specify the range for acceptable data.

Values equal to -1 for both `min` and `max` indicate that no range checking should be done.

```
getValue(This) -> unicode:charlist()
```

Types:

```
This = wxGridCellNumberEditor()
```

Returns the value currently in the editor control.

```
setParameters(This, Params) -> ok
```

Types:

```
This = wxGridCellNumberEditor()
Params = unicode:chardata()
```

Parameters string format is "min,max".

```
destroy(This :: wxGridCellNumberEditor()) -> ok
```

Destroys the object.

# wxGridCellNumberRenderer

Erlang module

This class may be used to format integer data in a cell.

See: `wxGridCellRenderer`, `wxGridCellAutoWrapStringRenderer` (not implemented in wx), `wxGridCellBoolRenderer`, `wxGridCellDateTimeRenderer` (not implemented in wx), `wxGridCellEnumRenderer` (not implemented in wx), `wxGridCellFloatRenderer`, `wxGridCellStringRenderer`

This class is derived (and can use functions) from: `wxGridCellStringRenderer` `wxGridCellRenderer`

wxWidgets docs: **wxGridCellNumberRenderer**

## Data Types

```
wxGridCellNumberRenderer() = wx:wx_object()
```

## Exports

```
new() -> wxGridCellNumberRenderer()
```

Default constructor.

```
destroy(This :: wxGridCellNumberRenderer()) -> ok
```

Destroys the object.

# wxGridCellRenderer

Erlang module

This class is responsible for actually drawing the cell in the grid. You may pass it to the `wxGridCellAttr` (below) to change the format of one given cell or to `wxGrid:setDefaultRenderer/2` to change the view of all cells. This is an abstract class, and you will normally use one of the predefined derived classes or derive your own class from it.

See: `wxGridCellAutoWrapStringRenderer` (not implemented in wx), `wxGridCellBoolRenderer`, `wxGridCellDateTimeRenderer` (not implemented in wx), `wxGridCellEnumRenderer` (not implemented in wx), `wxGridCellFloatRenderer`, `wxGridCellNumberRenderer`, `wxGridCellStringRenderer`

wxWidgets docs: **wxGridCellRenderer**

## Data Types

```
wxGridCellRenderer() = wx:wx_object()
```

## Exports

```
draw(This, Grid, Attr, Dc, Rect, Row, Col, IsSelected) -> ok
```

Types:

```
This = wxGridCellRenderer()
Grid = wxGrid:wxGrid()
Attr = wxGridCellAttr:wxGridCellAttr()
Dc = wxDC:wxDC()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
Row = Col = integer()
IsSelected = boolean()
```

Draw the given cell on the provided DC inside the given rectangle using the style specified by the attribute and the default or selected state corresponding to the isSelected value.

This pure virtual function has a default implementation which will prepare the DC using the given attribute: it will draw the rectangle with the background colour from attr and set the text colour and font.

```
getBestSize(This, Grid, Attr, Dc, Row, Col) ->
               {W :: integer(), H :: integer()}
```

Types:

```
This = wxGridCellRenderer()
Grid = wxGrid:wxGrid()
Attr = wxGridCellAttr:wxGridCellAttr()
Dc = wxDC:wxDC()
Row = Col = integer()
```

Get the preferred size of the cell for its contents.

This method must be overridden in the derived classes to return the minimal fitting size for displaying the content of the given grid cell.

See: `GetBestHeight()` (not implemented in wx), `GetBestWidth()` (not implemented in wx)

# wxGridCellStringRenderer

Erlang module

This class may be used to format string data in a cell; it is the default for string cells.

See: `wxGridCellRenderer`, `wxGridCellAutoWrapStringRenderer` (not implemented in wx), `wxGridCellBoolRenderer`, `wxGridCellDateTimeRenderer` (not implemented in wx), `wxGridCellEnumRenderer` (not implemented in wx), `wxGridCellFloatRenderer`, `wxGridCellNumberRenderer`

This class is derived (and can use functions) from: `wxGridCellRenderer`

wxWidgets docs: **wxGridCellStringRenderer**

## Data Types

```
wxGridCellStringRenderer() = wx:wx_object()
```

## Exports

```
new() -> wxGridCellStringRenderer()
```

```
destroy(This :: wxGridCellStringRenderer()) -> ok
```

Destroys the object.

# wxGridCellTextEditor

Erlang module

Grid cell editor for string/text data.

See: `wxGridCellEditor`, `wxGridCellAutoWrapStringEditor` (not implemented in wx), `wxGridCellBoolEditor`, `wxGridCellChoiceEditor`, `wxGridCellEnumEditor` (not implemented in wx), `wxGridCellFloatEditor`, `wxGridCellNumberEditor`, `wxGridCellDateEditor` (not implemented in wx)

This class is derived (and can use functions) from: `wxGridCellEditor`

wxWidgets docs: **wxGridCellTextEditor**

## Data Types

```
wxGridCellTextEditor() = wx:wx_object()
```

## Exports

```
new() -> wxGridCellTextEditor()
```

```
new(Options :: [Option]) -> wxGridCellTextEditor()
```

Types:

```
Option = {maxChars, integer()}
```

Text cell editor constructor.

```
setParameters(This, Params) -> ok
```

Types:

```
This = wxGridCellTextEditor()
Params = unicode:chardata()
```

The parameters string format is "n" where n is a number representing the maximum width.

```
destroy(This :: wxGridCellTextEditor()) -> ok
```

Destroys the object.

# wxGridEvent

Erlang module

This event class contains information about various grid events.

Notice that all grid event table macros are available in two versions: `EVT_GRID_XXX` and `EVT_GRID_CMD_XXX`. The only difference between the two is that the former doesn't allow to specify the grid window identifier and so takes a single parameter, the event handler, but is not suitable if there is more than one grid control in the window where the event table is used (as it would catch the events from all the grids). The version with `CMD` takes the id as first argument and the event handler as the second one and so can be used with multiple grids as well. Otherwise there are no difference between the two and only the versions without the id are documented below for brevity.

This class is derived (and can use functions) from: `wxNotifyEvent` `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxGridEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxGridEventType` to subscribe to events of this type.

## Data Types

```
wxGridEvent() = wx:wx_object()
wxGrid() =
    #wxGrid{type = wxGridEvent:wxGridEventType(),
            row = integer(),
            col = integer(),
            pos = {X :: integer(), Y :: integer()},
            selecting = boolean(),
            control = boolean(),
            meta = boolean(),
            shift = boolean(),
            alt = boolean()}
wxGridEventType() =
    grid_cell_left_click | grid_cell_right_click |
    grid_cell_left_dclick | grid_cell_right_dclick |
    grid_label_left_click | grid_label_right_click |
    grid_label_left_dclick | grid_label_right_dclick |
    grid_cell_changed | grid_select_cell | grid_cell_begin_drag |
    grid_editor_shown | grid_editor_hidden | grid_col_move |
    grid_col_sort | grid_tabbing
```

## Exports

```
altDown(This) -> boolean()
```

Types:

```
   This = wxGridEvent()
```

Returns true if the Alt key was down at the time of the event.

```
controlDown(This) -> boolean()
```

Types:

```
This = wxGridEvent()
```

Returns true if the Control key was down at the time of the event.

```
getCol(This) -> integer()
```

Types:

```
This = wxGridEvent()
```

Column at which the event occurred.

Notice that for a `wxEVT_GRID_SELECT_CELL` event this column is the column of the newly selected cell while the previously selected cell can be retrieved using `wxGrid:getGridCursorCol/1`.

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxGridEvent()
```

Position in pixels at which the event occurred.

```
getRow(This) -> integer()
```

Types:

```
This = wxGridEvent()
```

Row at which the event occurred.

Notice that for a `wxEVT_GRID_SELECT_CELL` event this row is the row of the newly selected cell while the previously selected cell can be retrieved using `wxGrid:getGridCursorRow/1`.

```
metaDown(This) -> boolean()
```

Types:

```
This = wxGridEvent()
```

Returns true if the Meta key was down at the time of the event.

```
selecting(This) -> boolean()
```

Types:

```
This = wxGridEvent()
```

Returns true if the user is selecting grid cells, or false if deselecting.

```
shiftDown(This) -> boolean()
```

Types:

```
This = wxGridEvent()
```

Returns true if the Shift key was down at the time of the event.

# wxGridSizer

Erlang module

A grid sizer is a sizer which lays out its children in a two-dimensional table with all table fields having the same size, i.e. the width of each field is the width of the widest child, the height of each field is the height of the tallest child.

See: `wxSizer`, **Overview sizer**

This class is derived (and can use functions) from: `wxSizer`

wxWidgets docs: **wxGridSizer**

## Data Types

```
wxGridSizer() = wx:wx_object()
```

## Exports

```
new(Cols) -> wxGridSizer()
```

Types:

```
   Cols = integer()
```

```
new(Cols, Options :: [Option]) -> wxGridSizer()
```

Types:

```
   Cols = integer()
   Option = {gap, {W :: integer(), H :: integer()}}
```

```
new(Cols, Vgap, Hgap) -> wxGridSizer()
new(Rows, Cols, Gap) -> wxGridSizer()
```

Types:

```
   Rows = Cols = integer()
   Gap = {W :: integer(), H :: integer()}
```

```
new(Rows, Cols, Vgap, Hgap) -> wxGridSizer()
```

Types:

```
   Rows = Cols = Vgap = Hgap = integer()
```

```
getCols(This) -> integer()
```

Types:

```
   This = wxGridSizer()
```

Returns the number of columns that has been specified for the sizer.

Returns zero if the sizer is automatically adjusting the number of columns depending on number of its children. To get the effective number of columns or rows being currently used, see `GetEffectiveColsCount()` (not implemented in wx)

```
getHGap(This) -> integer()
```

Types:

```
This = wxGridSizer()
```

Returns the horizontal gap (in pixels) between cells in the sizer.

```
getRows(This) -> integer()
```

Types:

```
This = wxGridSizer()
```

Returns the number of rows that has been specified for the sizer.

Returns zero if the sizer is automatically adjusting the number of rows depending on number of its children. To get the effective number of columns or rows being currently used, see `GetEffectiveRowsCount()` (not implemented in wx).

```
getVGap(This) -> integer()
```

Types:

```
This = wxGridSizer()
```

Returns the vertical gap (in pixels) between the cells in the sizer.

```
setCols(This, Cols) -> ok
```

Types:

```
This = wxGridSizer()
Cols = integer()
```

Sets the number of columns in the sizer.

```
setHGap(This, Gap) -> ok
```

Types:

```
This = wxGridSizer()
Gap = integer()
```

Sets the horizontal gap (in pixels) between cells in the sizer.

```
setRows(This, Rows) -> ok
```

Types:

```
This = wxGridSizer()
Rows = integer()
```

Sets the number of rows in the sizer.

```
setVGap(This, Gap) -> ok
```

Types:

```
This = wxGridSizer()
Gap = integer()
```

Sets the vertical gap (in pixels) between the cells in the sizer.

```
destroy(This :: wxGridSizer()) -> ok
```

Destroys the object.

# wxHelpEvent

Erlang module

A help event is sent when the user has requested context-sensitive help. This can either be caused by the application requesting context-sensitive help mode via `wxContextHelp` (not implemented in wx), or (on MS Windows) by the system generating a WM_HELP message when the user pressed F1 or clicked on the query button in a dialog caption.

A help event is sent to the window that the user clicked on, and is propagated up the window hierarchy until the event is processed or there are no more event handlers.

The application should call `wxEvent:getId/1` to check the identity of the clicked-on window, and then either show some suitable help or call `wxEvent:skip/2` if the identifier is unrecognised.

Calling Skip is important because it allows wxWidgets to generate further events for ancestors of the clicked-on window. Otherwise it would be impossible to show help for container windows, since processing would stop after the first window found.

See: `wxContextHelp` (not implemented in wx), `wxDialog`, **Overview events**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxHelpEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxHelpEventType` to subscribe to events of this type.

## Data Types

```
wxHelpEvent() = wx:wx_object()
wxHelp() = #wxHelp{type = wxHelpEvent:wxHelpEventType()}
wxHelpEventType() = help | detailed_help
```

## Exports

```
getOrigin(This) -> wx:wx_enum()
```

Types:

```
This = wxHelpEvent()
```

Returns the origin of the help event which is one of the `wxHelpEvent::Origin` (not implemented in wx) values.

The application may handle events generated using the keyboard or mouse differently, e.g. by using `wx_misc:getMousePosition/0` for the mouse events.

See: `setOrigin/2`

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxHelpEvent()
```

Returns the left-click position of the mouse, in screen coordinates.

This allows the application to position the help appropriately.

```
setOrigin(This, Origin) -> ok
```

Types:

```
This = wxHelpEvent()
Origin = wx:wx_enum()
```

Set the help event origin, only used internally by wxWidgets normally.

See: `getOrigin/1`

```
setPosition(This, Pt) -> ok
```

Types:

```
This = wxHelpEvent()
Pt = {X :: integer(), Y :: integer()}
```

Sets the left-click position of the mouse, in screen coordinates.

# wxHtmlEasyPrinting

Erlang module

This class provides very simple interface to printing architecture. It allows you to print HTML documents using only a few commands.

Note: Do not create this class on the stack only. You should create an instance on app startup and use this instance for all printing operations. The reason is that this class stores various settings in it.

wxWidgets docs: **wxHtmlEasyPrinting**

## Data Types

```
wxHtmlEasyPrinting() = wx:wx_object()
```

## Exports

```
new() -> wxHtmlEasyPrinting()
```

```
new(Options :: [Option]) -> wxHtmlEasyPrinting()
```

Types:

```
   Option =
       {name, unicode:chardata()} |
       {parentWindow, wxWindow:wxWindow()}
```

Constructor.

```
getPrintData(This) -> wxPrintData:wxPrintData()
```

Types:

```
   This = wxHtmlEasyPrinting()
```

Returns pointer to `wxPrintData` instance used by this class.

You can set its parameters (via SetXXXX methods).

```
getPageSetupData(This) ->
                    wxPageSetupDialogData:wxPageSetupDialogData()
```

Types:

```
   This = wxHtmlEasyPrinting()
```

Returns a pointer to `wxPageSetupDialogData` instance used by this class.

You can set its parameters (via SetXXXX methods).

```
previewFile(This, Htmlfile) -> boolean()
```

Types:

```
   This = wxHtmlEasyPrinting()
   Htmlfile = unicode:chardata()
```

Preview HTML file.

Returns false in case of error - call `wxPrinter:getLastError/0` to get detailed information about the kind of the error.

```
previewText(This, Htmltext) -> boolean()
```

Types:

```
   This = wxHtmlEasyPrinting()
   Htmltext = unicode:chardata()
```

```
previewText(This, Htmltext, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxHtmlEasyPrinting()
   Htmltext = unicode:chardata()
   Option = {basepath, unicode:chardata()}
```

Preview HTML text (not file!).

Returns false in case of error - call `wxPrinter:getLastError/0` to get detailed information about the kind of the error.

```
printFile(This, Htmlfile) -> boolean()
```

Types:

```
   This = wxHtmlEasyPrinting()
   Htmlfile = unicode:chardata()
```

Print HTML file.

Returns false in case of error - call `wxPrinter:getLastError/0` to get detailed information about the kind of the error.

```
printText(This, Htmltext) -> boolean()
```

Types:

```
   This = wxHtmlEasyPrinting()
   Htmltext = unicode:chardata()
```

```
printText(This, Htmltext, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxHtmlEasyPrinting()
   Htmltext = unicode:chardata()
   Option = {basepath, unicode:chardata()}
```

Print HTML text (not file!).

Returns false in case of error - call `wxPrinter:getLastError/0` to get detailed information about the kind of the error.

```
pageSetup(This) -> ok
```

Types:

```
   This = wxHtmlEasyPrinting()
```

Display page setup dialog and allows the user to modify settings.

```
setFonts(This, Normal_face, Fixed_face) -> ok
```

Types:

```
This = wxHtmlEasyPrinting()
Normal_face = Fixed_face = unicode:chardata()
```

```
setFonts(This, Normal_face, Fixed_face, Options :: [Option]) -> ok
```

Types:

```
This = wxHtmlEasyPrinting()
Normal_face = Fixed_face = unicode:chardata()
Option = {sizes, [integer()]}
```

Sets fonts.

See `wxHtmlDCRenderer::SetFonts` (not implemented in wx) for detailed description.

```
setHeader(This, Header) -> ok
```

Types:

```
This = wxHtmlEasyPrinting()
Header = unicode:chardata()
```

```
setHeader(This, Header, Options :: [Option]) -> ok
```

Types:

```
This = wxHtmlEasyPrinting()
Header = unicode:chardata()
Option = {pg, integer()}
```

Set page header.

The following macros can be used inside it:

```
setFooter(This, Footer) -> ok
```

Types:

```
This = wxHtmlEasyPrinting()
Footer = unicode:chardata()
```

```
setFooter(This, Footer, Options :: [Option]) -> ok
```

Types:

```
This = wxHtmlEasyPrinting()
Footer = unicode:chardata()
Option = {pg, integer()}
```

Set page footer.

The following macros can be used inside it: @DATE@ is replaced by the current date in default format @PAGENUM@ is replaced by page number @PAGESCNT@ is replaced by total number of pages @TIME@ is replaced by the current time in default format @TITLE@ is replaced with the title of the document

```
destroy(This :: wxHtmlEasyPrinting()) -> ok
```

Destroys the object.

# wxHtmlLinkEvent

Erlang module

This event class is used for the events generated by `wxHtmlWindow`.

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxHtmlLinkEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxHtmlLinkEventType` to subscribe to events of this type.

## Data Types

```
wxHtmlLinkEvent() = wx:wx_object()
wxHtmlLink() =
    #wxHtmlLink{type = wxHtmlLinkEvent:wxHtmlLinkEventType(),
                linkInfo = wx:wx_wxHtmlLinkInfo()}
wxHtmlLinkEventType() =
    command_html_link_clicked | html_cell_clicked |
    html_cell_hover
```

## Exports

```
getLinkInfo(This) -> wx:wx_wxHtmlLinkInfo()
```

Types:

```
   This = wxHtmlLinkEvent()
```

Returns the `wx_wxHtmlLinkInfo()` which contains info about the cell clicked and the hyperlink it contains.

# wxHtmlWindow

Erlang module

wxHtmlWindow is probably the only class you will directly use unless you want to do something special (like adding new tag handlers or MIME filters).

The purpose of this class is to display rich content pages (either local file or downloaded via HTTP protocol) in a window based on a subset of the HTML standard. The width of the window is constant, given in the constructor and virtual height is changed dynamically depending on page size. Once the window is created you can set its content by calling setPage/2 with raw HTML, loadPage/2 with a wxFileSystem (not implemented in wx) location or loadFile/2 with a filename.

Note: If you want complete HTML/CSS support as well as a Javascript engine, consider using wxWebView instead.

wxHtmlWindow uses the wxImage class for displaying images, so you need to initialize the handlers for any image formats you use before loading a page. See ?wxInitAllImageHandlers and wxImage::AddHandler (not implemented in wx).

Styles

This class supports the following styles:

See: wxHtmlLinkEvent, wxHtmlCellEvent (not implemented in wx)

This class is derived (and can use functions) from: wxScrolledWindow wxPanel wxWindow wxEvtHandler

wxWidgets docs: **wxHtmlWindow**

## Events

Event types emitted from this class: html_cell_clicked, html_cell_hover, command_html_link_clicked

## Data Types

```
wxHtmlWindow() = wx:wx_object()
```

## Exports

```
new() -> wxHtmlWindow()
```

Default ctor.

```
new(Parent) -> wxHtmlWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxHtmlWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
```

```
       {style, integer()}
```

Constructor.

The parameters are the same as `wxScrolled::wxScrolled()` (not implemented in wx) constructor.

```
appendToPage(This, Source) -> boolean()
```

Types:

```
   This = wxHtmlWindow()
   Source = unicode:chardata()
```

Appends HTML fragment to currently displayed text and refreshes the window.

Return: false if an error occurred, true otherwise.

```
getOpenedAnchor(This) -> unicode:charlist()
```

Types:

```
   This = wxHtmlWindow()
```

Returns anchor within currently opened page (see `getOpenedPage/1`).

If no page is opened or if the displayed page wasn't produced by call to `loadPage/2`, empty string is returned.

```
getOpenedPage(This) -> unicode:charlist()
```

Types:

```
   This = wxHtmlWindow()
```

Returns full location of the opened page.

If no page is opened or if the displayed page wasn't produced by call to `loadPage/2`, empty string is returned.

```
getOpenedPageTitle(This) -> unicode:charlist()
```

Types:

```
   This = wxHtmlWindow()
```

Returns title of the opened page or wxEmptyString if the current page does not contain <TITLE> tag.

```
getRelatedFrame(This) -> wxFrame:wxFrame()
```

Types:

```
   This = wxHtmlWindow()
```

Returns the related frame.

```
historyBack(This) -> boolean()
```

Types:

```
   This = wxHtmlWindow()
```

Moves back to the previous page.

Only pages displayed using `loadPage/2` are stored in history list.

```
historyCanBack(This) -> boolean()
```

Types:

```
This = wxHtmlWindow()
```

Returns true if it is possible to go back in the history i.e.

`historyBack/1` won't fail.

```
historyCanForward(This) -> boolean()
```

Types:

```
This = wxHtmlWindow()
```

Returns true if it is possible to go forward in the history i.e.

`historyForward/1` won't fail.

```
historyClear(This) -> ok
```

Types:

```
This = wxHtmlWindow()
```

Clears history.

```
historyForward(This) -> boolean()
```

Types:

```
This = wxHtmlWindow()
```

Moves to next page in history.

Only pages displayed using `loadPage/2` are stored in history list.

```
loadFile(This, Filename) -> boolean()
```

Types:

```
This = wxHtmlWindow()
Filename = unicode:chardata()
```

Loads an HTML page from a file and displays it.

Return: false if an error occurred, true otherwise

See: `loadPage/2`

```
loadPage(This, Location) -> boolean()
```

Types:

```
This = wxHtmlWindow()
Location = unicode:chardata()
```

Unlike `setPage/2` this function first loads the HTML page from `location` and then displays it.

Return: false if an error occurred, true otherwise

See: `loadFile/2`

```
selectAll(This) -> ok
```

Types:

```
This = wxHtmlWindow()
```

Selects all text in the window.

See: selectLine/2, selectWord/2

```
selectionToText(This) -> unicode:charlist()
```

Types:

```
This = wxHtmlWindow()
```

Returns the current selection as plain text.

Returns an empty string if no text is currently selected.

```
selectLine(This, Pos) -> ok
```

Types:

```
This = wxHtmlWindow()
Pos = {X :: integer(), Y :: integer()}
```

Selects the line of text that pos points at.

Note that pos is relative to the top of displayed page, not to window's origin, use wxScrolledWindow:calcUnscrolledPosition/3 to convert physical coordinate.

See: selectAll/1, selectWord/2

```
selectWord(This, Pos) -> ok
```

Types:

```
This = wxHtmlWindow()
Pos = {X :: integer(), Y :: integer()}
```

Selects the word at position pos.

Note that pos is relative to the top of displayed page, not to window's origin, use wxScrolledWindow:calcUnscrolledPosition/3 to convert physical coordinate.

See: selectAll/1, selectLine/2

```
setBorders(This, B) -> ok
```

Types:

```
This = wxHtmlWindow()
B = integer()
```

This function sets the space between border of window and HTML contents.

See image:

```
setFonts(This, Normal_face, Fixed_face) -> ok
```

Types:

```
This = wxHtmlWindow()
Normal_face = Fixed_face = unicode:chardata()
```

```
setFonts(This, Normal_face, Fixed_face, Options :: [Option]) -> ok
```

Types:

```
This = wxHtmlWindow()
Normal_face = Fixed_face = unicode:chardata()
Option = {sizes, [integer()]}
```

This function sets font sizes and faces.

See `wxHtmlDCRenderer::SetFonts` (not implemented in wx) for detailed description.

See: SetSize()

```
setPage(This, Source) -> boolean()
```

Types:

```
This = wxHtmlWindow()
Source = unicode:chardata()
```

Sets the source of a page and displays it, for example:

If you want to load a document from some location use `loadPage/2` instead.

Return: false if an error occurred, true otherwise.

```
setRelatedFrame(This, Frame, Format) -> ok
```

Types:

```
This = wxHtmlWindow()
Frame = wxFrame:wxFrame()
Format = unicode:chardata()
```

Sets the frame in which page title will be displayed.

`format` is the format of the frame title, e.g. "HtmlHelp : %s". It must contain exactly one s. This s is substituted with HTML page title.

```
setRelatedStatusBar(This, Statusbar) -> ok
setRelatedStatusBar(This, Index) -> ok
```

Types:

```
This = wxHtmlWindow()
Index = integer()
```

`After` calling `setRelatedFrame/3`, this sets statusbar slot where messages will be displayed.

(Default is -1 = no messages.)

```
setRelatedStatusBar(This, Statusbar, Options :: [Option]) -> ok
```

Types:

```
This = wxHtmlWindow()
Statusbar = wxStatusBar:wxStatusBar()
Option = {index, integer()}
```

`Sets` the associated statusbar where messages will be displayed.

Call this instead of `setRelatedFrame/3` if you want statusbar updates only, no changing of the frame title.

Since: 2.9.0

```
toText(This) -> unicode:charlist()
```

Types:

```
This = wxHtmlWindow()
```

Returns content of currently displayed page as plain text.

```
destroy(This :: wxHtmlWindow()) -> ok
```

Destroys the object.

# wxIcon

Erlang module

An icon is a small rectangular bitmap usually used for denoting a minimized application.

It differs from a `wxBitmap` in always having a mask associated with it for transparent drawing. On some platforms, icons and bitmaps are implemented identically, since there is no real distinction between a `wxBitmap` with a mask and an icon; and there is no specific icon format on some platforms (X-based applications usually standardize on XPMs for small bitmaps and icons). However, some platforms (such as Windows) make the distinction, so a separate class is provided.

Remark: It is usually desirable to associate a pertinent icon with a frame. Icons can also be used for other purposes, for example with `wxTreeCtrl` and `wxListCtrl`. Icons have different formats on different platforms therefore separate icons will usually be created for the different environments. Platform-specific methods for creating a `wxIcon` structure are catered for, and this is an occasion where conditional compilation will probably be required. Note that a new icon must be created for every time the icon is to be used for a new window. In Windows, the icon will not be reloaded if it has already been used. An icon allocated to a frame will be deleted when the frame is deleted. For more information please see overview_bitmap.

Predefined objects (include wx.hrl): ?wxNullIcon

See: **Overview bitmap**, **Overview bitmap**, `wxIconBundle`, `wxDC:drawIcon/3`, `wxCursor`

This class is derived (and can use functions) from: `wxBitmap`

wxWidgets docs: **wxIcon**

## Data Types

```
wxIcon() = wx:wx_object()
```

## Exports

```
new() -> wxIcon()
```

Default ctor.

Constructs an icon object with no data; an assignment or another member function such as `wxBitmap:loadFile/3` must be called subsequently.

```
new(Name) -> wxIcon()
new(Icon) -> wxIcon()
```

Types:

```
Icon = wxIcon()
```

Copy ctor.

```
new(Name, Options :: [Option]) -> wxIcon()
```

Types:

```
Name = unicode:chardata()
Option =
    {type, wx:wx_enum()} |
    {desiredWidth, integer()} |
```

```
       {desiredHeight, integer()}
```

Loads an icon from a file or resource.

See: `wxBitmap:loadFile/3`

```
copyFromBitmap(This, Bmp) -> ok
```

Types:

```
   This = wxIcon()
   Bmp = wxBitmap:wxBitmap()
```

Copies `bmp` bitmap to this icon.

Under MS Windows the bitmap must have mask colour set.

See: `wxBitmap:loadFile/3`

```
destroy(This :: wxIcon()) -> ok
```

Destructor.

See overview_refcount_destruct for more info.

If the application omits to delete the icon explicitly, the icon will be destroyed automatically by wxWidgets when the application exits.

Warning: Do not delete an icon that is selected into a memory device context.

# wxIconBundle

Erlang module

This class contains multiple copies of an icon in different sizes. It is typically used in `wxDialog::SetIcons` (not implemented in wx) and `wxTopLevelWindow:setIcons/2`.

Predefined objects (include wx.hrl): ?wxNullIconBundle

wxWidgets docs: **wxIconBundle**

## Data Types

```
wxIconBundle() = wx:wx_object()
```

## Exports

```
new() -> wxIconBundle()
```

Default ctor.

```
new(Ic) -> wxIconBundle()
new(File) -> wxIconBundle()
```

Types:

```
File = unicode:chardata()
```

Initializes the bundle with the icon(s) found in the file.

```
new(File, Type) -> wxIconBundle()
```

Types:

```
File = unicode:chardata()
Type = wx:wx_enum()
```

```
destroy(This :: wxIconBundle()) -> ok
```

Destructor.

```
addIcon(This, File) -> ok
addIcon(This, Icon) -> ok
```

Types:

```
This = wxIconBundle()
Icon = wxIcon:wxIcon()
```

Adds the icon to the collection; if the collection already contains an icon with the same width and height, it is replaced by the new one.

```
addIcon(This, File, Type) -> ok
```

Types:

```
   This = wxIconBundle()
   File = unicode:chardata()
   Type = wx:wx_enum()

getIcon(This) -> wxIcon:wxIcon()
```

Types:

```
   This = wxIconBundle()

getIcon(This, Size) -> wxIcon:wxIcon()
getIcon(This, Size :: [Option]) -> wxIcon:wxIcon()
```

Types:

```
   This = wxIconBundle()
   Option = {size, integer()} | {flags, integer()}
```

Same as.

.

```
getIcon(This, Size, Options :: [Option]) -> wxIcon:wxIcon()
```

Types:

```
   This = wxIconBundle()
   Size = {W :: integer(), H :: integer()}
   Option = {flags, integer()}
```

Returns the icon with the given size.

If `size` is ?wxDefaultSize, it is interpreted as the standard system icon size, i.e. the size returned by `wxSystemSettings:getMetric/2` for `wxSYS_ICON_X` and `wxSYS_ICON_Y`.

If the bundle contains an icon with exactly the requested size, it's always returned. Otherwise, the behaviour depends on the flags. If only `wxIconBundle::FALLBACK_NONE` (not implemented in wx) is given, the function returns an invalid icon. If `wxIconBundle::FALLBACK_SYSTEM` (not implemented in wx) is given, it tries to find the icon of standard system size, regardless of the size passed as parameter. Otherwise, or if the icon system size is not found neither, but `wxIconBundle::FALLBACK_NEAREST_LARGER` (not implemented in wx) flag is specified, the function returns the smallest icon of the size larger than the requested one or, if this fails too, just the icon closest to the specified size.

The `flags` parameter is available only since wxWidgets 2.9.4.

# wxIconizeEvent

Erlang module

An event being sent when the frame is iconized (minimized) or restored.

See: **Overview events**, `wxTopLevelWindow:iconize/2`, `wxTopLevelWindow:isIconized/1`

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxIconizeEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxIconizeEventType` to subscribe to events of this type.

## Data Types

```
wxIconizeEvent() = wx:wx_object()
wxIconize() =
    #wxIconize{type = wxIconizeEvent:wxIconizeEventType(),
               iconized = boolean()}
wxIconizeEventType() = iconize
```

## Exports

```
isIconized(This) -> boolean()
```

Types:

```
   This = wxIconizeEvent()
```

Returns true if the frame has been iconized, false if it has been restored.

# wxIdleEvent

Erlang module

This class is used for idle events, which are generated when the system becomes idle. Note that, unless you do something specifically, the idle events are not sent if the system remains idle once it has become it, e.g. only a single idle event will be generated until something else resulting in more normal events happens and only then is the next idle event sent again.

If you need to ensure a continuous stream of idle events, you can either use `requestMore/2` method in your handler or call ?wxWakeUpIdle() periodically (for example from a timer event handler), but note that both of these approaches (and especially the first one) increase the system load and so should be avoided if possible.

By default, idle events are sent to all windows, including even the hidden ones because they may be shown if some condition is met from their `wxEVT_IDLE` (or related `wxEVT_UPDATE_UI`) handler. The children of hidden windows do not receive idle events however as they can't change their state in any way noticeable by the user. Finally, the global `wxApp` (not implemented in wx) object also receives these events, as usual, so it can be used for any global idle time processing.

If sending idle events to all windows is causing a significant overhead in your application, you can call `setMode/1` with the value wxIDLE_PROCESS_SPECIFIED, and set the wxWS_EX_PROCESS_IDLE extra window style for every window which should receive idle events, all the other ones will not receive them in this case.

Delayed Action Mechanism

`wxIdleEvent` can be used to perform some action "at slightly later time". This can be necessary in several circumstances when, for whatever reason, something can't be done in the current event handler. For example, if a mouse event handler is called with the mouse button pressed, the mouse can be currently captured and some operations with it - notably capturing it again - might be impossible or lead to undesirable results. If you still want to capture it, you can do it from `wxEVT_IDLE` handler when it is called the next time instead of doing it immediately.

This can be achieved in two different ways: when using static event tables, you will need a flag indicating to the (always connected) idle event handler whether the desired action should be performed. The originally called handler would then set it to indicate that it should indeed be done and the idle handler itself would reset it to prevent it from doing the same action again.

Using dynamically connected event handlers things are even simpler as the original event handler can simply `wxEvtHandler::Connect()` (not implemented in wx) or `wxEvtHandler::Bind()` (not implemented in wx) the idle event handler which would only be executed then and could `wxEvtHandler::Disconnect()` (not implemented in wx) or `wxEvtHandler::Unbind()` (not implemented in wx) itself.

See: **Overview events**, `wxUpdateUIEvent`, `wxWindow::OnInternalIdle` (not implemented in wx)

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxIdleEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxIdleEventType` to subscribe to events of this type.

## Data Types

```
wxIdleEvent() = wx:wx_object()
wxIdle() = #wxIdle{type = wxIdleEvent:wxIdleEventType()}
wxIdleEventType() = idle
```

## Exports

`getMode() -> wx:wx_enum()`

Static function returning a value specifying how wxWidgets will send idle events: to all windows, or only to those which specify that they will process the events.

See: `setMode/1`

`requestMore(This) -> ok`

Types:

```
This = wxIdleEvent()
```

`requestMore(This, Options :: [Option]) -> ok`

Types:

```
This = wxIdleEvent()
Option = {needMore, boolean()}
```

Tells wxWidgets that more processing is required.

This function can be called by an OnIdle handler for a window or window event handler to indicate that wxApp::OnIdle should forward the OnIdle event once more to the application windows.

If no window calls this function during OnIdle, then the application will remain in a passive event loop (not calling OnIdle) until a new event is posted to the application by the windowing system.

See: `moreRequested/1`

`moreRequested(This) -> boolean()`

Types:

```
This = wxIdleEvent()
```

Returns true if the OnIdle function processing this event requested more processing time.

See: `requestMore/2`

`setMode(Mode) -> ok`

Types:

```
Mode = wx:wx_enum()
```

Static function for specifying how wxWidgets will send idle events: to all windows, or only to those which specify that they will process the events.

# wxImage

Erlang module

This class encapsulates a platform-independent image.

An image can be created from data, or using `wxBitmap:convertToImage/1`. An image can be loaded from a file in a variety of formats, and is extensible to new formats via image format handlers. Functions are available to set and get image bits, so it can be used for basic image manipulation.

A `wxImage` cannot (currently) be drawn directly to a `wxDC`. Instead, a platform-specific `wxBitmap` object must be created from it using the wxBitmap::wxBitmap(wxImage,int depth) constructor. This bitmap can then be drawn in a device context, using `wxDC:drawBitmap/4`.

More on the difference between `wxImage` and `wxBitmap`: `wxImage` is just a buffer of RGB bytes with an optional buffer for the alpha bytes. It is all generic, platform independent and image file format independent code. It includes generic code for scaling, resizing, clipping, and other manipulations of the image data. OTOH, `wxBitmap` is intended to be a wrapper of whatever is the native image format that is quickest/easiest to draw to a DC or to be the target of the drawing operations performed on a `wxMemoryDC`. By splitting the responsibilities between wxImage/wxBitmap like this then it's easier to use generic code shared by all platforms and image types for generic operations and platform specific code where performance or compatibility is needed.

One colour value of the image may be used as a mask colour which will lead to the automatic creation of a `wxMask` object associated to the bitmap object.

Alpha channel support

Starting from wxWidgets 2.5.0 `wxImage` supports alpha channel data, that is in addition to a byte for the red, green and blue colour components for each pixel it also stores a byte representing the pixel opacity.

An alpha value of 0 corresponds to a transparent pixel (null opacity) while a value of 255 means that the pixel is 100% opaque. The constants ?wxIMAGE_ALPHA_TRANSPARENT and ?wxIMAGE_ALPHA_OPAQUE can be used to indicate those values in a more readable form.

While all images have RGB data, not all images have an alpha channel. Before using `getAlpha/3` you should check if this image contains an alpha channel with `hasAlpha/1`. Currently the BMP, PNG, TGA, and TIFF format handlers have full alpha channel support for loading so if you want to use alpha you have to use one of these formats. If you initialize the image alpha channel yourself using `setAlpha/4`, you should save it in either PNG, TGA, or TIFF format to avoid losing it as these are the only handlers that currently support saving with alpha.

Available image handlers

The following image handlers are available. wxBMPHandler is always installed by default. To use other image formats, install the appropriate handler with `wxImage::AddHandler` (not implemented in wx) or call ? wxInitAllImageHandlers().

When saving in PCX format, `wxPCXHandler` (not implemented in wx) will count the number of different colours in the image; if there are 256 or less colours, it will save as 8 bit, else it will save as 24 bit.

Loading PNMs only works for ASCII or raw RGB images. When saving in PNM format, `wxPNMHandler` (not implemented in wx) will always save as raw RGB.

Saving GIFs requires images of maximum 8 bpp (see `wxQuantize` (not implemented in wx)), and the alpha channel converted to a mask (see `convertAlphaToMask/5`). Saving an animated GIF requires images of the same size (see `wxGIFHandler::SaveAnimation` (not implemented in wx))

Predefined objects (include wx.hrl): ?wxNullImage

See: `wxBitmap`, ?wxInitAllImageHandlers(), `wxPixelData` (not implemented in wx)

wxWidgets docs: **wxImage**

## Data Types

```
wxImage() = wx:wx_object()
```

## Exports

```
new() -> wxImage()
```

Creates an empty `wxImage` object without an alpha channel.

```
new(Name) -> wxImage()
new(Sz) -> wxImage()
```

Types:

```
   Sz = {W :: integer(), H :: integer()}
```

```
new(Width, Height) -> wxImage()
new(Name, Height :: [Option]) -> wxImage()
new(Sz, Data) -> wxImage()
new(Sz, Height :: [Option]) -> wxImage()
```

Types:

```
   Sz = {W :: integer(), H :: integer()}
   Option = {clear, boolean()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
new(Width, Height, Data) -> wxImage()
new(Width, Height, Data :: [Option]) -> wxImage()
new(Name, Mimetype, Data :: [Option]) -> wxImage()
new(Sz, Data, Alpha) -> wxImage()
```

Types:

```
   Sz = {W :: integer(), H :: integer()}
   Data = Alpha = binary()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
new(Width, Height, Data, Alpha) -> wxImage()
```

Types:

```
   Width = Height = integer()
   Data = Alpha = binary()
```

Creates an image from data in memory.

If `static_data` is false then the `wxImage` will take ownership of the data and free it afterwards. For this, it has to be allocated with `malloc`.

```
destroy(This :: wxImage()) -> ok
```

Destructor.

See reference-counted object destruction for more info.

```
blur(This, BlurRadius) -> wxImage()
```

Types:

```
This = wxImage()
BlurRadius = integer()
```

Blurs the image in both horizontal and vertical directions by the specified pixel `blurRadius`.

This should not be used when using a single mask colour for transparency.

See: `blurHorizontal/2`, `blurVertical/2`

```
blurHorizontal(This, BlurRadius) -> wxImage()
```

Types:

```
This = wxImage()
BlurRadius = integer()
```

Blurs the image in the horizontal direction only.

This should not be used when using a single mask colour for transparency.

See: `blur/2`, `blurVertical/2`

```
blurVertical(This, BlurRadius) -> wxImage()
```

Types:

```
This = wxImage()
BlurRadius = integer()
```

Blurs the image in the vertical direction only.

This should not be used when using a single mask colour for transparency.

See: `blur/2`, `blurHorizontal/2`

```
convertAlphaToMask(This) -> boolean()
```

Types:

```
This = wxImage()
```

```
convertAlphaToMask(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxImage()
Option = {threshold, integer()}
```

If the image has alpha channel, this method converts it to mask.

If the image has an alpha channel, all pixels with alpha value less than `threshold` are replaced with the mask colour and the alpha channel is removed. Otherwise nothing is done.

The mask colour is chosen automatically using `findFirstUnusedColour/2`, see the overload below if this is not appropriate.

Return: Returns true on success, false on error.

```
convertAlphaToMask(This, Mr, Mg, Mb) -> boolean()
```

Types:

```
This = wxImage()
Mr = Mg = Mb = integer()
```

```
convertAlphaToMask(This, Mr, Mg, Mb, Options :: [Option]) ->
                      boolean()
```

Types:

```
This = wxImage()
Mr = Mg = Mb = integer()
Option = {threshold, integer()}
```

If the image has alpha channel, this method converts it to mask using the specified colour as the mask colour.

If the image has an alpha channel, all pixels with alpha value less than `threshold` are replaced with the mask colour and the alpha channel is removed. Otherwise nothing is done.

Since: 2.9.0

Return: Returns true on success, false on error.

```
convertToGreyscale(This) -> wxImage()
```

Types:

```
This = wxImage()
```

Returns a greyscale version of the image.

Since: 2.9.0

```
convertToGreyscale(This, Weight_r, Weight_g, Weight_b) ->
                      wxImage()
```

Types:

```
This = wxImage()
Weight_r = Weight_g = Weight_b = number()
```

Returns a greyscale version of the image.

The returned image uses the luminance component of the original to calculate the greyscale. Defaults to using the standard ITU-T BT.601 when converting to YUV, where every pixel equals (R * `weight_r`) + (G * `weight_g`) + (B * `weight_b`).

```
convertToMono(This, R, G, B) -> wxImage()
```

Types:

```
This = wxImage()
R = G = B = integer()
```

Returns monochromatic version of the image.

The returned image has white colour where the original has (r,g,b) colour and black colour everywhere else.

```
copy(This) -> wxImage()
```

Types:

```
This = wxImage()
```

Returns an identical copy of this image.

```
create(This, Sz) -> boolean()
```

Types:

```
This = wxImage()
Sz = {W :: integer(), H :: integer()}
```

```
create(This, Width, Height) -> boolean()
create(This, Sz, Data) -> boolean()
create(This, Sz, Height :: [Option]) -> boolean()
```

Types:

```
This = wxImage()
Sz = {W :: integer(), H :: integer()}
Option = {clear, boolean()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
create(This, Width, Height, Data) -> boolean()
create(This, Width, Height, Data :: [Option]) -> boolean()
create(This, Sz, Data, Alpha) -> boolean()
```

Types:

```
This = wxImage()
Sz = {W :: integer(), H :: integer()}
Data = Alpha = binary()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
create(This, Width, Height, Data, Alpha) -> boolean()
```

Types:

```
This = wxImage()
Width = Height = integer()
Data = Alpha = binary()
```

Creates a fresh image.

See `new/4` for more info.

Return: true if the call succeeded, false otherwise.

```
'Destroy'(This) -> ok
```

Types:

```
This = wxImage()
```

Destroys the image data.

```
findFirstUnusedColour(This) -> Result
```

Types:

```
Result =
    {Res :: boolean(),
     R :: integer(),
     G :: integer(),
     B :: integer()}
This = wxImage()
```

```
findFirstUnusedColour(This, Options :: [Option]) -> Result
```

Types:

```
Result =
    {Res :: boolean(),
     R :: integer(),
     G :: integer(),
     B :: integer()}
This = wxImage()
Option =
    {startR, integer()} |
    {startG, integer()} |
    {startB, integer()}
```

Finds the first colour that is never used in the image.

The search begins at given initial colour and continues by increasing R, G and B components (in this order) by 1 until an unused colour is found or the colour space exhausted.

The parameters `r`, `g`, `b` are pointers to variables to save the colour.

The parameters `startR`, `startG`, `startB` define the initial values of the colour. The returned colour will have RGB values equal to or greater than these.

Return: Returns false if there is no unused colour left, true on success.

Note: This method involves computing the histogram, which is a computationally intensive operation.

```
getImageExtWildcard() -> unicode:charlist()
```

Iterates all registered `wxImageHandler` (not implemented in wx) objects, and returns a string containing file extension masks suitable for passing to file open/save dialog boxes.

Return: The format of the returned string is `"(*.ext1;*.ext2)|*.ext1;*.ext2"`. It is usually a good idea to prepend a description before passing the result to the dialog. Example:

See: `wxImageHandler` (not implemented in wx)

```
getAlpha(This) -> binary()
```

Types:

```
This = wxImage()
```

Returns pointer to the array storing the alpha values for this image.

This pointer is NULL for the images without the alpha channel. If the image does have it, this pointer may be used to directly manipulate the alpha values which are stored as the RGB ones.

```
getAlpha(This, X, Y) -> integer()
```

Types:

```
This = wxImage()
X = Y = integer()
```

Return alpha value at given pixel location.

```
getBlue(This, X, Y) -> integer()
```

Types:

```
This = wxImage()
X = Y = integer()
```

Returns the blue intensity at the given coordinate.

```
getData(This) -> binary()
```

Types:

```
This = wxImage()
```

Returns the image data as an array.

This is most often used when doing direct image manipulation. The return value points to an array of characters in RGBRGBRGB... format in the top-to-bottom, left-to-right order, that is the first RGB triplet corresponds to the first pixel of the first row, the second one - to the second pixel of the first row and so on until the end of the first row, with second row following after it and so on.

You should not delete the returned pointer nor pass it to `setData/4`.

```
getGreen(This, X, Y) -> integer()
```

Types:

```
This = wxImage()
X = Y = integer()
```

Returns the green intensity at the given coordinate.

```
getImageCount(Filename) -> integer()
```

Types:

```
Filename = unicode:chardata()
```

```
getImageCount(Filename, Options :: [Option]) -> integer()
```

Types:

```
Filename = unicode:chardata()
Option = {type, wx:wx_enum()}
```

If the image file contains more than one image and the image handler is capable of retrieving these individually, this function will return the number of available images.

For the overload taking the parameter `filename`, that's the name of the file to query. For the overload taking the parameter `stream`, that's the opened input stream with image data.

See `wxImageHandler::GetImageCount()` (not implemented in wx) for more info.

The parameter `type` may be one of the following values:

Return: Number of available images. For most image handlers, this is 1 (exceptions are TIFF and ICO formats as well as animated GIFs for which this function returns the number of frames in the animation).

```
getHeight(This) -> integer()
```

Types:

```
   This = wxImage()
```

Gets the height of the image in pixels.

See: `getWidth/1`, `GetSize()` (not implemented in wx)

```
getMaskBlue(This) -> integer()
```

Types:

```
   This = wxImage()
```

Gets the blue value of the mask colour.

```
getMaskGreen(This) -> integer()
```

Types:

```
   This = wxImage()
```

Gets the green value of the mask colour.

```
getMaskRed(This) -> integer()
```

Types:

```
   This = wxImage()
```

Gets the red value of the mask colour.

```
getOrFindMaskColour(This) -> Result
```

Types:

```
   Result =
       {Res :: boolean(),
        R :: integer(),
        G :: integer(),
        B :: integer()}
   This = wxImage()
```

Get the current mask colour or find a suitable unused colour that could be used as a mask colour.

Returns true if the image currently has a mask.

```
getPalette(This) -> wxPalette:wxPalette()
```

Types:

```
   This = wxImage()
```

Returns the palette associated with the image.

Currently the palette is only used when converting to `wxBitmap` under Windows.

Some of the `wxImage` handlers have been modified to set the palette if one exists in the image file (usually 256 or less colour images in GIF or PNG format).

```
getRed(This, X, Y) -> integer()
```

Types:

```
This = wxImage()
X = Y = integer()
```

Returns the red intensity at the given coordinate.

```
getSubImage(This, Rect) -> wxImage()
```

Types:

```
This = wxImage()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
```

Returns a sub image of the current one as long as the rect belongs entirely to the image.

```
getWidth(This) -> integer()
```

Types:

```
This = wxImage()
```

Gets the width of the image in pixels.

See: `getHeight/1`, `GetSize()` (not implemented in wx)

```
hasAlpha(This) -> boolean()
```

Types:

```
This = wxImage()
```

Returns true if this image has alpha channel, false otherwise.

See: `getAlpha/3`, `setAlpha/4`

```
hasMask(This) -> boolean()
```

Types:

```
This = wxImage()
```

Returns true if there is a mask active, false otherwise.

```
getOption(This, Name) -> unicode:charlist()
```

Types:

```
This = wxImage()
Name = unicode:chardata()
```

Gets a user-defined string-valued option.

Generic options:

Options specific to `wxGIFHandler` (not implemented in wx):

Return: The value of the option or an empty string if not found. Use `hasOption/2` if an empty string can be a valid option value.

See: setOption/3, getOptionInt/2, hasOption/2

getOptionInt(This, Name) -> integer()

Types:

```
This = wxImage()
Name = unicode:chardata()
```

Gets a user-defined integer-valued option.

The function is case-insensitive to name. If the given option is not present, the function returns 0. Use hasOption/2 if 0 is a possibly valid value for the option.

Generic options:

Since: 2.9.3

Options specific to wxPNGHandler (not implemented in wx):

Options specific to wxTIFFHandler (not implemented in wx):

Options specific to wxGIFHandler (not implemented in wx):

Note: Be careful when combining the options wxIMAGE_OPTION_TIFF_SAMPLESPERPIXEL, wxIMAGE_OPTION_TIFF_BITSPERSAMPLE, and wxIMAGE_OPTION_TIFF_PHOTOMETRIC. While some measures are taken to prevent illegal combinations and/or values, it is still easy to abuse them and come up with invalid results in the form of either corrupted images or crashes.

Return: The value of the option or 0 if not found. Use hasOption/2 if 0 can be a valid option value.

See: setOption/3, getOption/2

hasOption(This, Name) -> boolean()

Types:

```
This = wxImage()
Name = unicode:chardata()
```

Returns true if the given option is present.

The function is case-insensitive to name.

The lists of the currently supported options are in getOption/2 and getOptionInt/2 function docs.

See: setOption/3, getOption/2, getOptionInt/2

initAlpha(This) -> ok

Types:

```
This = wxImage()
```

Initializes the image alpha channel data.

It is an error to call it if the image already has alpha data. If it doesn't, alpha data will be by default initialized to all pixels being fully opaque. But if the image has a mask colour, all mask pixels will be completely transparent.

initStandardHandlers() -> ok

Internal use only.

Adds standard image format handlers. It only install wxBMPHandler for the time being, which is used by wxBitmap.

This function is called by wxWidgets on startup, and shouldn't be called by the user.

See: wxImageHandler (not implemented in wx), ?wxInitAllImageHandlers(), wxQuantize (not implemented in wx)

```
isTransparent(This, X, Y) -> boolean()
```

Types:

```
This = wxImage()
X = Y = integer()
```

```
isTransparent(This, X, Y, Options :: [Option]) -> boolean()
```

Types:

```
This = wxImage()
X = Y = integer()
Option = {threshold, integer()}
```

Returns true if the given pixel is transparent, i.e. either has the mask colour if this image has a mask or if this image has alpha channel and alpha value of this pixel is strictly less than threshold.

```
loadFile(This, Name) -> boolean()
```

Types:

```
This = wxImage()
Name = unicode:chardata()
```

```
loadFile(This, Name, Options :: [Option]) -> boolean()
```

Types:

```
This = wxImage()
Name = unicode:chardata()
Option = {type, wx:wx_enum()} | {index, integer()}
```

Loads an image from a file.

If no handler type is provided, the library will try to autodetect the format.

```
loadFile(This, Name, Mimetype, Options :: [Option]) -> boolean()
```

Types:

```
This = wxImage()
Name = Mimetype = unicode:chardata()
Option = {index, integer()}
```

Loads an image from a file.

If no handler type is provided, the library will try to autodetect the format.

```
ok(This) -> boolean()
```

Types:

```
This = wxImage()
```

See: isOk/1.

```
isOk(This) -> boolean()
```
Types:
```
   This = wxImage()
```
Returns true if image data is present.

```
removeHandler(Name) -> boolean()
```
Types:
```
   Name = unicode:chardata()
```
Finds the handler with the given name, and removes it.

The handler is also deleted.

Return: true if the handler was found and removed, false otherwise.

See: `wxImageHandler` (not implemented in wx)

```
mirror(This) -> wxImage()
```
Types:
```
   This = wxImage()
```

```
mirror(This, Options :: [Option]) -> wxImage()
```
Types:
```
   This = wxImage()
   Option = {horizontally, boolean()}
```
Returns a mirrored copy of the image.

The parameter `horizontally` indicates the orientation.

```
replace(This, R1, G1, B1, R2, G2, B2) -> ok
```
Types:
```
   This = wxImage()
   R1 = G1 = B1 = R2 = G2 = B2 = integer()
```
Replaces the colour specified by `r1`,g1,b1 by the colour `r2`,g2,b2.

```
rescale(This, Width, Height) -> wxImage()
```
Types:
```
   This = wxImage()
   Width = Height = integer()
```

```
rescale(This, Width, Height, Options :: [Option]) -> wxImage()
```
Types:

```
This = wxImage()
Width = Height = integer()
Option = {quality, wx:wx_enum()}
```

Changes the size of the image in-place by scaling it: after a call to this function,the image will have the given width and height.

For a description of the `quality` parameter, see the `scale/4` function. Returns the (modified) image itself.

See: `scale/4`

```
resize(This, Size, Pos) -> wxImage()
```

Types:

```
This = wxImage()
Size = {W :: integer(), H :: integer()}
Pos = {X :: integer(), Y :: integer()}
```

```
resize(This, Size, Pos, Options :: [Option]) -> wxImage()
```

Types:

```
This = wxImage()
Size = {W :: integer(), H :: integer()}
Pos = {X :: integer(), Y :: integer()}
Option = {r, integer()} | {g, integer()} | {b, integer()}
```

Changes the size of the image in-place without scaling it by adding either a border with the given colour or cropping as necessary.

The image is pasted into a new image with the given `size` and background colour at the position `pos` relative to the upper left of the new image.

If `red` = green = blue = -1 then use either the current mask colour if set or find, use, and set a suitable mask colour for any newly exposed areas.

Return: The (modified) image itself.

See: `size/4`

```
rotate(This, Angle, RotationCentre) -> wxImage()
```

Types:

```
This = wxImage()
Angle = number()
RotationCentre = {X :: integer(), Y :: integer()}
```

```
rotate(This, Angle, RotationCentre, Options :: [Option]) ->
         wxImage()
```

Types:

```
This = wxImage()
Angle = number()
RotationCentre = {X :: integer(), Y :: integer()}
Option =
    {interpolating, boolean()} |
```

```
      {offset_after_rotation, {X :: integer(), Y :: integer()}}
```

Rotates the image about the given point, by `angle` radians.

Passing true to `interpolating` results in better image quality, but is slower.

If the image has a mask, then the mask colour is used for the uncovered pixels in the rotated image background. Else, black (rgb 0, 0, 0) will be used.

Returns the rotated image, leaving this image intact.

```
rotateHue(This, Angle) -> ok
```

Types:

```
   This = wxImage()
   Angle = number()
```

Rotates the hue of each pixel in the image by `angle`, which is a double in the range of -1.0 to +1.0, where -1.0 corresponds to -360 degrees and +1.0 corresponds to +360 degrees.

```
rotate90(This) -> wxImage()
```

Types:

```
   This = wxImage()
```

```
rotate90(This, Options :: [Option]) -> wxImage()
```

Types:

```
   This = wxImage()
   Option = {clockwise, boolean()}
```

Returns a copy of the image rotated 90 degrees in the direction indicated by `clockwise`.

```
saveFile(This, Name) -> boolean()
```

Types:

```
   This = wxImage()
   Name = unicode:chardata()
```

Saves an image in the named file.

File type is determined from the extension of the file name. Note that this function may fail if the extension is not recognized! You can use one of the forms above to save images to files with non-standard extensions.

```
saveFile(This, Name, Type) -> boolean()
saveFile(This, Name, Mimetype) -> boolean()
```

Types:

```
   This = wxImage()
   Name = Mimetype = unicode:chardata()
```

Saves an image in the named file.

```
scale(This, Width, Height) -> wxImage()
```

Types:

```
   This = wxImage()
   Width = Height = integer()

scale(This, Width, Height, Options :: [Option]) -> wxImage()
```

Types:

```
   This = wxImage()
   Width = Height = integer()
   Option = {quality, wx:wx_enum()}
```

Returns a scaled version of the image.

This is also useful for scaling bitmaps in general as the only other way to scale bitmaps is to blit a `wxMemoryDC` into another `wxMemoryDC`.

The parameter `quality` determines what method to use for resampling the image, see wxImageResizeQuality documentation.

It should be noted that although using `wxIMAGE_QUALITY_HIGH` produces much nicer looking results it is a slower method. Downsampling will use the box averaging method which seems to operate very fast. If you are upsampling larger images using this method you will most likely notice that it is a bit slower and in extreme cases it will be quite substantially slower as the bicubic algorithm has to process a lot of data.

It should also be noted that the high quality scaling may not work as expected when using a single mask colour for transparency, as the scaling will blur the image and will therefore remove the mask partially. Using the alpha channel will work.

Example:

See: `rescale/4`

```
size(This, Size, Pos) -> wxImage()
```

Types:

```
   This = wxImage()
   Size = {W :: integer(), H :: integer()}
   Pos = {X :: integer(), Y :: integer()}

size(This, Size, Pos, Options :: [Option]) -> wxImage()
```

Types:

```
   This = wxImage()
   Size = {W :: integer(), H :: integer()}
   Pos = {X :: integer(), Y :: integer()}
   Option = {r, integer()} | {g, integer()} | {b, integer()}
```

Returns a resized version of this image without scaling it by adding either a border with the given colour or cropping as necessary.

The image is pasted into a new image with the given `size` and background colour at the position `pos` relative to the upper left of the new image.

If `red` = green = blue = -1 then the areas of the larger image not covered by this image are made transparent by filling them with the image mask colour (which will be allocated automatically if it isn't currently set).

Otherwise, the areas will be filled with the colour with the specified RGB components.

See: `resize/4`

```
setAlpha(This, Alpha) -> ok
```

Types:

```
This = wxImage()
Alpha = binary()
```

This function is similar to `setData/4` and has similar restrictions.

The pointer passed to it may however be NULL in which case the function will allocate the alpha array internally - this is useful to add alpha channel data to an image which doesn't have any.

If the pointer is not NULL, it must have one byte for each image pixel and be allocated with malloc(). `wxImage` takes ownership of the pointer and will free it unless `static_data` parameter is set to true - in this case the caller should do it.

```
setAlpha(This, X, Y, Alpha) -> ok
```

Types:

```
This = wxImage()
X = Y = Alpha = integer()
```

Sets the alpha value for the given pixel.

This function should only be called if the image has alpha channel data, use `hasAlpha/1` to check for this.

```
setData(This, Data) -> ok
```

Types:

```
This = wxImage()
Data = binary()
```

Sets the image data without performing checks.

The data given must have the size (width*height*3) or results will be unexpected. Don't use this method if you aren't sure you know what you are doing.

The data must have been allocated with `malloc()`, `NOT` with `operator` new.

If `static_data` is false, after this call the pointer to the data is owned by the `wxImage` object, that will be responsible for deleting it. Do not pass to this function a pointer obtained through `getData/1`.

```
setData(This, Data, New_width, New_height) -> ok
```

Types:

```
This = wxImage()
Data = binary()
New_width = New_height = integer()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setMask(This) -> ok
```

Types:

```
This = wxImage()
```

```
setMask(This, Options :: [Option]) -> ok
```

Types:

```
This = wxImage()
Option = {mask, boolean()}
```

Specifies whether there is a mask or not.

The area of the mask is determined by the current mask colour.

```
setMaskColour(This, Red, Green, Blue) -> ok
```

Types:

```
This = wxImage()
Red = Green = Blue = integer()
```

Sets the mask colour for this image (and tells the image to use the mask).

```
setMaskFromImage(This, Mask, Mr, Mg, Mb) -> boolean()
```

Types:

```
This = Mask = wxImage()
Mr = Mg = Mb = integer()
```

Sets image's mask so that the pixels that have RGB value of mr,mg,mb in mask will be masked in the image.

This is done by first finding an unused colour in the image, setting this colour as the mask colour and then using this colour to draw all pixels in the image who corresponding pixel in mask has given RGB value.

The parameter `mask` is the mask image to extract mask shape from. It must have the same dimensions as the image.

The parameters `mr`, `mg`, `mb` are the RGB values of the pixels in mask that will be used to create the mask.

Return: Returns false if mask does not have same dimensions as the image or if there is no unused colour left. Returns true if the mask was successfully applied.

Note: Note that this method involves computing the histogram, which is a computationally intensive operation.

```
setOption(This, Name, Value) -> ok
setOption(This, Name, Value) -> ok
```

Types:

```
This = wxImage()
Name = Value = unicode:chardata()
```

Sets a user-defined option.

The function is case-insensitive to `name`.

For example, when saving as a JPEG file, the option `quality` is used, which is a number between 0 and 100 (0 is terrible, 100 is very good).

The lists of the currently supported options are in `getOption/2` and `getOptionInt/2` function docs.

See: `getOption/2`, `getOptionInt/2`, `hasOption/2`

```
setPalette(This, Palette) -> ok
```

Types:

```
This = wxImage()
Palette = wxPalette:wxPalette()
```

Associates a palette with the image.

The palette may be used when converting `wxImage` to `wxBitmap` (MSW only at present) or in file save operations (none as yet).

```
setRGB(This, Rect, Red, Green, Blue) -> ok
```

Types:

```
   This = wxImage()
   Rect =
       {X :: integer(),
        Y :: integer(),
        W :: integer(),
        H :: integer()}
   Red = Green = Blue = integer()
```

Sets the colour of the pixels within the given rectangle.

This routine performs bounds-checks for the coordinate so it can be considered a safe way to manipulate the data.

```
setRGB(This, X, Y, R, G, B) -> ok
```

Types:

```
   This = wxImage()
   X = Y = R = G = B = integer()
```

Set the color of the pixel at the given x and y coordinate.

# wxImageList

Erlang module

A `wxImageList` contains a list of images, which are stored in an unspecified form. Images can have masks for transparent drawing, and can be made from a variety of sources including bitmaps and icons.

`wxImageList` is used principally in conjunction with `wxTreeCtrl` and `wxListCtrl` classes.

See: `wxTreeCtrl`, `wxListCtrl`

wxWidgets docs: **wxImageList**

## Data Types

```
wxImageList() = wx:wx_object()
```

## Exports

```
new() -> wxImageList()
```

Default ctor.

```
new(Width, Height) -> wxImageList()
```

Types:

```
Width = Height = integer()
```

```
new(Width, Height, Options :: [Option]) -> wxImageList()
```

Types:

```
Width = Height = integer()
Option = {mask, boolean()} | {initialCount, integer()}
```

Constructor specifying the image size, whether image masks should be created, and the initial size of the list.

See: `create/4`

```
add(This, Icon) -> integer()
```

Types:

```
This = wxImageList()
Icon = wxIcon:wxIcon() | wxBitmap:wxBitmap()
```

Adds a new image using an icon.

Return: The new zero-based image index.

Remark: The original bitmap or icon is not affected by the `add/3` operation, and can be deleted afterwards. If the bitmap is wider than the images in the list, then the bitmap will automatically be split into smaller images, each matching the dimensions of the image list. This does not apply when adding icons.

Only for:wxmsw,wxosx

```
add(This, Bitmap, Mask) -> integer()
add(This, Bitmap, MaskColour) -> integer()
```

Types:

```
This = wxImageList()
Bitmap = wxBitmap:wxBitmap()
MaskColour = wx:wx_colour()
```

Adds a new image or images using a bitmap and mask colour.

Return: The new zero-based image index.

Remark: The original bitmap or icon is not affected by the `add/3` operation, and can be deleted afterwards. If the bitmap is wider than the images in the list, then the bitmap will automatically be split into smaller images, each matching the dimensions of the image list. This does not apply when adding icons.

```
create(This, Width, Height) -> boolean()
```

Types:

```
This = wxImageList()
Width = Height = integer()
```

```
create(This, Width, Height, Options :: [Option]) -> boolean()
```

Types:

```
This = wxImageList()
Width = Height = integer()
Option = {mask, boolean()} | {initialCount, integer()}
```

Initializes the list.

See `new/3` for details.

```
draw(This, Index, Dc, X, Y) -> boolean()
```

Types:

```
This = wxImageList()
Index = integer()
Dc = wxDC:wxDC()
X = Y = integer()
```

```
draw(This, Index, Dc, X, Y, Options :: [Option]) -> boolean()
```

Types:

```
This = wxImageList()
Index = integer()
Dc = wxDC:wxDC()
X = Y = integer()
Option = {flags, integer()} | {solidBackground, boolean()}
```

Draws a specified image onto a device context.

```
getBitmap(This, Index) -> wxBitmap:wxBitmap()
```

Types:

```
   This = wxImageList()
   Index = integer()
```

Returns the bitmap corresponding to the given index.

```
getIcon(This, Index) -> wxIcon:wxIcon()
```

Types:

```
   This = wxImageList()
   Index = integer()
```

Returns the icon corresponding to the given index.

```
getImageCount(This) -> integer()
```

Types:

```
   This = wxImageList()
```

Returns the number of images in the list.

```
getSize(This, Index) -> Result
```

Types:

```
   Result =
       {Res :: boolean(), Width :: integer(), Height :: integer()}
   This = wxImageList()
   Index = integer()
```

Retrieves the size of the images in the list.

Currently, the `index` parameter is ignored as all images in the list have the same size.

Return: true if the function succeeded, false if it failed (for example, if the image list was not yet initialized).

```
remove(This, Index) -> boolean()
```

Types:

```
   This = wxImageList()
   Index = integer()
```

Removes the image at the given position.

```
removeAll(This) -> boolean()
```

Types:

```
   This = wxImageList()
```

Removes all the images in the list.

```
replace(This, Index, Icon) -> boolean()
```

Types:

```
   This = wxImageList()
   Index = integer()
   Icon = wxIcon:wxIcon() | wxBitmap:wxBitmap()
```

Replaces the existing image with the new image.

Return: true if the replacement was successful, false otherwise.

Remark: The original bitmap or icon is not affected by the `replace/4` operation, and can be deleted afterwards.

Only for:wxmsw,wxosx

```
replace(This, Index, Bitmap, Mask) -> boolean()
```

Types:

```
   This = wxImageList()
   Index = integer()
   Bitmap = Mask = wxBitmap:wxBitmap()
```

Replaces the existing image with the new image.

Windows only.

Return: true if the replacement was successful, false otherwise.

Remark: The original bitmap or icon is not affected by the `replace/4` operation, and can be deleted afterwards.

```
destroy(This :: wxImageList()) -> ok
```

Destroys the object.

# wxInitDialogEvent

Erlang module

A `wxInitDialogEvent` is sent as a dialog or panel is being initialised. Handlers for this event can transfer data to the window.

The default handler calls `wxWindow:transferDataToWindow/1`.

See: **Overview events**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxInitDialogEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxInitDialogEventType` to subscribe to events of this type.

## Data Types

```
wxInitDialogEvent() = wx:wx_object()
wxInitDialog() =
    #wxInitDialog{type =
                      wxInitDialogEvent:wxInitDialogEventType()}
wxInitDialogEventType() = init_dialog
```

# wxJoystickEvent

Erlang module

This event class contains information about joystick events, particularly events received by windows.

See: `wxJoystick` (not implemented in wx)

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxJoystickEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxJoystickEventType` to subscribe to events of this type.

## Data Types

```
wxJoystickEvent() = wx:wx_object()
wxJoystick() =
    #wxJoystick{type = wxJoystickEvent:wxJoystickEventType(),
                pos = {X :: integer(), Y :: integer()},
                zPosition = integer(),
                buttonChange = integer(),
                buttonState = integer(),
                joyStick = integer()}
wxJoystickEventType() =
    joy_button_down | joy_button_up | joy_move | joy_zmove
```

## Exports

```
buttonDown(This) -> boolean()
```

Types:

```
   This = wxJoystickEvent()
```

```
buttonDown(This, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxJoystickEvent()
   Option = {but, integer()}
```

Returns true if the event was a down event from the specified button (or any button).

```
buttonIsDown(This) -> boolean()
```

Types:

```
   This = wxJoystickEvent()
```

```
buttonIsDown(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxJoystickEvent()
Option = {but, integer()}
```

Returns true if the specified button (or any button) was in a down state.

```
buttonUp(This) -> boolean()
```

Types:

```
This = wxJoystickEvent()
```

```
buttonUp(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxJoystickEvent()
Option = {but, integer()}
```

Returns true if the event was an up event from the specified button (or any button).

```
getButtonChange(This) -> integer()
```

Types:

```
This = wxJoystickEvent()
```

Returns the identifier of the button changing state.

The return value is where `n` is the index of the button changing state, which can also be retrieved using `GetButtonOrdinal()` (not implemented in wx).

Note that for `n` equal to 1, 2, 3 or 4 there are predefined `wxJOY_BUTTONn` constants which can be used for more clarity, however these constants are not defined for the buttons beyond the first four.

```
getButtonState(This) -> integer()
```

Types:

```
This = wxJoystickEvent()
```

Returns the down state of the buttons.

This is a `wxJOY_BUTTONn` identifier, where `n` is one of 1, 2, 3, 4.

```
getJoystick(This) -> integer()
```

Types:

```
This = wxJoystickEvent()
```

Returns the identifier of the joystick generating the event - one of wxJOYSTICK1 and wxJOYSTICK2.

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxJoystickEvent()
```

Returns the x, y position of the joystick event.

These coordinates are valid for all the events except wxEVT_JOY_ZMOVE.

```
getZPosition(This) -> integer()
```

Types:

```
This = wxJoystickEvent()
```

Returns the z position of the joystick event.

This method can only be used for wxEVT_JOY_ZMOVE events.

```
isButton(This) -> boolean()
```

Types:

```
This = wxJoystickEvent()
```

Returns true if this was a button up or down event (`not` 'is any button down?').

```
isMove(This) -> boolean()
```

Types:

```
This = wxJoystickEvent()
```

Returns true if this was an x, y move event.

```
isZMove(This) -> boolean()
```

Types:

```
This = wxJoystickEvent()
```

Returns true if this was a z move event.

# wxKeyEvent

Erlang module

This event class contains information about key press and release events.

The main information carried by this event is the key being pressed or released. It can be accessed using either `getKeyCode/1` function or `getUnicodeKey/1`. For the printable characters, the latter should be used as it works for any keys, including non-Latin-1 characters that can be entered when using national keyboard layouts. `getKeyCode/1` should be used to handle special characters (such as cursor arrows keys or `HOME` or `INS` and so on) which correspond to ?wxKeyCode enum elements above the `WXK_START` constant. While `getKeyCode/1` also returns the character code for Latin-1 keys for compatibility, it doesn't work for Unicode characters in general and will return `WXK_NONE` for any non-Latin-1 ones. For this reason, it's recommended to always use `getUnicodeKey/1` and only fall back to `getKeyCode/1` if `getUnicodeKey/1` returned `WXK_NONE` meaning that the event corresponds to a non-printable special keys.

While both of these functions can be used with the events of `wxEVT_KEY_DOWN`, `wxEVT_KEY_UP` and `wxEVT_CHAR` types, the values returned by them are different for the first two events and the last one. For the latter, the key returned corresponds to the character that would appear in e.g. a text zone if the user pressed the key in it. As such, its value depends on the current state of the Shift key and, for the letters, on the state of Caps Lock modifier. For example, if `A` key is pressed without Shift being held down, `wxKeyEvent` of type `wxEVT_CHAR` generated for this key press will return (from either `getKeyCode/1` or `getUnicodeKey/1` as their meanings coincide for ASCII characters) key code of 97 corresponding the ASCII value of `a`. And if the same key is pressed but with Shift being held (or Caps Lock being active), then the key could would be 65, i.e. ASCII value of capital `A`.

However for the key down and up events the returned key code will instead be `A` independently of the state of the modifier keys i.e. it depends only on physical key being pressed and is not translated to its logical representation using the current keyboard state. Such untranslated key codes are defined as follows:

Notice that the first rule applies to all Unicode letters, not just the usual Latin-1 ones. However for non-Latin-1 letters only `getUnicodeKey/1` can be used to retrieve the key code as `getKeyCode/1` just returns `WXK_NONE` in this case.

To summarize: you should handle `wxEVT_CHAR` if you need the translated key and `wxEVT_KEY_DOWN` if you only need the value of the key itself, independent of the current keyboard state.

Note: Not all key down events may be generated by the user. As an example, `wxEVT_KEY_DOWN` with `=` key code can be generated using the standard US keyboard layout but not using the German one because the `=` key corresponds to Shift-0 key combination in this layout and the key code for it is `0`, not `=`. Because of this you should avoid requiring your users to type key events that might be impossible to enter on their keyboard.

Another difference between key and char events is that another kind of translation is done for the latter ones when the Control key is pressed: char events for ASCII letters in this case carry codes corresponding to the ASCII value of Ctrl +Latter, i.e. 1 for Ctrl+A, 2 for Ctrl+B and so on until 26 for Ctrl+Z. This is convenient for terminal-like applications and can be completely ignored by all the other ones (if you need to handle Ctrl+A it is probably a better idea to use the key event rather than the char one). Notice that currently no translation is done for the presses of [, \, ], ^ and _ keys which might be mapped to ASCII values from 27 to 31. Since version 2.9.2, the enum values `WXK_CONTROL_A - WXK_CONTROL_Z` can be used instead of the non-descriptive constant values 1-26.

Finally, modifier keys only generate key events but no char events at all. The modifiers keys are `WXK_SHIFT`, `WXK_CONTROL`, `WXK_ALT` and various `WXK_WINDOWS_XXX` from ?wxKeyCode enum.

Modifier keys events are special in one additional aspect: usually the keyboard state associated with a key press is well defined, e.g. `shiftDown/1` returns `true` only if the Shift key was held pressed when the key that generated this event itself was pressed. There is an ambiguity for the key press events for Shift key itself however. By convention, it is considered to be already pressed when it is pressed and already released when it is released. In

other words, wxEVT_KEY_DOWN event for the Shift key itself will have wxMOD_SHIFT in getModifiers/1 and shiftDown/1 will return true while the wxEVT_KEY_UP event for Shift itself will not have wxMOD_SHIFT in its modifiers and shiftDown/1 will return false.

Tip: You may discover the key codes and modifiers generated by all the keys on your system interactively by running the page_samples_keyboard wxWidgets sample and pressing some keys in it.

Note: If a key down (EVT_KEY_DOWN) event is caught and the event handler does not call event.Skip() then the corresponding char event (EVT_CHAR) will not happen. This is by design and enables the programs that handle both types of events to avoid processing the same key twice. As a consequence, if you do not want to suppress the wxEVT_CHAR events for the keys you handle, always call event.Skip() in your wxEVT_KEY_DOWN handler. Not doing may also prevent accelerators defined using this key from working.

Note: If a key is maintained in a pressed state, you will typically get a lot of (automatically generated) key down events but only one key up one at the end when the key is released so it is wrong to assume that there is one up event corresponding to each down one.

Note: For Windows programmers: The key and char events in wxWidgets are similar to but slightly different from Windows WM_KEYDOWN and WM_CHAR events. In particular, Alt+x combination will generate a char event in wxWidgets (unless it is used as an accelerator) and almost all keys, including ones without ASCII equivalents, generate char events too.

See: wxKeyboardState (not implemented in wx)

This class is derived (and can use functions) from: wxEvent

wxWidgets docs: **wxKeyEvent**

## Events

Use wxEvtHandler:connect/3 with wxKeyEventType to subscribe to events of this type.

## Data Types

```
wxKeyEvent() = wx:wx_object()
wxKey() =
    #wxKey{type = wxKeyEvent:wxKeyEventType(),
           x = integer(),
           y = integer(),
           keyCode = integer(),
           controlDown = boolean(),
           shiftDown = boolean(),
           altDown = boolean(),
           metaDown = boolean(),
           uniChar = integer(),
           rawCode = integer(),
           rawFlags = integer()}
wxKeyEventType() = char | char_hook | key_down | key_up
```

## Exports

```
altDown(This) -> boolean()
```

Types:

```
   This = wxKeyEvent()
```

Returns true if the Alt key is pressed.

Notice that `getModifiers/1` should usually be used instead of this one.

`cmdDown(This) -> boolean()`

Types:

```
This = wxKeyEvent()
```

Returns true if the key used for command accelerators is pressed.

Same as `controlDown/1`. Deprecated.

Notice that `getModifiers/1` should usually be used instead of this one.

`controlDown(This) -> boolean()`

Types:

```
This = wxKeyEvent()
```

Returns true if the Control key or Apple/Command key under macOS is pressed.

This function doesn't distinguish between right and left control keys.

Notice that `getModifiers/1` should usually be used instead of this one.

`getKeyCode(This) -> integer()`

Types:

```
This = wxKeyEvent()
```

Returns the key code of the key that generated this event.

ASCII symbols return normal ASCII values, while events from special keys such as "left cursor arrow" (`WXK_LEFT`) return values outside of the ASCII range. See ?wxKeyCode for a full list of the virtual key codes.

Note that this method returns a meaningful value only for special non-alphanumeric keys or if the user entered a Latin-1 character (this includes ASCII and the accented letters found in Western European languages but not letters of other alphabets such as e.g. Cyrillic). Otherwise it simply method returns `WXK_NONE` and `getUnicodeKey/1` should be used to obtain the corresponding Unicode character.

Using `getUnicodeKey/1` is in general the right thing to do if you are interested in the characters typed by the user, `getKeyCode/1` should be only used for special keys (for which `getUnicodeKey/1` returns `WXK_NONE`). To handle both kinds of keys you might write:

`getModifiers(This) -> integer()`

Types:

```
This = wxKeyEvent()
```

Return the bit mask of all pressed modifier keys.

The return value is a combination of `wxMOD_ALT`, `wxMOD_CONTROL`, `wxMOD_SHIFT` and `wxMOD_META` bit masks. Additionally, `wxMOD_NONE` is defined as 0, i.e. corresponds to no modifiers (see `HasAnyModifiers()` (not implemented in wx)) and `wxMOD_CMD` is either `wxMOD_CONTROL` (MSW and Unix) or `wxMOD_META` (Mac), see `cmdDown/1`. See ?wxKeyModifier for the full list of modifiers.

Notice that this function is easier to use correctly than, for example, `controlDown/1` because when using the latter you also have to remember to test that none of the other modifiers is pressed:

and forgetting to do it can result in serious program bugs (e.g. program not working with European keyboard layout where `AltGr` key which is seen by the program as combination of CTRL and ALT is used). On the other hand, you can simply write:

with this function.

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxKeyEvent()
```

Obtains the position (in client coordinates) at which the key was pressed.

Notice that under most platforms this position is simply the current mouse pointer position and has no special relationship to the key event itself.

`x` and `y` may be NULL if the corresponding coordinate is not needed.

```
getRawKeyCode(This) -> integer()
```

Types:

```
This = wxKeyEvent()
```

Returns the raw key code for this event.

The flags are platform-dependent and should only be used if the functionality provided by other `wxKeyEvent` methods is insufficient.

Under MSW, the raw key code is the value of `wParam` parameter of the corresponding message.

Under GTK, the raw key code is the `keyval` field of the corresponding GDK event.

Under macOS, the raw key code is the `keyCode` field of the corresponding NSEvent.

Note: Currently the raw key codes are not supported by all ports, use #ifdef wxHAS_RAW_KEY_CODES to determine if this feature is available.

```
getRawKeyFlags(This) -> integer()
```

Types:

```
This = wxKeyEvent()
```

Returns the low level key flags for this event.

The flags are platform-dependent and should only be used if the functionality provided by other `wxKeyEvent` methods is insufficient.

Under MSW, the raw flags are just the value of `lParam` parameter of the corresponding message.

Under GTK, the raw flags contain the `hardware_keycode` field of the corresponding GDK event.

Under macOS, the raw flags contain the modifiers state.

Note: Currently the raw key flags are not supported by all ports, use #ifdef wxHAS_RAW_KEY_CODES to determine if this feature is available.

```
getUnicodeKey(This) -> integer()
```

Types:

```
This = wxKeyEvent()
```

Returns the Unicode character corresponding to this key event.

If the key pressed doesn't have any character value (e.g. a cursor key) this method will return `WXK_NONE`. In this case you should use `getKeyCode/1` to retrieve the value of the key.

This function is only available in Unicode build, i.e. when `wxUSE_UNICODE` is 1.

```
getX(This) -> integer()
```

Types:

```
This = wxKeyEvent()
```

Returns the X position (in client coordinates) of the event.

See: `getPosition/1`

```
getY(This) -> integer()
```

Types:

```
This = wxKeyEvent()
```

Returns the Y position (in client coordinates) of the event.

See: `getPosition/1`

```
hasModifiers(This) -> boolean()
```

Types:

```
This = wxKeyEvent()
```

Returns true if Control or Alt are pressed.

Checks if Control, Alt or, under macOS only, Command key are pressed (notice that the real Control key is still taken into account under OS X too).

This method returns false if only Shift is pressed for compatibility reasons and also because pressing Shift usually doesn't change the interpretation of key events, see `HasAnyModifiers()` (not implemented in wx) if you want to take Shift into account as well.

```
metaDown(This) -> boolean()
```

Types:

```
This = wxKeyEvent()
```

Returns true if the Meta/Windows/Apple key is pressed.

This function tests the state of the key traditionally called Meta under Unix systems, Windows keys under MSW Notice that `getModifiers/1` should usually be used instead of this one.

See: `cmdDown/1`

```
shiftDown(This) -> boolean()
```

Types:

```
This = wxKeyEvent()
```

Returns true if the Shift key is pressed.

This function doesn't distinguish between right and left shift keys.

Notice that `getModifiers/1` should usually be used instead of this one.

# wxLayoutAlgorithm

Erlang module

`wxLayoutAlgorithm` implements layout of subwindows in MDI or SDI frames. It sends a `wxCalculateLayoutEvent` (not implemented in wx) event to children of the frame, asking them for information about their size. For MDI parent frames, the algorithm allocates the remaining space to the MDI client window (which contains the MDI child frames).

For SDI (normal) frames, a 'main' window is specified as taking up the remaining space.

Because the event system is used, this technique can be applied to any windows, which are not necessarily 'aware' of the layout classes (no virtual functions in `wxWindow` refer to `wxLayoutAlgorithm` or its events). However, you may wish to use `wxSashLayoutWindow` for your subwindows since this class provides handlers for the required events, and accessors to specify the desired size of the window. The sash behaviour in the base class can be used, optionally, to make the windows user-resizable.

`wxLayoutAlgorithm` is typically used in IDE (integrated development environment) applications, where there are several resizable windows in addition to the MDI client window, or other primary editing window. Resizable windows might include toolbars, a project window, and a window for displaying error and warning messages.

When a window receives an OnCalculateLayout event, it should call SetRect in the given event object, to be the old supplied rectangle minus whatever space the window takes up. It should also set its own size accordingly. `wxSashLayoutWindow::OnCalculateLayout` (not implemented in wx) generates an OnQueryLayoutInfo event which it sends to itself to determine the orientation, alignment and size of the window, which it gets from internal member variables set by the application.

The algorithm works by starting off with a rectangle equal to the whole frame client area. It iterates through the frame children, generating wxLayoutAlgorithm::OnCalculateLayout events which subtract the window size and return the remaining rectangle for the next window to process. It is assumed (by `wxSashLayoutWindow::OnCalculateLayout` (not implemented in wx)) that a window stretches the full dimension of the frame client, according to the orientation it specifies. For example, a horizontal window will stretch the full width of the remaining portion of the frame client area. In the other orientation, the window will be fixed to whatever size was specified by wxLayoutAlgorithm::OnQueryLayoutInfo. An alignment setting will make the window 'stick' to the left, top, right or bottom of the remaining client area. This scheme implies that order of window creation is important. Say you wish to have an extra toolbar at the top of the frame, a project window to the left of the MDI client window, and an output window above the status bar. You should therefore create the windows in this order: toolbar, output window, project window. This ensures that the toolbar and output window take up space at the top and bottom, and then the remaining height in-between is used for the project window.

`wxLayoutAlgorithm` is quite independent of the way in which wxLayoutAlgorithm::OnCalculateLayout chooses to interpret a window's size and alignment. Therefore you could implement a different window class with a new wxLayoutAlgorithm::OnCalculateLayout event handler, that has a more sophisticated way of laying out the windows. It might allow specification of whether stretching occurs in the specified orientation, for example, rather than always assuming stretching. (This could, and probably should, be added to the existing implementation).

Note: `wxLayoutAlgorithm` has nothing to do with `wxLayoutConstraints` (not implemented in wx). It is an alternative way of specifying layouts for which the normal constraint system is unsuitable.

See: `wxSashEvent`, `wxSashLayoutWindow`, **Overview events**

wxWidgets docs: **wxLayoutAlgorithm**

## Data Types

```
wxLayoutAlgorithm() = wx:wx_object()
```

## Exports

```
new() -> wxLayoutAlgorithm()
```

Default constructor.

```
destroy(This :: wxLayoutAlgorithm()) -> ok
```

Destructor.

```
layoutFrame(This, Frame) -> boolean()
```

Types:

```
   This = wxLayoutAlgorithm()
   Frame = wxFrame:wxFrame()
```

```
layoutFrame(This, Frame, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxLayoutAlgorithm()
   Frame = wxFrame:wxFrame()
   Option = {mainWindow, wxWindow:wxWindow()}
```

Lays out the children of a normal frame.

`mainWindow` is set to occupy the remaining space. This function simply calls `layoutWindow/3`.

```
layoutMDIFrame(This, Frame) -> boolean()
```

Types:

```
   This = wxLayoutAlgorithm()
   Frame = wxMDIParentFrame:wxMDIParentFrame()
```

```
layoutMDIFrame(This, Frame, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxLayoutAlgorithm()
   Frame = wxMDIParentFrame:wxMDIParentFrame()
   Option =
       {rect,
        {X :: integer(),
         Y :: integer(),
         W :: integer(),
         H :: integer()}}
```

Lays out the children of an MDI parent frame.

If `rect` is non-NULL, the given rectangle will be used as a starting point instead of the frame's client area. The MDI client window is set to occupy the remaining space.

```
layoutWindow(This, Parent) -> boolean()
```

Types:

```
   This = wxLayoutAlgorithm()
   Parent = wxWindow:wxWindow()
```

```
layoutWindow(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxLayoutAlgorithm()
   Parent = wxWindow:wxWindow()
   Option = {mainWindow, wxWindow:wxWindow()}
```

Lays out the children of a normal frame or other window.

`mainWindow` is set to occupy the remaining space. If this is not specified, then the last window that responds to a calculate layout event in query mode will get the remaining space (that is, a non-query OnCalculateLayout event will not be sent to this window and the window will be set to the remaining size).

# wxListBox

Erlang module

A listbox is used to select one or more of a list of strings.

The strings are displayed in a scrolling box, with the selected string(s) marked in reverse video. A listbox can be single selection (if an item is selected, the previous selection is removed) or multiple selection (clicking an item toggles the item on or off independently of other selections).

List box elements are numbered from zero and while the maximal number of elements is unlimited, it is usually better to use a virtual control, not requiring to add all the items to it at once, such as `wxDataViewCtrl` (not implemented in wx) or `wxListCtrl` with `wxLC_VIRTUAL` style, once more than a few hundreds items need to be displayed because this control is not optimized, neither from performance nor from user interface point of view, for large number of items.

Notice that the list box doesn't support control characters other than `TAB`.

Styles

This class supports the following styles:

See: `wxEditableListBox` (not implemented in wx), `wxChoice`, `wxComboBox`, `wxListCtrl`, `wxCommandEvent`

This class is derived (and can use functions) from: `wxControlWithItems wxControl wxWindow wxEvtHandler`

wxWidgets docs: **wxListBox**

## Events

Event types emitted from this class: `command_listbox_selected`, `command_listbox_doubleclicked`

## Data Types

```
wxListBox() = wx:wx_object()
```

## Exports

```
new() -> wxListBox()
```

Default constructor.

```
new(Parent, Id) -> wxListBox()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxListBox()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
```

```
       {choices, [unicode:chardata()]} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Constructor, creating and showing a list box.

See the other `new/3` constructor; the only difference is that this overload takes a `wxArrayString` (not implemented in wx) instead of a pointer to an array of `wxString` (not implemented in wx).

```
destroy(This :: wxListBox()) -> ok
```

Destructor, destroying the list box.

```
create(This, Parent, Id, Pos, Size, Choices) -> boolean()
```

Types:

```
   This = wxListBox()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Pos = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Choices = [unicode:chardata()]
```

```
create(This, Parent, Id, Pos, Size, Choices, Options :: [Option]) ->
          boolean()
```

Types:

```
   This = wxListBox()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Pos = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Choices = [unicode:chardata()]
   Option = {style, integer()} | {validator, wx:wx_object()}
```

```
deselect(This, N) -> ok
```

Types:

```
   This = wxListBox()
   N = integer()
```

Deselects an item in the list box.

Remark: This applies to multiple selection listboxes only.

```
getSelections(This) -> Result
```

Types:

```
   Result = {Res :: integer(), Selections :: [integer()]}
   This = wxListBox()
```

Fill an array of ints with the positions of the currently selected items.

Return: The number of selections.

Remark: Use this with a multiple selection listbox.

See: `wxControlWithItems:getSelection/1`, `wxControlWithItems:getStringSelection/1`, `wxControlWithItems:setSelection/2`

```
insertItems(This, Items, Pos) -> ok
```

Types:

```
This = wxListBox()
Items = [unicode:chardata()]
Pos = integer()
```

Insert the given number of strings before the specified position.

```
isSelected(This, N) -> boolean()
```

Types:

```
This = wxListBox()
N = integer()
```

Determines whether an item is selected.

Return: true if the given item is selected, false otherwise.

```
set(This, Items) -> ok
```

Types:

```
This = wxListBox()
Items = [unicode:chardata()]
```

Replaces the current control contents with the given items.

Notice that calling this method is usually much faster than appending them one by one if you need to add a lot of items.

```
hitTest(This, Point) -> integer()
```

Types:

```
This = wxListBox()
Point = {X :: integer(), Y :: integer()}
```

Returns the item located at `point`, or `wxNOT_FOUND` if there is no item located at `point`.

It is currently implemented for wxMSW, wxMac and wxGTK2 ports.

Return: Item located at point, or wxNOT_FOUND if unimplemented or the item does not exist.

Since: 2.7.0

```
hitTest(This, X, Y) -> integer()
```

Types:

```
This = wxListBox()
X = Y = integer()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setFirstItem(This, N) -> ok
setFirstItem(This, String) -> ok
```

Types:

```
This = wxListBox()
String = unicode:chardata()
```

Set the specified item to be the first visible item.

# wxListCtrl

Erlang module

A list control presents lists in a number of formats: list view, report view, icon view and small icon view. In any case, elements are numbered from zero. For all these modes, the items are stored in the control and must be added to it using `insertItem/4` method.

A special case of report view quite different from the other modes of the list control is a virtual control in which the items data (including text, images and attributes) is managed by the main program and is requested by the control itself only when needed which allows having controls with millions of items without consuming much memory. To use virtual list control you must use `setItemCount/2` first and override at least `wxListCtrl::OnGetItemText` (not implemented in wx) (and optionally `wxListCtrl::OnGetItemImage` (not implemented in wx) or `wxListCtrl::OnGetItemColumnImage` (not implemented in wx) and `wxListCtrl::OnGetItemAttr` (not implemented in wx)) to return the information about the items when the control requests it.

Virtual list control can be used as a normal one except that no operations which can take time proportional to the number of items in the control happen - this is required to allow having a practically infinite number of items. For example, in a multiple selection virtual list control, the selections won't be sent when many items are selected at once because this could mean iterating over all the items.

Using many of `wxListCtrl` features is shown in the corresponding sample.

To intercept events from a list control, use the event table macros described in `wxListEvent`.

`wxMac Note`: Starting with wxWidgets 2.8, `wxListCtrl` uses a native implementation for report mode, and uses a generic implementation for other modes. You can use the generic implementation for report mode as well by setting the `mac.listctrl.always_use_generic` system option (see `wxSystemOptions`) to 1.

Styles

This class supports the following styles:

Note: Under wxMSW this control uses `wxSystemThemedControl` (not implemented in wx) for an explorer style appearance by default since wxWidgets 3.1.0. If this is not desired, you can call `wxSystemThemedControl::EnableSystemTheme` (not implemented in wx) with `false` argument to disable this.

See: **Overview listctrl**, `wxListView`, `wxListBox`, `wxTreeCtrl`, `wxImageList`, `wxListEvent`, `wxListItem`, `wxEditableListBox` (not implemented in wx)

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxListCtrl**

## Events

Event types emitted from this class: `command_list_begin_drag`, `command_list_begin_rdrag`, `command_list_begin_label_edit`, `command_list_end_label_edit`, `command_list_delete_item`, `command_list_delete_all_items`, `command_list_item_selected`, `command_list_item_deselected`, `command_list_item_activated`, `command_list_item_focused`, `command_list_item_middle_click`, `command_list_item_right_click`, `command_list_key_down`, `command_list_insert_item`, `command_list_col_click`, `command_list_col_right_click`, `command_list_col_begin_drag`, `command_list_col_dragging`, `command_list_col_end_drag`, `command_list_cache_hint`

## Data Types

```
wxListCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxListCtrl()
```

Default constructor.

```
new(Parent, Options :: [Option]) -> wxListCtrl()
```

Types:

```
Parent = wxWindow:wxWindow()
Option =
    {winid, integer()} |
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {validator, wx:wx_object()} |
    {onGetItemText, function()} |
    {onGetItemAttr, function()} |
    {onGetItemColumnImage, function()}
```

Constructor, creating and showing a list control.

See: `create/3`, `wxValidator` (not implemented in wx)

```
destroy(This :: wxListCtrl()) -> ok
```

Destructor, destroying the list control.

```
arrange(This) -> boolean()
```

Types:

```
This = wxListCtrl()
```

```
arrange(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxListCtrl()
Option = {flag, integer()}
```

Arranges the items in icon or small icon view.

This only has effect on Win32. `flag` is one of:

```
assignImageList(This, ImageList, Which) -> ok
```

Types:

```
This = wxListCtrl()
ImageList = wxImageList:wxImageList()
Which = integer()
```

Sets the image list associated with the control and takes ownership of it (i.e.

the control will, unlike when using setImageList/3, delete the list when destroyed). which is one of wxIMAGE_LIST_NORMAL, wxIMAGE_LIST_SMALL, wxIMAGE_LIST_STATE (the last is unimplemented).

See: setImageList/3

```
clearAll(This) -> ok
```

Types:

```
   This = wxListCtrl()
```

Deletes all items and all columns.

Note: This sends an event of type wxEVT_LIST_DELETE_ALL_ITEMS under all platforms.

```
create(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
   This = Parent = wxWindow:wxWindow()
   Option =
       {winid, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()} |
       {onGetItemText, function()} |
       {onGetItemAttr, function()} |
       {onGetItemColumnImage, function()}
```

Creates the list control.

See new/2 for further details.

```
deleteAllItems(This) -> boolean()
```

Types:

```
   This = wxListCtrl()
```

Deletes all items in the list control.

This function does not send the wxEVT_LIST_DELETE_ITEM event because deleting many items from the control would be too slow then (unlike deleteItem/2) but it does send the special wxEVT_LIST_DELETE_ALL_ITEMS event if the control was not empty. If it was already empty, nothing is done and no event is sent.

Return: true if the items were successfully deleted or if the control was already empty, false if an error occurred while deleting the items.

```
deleteColumn(This, Col) -> boolean()
```

Types:

```
   This = wxListCtrl()
   Col = integer()
```

Deletes a column.

```
deleteItem(This, Item) -> boolean()
```

Types:

```
This = wxListCtrl()
Item = integer()
```

Deletes the specified item.

This function sends the `wxEVT_LIST_DELETE_ITEM` event for the item being deleted.

See: `deleteAllItems/1`

```
editLabel(This, Item) -> wxTextCtrl:wxTextCtrl()
```

Types:

```
This = wxListCtrl()
Item = integer()
```

Starts editing the label of the given item.

This function generates a `EVT_LIST_BEGIN_LABEL_EDIT` event which can be vetoed so that no text control will appear for in-place editing.

If the user changed the label (i.e. s/he does not press ESC or leave the text control without changes, a `EVT_LIST_END_LABEL_EDIT` event will be sent which can be vetoed as well.

```
ensureVisible(This, Item) -> boolean()
```

Types:

```
This = wxListCtrl()
Item = integer()
```

Ensures this item is visible.

```
findItem(This, Start, Str) -> integer()
```

Types:

```
This = wxListCtrl()
Start = integer()
Str = unicode:chardata()
```

```
findItem(This, Start, Str, Options :: [Option]) -> integer()
findItem(This, Start, Pt, Direction) -> integer()
```

Types:

```
This = wxListCtrl()
Start = integer()
Pt = {X :: integer(), Y :: integer()}
Direction = integer()
```

Find an item nearest this position in the specified direction, starting from `start` or the beginning if `start` is -1.

Return: The next matching item if any or `-1` (wxNOT_FOUND) otherwise.

```
getColumn(This, Col, Item) -> boolean()
```

Types:

```
This = wxListCtrl()
Col = integer()
Item = wxListItem:wxListItem()
```

Gets information about this column.

See `setItem/5` for more information.

```
getColumnCount(This) -> integer()
```

Types:

```
This = wxListCtrl()
```

Returns the number of columns.

```
getColumnWidth(This, Col) -> integer()
```

Types:

```
This = wxListCtrl()
Col = integer()
```

Gets the column width (report view only).

```
getCountPerPage(This) -> integer()
```

Types:

```
This = wxListCtrl()
```

Gets the number of items that can fit vertically in the visible area of the list control (list or report view) or the total number of items in the list control (icon or small icon view).

```
getEditControl(This) -> wxTextCtrl:wxTextCtrl()
```

Types:

```
This = wxListCtrl()
```

Returns the edit control being currently used to edit a label.

Returns NULL if no label is being edited.

Note: It is currently only implemented for wxMSW and the generic version, not for the native macOS version.

```
getImageList(This, Which) -> wxImageList:wxImageList()
```

Types:

```
This = wxListCtrl()
Which = integer()
```

Returns the specified image list.

`which` may be one of:

```
getItem(This, Info) -> boolean()
```

Types:

```
This = wxListCtrl()
Info = wxListItem:wxListItem()
```

Gets information about the item.

See `setItem/5` for more information.

You must call `info.SetId()` to set the ID of item you're interested in before calling this method, and `info.SetMask()` with the flags indicating what fields you need to retrieve from `info`.

```
getItemBackgroundColour(This, Item) -> wx:wx_colour4()
```

Types:

```
   This = wxListCtrl()
   Item = integer()
```

Returns the colour for this item.

If the item has no specific colour, returns an invalid colour (and not the default background control of the control itself).

See: `getItemTextColour/2`

```
getItemCount(This) -> integer()
```

Types:

```
   This = wxListCtrl()
```

Returns the number of items in the list control.

```
getItemData(This, Item) -> integer()
```

Types:

```
   This = wxListCtrl()
   Item = integer()
```

Gets the application-defined data associated with this item.

```
getItemFont(This, Item) -> wxFont:wxFont()
```

Types:

```
   This = wxListCtrl()
   Item = integer()
```

Returns the item's font.

```
getItemPosition(This, Item) -> Result
```

Types:

```
   Result =
       {Res :: boolean(), Pos :: {X :: integer(), Y :: integer()}}
   This = wxListCtrl()
   Item = integer()
```

Returns the position of the item, in icon or small icon view.

```
getItemRect(This, Item) -> Result
```

Types:

```
   Result =
       {Res :: boolean(),
        Rect ::
            {X :: integer(),
             Y :: integer(),
```

```
            W :: integer(),
            H :: integer()}}
   This = wxListCtrl()
   Item = integer()

getItemRect(This, Item, Options :: [Option]) -> Result
```

Types:

```
   Result =
       {Res :: boolean(),
        Rect ::
            {X :: integer(),
             Y :: integer(),
             W :: integer(),
             H :: integer()}}
   This = wxListCtrl()
   Item = integer()
   Option = {code, integer()}
```

Returns the rectangle representing the item's size and position, in physical coordinates.

`code` is one of wxLIST_RECT_BOUNDS, wxLIST_RECT_ICON, wxLIST_RECT_LABEL.

```
getItemSpacing(This) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxListCtrl()
```

Retrieves the spacing between icons in pixels: horizontal spacing is returned as `x` component of the {Width,Height} object and the vertical spacing as its `y` component.

```
getItemState(This, Item, StateMask) -> integer()
```

Types:

```
   This = wxListCtrl()
   Item = StateMask = integer()
```

Gets the item state.

For a list of state flags, see `setItem/5`. The `stateMask` indicates which state flags are of interest.

```
getItemText(This, Item) -> unicode:charlist()
```

Types:

```
   This = wxListCtrl()
   Item = integer()

getItemText(This, Item, Options :: [Option]) -> unicode:charlist()
```

Types:

```
   This = wxListCtrl()
   Item = integer()
   Option = {col, integer()}
```

Gets the item text for this item.

```
getItemTextColour(This, Item) -> wx:wx_colour4()
```

Types:

```
   This = wxListCtrl()
   Item = integer()
```

Returns the colour for this item.

If the item has no specific colour, returns an invalid colour (and not the default foreground control of the control itself as this wouldn't allow distinguishing between items having the same colour as the current control foreground and items with default colour which, hence, have always the same colour as the control).

```
getNextItem(This, Item) -> integer()
```

Types:

```
   This = wxListCtrl()
   Item = integer()
```

```
getNextItem(This, Item, Options :: [Option]) -> integer()
```

Types:

```
   This = wxListCtrl()
   Item = integer()
   Option = {geometry, integer()} | {state, integer()}
```

Searches for an item with the given geometry or state, starting from `item` but excluding the `item` itself.

If `item` is -1, the first item that matches the specified flags will be returned. Returns the first item with given state following `item` or -1 if no such item found. This function may be used to find all selected items in the control like this:

`geometry` can be one of:

Note: this parameter is only supported by wxMSW currently and ignored on other platforms.

`state` can be a bitlist of the following:

```
getSelectedItemCount(This) -> integer()
```

Types:

```
   This = wxListCtrl()
```

Returns the number of selected items in the list control.

```
getTextColour(This) -> wx:wx_colour4()
```

Types:

```
   This = wxListCtrl()
```

Gets the text colour of the list control.

```
getTopItem(This) -> integer()
```

Types:

```
   This = wxListCtrl()
```

Gets the index of the topmost visible item when in list or report view.

```
getViewRect(This) ->
               {X :: integer(),
```

```
               Y :: integer(),
               W :: integer(),
               H :: integer()}
```

Types:

```
This = wxListCtrl()
```

Returns the rectangle taken by all items in the control.

In other words, if the controls client size were equal to the size of this rectangle, no scrollbars would be needed and no free space would be left.

Note that this function only works in the icon and small icon views, not in list or report views (this is a limitation of the native Win32 control).

```
hitTest(This, Point) -> Result
```

Types:

```
Result =
    {Res :: integer(),
     Flags :: integer(),
     PtrSubItem :: integer()}
This = wxListCtrl()
Point = {X :: integer(), Y :: integer()}
```

Determines which item (if any) is at the specified point, giving details in `flags`.

Returns index of the item or `wxNOT_FOUND` if no item is at the specified point.

`flags` will be a combination of the following flags:

If `ptrSubItem` is not NULL and the `wxListCtrl` is in the report mode the subitem (or column) number will also be provided. This feature is only available in version 2.7.0 or higher and is currently only implemented under wxMSW and requires at least comctl32.dll of version 4.70 on the host system or the value stored in `ptrSubItem` will be always -1. To compile this feature into wxWidgets library you need to have access to commctrl.h of version 4.70 that is provided by Microsoft.

```
insertColumn(This, Col, Heading) -> integer()
insertColumn(This, Col, Info) -> integer()
```

Types:

```
This = wxListCtrl()
Col = integer()
Info = wxListItem:wxListItem()
```

For report view mode (only), inserts a column.

For more details, see `setItem/5`. Also see `insertColumn/4` overload for a usually more convenient alternative to this method and the description of how the item width is interpreted by this method.

```
insertColumn(This, Col, Heading, Options :: [Option]) -> integer()
```

Types:

```
This = wxListCtrl()
Col = integer()
Heading = unicode:chardata()
Option = {format, integer()} | {width, integer()}
```

For report view mode (only), inserts a column.

Insert a new column in the list control in report view mode at the given position specifying its most common attributes.

Notice that to set the image for the column you need to use `insertColumn/4` overload and specify ? wxLIST_MASK_IMAGE in the item mask.

Return: The index of the inserted column or -1 if adding it failed.

```
insertItem(This, Info) -> integer()
```

Types:

```
This = wxListCtrl()
Info = wxListItem:wxListItem()
```

Inserts an item, returning the index of the new item if successful, -1 otherwise.

```
insertItem(This, Index, ImageIndex) -> integer()
insertItem(This, Index, Label) -> integer()
```

Types:

```
This = wxListCtrl()
Index = integer()
Label = unicode:chardata()
```

Insert a string item.

```
insertItem(This, Index, Label, ImageIndex) -> integer()
```

Types:

```
This = wxListCtrl()
Index = integer()
Label = unicode:chardata()
ImageIndex = integer()
```

Insert an image/string item.

```
refreshItem(This, Item) -> ok
```

Types:

```
This = wxListCtrl()
Item = integer()
```

Redraws the given `item`.

This is only useful for the virtual list controls as without calling this function the displayed value of the item doesn't change even when the underlying data does change.

See: `refreshItems/3`

```
refreshItems(This, ItemFrom, ItemTo) -> ok
```

Types:

```
This = wxListCtrl()
ItemFrom = ItemTo = integer()
```

Redraws the items between `itemFrom` and `itemTo`.

The starting item must be less than or equal to the ending one.

Just as `refreshItem/2` this is only useful for virtual list controls.

```
scrollList(This, Dx, Dy) -> boolean()
```

Types:

```
This = wxListCtrl()
Dx = Dy = integer()
```

Scrolls the list control.

If in icon, small icon or report view mode, `dx` specifies the number of pixels to scroll. If in list view mode, `dx` specifies the number of columns to scroll. `dy` always specifies the number of pixels to scroll vertically.

Note: This method is currently only implemented in the Windows version.

```
setBackgroundColour(This, Col) -> boolean()
```

Types:

```
This = wxListCtrl()
Col = wx:wx_colour()
```

Sets the background colour.

Note that the `wxWindow:getBackgroundColour/1` function of `wxWindow` base class can be used to retrieve the current background colour.

```
setColumn(This, Col, Item) -> boolean()
```

Types:

```
This = wxListCtrl()
Col = integer()
Item = wxListItem:wxListItem()
```

Sets information about this column.

See `setItem/5` for more information.

```
setColumnWidth(This, Col, Width) -> boolean()
```

Types:

```
This = wxListCtrl()
Col = Width = integer()
```

Sets the column width.

`width` can be a width in pixels or `wxLIST_AUTOSIZE` (-1) or `wxLIST_AUTOSIZE_USEHEADER` (-2).

`wxLIST_AUTOSIZE` will resize the column to the length of its longest item.

wxLIST_AUTOSIZE_USEHEADER will resize the column to the length of the header (Win32) or 80 pixels (other platforms).

In small or normal icon view, col must be -1, and the column width is set for all columns.

```
setImageList(This, ImageList, Which) -> ok
```

Types:

```
This = wxListCtrl()
ImageList = wxImageList:wxImageList()
Which = integer()
```

Sets the image list associated with the control.

which is one of wxIMAGE_LIST_NORMAL, wxIMAGE_LIST_SMALL, wxIMAGE_LIST_STATE (the last is unimplemented).

This method does not take ownership of the image list, you have to delete it yourself.

See: assignImageList/3

```
setItem(This, Info) -> boolean()
```

Types:

```
This = wxListCtrl()
Info = wxListItem:wxListItem()
```

Sets the data of an item.

Using the wxListItem's mask and state mask, you can change only selected attributes of a wxListCtrl item.

Return: true if the item was successfully updated or false if the update failed for some reason (e.g. an invalid item index).

```
setItem(This, Index, Column, Label) -> boolean()
```

Types:

```
This = wxListCtrl()
Index = Column = integer()
Label = unicode:chardata()
```

```
setItem(This, Index, Column, Label, Options :: [Option]) ->
          boolean()
```

Types:

```
This = wxListCtrl()
Index = Column = integer()
Label = unicode:chardata()
Option = {imageId, integer()}
```

Sets an item string field at a particular column.

Return: true if the item was successfully updated or false if the update failed for some reason (e.g. an invalid item index).

```
setItemBackgroundColour(This, Item, Col) -> ok
```

Types:

```
This = wxListCtrl()
Item = integer()
Col = wx:wx_colour()
```

Sets the background colour for this item.

This function only works in report view mode. The colour can be retrieved using `getItemBackgroundColour/2`.

```
setItemCount(This, Count) -> ok
```

Types:

```
This = wxListCtrl()
Count = integer()
```

This method can only be used with virtual list controls.

It is used to indicate to the control the number of items it contains. After calling it, the main program should be ready to handle calls to various item callbacks (such as `wxListCtrl::OnGetItemText` (not implemented in wx)) for all items in the range from 0 to `count`.

Notice that the control is not necessarily redrawn after this call as it may be undesirable if an item which is not visible on the screen anyhow was added to or removed from a control displaying many items, if you do need to refresh the display you can just call `wxWindow:refresh/2` manually.

```
setItemData(This, Item, Data) -> boolean()
```

Types:

```
This = wxListCtrl()
Item = Data = integer()
```

Associates application-defined data with this item.

Notice that this function cannot be used to associate pointers with the control items, use `SetItemPtrData()` (not implemented in wx) instead.

```
setItemFont(This, Item, Font) -> ok
```

Types:

```
This = wxListCtrl()
Item = integer()
Font = wxFont:wxFont()
```

Sets the item's font.

```
setItemImage(This, Item, Image) -> boolean()
```

Types:

```
This = wxListCtrl()
Item = Image = integer()
```

```
setItemImage(This, Item, Image, Options :: [Option]) -> boolean()
```

Types:

```
This = wxListCtrl()
Item = Image = integer()
Option = {selImage, integer()}
```

Sets the unselected and selected images associated with the item.

The images are indices into the image list associated with the list control.

```
setItemColumnImage(This, Item, Column, Image) -> boolean()
```

Types:

```
This = wxListCtrl()
Item = Column = Image = integer()
```

Sets the image associated with the item.

In report view, you can specify the column. The image is an index into the image list associated with the list control.

```
setItemPosition(This, Item, Pos) -> boolean()
```

Types:

```
This = wxListCtrl()
Item = integer()
Pos = {X :: integer(), Y :: integer()}
```

Sets the position of the item, in icon or small icon view.

Windows only.

```
setItemState(This, Item, State, StateMask) -> boolean()
```

Types:

```
This = wxListCtrl()
Item = State = StateMask = integer()
```

Sets the item state.

The `stateMask` is a combination of `wxLIST_STATE_XXX` constants described in `wxListItem` documentation. For each of the bits specified in `stateMask`, the corresponding state is set or cleared depending on whether `state` argument contains the same bit or not.

So to select an item you can use while to deselect it you should use

Consider using `wxListView` if possible to avoid dealing with this error-prone and confusing method.

Also notice that contrary to the usual rule that only user actions generate events, this method does generate wxEVT_LIST_ITEM_SELECTED event when it is used to select an item.

```
setItemText(This, Item, Text) -> ok
```

Types:

```
This = wxListCtrl()
Item = integer()
Text = unicode:chardata()
```

Sets the item text for this item.

```
setItemTextColour(This, Item, Col) -> ok
```

Types:

```
This = wxListCtrl()
Item = integer()
Col = wx:wx_colour()
```

Sets the colour for this item.

This function only works in report view. The colour can be retrieved using `getItemTextColour/2`.

```
setSingleStyle(This, Style) -> ok
```

Types:

```
This = wxListCtrl()
Style = integer()
```

```
setSingleStyle(This, Style, Options :: [Option]) -> ok
```

Types:

```
This = wxListCtrl()
Style = integer()
Option = {add, boolean()}
```

Adds or removes a single window style.

```
setTextColour(This, Col) -> ok
```

Types:

```
This = wxListCtrl()
Col = wx:wx_colour()
```

Sets the text colour of the list control.

```
setWindowStyleFlag(This, Style) -> ok
```

Types:

```
This = wxListCtrl()
Style = integer()
```

Sets the whole window style, deleting all items.

```
sortItems(This :: wxListCtrl(), SortCallBack) -> boolean()
```

Types:

```
SortCallBack = fun((integer(), integer()) -> integer())
```

Sort the items in the list control.

Sorts the items with supplied `SortCallBack` fun.

SortCallBack receives the client data associated with two items to compare (`NOT` the index), and should return 0 if the items are equal, a negative value if the first item is less than the second one and a positive value if the first item is greater than the second one.

Remark: Notice that the control may only be sorted on client data associated with the items, so you must use SetItemData if you want to be able to sort the items in the control.

The callback may not call other (wx) processes.

# wxListEvent

Erlang module

A list event holds information about events associated with `wxListCtrl` objects.

See: `wxListCtrl`

This class is derived (and can use functions) from: `wxNotifyEvent` `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxListEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxListEventType` to subscribe to events of this type.

## Data Types

```
wxListEvent() = wx:wx_object()
wxList() =
    #wxList{type = wxListEvent:wxListEventType(),
            code = integer(),
            oldItemIndex = integer(),
            itemIndex = integer(),
            col = integer(),
            pointDrag = {X :: integer(), Y :: integer()}}
wxListEventType() =
    command_list_begin_drag | command_list_begin_rdrag |
    command_list_begin_label_edit | command_list_end_label_edit |
    command_list_delete_item | command_list_delete_all_items |
    command_list_key_down | command_list_insert_item |
    command_list_col_click | command_list_col_right_click |
    command_list_col_begin_drag | command_list_col_dragging |
    command_list_col_end_drag | command_list_item_selected |
    command_list_item_deselected | command_list_item_right_click |
    command_list_item_middle_click | command_list_item_activated |
    command_list_item_focused | command_list_cache_hint
```

## Exports

```
getCacheFrom(This) -> integer()
```

Types:

```
   This = wxListEvent()
```

For `EVT_LIST_CACHE_HINT` event only: return the first item which the list control advises us to cache.

```
getCacheTo(This) -> integer()
```

Types:

```
   This = wxListEvent()
```

For `EVT_LIST_CACHE_HINT` event only: return the last item (inclusive) which the list control advises us to cache.

```
getKeyCode(This) -> integer()
```

Types:

```
This = wxListEvent()
```

Key code if the event is a keypress event.

```
getIndex(This) -> integer()
```

Types:

```
This = wxListEvent()
```

The item index.

```
getColumn(This) -> integer()
```

Types:

```
This = wxListEvent()
```

The column position: it is only used with `COL` events.

For the column dragging events, it is the column to the left of the divider being dragged, for the column click events it may be -1 if the user clicked in the list control header outside any column.

```
getPoint(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxListEvent()
```

The position of the mouse pointer if the event is a drag event.

```
getLabel(This) -> unicode:charlist()
```

Types:

```
This = wxListEvent()
```

The (new) item label for `EVT_LIST_END_LABEL_EDIT` event.

```
getText(This) -> unicode:charlist()
```

Types:

```
This = wxListEvent()
```

The text.

```
getImage(This) -> integer()
```

Types:

```
This = wxListEvent()
```

The image.

```
getData(This) -> integer()
```

Types:

```
This = wxListEvent()
```

The data.

```
getMask(This) -> integer()
```

Types:

```
This = wxListEvent()
```

The mask.

```
getItem(This) -> wxListItem:wxListItem()
```

Types:

```
This = wxListEvent()
```

An item object, used by some events.

See also `wxListCtrl:setItem/5`.

```
isEditCancelled(This) -> boolean()
```

Types:

```
This = wxListEvent()
```

This method only makes sense for `EVT_LIST_END_LABEL_EDIT` message and returns true if it the label editing has been cancelled by the user (`getLabel/1` returns an empty string in this case but it doesn't allow the application to distinguish between really cancelling the edit and the admittedly rare case when the user wants to rename it to an empty string).

# wxListItem

Erlang module

This class stores information about a `wxListCtrl` item or column.

`wxListItem` is a class which contains information about:

The `wxListItem` object can also contain item-specific colour and font information: for this you need to call one of `setTextColour/2`, `setBackgroundColour/2` or `setFont/2` functions on it passing it the colour/font to use. If the colour/font is not specified, the default list control colour/font is used.

See: `wxListCtrl`

wxWidgets docs: **wxListItem**

## Data Types

```
wxListItem() = wx:wx_object()
```

## Exports

```
new() -> wxListItem()
```

Constructor.

```
new(Item) -> wxListItem()
```

Types:

```
   Item = wxListItem()
```

```
clear(This) -> ok
```

Types:

```
   This = wxListItem()
```

Resets the item state to the default.

```
getAlign(This) -> wx:wx_enum()
```

Types:

```
   This = wxListItem()
```

Returns the alignment for this item.

Can be one of `wxLIST_FORMAT_LEFT`, `wxLIST_FORMAT_RIGHT` or `wxLIST_FORMAT_CENTRE`.

```
getBackgroundColour(This) -> wx:wx_colour4()
```

Types:

```
   This = wxListItem()
```

Returns the background colour for this item.

```
getColumn(This) -> integer()
```

Types:

```
This = wxListItem()
```

Returns the zero-based column; meaningful only in report mode.

```
getFont(This) -> wxFont:wxFont()
```

Types:

```
This = wxListItem()
```

Returns the font used to display the item.

```
getId(This) -> integer()
```

Types:

```
This = wxListItem()
```

Returns the zero-based item position.

```
getImage(This) -> integer()
```

Types:

```
This = wxListItem()
```

Returns the zero-based index of the image associated with the item into the image list.

```
getMask(This) -> integer()
```

Types:

```
This = wxListItem()
```

Returns a bit mask indicating which fields of the structure are valid.

Can be any combination of the following values:

```
getState(This) -> integer()
```

Types:

```
This = wxListItem()
```

Returns a bit field representing the state of the item.

Can be any combination of:

```
getText(This) -> unicode:charlist()
```

Types:

```
This = wxListItem()
```

Returns the label/header text.

```
getTextColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxListItem()
```

Returns the text colour.

```
getWidth(This) -> integer()
```

Types:

```
This = wxListItem()
```

Meaningful only for column headers in report mode.

Returns the column width.

`setAlign(This, Align) -> ok`

Types:

```
This = wxListItem()
Align = wx:wx_enum()
```

Sets the alignment for the item.

See also `getAlign/1`

`setBackgroundColour(This, ColBack) -> ok`

Types:

```
This = wxListItem()
ColBack = wx:wx_colour()
```

Sets the background colour for the item.

`setColumn(This, Col) -> ok`

Types:

```
This = wxListItem()
Col = integer()
```

Sets the zero-based column.

Meaningful only in report mode.

`setFont(This, Font) -> ok`

Types:

```
This = wxListItem()
Font = wxFont:wxFont()
```

Sets the font for the item.

`setId(This, Id) -> ok`

Types:

```
This = wxListItem()
Id = integer()
```

Sets the zero-based item position.

`setImage(This, Image) -> ok`

Types:

```
This = wxListItem()
Image = integer()
```

Sets the zero-based index of the image associated with the item into the image list.

```
setMask(This, Mask) -> ok
```

Types:

```
This = wxListItem()
Mask = integer()
```

Sets the mask of valid fields.

See `getMask/1`.

```
setState(This, State) -> ok
```

Types:

```
This = wxListItem()
State = integer()
```

Sets the item state flags (note that the valid state flags are influenced by the value of the state mask, see `setStateMask/2`).

See `getState/1` for valid flag values.

```
setStateMask(This, StateMask) -> ok
```

Types:

```
This = wxListItem()
StateMask = integer()
```

Sets the bitmask that is used to determine which of the state flags are to be set.

See also `setState/2`.

```
setText(This, Text) -> ok
```

Types:

```
This = wxListItem()
Text = unicode:chardata()
```

Sets the text label for the item.

```
setTextColour(This, ColText) -> ok
```

Types:

```
This = wxListItem()
ColText = wx:wx_colour()
```

Sets the text colour for the item.

```
setWidth(This, Width) -> ok
```

Types:

```
This = wxListItem()
Width = integer()
```

Meaningful only for column headers in report mode.

Sets the column width.

```
destroy(This :: wxListItem()) -> ok
```

Destroys the object.

# wxListItemAttr

Erlang module

wxWidgets docs: **wxListItemAttr**

## Data Types

```
wxListItemAttr() = wx:wx_object()
```

## Exports

```
new() -> wxListItemAttr()
```

```
new(ColText, ColBack, Font) -> wxListItemAttr()
```

Types:

```
   ColText = ColBack = wx:wx_colour()
   Font = wxFont:wxFont()
```

```
getBackgroundColour(This) -> wx:wx_colour4()
```

Types:

```
   This = wxListItemAttr()
```

```
getFont(This) -> wxFont:wxFont()
```

Types:

```
   This = wxListItemAttr()
```

```
getTextColour(This) -> wx:wx_colour4()
```

Types:

```
   This = wxListItemAttr()
```

```
hasBackgroundColour(This) -> boolean()
```

Types:

```
   This = wxListItemAttr()
```

```
hasFont(This) -> boolean()
```

Types:

```
   This = wxListItemAttr()
```

```
hasTextColour(This) -> boolean()
```

Types:

```
   This = wxListItemAttr()
```

```
setBackgroundColour(This, ColBack) -> ok
```

Types:

```
   This = wxListItemAttr()
   ColBack = wx:wx_colour()
```

```
setFont(This, Font) -> ok
```

Types:

```
   This = wxListItemAttr()
   Font = wxFont:wxFont()
```

```
setTextColour(This, ColText) -> ok
```

Types:

```
   This = wxListItemAttr()
   ColText = wx:wx_colour()
```

```
destroy(This :: wxListItemAttr()) -> ok
```

Destroys the object.

# wxListView

Erlang module

This class currently simply presents a simpler to use interface for the `wxListCtrl` - it can be thought of as a `façade` for that complicated class.

Using it is preferable to using `wxListCtrl` directly whenever possible because in the future some ports might implement `wxListView` but not the full set of `wxListCtrl` features.

Other than different interface, this class is identical to `wxListCtrl`. In particular, it uses the same events, same window styles and so on.

See: `setColumnImage/3`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxListView**

## Data Types

```
wxListView() = wx:wx_object()
```

## Exports

```
clearColumnImage(This, Col) -> ok
```

Types:

```
This = wxListView()
Col = integer()
```

Resets the column image - after calling this function, no image will be shown.

See: `setColumnImage/3`

```
focus(This, Index) -> ok
```

Types:

```
This = wxListView()
Index = integer()
```

Sets focus to the item with the given `index`.

```
getFirstSelected(This) -> integer()
```

Types:

```
This = wxListView()
```

Returns the first selected item in a (presumably) multiple selection control.

Together with `getNextSelected/2` it can be used to iterate over all selected items in the control.

Return: The first selected item, if any, -1 otherwise.

```
getFocusedItem(This) -> integer()
```

Types:

```
This = wxListView()
```

Returns the currently focused item or -1 if none.

See: `isSelected/2`, `focus/2`

```
getNextSelected(This, Item) -> integer()
```

Types:

```
This = wxListView()
Item = integer()
```

Used together with `getFirstSelected/1` to iterate over all selected items in the control.

Return: Returns the next selected item or -1 if there are no more of them.

```
isSelected(This, Index) -> boolean()
```

Types:

```
This = wxListView()
Index = integer()
```

Returns true if the item with the given `index` is selected, false otherwise.

See: `getFirstSelected/1`, `getNextSelected/2`

```
select(This, N) -> ok
```

Types:

```
This = wxListView()
N = integer()
```

```
select(This, N, Options :: [Option]) -> ok
```

Types:

```
This = wxListView()
N = integer()
Option = {on, boolean()}
```

Selects or unselects the given item.

Notice that this method inherits the unusual behaviour of `wxListCtrl:setItemState/4` which sends a wxEVT_LIST_ITEM_SELECTED event when it is used to select an item, contrary to the usual rule that only the user actions result in selection.

```
setColumnImage(This, Col, Image) -> ok
```

Types:

```
This = wxListView()
Col = Image = integer()
```

Sets the column image for the specified column.

To use the column images, the control must have a valid image list with at least one image.

# wxListbook

Erlang module

`wxListbook` is a class similar to `wxNotebook` but which uses a `wxListCtrl` to show the labels instead of the tabs.

The underlying `wxListCtrl` displays page labels in a one-column report view by default. Calling wxBookCtrl::SetImageList will implicitly switch the control to use an icon view.

For usage documentation of this class, please refer to the base abstract class wxBookCtrl. You can also use the page_samples_notebook to see `wxListbook` in action.

Styles

This class supports the following styles:

See: ?wxBookCtrl, `wxNotebook`, **Examples**

This class is derived (and can use functions) from: `wxBookCtrlBase` `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxListbook**

## Events

Event types emitted from this class: `listbook_page_changed`, `listbook_page_changing`

## Data Types

```
wxListbook() = wx:wx_object()
```

## Exports

```
new() -> wxListbook()
```

Default ctor.

```
new(Parent, Id) -> wxListbook()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxListbook()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Constructs a listbook control.

```
addPage(This, Page, Text) -> boolean()
```

Types:

```
This = wxListbook()
Page = wxWindow:wxWindow()
Text = unicode:chardata()
```

```
addPage(This, Page, Text, Options :: [Option]) -> boolean()
```

Types:

```
This = wxListbook()
Page = wxWindow:wxWindow()
Text = unicode:chardata()
Option = {bSelect, boolean()} | {imageId, integer()}
```

Adds a new page.

The page must have the book control itself as the parent and must not have been added to this control previously.

The call to this function will generate the page changing and page changed events if `select` is true, but not when inserting the very first page (as there is no previous page selection to switch from in this case and so it wouldn't make sense to e.g. veto such event).

Return: true if successful, false otherwise.

Remark: Do not delete the page, it will be deleted by the book control.

See: `insertPage/5`

```
advanceSelection(This) -> ok
```

Types:

```
This = wxListbook()
```

```
advanceSelection(This, Options :: [Option]) -> ok
```

Types:

```
This = wxListbook()
Option = {forward, boolean()}
```

Cycles through the tabs.

The call to this function generates the page changing events.

```
assignImageList(This, ImageList) -> ok
```

Types:

```
This = wxListbook()
ImageList = wxImageList:wxImageList()
```

Sets the image list for the page control and takes ownership of the list.

See: `wxImageList`, `setImageList/2`

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = wxListbook()
   Parent = wxWindow:wxWindow()
   Id = integer()

create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxListbook()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Create the list book control that has already been constructed with the default constructor.

```
deleteAllPages(This) -> boolean()
```

Types:

```
   This = wxListbook()
```

Deletes all pages.

```
getCurrentPage(This) -> wxWindow:wxWindow()
```

Types:

```
   This = wxListbook()
```

Returns the currently selected page or NULL.

```
getImageList(This) -> wxImageList:wxImageList()
```

Types:

```
   This = wxListbook()
```

Returns the associated image list, may be NULL.

See: `wxImageList`, `setImageList/2`

```
getPage(This, Page) -> wxWindow:wxWindow()
```

Types:

```
   This = wxListbook()
   Page = integer()
```

Returns the window at the given page position.

```
getPageCount(This) -> integer()
```

Types:

```
   This = wxListbook()
```

Returns the number of pages in the control.

```
getPageImage(This, NPage) -> integer()
```

Types:

```
This = wxListbook()
NPage = integer()
```

Returns the image index for the given page.

```
getPageText(This, NPage) -> unicode:charlist()
```

Types:

```
This = wxListbook()
NPage = integer()
```

Returns the string for the given page.

```
getSelection(This) -> integer()
```

Types:

```
This = wxListbook()
```

Returns the currently selected page, or `wxNOT_FOUND` if none was selected.

Note that this method may return either the previously or newly selected page when called from the `EVT_BOOKCTRL_PAGE_CHANGED` handler depending on the platform and so `wxBookCtrlEvent:getSelection/1` should be used instead in this case.

```
hitTest(This, Pt) -> Result
```

Types:

```
Result = {Res :: integer(), Flags :: integer()}
This = wxListbook()
Pt = {X :: integer(), Y :: integer()}
```

Returns the index of the tab at the specified position or `wxNOT_FOUND` if none.

If `flags` parameter is non-NULL, the position of the point inside the tab is returned as well.

Return: Returns the zero-based tab index or `wxNOT_FOUND` if there is no tab at the specified position.

```
insertPage(This, Index, Page, Text) -> boolean()
```

Types:

```
This = wxListbook()
Index = integer()
Page = wxWindow:wxWindow()
Text = unicode:chardata()
```

```
insertPage(This, Index, Page, Text, Options :: [Option]) ->
              boolean()
```

Types:

```
This = wxListbook()
Index = integer()
Page = wxWindow:wxWindow()
Text = unicode:chardata()
Option = {bSelect, boolean()} | {imageId, integer()}
```

Inserts a new page at the specified position.

Return: true if successful, false otherwise.

Remark: Do not delete the page, it will be deleted by the book control.

See: `addPage/4`

```
setImageList(This, ImageList) -> ok
```

Types:

```
This = wxListbook()
ImageList = wxImageList:wxImageList()
```

Sets the image list to use.

It does not take ownership of the image list, you must delete it yourself.

See: `wxImageList`, `assignImageList/2`

```
setPageSize(This, Size) -> ok
```

Types:

```
This = wxListbook()
Size = {W :: integer(), H :: integer()}
```

Sets the width and height of the pages.

Note: This method is currently not implemented for wxGTK.

```
setPageImage(This, Page, Image) -> boolean()
```

Types:

```
This = wxListbook()
Page = Image = integer()
```

Sets the image index for the given page.

`image` is an index into the image list which was set with `setImageList/2`.

```
setPageText(This, Page, Text) -> boolean()
```

Types:

```
This = wxListbook()
Page = integer()
Text = unicode:chardata()
```

Sets the text for the given page.

```
setSelection(This, Page) -> integer()
```

Types:

```
This = wxListbook()
Page = integer()
```

Sets the selection to the given page, returning the previous selection.

Notice that the call to this function generates the page changing events, use the `changeSelection/2` function if you don't want these events to be generated.

See: `getSelection/1`

```
changeSelection(This, Page) -> integer()
```

Types:

```
This = wxListbook()
Page = integer()
```

Changes the selection to the given page, returning the previous selection.

This function behaves as `setSelection/2` but does `not` generate the page changing events.

See overview_events_prog for more information.

```
destroy(This :: wxListbook()) -> ok
```

Destroys the object.

# wxLocale

Erlang module

`wxLocale` class encapsulates all language-dependent settings and is a generalization of the C locale concept.

In wxWidgets this class manages current locale. It also initializes and activates `wxTranslations` (not implemented in wx) object that manages message catalogs.

For a list of the supported languages, please see ?wxLanguage enum values. These constants may be used to specify the language in `init/3` and are returned by `getSystemLanguage/0`.

See: **Overview i18n**, **Examples**, `wxXLocale` (not implemented in wx), `wxTranslations` (not implemented in wx)

wxWidgets docs: **wxLocale**

## Data Types

```
wxLocale() = wx:wx_object()
```

## Exports

```
new() -> wxLocale()
```

This is the default constructor and it does nothing to initialize the object: `init/3` must be used to do that.

```
new(Language) -> wxLocale()
new(Name) -> wxLocale()
```

Types:

```
Name = unicode:chardata()
```

```
new(Language, Options :: [Option]) -> wxLocale()
new(Name, Options :: [Option]) -> wxLocale()
```

Types:

```
Name = unicode:chardata()
Option =
    {shortName, unicode:chardata()} |
    {locale, unicode:chardata()} |
    {bLoadDefault, boolean()}
```

See `init/3` for parameters description.

The call of this function has several global side effects which you should understand: first of all, the application locale is changed - note that this will affect many of standard C library functions such as printf() or strftime(). Second, this `wxLocale` object becomes the new current global locale for the application and so all subsequent calls to ? wxGetTranslation() will try to translate the messages using the message catalogs for this locale.

```
destroy(This :: wxLocale()) -> ok
```

The destructor, like the constructor, also has global side effects: the previously set locale is restored and so the changes described in `init/3` documentation are rolled back.

```
init(This) -> boolean()
```
Types:
```
   This = wxLocale()
```

```
init(This, Options :: [Option]) -> boolean()
```
Types:
```
   This = wxLocale()
   Option = {language, integer()} | {flags, integer()}
```
Initializes the `wxLocale` instance.

The call of this function has several global side effects which you should understand: first of all, the application locale is changed - note that this will affect many of standard C library functions such as printf() or strftime(). Second, this `wxLocale` object becomes the new current global locale for the application and so all subsequent calls to ? wxGetTranslation() will try to translate the messages using the message catalogs for this locale.

Return: true on success or false if the given locale couldn't be set.

```
init(This, Name, Options :: [Option]) -> boolean()
```
Types:
```
   This = wxLocale()
   Name = unicode:chardata()
   Option =
       {shortName, unicode:chardata()} |
       {locale, unicode:chardata()} |
       {bLoadDefault, boolean()}
```
Deprecated: This form is deprecated, use the other one unless you know what you are doing.

```
addCatalog(This, Domain) -> boolean()
```
Types:
```
   This = wxLocale()
   Domain = unicode:chardata()
```
Calls wxTranslations::AddCatalog(const wxString&).

```
addCatalog(This, Domain, MsgIdLanguage) -> boolean()
```
Types:
```
   This = wxLocale()
   Domain = unicode:chardata()
   MsgIdLanguage = wx:wx_enum()
```
Calls `wxTranslations::AddCatalog(const wxString&, wxLanguage)` (not implemented in wx).

```
addCatalog(This, Domain, MsgIdLanguage, MsgIdCharset) -> boolean()
```
Types:

```
This = wxLocale()
Domain = unicode:chardata()
MsgIdLanguage = wx:wx_enum()
MsgIdCharset = unicode:chardata()
```

Calls `wxTranslations::AddCatalog(const wxString&, wxLanguage, const wxString&)` (not implemented in wx).

### addCatalogLookupPathPrefix(Prefix) -> ok

Types:

```
Prefix = unicode:chardata()
```

Calls `wxFileTranslationsLoader::AddCatalogLookupPathPrefix()` (not implemented in wx).

### getCanonicalName(This) -> unicode:charlist()

Types:

```
This = wxLocale()
```

Returns the canonical form of current locale name.

Canonical form is the one that is used on UNIX systems: it is a two- or five-letter string in xx or xx_YY format, where xx is ISO 639 code of language and YY is ISO 3166 code of the country. Examples are "en", "en_GB", "en_US" or "fr_FR". This form is internally used when looking up message catalogs. Compare `getSysName/1`.

### getLanguage(This) -> integer()

Types:

```
This = wxLocale()
```

Returns the ?wxLanguage constant of current language.

Note that you can call this function only if you used the form of `init/3` that takes ?wxLanguage argument.

### getLanguageName(Lang) -> unicode:charlist()

Types:

```
Lang = integer()
```

Returns English name of the given language or empty string if this language is unknown.

See `GetLanguageInfo()` (not implemented in wx) for a remark about special meaning of `wxLANGUAGE_DEFAULT`.

### getLocale(This) -> unicode:charlist()

Types:

```
This = wxLocale()
```

Returns the locale name as passed to the constructor or `init/3`.

This is a full, human-readable name, e.g. "English" or "French".

### getName(This) -> unicode:charlist()

Types:

```
This = wxLocale()
```

Returns the current short name for the locale (as given to the constructor or the `init/3` function).

```
getString(This, OrigString) -> unicode:charlist()
```

Types:

```
   This = wxLocale()
   OrigString = unicode:chardata()
```

```
getString(This, OrigString, Options :: [Option]) ->
             unicode:charlist()
```

Types:

```
   This = wxLocale()
   OrigString = unicode:chardata()
   Option = {szDomain, unicode:chardata()}
```

Calls wxGetTranslation(const wxString&, const wxString&).

```
getString(This, OrigString, OrigString2, N) -> unicode:charlist()
```

Types:

```
   This = wxLocale()
   OrigString = OrigString2 = unicode:chardata()
   N = integer()
```

```
getString(This, OrigString, OrigString2, N, Options :: [Option]) ->
             unicode:charlist()
```

Types:

```
   This = wxLocale()
   OrigString = OrigString2 = unicode:chardata()
   N = integer()
   Option = {szDomain, unicode:chardata()}
```

Calls wxGetTranslation(const wxString&, const wxString&, unsigned, const wxString&).

```
getHeaderValue(This, Header) -> unicode:charlist()
```

Types:

```
   This = wxLocale()
   Header = unicode:chardata()
```

```
getHeaderValue(This, Header, Options :: [Option]) ->
                  unicode:charlist()
```

Types:

```
   This = wxLocale()
   Header = unicode:chardata()
   Option = {szDomain, unicode:chardata()}
```

Calls `wxTranslations::GetHeaderValue()` (not implemented in wx).

```
getSysName(This) -> unicode:charlist()
```

Types:

```
This = wxLocale()
```

Returns current platform-specific locale name as passed to setlocale().

Compare `getCanonicalName/1`.

### `getSystemEncoding() -> wx:wx_enum()`

Tries to detect the user's default font encoding.

Returns ?wxFontEncoding() value or `wxFONTENCODING_SYSTEM` if it couldn't be determined.

### `getSystemEncodingName() -> unicode:charlist()`

Tries to detect the name of the user's default font encoding.

This string isn't particularly useful for the application as its form is platform-dependent and so you should probably use `getSystemEncoding/0` instead.

Returns a user-readable string value or an empty string if it couldn't be determined.

### `getSystemLanguage() -> integer()`

Tries to detect the user's default locale setting.

Returns the ?wxLanguage value or `wxLANGUAGE_UNKNOWN` if the language-guessing algorithm failed.

Note: This function works with `locales` and returns the user's default locale. This may be, and usually is, the same as their preferred UI language, but it's not the same thing. Use wxTranslation to obtain `language` information.

See: `wxTranslations::GetBestTranslation()` (not implemented in wx)

### `isLoaded(This, Domain) -> boolean()`

Types:

```
This = wxLocale()
Domain = unicode:chardata()
```

Calls `wxTranslations::IsLoaded()` (not implemented in wx).

### `isOk(This) -> boolean()`

Types:

```
This = wxLocale()
```

Returns true if the locale could be set successfully.

# wxLogNull

Erlang module

This class allows you to temporarily suspend logging. All calls to the log functions during the life time of an object of this class are just ignored.

In particular, it can be used to suppress the log messages given by wxWidgets itself but it should be noted that it is rarely the best way to cope with this problem as `all` log messages are suppressed, even if they indicate a completely different error than the one the programmer wanted to suppress.

For instance, the example of the overview:

would be better written as:

wxWidgets docs: **wxLogNull**

## Data Types

```
wxLogNull() = wx:wx_object()
```

## Exports

```
new() -> wxLogNull()
```

Suspends logging.

```
destroy(This :: wxLogNull()) -> ok
```

Resumes logging.

# wxMDIChildFrame

Erlang module

An MDI child frame is a frame that can only exist inside a `wxMDIClientWindow`, which is itself a child of `wxMDIParentFrame`.

Styles

This class supports the following styles:

All of the standard `wxFrame` styles can be used but most of them are ignored by TDI-based MDI implementations.

Remark: Although internally an MDI child frame is a child of the MDI client window, in wxWidgets you create it as a child of `wxMDIParentFrame`. In fact, you can usually forget that the client window exists. MDI child frames are clipped to the area of the MDI client window, and may be iconized on the client window. You can associate a menubar with a child frame as usual, although an MDI child doesn't display its menubar under its own title bar. The MDI parent frame's menubar will be changed to reflect the currently active child frame. If there are currently no children, the parent frame's own menubar will be displayed.

See: `wxMDIClientWindow`, `wxMDIParentFrame`, `wxFrame`

This class is derived (and can use functions) from: `wxFrame` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxMDIChildFrame**

## Data Types

```
wxMDIChildFrame() = wx:wx_object()
```

## Exports

```
new() -> wxMDIChildFrame()
```

Default constructor.

```
new(Parent, Id, Title) -> wxMDIChildFrame()
```

Types:

```
Parent = wxMDIParentFrame:wxMDIParentFrame()
Id = integer()
Title = unicode:chardata()
```

```
new(Parent, Id, Title, Options :: [Option]) -> wxMDIChildFrame()
```

Types:

```
Parent = wxMDIParentFrame:wxMDIParentFrame()
Id = integer()
Title = unicode:chardata()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Constructor, creating the window.

See: create/5

```
destroy(This :: wxMDIChildFrame()) -> ok
```

Destructor.

Destroys all child windows and menu bar if present.

```
activate(This) -> ok
```

Types:

```
This = wxMDIChildFrame()
```

Activates this MDI child frame.

See: maximize/2, restore/1

```
create(This, Parent, Id, Title) -> boolean()
```

Types:

```
This = wxMDIChildFrame()
Parent = wxMDIParentFrame:wxMDIParentFrame()
Id = integer()
Title = unicode:chardata()
```

```
create(This, Parent, Id, Title, Options :: [Option]) -> boolean()
```

Types:

```
This = wxMDIChildFrame()
Parent = wxMDIParentFrame:wxMDIParentFrame()
Id = integer()
Title = unicode:chardata()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Used in two-step frame construction.

See new/4 for further details.

```
maximize(This) -> ok
```

Types:

```
This = wxMDIChildFrame()
```

```
maximize(This, Options :: [Option]) -> ok
```

Types:

```
This = wxMDIChildFrame()
Option = {maximize, boolean()}
```

Maximizes this MDI child frame.

This function doesn't do anything if IsAlwaysMaximized() (not implemented in wx) returns true.

See: activate/1, restore/1

```
restore(This) -> ok
```

Types:

```
This = wxMDIChildFrame()
```

Restores this MDI child frame (unmaximizes).

This function doesn't do anything if `IsAlwaysMaximized()` (not implemented in wx) returns true.

See: `activate/1`, `maximize/2`

# wxMDIClientWindow

Erlang module

An MDI client window is a child of `wxMDIParentFrame`, and manages zero or more `wxMDIChildFrame` objects.

The client window is the area where MDI child windows exist. It doesn't have to cover the whole parent frame; other windows such as toolbars and a help window might coexist with it. There can be scrollbars on a client window, which are controlled by the parent window style.

The `wxMDIClientWindow` class is usually adequate without further derivation, and it is created automatically when the MDI parent frame is created. If the application needs to derive a new class, the function `wxMDIParentFrame::OnCreateClient()` (not implemented in wx) must be overridden in order to give an opportunity to use a different class of client window.

Under wxMSW, the client window will automatically have a sunken border style when the active child is not maximized, and no border style when a child is maximized.

See: `wxMDIChildFrame`, `wxMDIParentFrame`, `wxFrame`

This class is derived (and can use functions) from: `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxMDIClientWindow**

## Data Types

```
wxMDIClientWindow() = wx:wx_object()
```

## Exports

```
new() -> wxMDIClientWindow()
```

Default constructor.

Objects of this class are only created by `wxMDIParentFrame` which uses the default constructor and calls `createClient/3` immediately afterwards.

```
createClient(This, Parent) -> boolean()
```

Types:

```
This = wxMDIClientWindow()
Parent = wxMDIParentFrame:wxMDIParentFrame()
```

```
createClient(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
This = wxMDIClientWindow()
Parent = wxMDIParentFrame:wxMDIParentFrame()
Option = {style, integer()}
```

Called by `wxMDIParentFrame` immediately after creating the client window.

This function may be overridden in the derived class but the base class version must usually be called first to really create the window.

```
destroy(This :: wxMDIClientWindow()) -> ok
```

Destroys the object.

# wxMDIParentFrame

Erlang module

An MDI (Multiple Document Interface) parent frame is a window which can contain MDI child frames in its client area which emulates the full desktop.

MDI is a user-interface model in which all the window reside inside the single parent window as opposed to being separate from each other. It remains popular despite dire warnings from Microsoft itself (which popularized this model in the first model) that MDI is obsolete.

An MDI parent frame always has a `wxMDIClientWindow` associated with it, which is the parent for MDI child frames. In the simplest case, the client window takes up the entire parent frame area but it is also possible to resize it to be smaller in order to have other windows in the frame, a typical example is using a sidebar along one of the window edges.

The appearance of MDI applications differs between different ports. The classic MDI model, with child windows which can be independently moved, resized etc, is only available under MSW, which provides native support for it. In Mac ports, multiple top level windows are used for the MDI children too and the MDI parent frame itself is invisible, to accommodate the native look and feel requirements. In all the other ports, a tab-based MDI implementation (sometimes called TDI) is used and so at most one MDI child is visible at any moment (child frames are always maximized).

Although it is possible to have multiple MDI parent frames, a typical MDI application has a single MDI parent frame window inside which multiple MDI child frames, i.e. objects of class `wxMDIChildFrame`, can be created.

Styles

This class supports the following styles:

There are no special styles for this class, all `wxFrame` styles apply to it in the usual way. The only exception is that wxHSCROLL and wxVSCROLL styles apply not to the frame itself but to the client window, so that using them enables horizontal and vertical scrollbars for this window and not the frame.

See: `wxMDIChildFrame`, `wxMDIClientWindow`, `wxFrame`, `wxDialog`

This class is derived (and can use functions) from: `wxFrame` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxMDIParentFrame**

## Data Types

```
wxMDIParentFrame() = wx:wx_object()
```

## Exports

```
new() -> wxMDIParentFrame()
```

Default constructor.

Use `create/5` for the objects created using this constructor.

```
new(Parent, Id, Title) -> wxMDIParentFrame()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()

new(Parent, Id, Title, Options :: [Option]) -> wxMDIParentFrame()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructor, creating the window.

Notice that if you override virtual `OnCreateClient()` (not implemented in wx) method you shouldn't be using this constructor but the default constructor and `create/5` as otherwise your overridden method is never going to be called because of the usual C++ virtual call resolution rules.

Under wxMSW, the client window will automatically have a sunken border style when the active child is not maximized, and no border style when a child is maximized.

See: `create/5`, `OnCreateClient()` (not implemented in wx)

```
destroy(This :: wxMDIParentFrame()) -> ok
```

Destructor.

Destroys all child windows and menu bar if present.

```
activateNext(This) -> ok
```

Types:

```
   This = wxMDIParentFrame()
```

Activates the MDI child following the currently active one.

The MDI children are maintained in an ordered list and this function switches to the next element in this list, wrapping around the end of it if the currently active child is the last one.

See: `activatePrevious/1`

```
activatePrevious(This) -> ok
```

Types:

```
   This = wxMDIParentFrame()
```

Activates the MDI child preceding the currently active one.

See: `activateNext/1`

```
arrangeIcons(This) -> ok
```

Types:

```
   This = wxMDIParentFrame()
```

Arranges any iconized (minimized) MDI child windows.

This method is only implemented in MSW MDI implementation and does nothing under the other platforms.

See: `cascade/1`, `tile/2`

```
cascade(This) -> ok
```

Types:

```
This = wxMDIParentFrame()
```

Arranges the MDI child windows in a cascade.

This method is only implemented in MSW MDI implementation and does nothing under the other platforms.

See: `tile/2`, `arrangeIcons/1`

```
create(This, Parent, Id, Title) -> boolean()
```

Types:

```
This = wxMDIParentFrame()
Parent = wxWindow:wxWindow()
Id = integer()
Title = unicode:chardata()
```

```
create(This, Parent, Id, Title, Options :: [Option]) -> boolean()
```

Types:

```
This = wxMDIParentFrame()
Parent = wxWindow:wxWindow()
Id = integer()
Title = unicode:chardata()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Used in two-step frame construction.

See `new/4` for further details.

```
getActiveChild(This) -> wxMDIChildFrame:wxMDIChildFrame()
```

Types:

```
This = wxMDIParentFrame()
```

Returns a pointer to the active MDI child, if there is one.

If there are any children at all this function returns a non-NULL pointer.

```
getClientWindow(This) -> wxMDIClientWindow:wxMDIClientWindow()
```

Types:

```
This = wxMDIParentFrame()
```

Returns a pointer to the client window.

See: `OnCreateClient()` (not implemented in wx)

```
tile(This) -> ok
```

Types:

```
This = wxMDIParentFrame()
```

```
tile(This, Options :: [Option]) -> ok
```

Types:

```
This = wxMDIParentFrame()
Option = {orient, wx:wx_enum()}
```

Tiles the MDI child windows either horizontally or vertically depending on whether `orient` is `wxHORIZONTAL` or `wxVERTICAL`.

This method is only implemented in MSW MDI implementation and does nothing under the other platforms.

# wxMask

Erlang module

This class encapsulates a monochrome mask bitmap, where the masked area is black and the unmasked area is white.

When associated with a bitmap and drawn in a device context, the unmasked area of the bitmap will be drawn, and the masked area will not be drawn.

Note: A mask can be associated also with a bitmap with an alpha channel but drawing such bitmaps under wxMSW may be slow so using them should be avoided if drawing performance is an important factor.

See: `wxBitmap`, `wxDC:blit/6`, `wxMemoryDC`

wxWidgets docs: **wxMask**

## Data Types

```
wxMask() = wx:wx_object()
```

## Exports

```
new() -> wxMask()
```

Default constructor.

```
new(Bitmap) -> wxMask()
```

Types:

```
Bitmap = wxBitmap:wxBitmap()
```

Constructs a mask from a monochrome bitmap.

```
new(Bitmap, Index) -> wxMask()
new(Bitmap, Colour) -> wxMask()
```

Types:

```
Bitmap = wxBitmap:wxBitmap()
Colour = wx:wx_colour()
```

Constructs a mask from a bitmap and a colour that indicates the background.

```
destroy(This :: wxMask()) -> ok
```

Destroys the `wxMask` object and the underlying bitmap data.

```
create(This, Bitmap) -> boolean()
```

Types:

```
This = wxMask()
Bitmap = wxBitmap:wxBitmap()
```

Constructs a mask from a monochrome bitmap.

```
create(This, Bitmap, Index) -> boolean()
create(This, Bitmap, Colour) -> boolean()
```

Types:

```
This = wxMask()
Bitmap = wxBitmap:wxBitmap()
Colour = wx:wx_colour()
```

Constructs a mask from a bitmap and a colour that indicates the background.

# wxMaximizeEvent

Erlang module

An event being sent when a top level window is maximized. Notice that it is not sent when the window is restored to its original size after it had been maximized, only a normal `wxSizeEvent` is generated in this case.

Currently this event is only generated in wxMSW, wxGTK and wxOSX/Cocoa ports so portable programs should only rely on receiving `wxEVT_SIZE` and not necessarily this event when the window is maximized.

See: **Overview events**, `wxTopLevelWindow:maximize/2`, `wxTopLevelWindow:isMaximized/1`

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxMaximizeEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxMaximizeEventType` to subscribe to events of this type.

## Data Types

```
wxMaximizeEvent() = wx:wx_object()
wxMaximize() =
    #wxMaximize{type = wxMaximizeEvent:wxMaximizeEventType()}
wxMaximizeEventType() = maximize
```

# wxMemoryDC

Erlang module

A memory device context provides a means to draw graphics onto a bitmap. When drawing in to a mono-bitmap, using `wxWHITE`, `wxWHITE_PEN` and `wxWHITE_BRUSH` will draw the background colour (i.e. 0) whereas all other colours will draw the foreground colour (i.e. 1).

A bitmap must be selected into the new memory DC before it may be used for anything. Typical usage is as follows:

Note that the memory DC must be deleted (or the bitmap selected out of it) before a bitmap can be reselected into another memory DC.

And, before performing any other operations on the bitmap data, the bitmap must be selected out of the memory DC:

This happens automatically when `wxMemoryDC` object goes out of scope.

See: `wxBitmap`, `wxDC`

This class is derived (and can use functions) from: `wxDC`

wxWidgets docs: **wxMemoryDC**

## Data Types

```
wxMemoryDC() = wx:wx_object()
```

## Exports

```
new() -> wxMemoryDC()
```

Constructs a new memory device context.

Use the `wxDC:isOk/1` member to test whether the constructor was successful in creating a usable device context. Don't forget to select a bitmap into the DC before drawing on it.

```
new(Dc) -> wxMemoryDC()
```

Types:

```
Dc = wxDC:wxDC() | wxBitmap:wxBitmap()
```

Constructs a new memory device context having the same characteristics as the given existing device context.

This constructor creates a memory device context `compatible` with `dc` in wxMSW, the argument is ignored in the other ports. If `dc` is NULL, a device context compatible with the screen is created, just as with the default constructor.

```
selectObject(This, Bitmap) -> ok
```

Types:

```
This = wxMemoryDC()
Bitmap = wxBitmap:wxBitmap()
```

Works exactly like `selectObjectAsSource/2` but this is the function you should use when you select a bitmap because you want to modify it, e.g.

drawing on this DC.

Using `selectObjectAsSource/2` when modifying the bitmap may incur some problems related to `wxBitmap` being a reference counted object (see overview_refcount).

Before using the updated bitmap data, make sure to select it out of context first either by selecting ?wxNullBitmap into the device context or destroying the device context entirely.

If the bitmap is already selected in this device context, nothing is done. If it is selected in another context, the function asserts and drawing on the bitmap won't work correctly.

See: `wxDC:drawBitmap/4`

```
selectObjectAsSource(This, Bitmap) -> ok
```

Types:

```
This = wxMemoryDC()
Bitmap = wxBitmap:wxBitmap()
```

Selects the given bitmap into the device context, to use as the memory bitmap.

Selecting the bitmap into a memory DC allows you to draw into the DC (and therefore the bitmap) and also to use `wxDC:blit/6` to copy the bitmap to a window. For this purpose, you may find `wxDC:drawIcon/3` easier to use instead.

If the argument is ?wxNullBitmap (or some other uninitialised `wxBitmap`) the current bitmap is selected out of the device context, and the original bitmap restored, allowing the current bitmap to be destroyed safely.

```
destroy(This :: wxMemoryDC()) -> ok
```

Destroys the object.

# wxMenu

Erlang module

A menu is a popup (or pull down) list of items, one of which may be selected before the menu goes away (clicking elsewhere dismisses the menu). Menus may be used to construct either menu bars or popup menus.

A menu item has an integer ID associated with it which can be used to identify the selection, or to change the menu item in some way. A menu item with a special identifier `wxID_SEPARATOR` is a separator item and doesn't have an associated command but just makes a separator line appear in the menu.

Note: Please note that `wxID_ABOUT` and `wxID_EXIT` are predefined by wxWidgets and have a special meaning since entries using these IDs will be taken out of the normal menus under macOS and will be inserted into the system menu (following the appropriate macOS interface guideline).

Menu items may be either `normal` items, `check` items or `radio` items. Normal items don't have any special properties while the check items have a boolean flag associated to them and they show a checkmark in the menu when the flag is set. wxWidgets automatically toggles the flag value when the item is clicked and its value may be retrieved using either `isChecked/2` method of `wxMenu` or `wxMenuBar` itself or by using wxEvent::IsChecked when you get the menu notification for the item in question.

The radio items are similar to the check items except that all the other items in the same radio group are unchecked when a radio item is checked. The radio group is formed by a contiguous range of radio items, i.e. it starts at the first item of this kind and ends with the first item of a different kind (or the end of the menu). Notice that because the radio groups are defined in terms of the item positions inserting or removing the items in the menu containing the radio items risks to not work correctly.

Allocation strategy

All menus must be created on the `heap` because all menus attached to a menubar or to another menu will be deleted by their parent when it is deleted. The only exception to this rule are the popup menus (i.e. menus used with `wxWindow:popupMenu/4`) as wxWidgets does not destroy them to allow reusing the same menu more than once. But the exception applies only to the menus themselves and not to any submenus of popup menus which are still destroyed by wxWidgets as usual and so must be heap-allocated.

As the frame menubar is deleted by the frame itself, it means that normally all menus used are deleted automatically.

Event handling

Event handlers for the commands generated by the menu items can be connected directly to the menu object itself using `wxEvtHandler::Bind()` (not implemented in wx). If this menu is a submenu of another one, the events from its items can also be processed in the parent menu and so on, recursively.

If the menu is part of a menu bar, then events can also be handled in `wxMenuBar` object.

Finally, menu events can also be handled in the associated window, which is either the `wxFrame` associated with the menu bar this menu belongs to or the window for which `wxWindow:popupMenu/4` was called for the popup menus.

See overview_events_bind for how to bind event handlers to the various objects.

See: `wxMenuBar`, `wxWindow:popupMenu/4`, **Overview events**, `wxFileHistory` (not implemented in wx)

This class is derived (and can use functions) from: `wxEvtHandler`

wxWidgets docs: **wxMenu**

## Data Types

```
wxMenu() = wx:wx_object()
```

## Exports

```
new() -> wxMenu()
```

Constructs a wxMenu object.

```
new(Options :: [Option]) -> wxMenu()
```

Types:

```
Option = {style, integer()}
```

Constructs a wxMenu object.

```
new(Title, Options :: [Option]) -> wxMenu()
```

Types:

```
Title = unicode:chardata()
Option = {style, integer()}
```

Constructs a wxMenu object with a title.

```
destroy(This :: wxMenu()) -> ok
```

Destructor, destroying the menu.

Note: Under Motif, a popup menu must have a valid parent (the window it was last popped up on) when being destroyed. Therefore, make sure you delete or re-use the popup menu `before` destroying the parent window. Re-use in this context means popping up the menu on a different window from last time, which causes an implicit destruction and recreation of internal data structures.

```
append(This, MenuItem) -> wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
MenuItem = wxMenuItem:wxMenuItem()
```

Adds a menu item object.

This is the most generic variant of `append/5` method because it may be used for both items (including separators) and submenus and because you can also specify various extra properties of a menu item this way, such as bitmaps and fonts.

Remark: See the remarks for the other `append/5` overloads.

See: `appendSeparator/1`, `appendCheckItem/4`, `appendRadioItem/4`, `AppendSubMenu()` (not implemented in wx), `insert/6`, `setLabel/3`, `getHelpString/2`, `setHelpString/3`, `wxMenuItem`

```
append(This, Id, Item) -> wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()

append(This, Id, Item, SubMenu) -> wxMenuItem:wxMenuItem()
append(This, Id, Item, SubMenu :: [Option]) ->
          wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()
   Option = {help, unicode:chardata()} | {kind, wx:wx_enum()}
```

Adds a menu item.

Example:

or even better for stock menu items (see `wxMenuItem:new/1`):

Remark: This command can be used after the menu has been shown, as well as on initial creation of a menu or menubar.

See: `appendSeparator/1`, `appendCheckItem/4`, `appendRadioItem/4`, `AppendSubMenu()` (not implemented in wx), `insert/6`, `setLabel/3`, `getHelpString/2`, `setHelpString/3`, `wxMenuItem`

```
append(This, Id, Item, SubMenu, Options :: [Option]) ->
          wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()
   SubMenu = wxMenu()
   Option = {help, unicode:chardata()}
```

Adds a submenu.

Deprecated: This function is deprecated, use `AppendSubMenu()` (not implemented in wx) instead.

See: `appendSeparator/1`, `appendCheckItem/4`, `appendRadioItem/4`, `AppendSubMenu()` (not implemented in wx), `insert/6`, `setLabel/3`, `getHelpString/2`, `setHelpString/3`, `wxMenuItem`

```
appendCheckItem(This, Id, Item) -> wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()

appendCheckItem(This, Id, Item, Options :: [Option]) ->
                   wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()
   Option = {help, unicode:chardata()}
```

Adds a checkable item to the end of the menu.

See: `append/5`, `insertCheckItem/5`

```
appendRadioItem(This, Id, Item) -> wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()
```

```
appendRadioItem(This, Id, Item, Options :: [Option]) ->
                  wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()
   Option = {help, unicode:chardata()}
```

Adds a radio item to the end of the menu.

All consequent radio items form a group and when an item in the group is checked, all the others are automatically unchecked.

Note: Radio items are not supported under wxMotif.

See: `append/5`, `insertRadioItem/5`

```
appendSeparator(This) -> wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
```

Adds a separator to the end of the menu.

See: `append/5`, `insertSeparator/2`

```
break(This) -> ok
```

Types:

```
   This = wxMenu()
```

Inserts a break in a menu, causing the next appended item to appear in a new column.

This function only actually inserts a break in wxMSW and does nothing under the other platforms.

```
check(This, Id, Check) -> ok
```

Types:

```
This = wxMenu()
Id = integer()
Check = boolean()
```

Checks or unchecks the menu item.

See: `isChecked/2`

```
delete(This, Id) -> boolean()
delete(This, Item) -> boolean()
```

Types:

```
This = wxMenu()
Item = wxMenuItem:wxMenuItem()
```

Deletes the menu item from the menu.

If the item is a submenu, it will `not` be deleted. Use `'Destroy'/2` if you want to delete a submenu.

See: `findItem/2`, `'Destroy'/2`, `remove/2`

```
'Destroy'(This, Id) -> boolean()
'Destroy'(This, Item) -> boolean()
```

Types:

```
This = wxMenu()
Item = wxMenuItem:wxMenuItem()
```

Deletes the menu item from the menu.

If the item is a submenu, it will be deleted. Use `remove/2` if you want to keep the submenu (for example, to reuse it later).

See: `findItem/2`, `delete/2`, `remove/2`

```
enable(This, Id, Enable) -> ok
```

Types:

```
This = wxMenu()
Id = integer()
Enable = boolean()
```

Enables or disables (greys out) a menu item.

See: `isEnabled/2`

```
findItem(This, Id) -> wxMenuItem:wxMenuItem()
findItem(This, ItemString) -> integer()
```

Types:

```
This = wxMenu()
ItemString = unicode:chardata()
```

Finds the menu id for a menu item string.

Return: Menu item identifier, or wxNOT_FOUND if none is found.

Remark: Any special menu codes are stripped out of source and target strings before matching.

```
findItemByPosition(This, Position) -> wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
Position = integer()
```

Returns the `wxMenuItem` given a position in the menu.

```
getHelpString(This, Id) -> unicode:charlist()
```

Types:

```
This = wxMenu()
Id = integer()
```

Returns the help string associated with a menu item.

Return: The help string, or the empty string if there is no help string or the item was not found.

See: `setHelpString/3`, `append/5`

```
getLabel(This, Id) -> unicode:charlist()
```

Types:

```
This = wxMenu()
Id = integer()
```

Returns a menu item label.

Return: The item label, or the empty string if the item was not found.

See: `GetLabelText()` (not implemented in wx), `setLabel/3`

```
getMenuItemCount(This) -> integer()
```

Types:

```
This = wxMenu()
```

Returns the number of items in the menu.

```
getMenuItems(This) -> [wxMenuItem:wxMenuItem()]
```

Types:

```
This = wxMenu()
```

```
getTitle(This) -> unicode:charlist()
```

Types:

```
This = wxMenu()
```

Returns the title of the menu.

See: `setTitle/2`

```
insert(This, Pos, Id) -> wxMenuItem:wxMenuItem()
insert(This, Pos, MenuItem) -> wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
Pos = integer()
MenuItem = wxMenuItem:wxMenuItem()
```

Inserts the given `item` before the position `pos`.

Inserting the item at position `getMenuItemCount/1` is the same as appending it.

See: `append/5`, `prepend/5`

```
insert(This, Pos, Id, Options :: [Option]) ->
          wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
Pos = Id = integer()
Option =
    {text, unicode:chardata()} |
    {help, unicode:chardata()} |
    {kind, wx:wx_enum()}
```

Inserts the given `item` before the position `pos`.

Inserting the item at position `getMenuItemCount/1` is the same as appending it.

See: `append/5`, `prepend/5`

```
insert(This, Pos, Id, Text, Submenu) -> wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
Pos = Id = integer()
Text = unicode:chardata()
Submenu = wxMenu()
```

```
insert(This, Pos, Id, Text, Submenu, Options :: [Option]) ->
          wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
Pos = Id = integer()
Text = unicode:chardata()
Submenu = wxMenu()
Option = {help, unicode:chardata()}
```

Inserts the given `submenu` before the position `pos`.

`text` is the text shown in the menu for it and `help` is the help string shown in the status bar when the submenu item is selected.

See: `AppendSubMenu()` (not implemented in wx), `prepend/5`

```
insertCheckItem(This, Pos, Id, Item) -> wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Pos = Id = integer()
   Item = unicode:chardata()

insertCheckItem(This, Pos, Id, Item, Options :: [Option]) ->
                   wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Pos = Id = integer()
   Item = unicode:chardata()
   Option = {help, unicode:chardata()}
```

Inserts a checkable item at the given position.

See: `insert/6`, `appendCheckItem/4`

```
insertRadioItem(This, Pos, Id, Item) -> wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Pos = Id = integer()
   Item = unicode:chardata()

insertRadioItem(This, Pos, Id, Item, Options :: [Option]) ->
                   wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Pos = Id = integer()
   Item = unicode:chardata()
   Option = {help, unicode:chardata()}
```

Inserts a radio item at the given position.

See: `insert/6`, `appendRadioItem/4`

```
insertSeparator(This, Pos) -> wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Pos = integer()
```

Inserts a separator at the given position.

See: `insert/6`, `appendSeparator/1`

```
isChecked(This, Id) -> boolean()
```

Types:

```
   This = wxMenu()
   Id = integer()
```

Determines whether a menu item is checked.

Return: true if the menu item is checked, false otherwise.

See: check/3

```
isEnabled(This, Id) -> boolean()
```

Types:

```
This = wxMenu()
Id = integer()
```

Determines whether a menu item is enabled.

Return: true if the menu item is enabled, false otherwise.

See: enable/3

```
prepend(This, Id) -> wxMenuItem:wxMenuItem()
prepend(This, Item) -> wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
Item = wxMenuItem:wxMenuItem()
```

Inserts the given item at position 0, i.e. before all the other existing items.

See: append/5, insert/6

```
prepend(This, Id, Options :: [Option]) -> wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
Id = integer()
Option =
    {text, unicode:chardata()} |
    {help, unicode:chardata()} |
    {kind, wx:wx_enum()}
```

Inserts the given item at position 0, i.e. before all the other existing items.

See: append/5, insert/6

```
prepend(This, Id, Text, Submenu) -> wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
Id = integer()
Text = unicode:chardata()
Submenu = wxMenu()
```

```
prepend(This, Id, Text, Submenu, Options :: [Option]) ->
          wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Text = unicode:chardata()
   Submenu = wxMenu()
   Option = {help, unicode:chardata()}
```

Inserts the given `submenu` at position 0.

See: `AppendSubMenu()` (not implemented in wx), `insert/6`

```
prependCheckItem(This, Id, Item) -> wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()
```

```
prependCheckItem(This, Id, Item, Options :: [Option]) ->
                    wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()
   Option = {help, unicode:chardata()}
```

Inserts a checkable item at position 0.

See: `prepend/5`, `appendCheckItem/4`

```
prependRadioItem(This, Id, Item) -> wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()
```

```
prependRadioItem(This, Id, Item, Options :: [Option]) ->
                    wxMenuItem:wxMenuItem()
```

Types:

```
   This = wxMenu()
   Id = integer()
   Item = unicode:chardata()
   Option = {help, unicode:chardata()}
```

Inserts a radio item at position 0.

See: `prepend/5`, `appendRadioItem/4`

```
prependSeparator(This) -> wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
```

Inserts a separator at position 0.

See: `prepend/5`, `appendSeparator/1`

```
remove(This, Id) -> wxMenuItem:wxMenuItem()
remove(This, Item) -> wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenu()
Item = wxMenuItem:wxMenuItem()
```

Removes the menu item from the menu but doesn't delete the associated C++ object.

This allows you to reuse the same item later by adding it back to the menu (especially useful with submenus).

Return: A pointer to the item which was detached from the menu.

```
setHelpString(This, Id, HelpString) -> ok
```

Types:

```
This = wxMenu()
Id = integer()
HelpString = unicode:chardata()
```

Sets an item's help string.

See: `getHelpString/2`

```
setLabel(This, Id, Label) -> ok
```

Types:

```
This = wxMenu()
Id = integer()
Label = unicode:chardata()
```

Sets the label of a menu item.

See: `append/5`, `getLabel/2`

```
setTitle(This, Title) -> ok
```

Types:

```
This = wxMenu()
Title = unicode:chardata()
```

Sets the title of the menu.

Remark: Notice that you can only call this method directly for the popup menus, to change the title of a menu that is part of a menu bar you need to use `wxMenuBar:setLabelTop/3`.

See: `getTitle/1`

# wxMenuBar

Erlang module

A menu bar is a series of menus accessible from the top of a frame.

Remark: To respond to a menu selection, provide a handler for EVT_MENU, in the frame that contains the menu bar.

If you have a toolbar which uses the same identifiers as your EVT_MENU entries, events from the toolbar will also be processed by your EVT_MENU event handlers.

Tip: under Windows, if you discover that menu shortcuts (for example, Alt+F to show the file menu) are not working, check any EVT_CHAR events you are handling in child windows. If you are not calling event.Skip() for events that you don't process in these event handlers, menu shortcuts may cease to work.

See: `wxMenu`, **Overview events**

This class is derived (and can use functions) from: `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxMenuBar**

## Data Types

```
wxMenuBar() = wx:wx_object()
```

## Exports

```
new() -> wxMenuBar()
```

Construct an empty menu bar.

```
new(Style) -> wxMenuBar()
```

Types:

```
   Style = integer()
```

```
destroy(This :: wxMenuBar()) -> ok
```

Destructor, destroying the menu bar and removing it from the parent frame (if any).

```
append(This, Menu, Title) -> boolean()
```

Types:

```
   This = wxMenuBar()
   Menu = wxMenu:wxMenu()
   Title = unicode:chardata()
```

Adds the item to the end of the menu bar.

Return: true on success, false if an error occurred.

See: `insert/4`

```
check(This, Id, Check) -> ok
```

Types:

```
This = wxMenuBar()
Id = integer()
Check = boolean()
```

Checks or unchecks a menu item.

Remark: Only use this when the menu bar has been associated with a frame; otherwise, use the `wxMenu` equivalent call.

```
enable(This, Id, Enable) -> ok
```

Types:

```
This = wxMenuBar()
Id = integer()
Enable = boolean()
```

Enables or disables (greys out) a menu item.

Remark: Only use this when the menu bar has been associated with a frame; otherwise, use the `wxMenu` equivalent call.

```
enableTop(This, Pos, Enable) -> ok
```

Types:

```
This = wxMenuBar()
Pos = integer()
Enable = boolean()
```

Enables or disables a whole menu.

Remark: Only use this when the menu bar has been associated with a frame.

```
findMenu(This, Title) -> integer()
```

Types:

```
This = wxMenuBar()
Title = unicode:chardata()
```

Returns the index of the menu with the given `title` or `wxNOT_FOUND` if no such menu exists in this menubar.

The `title` parameter may specify either the menu title (with accelerator characters, i.e. `"&File"`) or just the menu label (`"File"`) indifferently.

```
findMenuItem(This, MenuString, ItemString) -> integer()
```

Types:

```
This = wxMenuBar()
MenuString = ItemString = unicode:chardata()
```

Finds the menu item id for a menu name/menu item string pair.

Return: The menu item identifier, or wxNOT_FOUND if none was found.

Remark: Any special menu codes are stripped out of source and target strings before matching.

```
findItem(This, Id) -> wxMenuItem:wxMenuItem()
```

Types:

```
This = wxMenuBar()
Id = integer()
```

Finds the menu item object associated with the given menu item identifier.

Return: The found menu item object, or NULL if one was not found.

```
getHelpString(This, Id) -> unicode:charlist()
```

Types:

```
This = wxMenuBar()
Id = integer()
```

Gets the help string associated with the menu item identifier.

Return: The help string, or the empty string if there was no help string or the menu item was not found.

See: `setHelpString/3`

```
getLabel(This, Id) -> unicode:charlist()
```

Types:

```
This = wxMenuBar()
Id = integer()
```

Gets the label associated with a menu item.

Return: The menu item label, or the empty string if the item was not found.

Remark: Use only after the menubar has been associated with a frame.

```
getLabelTop(This, Pos) -> unicode:charlist()
```

Types:

```
This = wxMenuBar()
Pos = integer()
```

See: `getMenuLabel/2`.

```
getMenuLabel(This, Pos) -> unicode:charlist()
```

Types:

```
This = wxMenuBar()
Pos = integer()
```

Returns the label of a top-level menu.

Note that the returned string includes the accelerator characters that have been specified in the menu title string during its construction.

Return: The menu label, or the empty string if the menu was not found.

Remark: Use only after the menubar has been associated with a frame.

See: `getMenuLabelText/2`, `setMenuLabel/3`

```
getMenuLabelText(This, Pos) -> unicode:charlist()
```

Types:

```
This = wxMenuBar()
Pos = integer()
```

Returns the label of a top-level menu.

Note that the returned string does not include any accelerator characters that may have been specified in the menu title string during its construction.

Return: The menu label, or the empty string if the menu was not found.

Remark: Use only after the menubar has been associated with a frame.

See: `getMenuLabel/2`, `setMenuLabel/3`

```
getMenu(This, MenuIndex) -> wxMenu:wxMenu()
```

Types:

```
This = wxMenuBar()
MenuIndex = integer()
```

Returns the menu at `menuIndex` (zero-based).

```
getMenuCount(This) -> integer()
```

Types:

```
This = wxMenuBar()
```

Returns the number of menus in this menubar.

```
insert(This, Pos, Menu, Title) -> boolean()
```

Types:

```
This = wxMenuBar()
Pos = integer()
Menu = wxMenu:wxMenu()
Title = unicode:chardata()
```

Inserts the menu at the given position into the menu bar.

Inserting menu at position 0 will insert it in the very beginning of it, inserting at position `getMenuCount/1` is the same as calling `append/3`.

Return: true on success, false if an error occurred.

See: `append/3`

```
isChecked(This, Id) -> boolean()
```

Types:

```
This = wxMenuBar()
Id = integer()
```

Determines whether an item is checked.

Return: true if the item was found and is checked, false otherwise.

```
setAutoWindowMenu(Enable) -> ok
```

Types:

```
   Enable = boolean()
```

```
getAutoWindowMenu() -> boolean()
```

```
oSXGetAppleMenu(This) -> wxMenu:wxMenu()
```

Types:

```
   This = wxMenuBar()
```

Returns the Apple menu.

This is the leftmost menu with application's name as its title. You shouldn't remove any items from it, but it is safe to insert extra menu items or submenus into it.

Only for:wxosx

Since: 3.0.1

```
macGetCommonMenuBar() -> wxMenuBar()
```

Enables you to get the global menubar on Mac, that is, the menubar displayed when the app is running without any frames open.

Return: The global menubar.

Remark: Only exists on Mac, other platforms do not have this method.

Only for:wxosx

```
macSetCommonMenuBar(Menubar) -> ok
```

Types:

```
   Menubar = wxMenuBar()
```

Enables you to set the global menubar on Mac, that is, the menubar displayed when the app is running without any frames open.

Remark: Only exists on Mac, other platforms do not have this method.

Only for:wxosx

```
isEnabled(This, Id) -> boolean()
```

Types:

```
   This = wxMenuBar()
   Id = integer()
```

Determines whether an item is enabled.

Return: true if the item was found and is enabled, false otherwise.

```
remove(This, Pos) -> wxMenu:wxMenu()
```

Types:

```
   This = wxMenuBar()
   Pos = integer()
```

Removes the menu from the menu bar and returns the menu object - the caller is responsible for deleting it.

This function may be used together with `insert/4` to change the menubar dynamically.

See: `replace/4`

```
replace(This, Pos, Menu, Title) -> wxMenu:wxMenu()
```

Types:

```
This = wxMenuBar()
Pos = integer()
Menu = wxMenu:wxMenu()
Title = unicode:chardata()
```

Replaces the menu at the given position with another one.

Return: The menu which was previously at position pos. The caller is responsible for deleting it.

See: `insert/4`, `remove/2`

```
setHelpString(This, Id, HelpString) -> ok
```

Types:

```
This = wxMenuBar()
Id = integer()
HelpString = unicode:chardata()
```

Sets the help string associated with a menu item.

See: `getHelpString/2`

```
setLabel(This, Id, Label) -> ok
```

Types:

```
This = wxMenuBar()
Id = integer()
Label = unicode:chardata()
```

Sets the label of a menu item.

Remark: Use only after the menubar has been associated with a frame.

See: `getLabel/2`

```
setLabelTop(This, Pos, Label) -> ok
```

Types:

```
This = wxMenuBar()
Pos = integer()
Label = unicode:chardata()
```

See: `setMenuLabel/3`.

```
setMenuLabel(This, Pos, Label) -> ok
```

Types:

```
This = wxMenuBar()
Pos = integer()
Label = unicode:chardata()
```

Sets the label of a top-level menu.

Remark: Use only after the menubar has been associated with a frame.

# wxMenuEvent

Erlang module

This class is used for a variety of menu-related events. Note that these do not include menu command events, which are handled using `wxCommandEvent` objects.

Events of this class are generated by both menus that are part of a `wxMenuBar`, attached to `wxFrame`, and popup menus shown by `wxWindow:popupMenu/4`. They are sent to the following objects until one of them handles the event: `-# The menu object itself, as returned by GetMenu(), if any. -# The wxMenuBar to which this menu is attached, if any. -# The window associated with the menu, e.g. the one calling PopupMenu() for the popup menus. -# The top level parent of that window if it's different from the window itself.`

This is similar to command events generated by the menu items, but, unlike them, `wxMenuEvent` are only sent to the window itself and its top level parent but not any intermediate windows in the hierarchy.

The default handler for `wxEVT_MENU_HIGHLIGHT` in `wxFrame` displays help text in the status bar, see `wxFrame:setStatusBarPane/2`.

See: `wxCommandEvent`, **Overview events**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxMenuEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxMenuEventType` to subscribe to events of this type.

## Data Types

```
wxMenuEvent() = wx:wx_object()
wxMenu() =
    #wxMenu{type = wxMenuEvent:wxMenuEventType(),
            menuId = integer(),
            menu = wxMenu:wxMenu()}
wxMenuEventType() = menu_open | menu_close | menu_highlight
```

## Exports

```
getMenu(This) -> wxMenu:wxMenu()
```

Types:

```
This = wxMenuEvent()
```

Returns the menu which is being opened or closed, or the menu containing the highlighted item.

Note that the returned value can be NULL if the menu being opened doesn't have a corresponding `wxMenu`, e.g. this happens when opening the system menu in wxMSW port.

Remark: Since 3.1.3 this function can be used with `OPEN`, `CLOSE` and `HIGHLIGHT` events. Before 3.1.3, this method can only be used with the `OPEN` and `CLOSE` events.

```
getMenuId(This) -> integer()
```

Types:

```
This = wxMenuEvent()
```

Returns the menu identifier associated with the event.

This method should be only used with the `HIGHLIGHT` events.

```
isPopup(This) -> boolean()
```

Types:

```
This = wxMenuEvent()
```

Returns true if the menu which is being opened or closed is a popup menu, false if it is a normal one.

This method should only be used with the `OPEN` and `CLOSE` events.

# wxMenuItem

Erlang module

A menu item represents an item in a menu.

Note that you usually don't have to deal with it directly as `wxMenu` methods usually construct an object of this class for you.

Also please note that the methods related to fonts and bitmaps are currently only implemented for Windows, Mac and GTK+.

See: `wxMenuBar`, `wxMenu`

wxWidgets docs: **wxMenuItem**

## Events

Event types emitted from this class: `menu_open`, `menu_close`, `menu_highlight`

## Data Types

```
wxMenuItem() = wx:wx_object()
```

## Exports

```
new() -> wxMenuItem()

new(Options :: [Option]) -> wxMenuItem()
```

Types:

```
   Option =
       {parentMenu, wxMenu:wxMenu()} |
       {id, integer()} |
       {text, unicode:chardata()} |
       {help, unicode:chardata()} |
       {kind, wx:wx_enum()} |
       {subMenu, wxMenu:wxMenu()}
```

Constructs a `wxMenuItem` object.

Menu items can be standard, or "stock menu items", or custom. For the standard menu items (such as commands to open a file, exit the program and so on, see page_stockitems for the full list) it is enough to specify just the stock ID and leave `text` and `help` string empty. Some platforms (currently wxGTK only, and see the remark in `setBitmap/2` documentation) will also show standard bitmaps for stock menu items.

Leaving at least `text` empty for the stock menu items is actually strongly recommended as they will have appearance and keyboard interface (including standard accelerators) familiar to the user.

For the custom (non-stock) menu items, `text` must be specified and while `help` string may be left empty, it's recommended to pass the item description (which is automatically shown by the library in the status bar when the menu item is selected) in this parameter.

Finally note that you can e.g. use a stock menu label without using its stock help string:

that is, stock properties are set independently one from the other.

```
destroy(This :: wxMenuItem()) -> ok
```

Destructor.

```
check(This) -> ok
```

Types:

```
   This = wxMenuItem()
```

```
check(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxMenuItem()
   Option = {check, boolean()}
```

Checks or unchecks the menu item.

Note that this only works when the item is already appended to a menu.

```
enable(This) -> ok
```

Types:

```
   This = wxMenuItem()
```

```
enable(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxMenuItem()
   Option = {enable, boolean()}
```

Enables or disables the menu item.

```
getBitmap(This) -> wxBitmap:wxBitmap()
```

Types:

```
   This = wxMenuItem()
```

Returns the checked or unchecked bitmap.

Only for:wxmsw

```
getHelp(This) -> unicode:charlist()
```

Types:

```
   This = wxMenuItem()
```

Returns the help string associated with the menu item.

```
getId(This) -> integer()
```

Types:

```
   This = wxMenuItem()
```

Returns the menu item identifier.

```
getKind(This) -> wx:wx_enum()
```

Types:

```
This = wxMenuItem()
```

Returns the item kind, one of `wxITEM_SEPARATOR`, `wxITEM_NORMAL`, `wxITEM_CHECK` or `wxITEM_RADIO`.

```
getLabelFromText(Text) -> unicode:charlist()
```

Types:

```
Text = unicode:chardata()
```

See: `getLabelText/1`.

```
getLabelText(Text) -> unicode:charlist()
```

Types:

```
Text = unicode:chardata()
```

Strips all accelerator characters and mnemonics from the given `text`.

For example:

will return just `"Hello"`.

See: `getItemLabelText/1`, `getItemLabel/1`

```
getText(This) -> unicode:charlist()
```

Types:

```
This = wxMenuItem()
```

See: `getItemLabel/1`.

```
getItemLabel(This) -> unicode:charlist()
```

Types:

```
This = wxMenuItem()
```

Returns the text associated with the menu item including any accelerator characters that were passed to the constructor or `setItemLabel/2`.

See: `getItemLabelText/1`, `getLabelText/1`

```
getLabel(This) -> unicode:charlist()
```

Types:

```
This = wxMenuItem()
```

See: `getItemLabelText/1`.

```
getItemLabelText(This) -> unicode:charlist()
```

Types:

```
This = wxMenuItem()
```

Returns the text associated with the menu item, without any accelerator characters.

See: `getItemLabel/1`, `getLabelText/1`

```
getMenu(This) -> wxMenu:wxMenu()
```

Types:

```
This = wxMenuItem()
```

Returns the menu this menu item is in, or NULL if this menu item is not attached.

```
getSubMenu(This) -> wxMenu:wxMenu()
```

Types:

```
This = wxMenuItem()
```

Returns the submenu associated with the menu item, or NULL if there isn't one.

```
isCheckable(This) -> boolean()
```

Types:

```
This = wxMenuItem()
```

Returns true if the item is checkable.

Notice that the radio buttons are considered to be checkable as well, so this method returns true for them too. Use `IsCheck()` (not implemented in wx) if you want to test for the check items only.

```
isChecked(This) -> boolean()
```

Types:

```
This = wxMenuItem()
```

Returns true if the item is checked.

```
isEnabled(This) -> boolean()
```

Types:

```
This = wxMenuItem()
```

Returns true if the item is enabled.

```
isSeparator(This) -> boolean()
```

Types:

```
This = wxMenuItem()
```

Returns true if the item is a separator.

```
isSubMenu(This) -> boolean()
```

Types:

```
This = wxMenuItem()
```

Returns true if the item is a submenu.

```
setBitmap(This, Bmp) -> ok
```

Types:

```
This = wxMenuItem()
Bmp = wxBitmap:wxBitmap()
```

Sets the bitmap for the menu item.

It is equivalent to wxMenuItem::SetBitmaps(bmp, wxNullBitmap) if `checked` is true (default value) or SetBitmaps(wxNullBitmap, bmp) otherwise.

`setBitmap/2` must be called before the item is appended to the menu, i.e. appending the item without a bitmap and setting one later is not guaranteed to work. But the bitmap can be changed or reset later if it had been set up initially.

Notice that GTK+ uses a global setting called `gtk-menu-images` to determine if the images should be shown in the menus at all. If it is off (which is the case in e.g. Gnome 2.28 by default), no images will be shown, consistently with the native behaviour.

Only for:wxmsw,wxosx,wxgtk

```
setHelp(This, HelpString) -> ok
```

Types:

```
This = wxMenuItem()
HelpString = unicode:chardata()
```

Sets the help string.

```
setMenu(This, Menu) -> ok
```

Types:

```
This = wxMenuItem()
Menu = wxMenu:wxMenu()
```

Sets the parent menu which will contain this menu item.

```
setSubMenu(This, Menu) -> ok
```

Types:

```
This = wxMenuItem()
Menu = wxMenu:wxMenu()
```

Sets the submenu of this menu item.

```
setText(This, Label) -> ok
```

Types:

```
This = wxMenuItem()
Label = unicode:chardata()
```

See: `setItemLabel/2`.

```
setItemLabel(This, Label) -> ok
```

Types:

```
This = wxMenuItem()
Label = unicode:chardata()
```

Sets the label associated with the menu item.

Note that if the ID of this menu item corresponds to a stock ID, then it is not necessary to specify a label: wxWidgets will automatically use the stock item label associated with that ID. See the `new/1` for more info.

The label string for the normal menu items (not separators) may include the accelerator which can be used to activate the menu item from keyboard. An accelerator key can be specified using the ampersand & character. In order to embed an ampersand character in the menu item text, the ampersand must be doubled.

Optionally you can specify also an accelerator string appending a tab character `\t` followed by a valid key combination (e.g. `CTRL+V`). Its general syntax is any combination of `"CTRL"`, `"RAWCTRL"`, `"ALT"` and `"SHIFT"` strings (case

doesn't matter) separated by either `'-'` or `'+'` characters and followed by the accelerator itself. Notice that `CTRL` corresponds to the "Ctrl" key on most platforms but not under macOS where it is mapped to "Cmd" key on Mac keyboard. Usually this is exactly what you want in portable code but if you really need to use the (rarely used for this purpose) "Ctrl" key even under Mac, you may use `RAWCTRL` to prevent this mapping. Under the other platforms `RAWCTRL` is the same as plain `CTRL`.

The accelerator may be any alphanumeric character, any function key (from `F1` to `F12`), any numpad digit key using `KP_` prefix (i.e. from `KP_0` to `KP_9`) or one of the special strings listed below (again, case doesn't matter) corresponding to the specified key code:

Examples:

Note: In wxGTK using `"SHIFT"` with non-alphabetic characters currently doesn't work, even in combination with other modifiers, due to GTK+ limitation. E.g. `Shift+Ctrl+A` works but `Shift+Ctrl+1` or `Shift+/` do not, so avoid using accelerators of this form in portable code.

Note: In wxGTk, the left/right/up/down arrow keys do not work as accelerator keys for a menu item unless a modifier key is used. Additionally, the following keycodes are not supported as menu accelerator keys:

See: `getItemLabel/1`, `getItemLabelText/1`

# wxMessageDialog

Erlang module

This class represents a dialog that shows a single or multi-line message, with a choice of OK, Yes, No and Cancel buttons.

Styles

This class supports the following styles:

See: **Overview cmndlg**

See: `wxRichMessageDialog` (not implemented in wx)

This class is derived (and can use functions) from: `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxMessageDialog**

## Data Types

```
wxMessageDialog() = wx:wx_object()
```

## Exports

```
new(Parent, Message) -> wxMessageDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Message = unicode:chardata()
```

```
new(Parent, Message, Options :: [Option]) -> wxMessageDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Message = unicode:chardata()
   Option =
       {caption, unicode:chardata()} |
       {style, integer()} |
       {pos, {X :: integer(), Y :: integer()}}
```

Constructor specifying the message box properties.

Use `wxDialog:showModal/1` to show the dialog.

`style` may be a bit list of the identifiers described above.

Notice that not all styles are compatible: only one of `wxOK` and `wxYES_NO` may be specified (and one of them must be specified) and at most one default button style can be used and it is only valid if the corresponding button is shown in the message box.

```
destroy(This :: wxMessageDialog()) -> ok
```

Destroys the object.

# wxMiniFrame

Erlang module

A miniframe is a frame with a small title bar. It is suitable for floating toolbars that must not take up too much screen area.

An example of mini frame can be seen in the page_samples_dialogs using the "Mini frame" command of the "Generic dialogs" submenu.

Styles

This class supports the following styles:

Remark: This class has miniframe functionality under Windows and GTK, i.e. the presence of mini frame will not be noted in the task bar and focus behaviour is different. On other platforms, it behaves like a normal frame.

See: `wxMDIParentFrame`, `wxMDIChildFrame`, `wxFrame`, `wxDialog`

This class is derived (and can use functions) from: `wxFrame` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxMiniFrame**

## Data Types

```
wxMiniFrame() = wx:wx_object()
```

## Exports

```
new() -> wxMiniFrame()
```

Default ctor.

```
new(Parent, Id, Title) -> wxMiniFrame()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Title = unicode:chardata()
```

```
new(Parent, Id, Title, Options :: [Option]) -> wxMiniFrame()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Title = unicode:chardata()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Constructor, creating the window.

Remark: The frame behaves like a normal frame on non-Windows platforms.

See: `create/5`

```
destroy(This :: wxMiniFrame()) -> ok
```

Destructor.

Destroys all child windows and menu bar if present.

```
create(This, Parent, Id, Title) -> boolean()
```

Types:

```
   This = wxMiniFrame()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()
```

```
create(This, Parent, Id, Title, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxMiniFrame()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Title = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Used in two-step frame construction.

See `new/4` for further details.

# wxMirrorDC

Erlang module

wxMirrorDC is a simple wrapper class which is always associated with a real wxDC object and either forwards all of its operations to it without changes (no mirroring takes place) or exchanges x and y coordinates which makes it possible to reuse the same code to draw a figure and its mirror - i.e. reflection related to the diagonal line x == y.

Since: 2.5.0

This class is derived (and can use functions) from: wxDC

wxWidgets docs: **wxMirrorDC**

## Data Types

```
wxMirrorDC() = wx:wx_object()
```

## Exports

```
new(Dc, Mirror) -> wxMirrorDC()
```

Types:

```
Dc = wxDC:wxDC()
Mirror = boolean()
```

Creates a (maybe) mirrored DC associated with the real dc.

Everything drawn on wxMirrorDC will appear (and maybe mirrored) on dc.

mirror specifies if we do mirror (if it is true) or not (if it is false).

```
destroy(This :: wxMirrorDC()) -> ok
```

Destroys the object.

# wxMouseCaptureChangedEvent

Erlang module

An mouse capture changed event is sent to a window that loses its mouse capture. This is called even if `wxWindow:releaseMouse/1` was called by the application code. Handling this event allows an application to cater for unexpected capture releases which might otherwise confuse mouse handling code.

Only for:wxmsw

See: `wxMouseCaptureLostEvent`, **Overview events**, `wxWindow:captureMouse/1`, `wxWindow:releaseMouse/1`, `wxWindow:getCapture/0`

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxMouseCaptureChangedEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxMouseCaptureChangedEventType` to subscribe to events of this type.

## Data Types

```
wxMouseCaptureChangedEvent() = wx:wx_object()
wxMouseCaptureChanged() =
    #wxMouseCaptureChanged{type =
                              wxMouseCaptureChangedEvent:wxMouseCaptureChangedEventType()}
wxMouseCaptureChangedEventType() = mouse_capture_changed
```

## Exports

```
getCapturedWindow(This) -> wxWindow:wxWindow()
```

Types:

```
   This = wxMouseCaptureChangedEvent()
```

Returns the window that gained the capture, or NULL if it was a non-wxWidgets window.

# wxMouseCaptureLostEvent

Erlang module

A mouse capture lost event is sent to a window that had obtained mouse capture, which was subsequently lost due to an "external" event (for example, when a dialog box is shown or if another application captures the mouse).

If this happens, this event is sent to all windows that are on the capture stack (i.e. called CaptureMouse, but didn't call ReleaseMouse yet). The event is not sent if the capture changes because of a call to CaptureMouse or ReleaseMouse.

This event is currently emitted under Windows only.

Only for:wxmsw

See: `wxMouseCaptureChangedEvent`, **Overview events**, `wxWindow:captureMouse/1`, `wxWindow:releaseMouse/1`, `wxWindow:getCapture/0`

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxMouseCaptureLostEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxMouseCaptureLostEventType` to subscribe to events of this type.

## Data Types

```
wxMouseCaptureLostEvent() = wx:wx_object()
wxMouseCaptureLost() =
    #wxMouseCaptureLost{type =
                           wxMouseCaptureLostEvent:wxMouseCaptureLostEventType()}
wxMouseCaptureLostEventType() = mouse_capture_lost
```

# wxMouseEvent

Erlang module

This event class contains information about the events generated by the mouse: they include mouse buttons press and release events and mouse move events.

All mouse events involving the buttons use `wxMOUSE_BTN_LEFT` for the left mouse button, `wxMOUSE_BTN_MIDDLE` for the middle one and `wxMOUSE_BTN_RIGHT` for the right one. And if the system supports more buttons, the `wxMOUSE_BTN_AUX1` and `wxMOUSE_BTN_AUX2` events can also be generated. Note that not all mice have even a middle button so a portable application should avoid relying on the events from it (but the right button click can be emulated using the left mouse button with the control key under Mac platforms with a single button mouse).

For the `wxEVT_ENTER_WINDOW` and `wxEVT_LEAVE_WINDOW` events purposes, the mouse is considered to be inside the window if it is in the window client area and not inside one of its children. In other words, the parent window receives `wxEVT_LEAVE_WINDOW` event not only when the mouse leaves the window entirely but also when it enters one of its children.

The position associated with a mouse event is expressed in the window coordinates of the window which generated the event, you can use `wxWindow:clientToScreen/3` to convert it to screen coordinates and possibly call `wxWindow:screenToClient/2` next to convert it to window coordinates of another window.

Note: Note the difference between methods like `leftDown/1` and the inherited `leftIsDown/1`: the former returns true when the event corresponds to the left mouse button click while the latter returns true if the left mouse button is currently being pressed. For example, when the user is dragging the mouse you can use `leftIsDown/1` to test whether the left mouse button is (still) depressed. Also, by convention, if `leftDown/1` returns true, `leftIsDown/1` will also return true in wxWidgets whatever the underlying GUI behaviour is (which is platform-dependent). The same applies, of course, to other mouse buttons as well.

See: `wxKeyEvent`

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxMouseEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxMouseEventType` to subscribe to events of this type.

## Data Types

```
wxMouseEvent() = wx:wx_object()
wxMouse() =
    #wxMouse{type = wxMouseEvent:wxMouseEventType(),
             x = integer(),
             y = integer(),
             leftDown = boolean(),
             middleDown = boolean(),
             rightDown = boolean(),
             controlDown = boolean(),
             shiftDown = boolean(),
             altDown = boolean(),
             metaDown = boolean(),
             wheelRotation = integer(),
             wheelDelta = integer(),
```

```
             linesPerAction = integer()}
wxMouseEventType() =
    left_down | left_up | middle_down | middle_up | right_down |
    right_up | motion | enter_window | leave_window |
    left_dclick | middle_dclick | right_dclick | mousewheel |
    aux1_down | aux1_up | aux1_dclick | aux2_down | aux2_up |
    aux2_dclick
```

## Exports

`altDown(This) -> boolean()`

Types:

```
This = wxMouseEvent()
```

Returns true if the Alt key is pressed.

Notice that `wxKeyEvent:getModifiers/1` should usually be used instead of this one.

`button(This, But) -> boolean()`

Types:

```
This = wxMouseEvent()
But = wx:wx_enum()
```

Returns true if the event was generated by the specified button.

See: wxMouseState::ButtoinIsDown()

`buttonDClick(This) -> boolean()`

Types:

```
This = wxMouseEvent()
```

`buttonDClick(This, Options :: [Option]) -> boolean()`

Types:

```
This = wxMouseEvent()
Option = {but, wx:wx_enum()}
```

If the argument is omitted, this returns true if the event was a mouse double click event.

Otherwise the argument specifies which double click event was generated (see `button/2` for the possible values).

`buttonDown(This) -> boolean()`

Types:

```
This = wxMouseEvent()
```

`buttonDown(This, Options :: [Option]) -> boolean()`

Types:

```
This = wxMouseEvent()
Option = {but, wx:wx_enum()}
```

If the argument is omitted, this returns true if the event was a mouse button down event.

Otherwise the argument specifies which button-down event was generated (see `button/2` for the possible values).

```
buttonUp(This) -> boolean()
```
Types:
```
   This = wxMouseEvent()
```

```
buttonUp(This, Options :: [Option]) -> boolean()
```
Types:
```
   This = wxMouseEvent()
   Option = {but, wx:wx_enum()}
```
If the argument is omitted, this returns true if the event was a mouse button up event.

Otherwise the argument specifies which button-up event was generated (see `button/2` for the possible values).

```
cmdDown(This) -> boolean()
```
Types:
```
   This = wxMouseEvent()
```
Returns true if the key used for command accelerators is pressed.

Same as `controlDown/1`. Deprecated.

Notice that `wxKeyEvent:getModifiers/1` should usually be used instead of this one.

```
controlDown(This) -> boolean()
```
Types:
```
   This = wxMouseEvent()
```
Returns true if the Control key or Apple/Command key under macOS is pressed.

This function doesn't distinguish between right and left control keys.

Notice that `wxKeyEvent:getModifiers/1` should usually be used instead of this one.

```
dragging(This) -> boolean()
```
Types:
```
   This = wxMouseEvent()
```
Returns true if this was a dragging event (motion while a button is depressed).

See: `moving/1`

```
entering(This) -> boolean()
```
Types:
```
   This = wxMouseEvent()
```
Returns true if the mouse was entering the window.

See: `leaving/1`

```
getButton(This) -> integer()
```
Types:
```
   This = wxMouseEvent()
```
Returns the mouse button which generated this event or `wxMOUSE_BTN_NONE` if no button is involved (for mouse move, enter or leave event, for example).

Otherwise wxMOUSE_BTN_LEFT is returned for the left button down, up and double click events, wxMOUSE_BTN_MIDDLE and wxMOUSE_BTN_RIGHT for the same events for the middle and the right buttons respectively.

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxMouseEvent()
```

Returns the physical mouse position.

```
getLogicalPosition(This, Dc) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxMouseEvent()
Dc = wxDC:wxDC()
```

Returns the logical mouse position in pixels (i.e. translated according to the translation set for the DC, which usually indicates that the window has been scrolled).

```
getLinesPerAction(This) -> integer()
```

Types:

```
This = wxMouseEvent()
```

Returns the configured number of lines (or whatever) to be scrolled per wheel action.

Default value under most platforms is three.

See: GetColumnsPerAction() (not implemented in wx)

```
getWheelRotation(This) -> integer()
```

Types:

```
This = wxMouseEvent()
```

Get wheel rotation, positive or negative indicates direction of rotation.

Current devices all send an event when rotation is at least +/-WheelDelta, but finer resolution devices can be created in the future.

Because of this you shouldn't assume that one event is equal to 1 line, but you should be able to either do partial line scrolling or wait until several events accumulate before scrolling.

```
getWheelDelta(This) -> integer()
```

Types:

```
This = wxMouseEvent()
```

Get wheel delta, normally 120.

This is the threshold for action to be taken, and one such action (for example, scrolling one increment) should occur for each delta.

```
getX(This) -> integer()
```

Types:

```
This = wxMouseEvent()
```

Returns X coordinate of the physical mouse event position.

```
getY(This) -> integer()
```

Types:

```
This = wxMouseEvent()
```

Returns Y coordinate of the physical mouse event position.

```
isButton(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the event was a mouse button event (not necessarily a button down event - that may be tested using `buttonDown/2`).

```
isPageScroll(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the system has been setup to do page scrolling with the mouse wheel instead of line scrolling.

```
leaving(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the mouse was leaving the window.

See: `entering/1`

```
leftDClick(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the event was a left double click.

```
leftDown(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the left mouse button changed to down.

```
leftIsDown(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the left mouse button is currently down.

```
leftUp(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the left mouse button changed to up.

```
metaDown(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the Meta key was down at the time of the event.

```
middleDClick(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the event was a middle double click.

```
middleDown(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the middle mouse button changed to down.

```
middleIsDown(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the middle mouse button is currently down.

```
middleUp(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the middle mouse button changed to up.

```
moving(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if this was a motion event and no mouse buttons were pressed.

If any mouse button is held pressed, then this method returns false and `dragging/1` returns true.

```
rightDClick(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the event was a right double click.

```
rightDown(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the right mouse button changed to down.

```
rightIsDown(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the right mouse button is currently down.

```
rightUp(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the right mouse button changed to up.

```
shiftDown(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the Shift key is pressed.

This function doesn't distinguish between right and left shift keys.

Notice that `wxKeyEvent:getModifiers/1` should usually be used instead of this one.

```
getWheelAxis(This) -> wx:wx_enum()
```

Types:

```
This = wxMouseEvent()
```

Gets the axis the wheel operation concerns.

Usually the mouse wheel is used to scroll vertically so `wxMOUSE_WHEEL_VERTICAL` is returned but some mice (and most trackpads) also allow to use the wheel to scroll horizontally in which case `wxMOUSE_WHEEL_HORIZONTAL` is returned.

Notice that before wxWidgets 2.9.4 this method returned `int`.

```
aux1DClick(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the event was a first extra button double click.

```
aux1Down(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the first extra button mouse button changed to down.

```
aux1Up(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the first extra button mouse button changed to up.

```
aux2DClick(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the event was a second extra button double click.

```
aux2Down(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the second extra button mouse button changed to down.

```
aux2Up(This) -> boolean()
```

Types:

```
This = wxMouseEvent()
```

Returns true if the second extra button mouse button changed to up.

# wxMoveEvent

Erlang module

A move event holds information about window position change.

These events are currently generated for top level (see `wxTopLevelWindow`) windows in all ports, but are not generated for the child windows in wxGTK.

See: {X,Y}, **Overview events**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxMoveEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxMoveEventType` to subscribe to events of this type.

## Data Types

```
wxMoveEvent() = wx:wx_object()
wxMove() =
    #wxMove{type = wxMoveEvent:wxMoveEventType(),
            pos = {X :: integer(), Y :: integer()},
            rect =
                {X :: integer(),
                 Y :: integer(),
                 W :: integer(),
                 H :: integer()}}
wxMoveEventType() = move
```

## Exports

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxMoveEvent()
```

Returns the position of the window generating the move change event.

```
getRect(This) ->
           {X :: integer(),
            Y :: integer(),
            W :: integer(),
            H :: integer()}
```

Types:

```
This = wxMoveEvent()
```

# wxMultiChoiceDialog

Erlang module

This class represents a dialog that shows a list of strings, and allows the user to select one or more.

Styles

This class supports the following styles:

See: **Overview cmndlg**, `wxSingleChoiceDialog`

This class is derived (and can use functions) from: `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxMultiChoiceDialog**

## Data Types

```
wxMultiChoiceDialog() = wx:wx_object()
```

## Exports

```
new(Parent, Message, Caption, Choices) -> wxMultiChoiceDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Message = Caption = unicode:chardata()
   Choices = [unicode:chardata()]
```

```
new(Parent, Message, Caption, Choices, Options :: [Option]) ->
       wxMultiChoiceDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Message = Caption = unicode:chardata()
   Choices = [unicode:chardata()]
   Option =
       {style, integer()} | {pos, {X :: integer(), Y :: integer()}}
```

Constructor taking an array of `wxString` (not implemented in wx) choices.

Remark: Use `wxDialog:showModal/1` to show the dialog.

```
getSelections(This) -> [integer()]
```

Types:

```
   This = wxMultiChoiceDialog()
```

Returns array with indexes of selected items.

```
setSelections(This, Selections) -> ok
```

Types:

```
This = wxMultiChoiceDialog()
Selections = [integer()]
```

Sets selected items from the array of selected items' indexes.

```
destroy(This :: wxMultiChoiceDialog()) -> ok
```

Destroys the object.

# wxNavigationKeyEvent

Erlang module

This event class contains information about navigation events, generated by navigation keys such as tab and page down.

This event is mainly used by wxWidgets implementations. A `wxNavigationKeyEvent` handler is automatically provided by wxWidgets when you enable keyboard navigation inside a window by inheriting it from wxNavigationEnabled<>.

See: `wxWindow:navigate/2`, `wxWindow::NavigateIn` (not implemented in wx)

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxNavigationKeyEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxNavigationKeyEventType` to subscribe to events of this type.

## Data Types

```
wxNavigationKeyEvent() = wx:wx_object()
wxNavigationKey() =
    #wxNavigationKey{type =
                         wxNavigationKeyEvent:wxNavigationKeyEventType(),
                     dir = boolean(),
                     focus = wxWindow:wxWindow()}
wxNavigationKeyEventType() = navigation_key
```

## Exports

```
getDirection(This) -> boolean()
```

Types:

```
This = wxNavigationKeyEvent()
```

Returns true if the navigation was in the forward direction.

```
setDirection(This, Direction) -> ok
```

Types:

```
This = wxNavigationKeyEvent()
Direction = boolean()
```

Sets the direction to forward if `direction` is true, or backward if false.

```
isWindowChange(This) -> boolean()
```

Types:

```
This = wxNavigationKeyEvent()
```

Returns true if the navigation event represents a window change (for example, from Ctrl-Page Down in a notebook).

```
setWindowChange(This, WindowChange) -> ok
```

Types:

```
   This = wxNavigationKeyEvent()
   WindowChange = boolean()
```

Marks the event as a window change event.

```
isFromTab(This) -> boolean()
```

Types:

```
   This = wxNavigationKeyEvent()
```

Returns true if the navigation event was from a tab key.

This is required for proper navigation over radio buttons.

```
setFromTab(This, FromTab) -> ok
```

Types:

```
   This = wxNavigationKeyEvent()
   FromTab = boolean()
```

Marks the navigation event as from a tab key.

```
getCurrentFocus(This) -> wxWindow:wxWindow()
```

Types:

```
   This = wxNavigationKeyEvent()
```

Returns the child that has the focus, or NULL.

```
setCurrentFocus(This, CurrentFocus) -> ok
```

Types:

```
   This = wxNavigationKeyEvent()
   CurrentFocus = wxWindow:wxWindow()
```

Sets the current focus window member.

# wxNotebook

Erlang module

This class represents a notebook control, which manages multiple windows with associated tabs.

To use the class, create a `wxNotebook` object and call `wxBookCtrlBase:addPage/4` or `wxBookCtrlBase:insertPage/5`, passing a window to be used as the page. Do not explicitly delete the window for a page that is currently managed by `wxNotebook`.

`wxNotebookPage` is a typedef for `wxWindow`.

Styles

This class supports the following styles:

Page backgrounds

On Windows, the default theme paints a background on the notebook's pages. If you wish to suppress this theme, for aesthetic or performance reasons, there are three ways of doing it. You can use `wxNB_NOPAGETHEME` to disable themed drawing for a particular notebook, you can call `wxSystemOptions:setOption/2` to disable it for the whole application, or you can disable it for individual pages by using `wxWindow:setBackgroundColour/2`.

To disable themed pages globally:

Set the value to 1 to enable it again. To give a single page a solid background that more or less fits in with the overall theme, use:

On platforms other than Windows, or if the application is not using Windows themes, `getThemeBackgroundColour/1` will return an uninitialised colour object, and the above code will therefore work on all platforms.

See: ?wxBookCtrl, `wxBookCtrlEvent`, `wxImageList`, **Examples**

This class is derived (and can use functions) from: `wxBookCtrlBase` `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxNotebook**

## Events

Event types emitted from this class: `command_notebook_page_changed`, `command_notebook_page_changing`

## Data Types

```
wxNotebook() = wx:wx_object()
```

## Exports

```
new() -> wxNotebook()
```

Constructs a notebook control.

```
new(Parent, Id) -> wxNotebook()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()

new(Parent, Id, Options :: [Option]) -> wxNotebook()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructs a notebook control.

Note that sometimes you can reduce flicker by passing the wxCLIP_CHILDREN window style.

```
destroy(This :: wxNotebook()) -> ok
```

Destroys the `wxNotebook` object.

```
assignImageList(This, ImageList) -> ok
```

Types:

```
   This = wxNotebook()
   ImageList = wxImageList:wxImageList()
```

Sets the image list for the page control and takes ownership of the list.

See: `wxImageList`, `setImageList/2`

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = wxNotebook()
   Parent = wxWindow:wxWindow()
   Id = integer()

create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxNotebook()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Creates a notebook control.

See `new/3` for a description of the parameters.

```
getImageList(This) -> wxImageList:wxImageList()
```

Types:

```
This = wxNotebook()
```

Returns the associated image list, may be NULL.

See: `wxImageList`, `setImageList/2`

```
getPageImage(This, NPage) -> integer()
```

Types:

```
This = wxNotebook()
NPage = integer()
```

Returns the image index for the given page.

```
getRowCount(This) -> integer()
```

Types:

```
This = wxNotebook()
```

Returns the number of rows in the notebook control.

```
getThemeBackgroundColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxNotebook()
```

If running under Windows and themes are enabled for the application, this function returns a suitable colour for painting the background of a notebook page, and can be passed to `wxWindow:setBackgroundColour/2`.

Otherwise, an uninitialised colour will be returned.

```
setImageList(This, ImageList) -> ok
```

Types:

```
This = wxNotebook()
ImageList = wxImageList:wxImageList()
```

Sets the image list to use.

It does not take ownership of the image list, you must delete it yourself.

See: `wxImageList`, `assignImageList/2`

```
setPadding(This, Padding) -> ok
```

Types:

```
This = wxNotebook()
Padding = {W :: integer(), H :: integer()}
```

Sets the amount of space around each page's icon and label, in pixels.

Note: The vertical padding cannot be changed in wxGTK.

```
setPageSize(This, Size) -> ok
```

Types:

```
This = wxNotebook()
Size = {W :: integer(), H :: integer()}
```

Sets the width and height of the pages.

Note: This method is currently not implemented for wxGTK.

```
setPageImage(This, Page, Image) -> boolean()
```

Types:

```
This = wxNotebook()
Page = Image = integer()
```

Sets the image index for the given page.

`image` is an index into the image list which was set with `setImageList/2`.

# wxNotificationMessage

Erlang module

This class allows showing the user a message non intrusively.

Currently it is implemented natively for Windows, macOS, GTK and uses generic toast notifications under the other platforms. It's not recommended but `wxGenericNotificationMessage` can be used instead of the native ones. This might make sense if your application requires features not available in the native implementation.

Notice that this class is not a window and so doesn't derive from `wxWindow`.

Platform Notes

Par: Up to Windows 8 balloon notifications are displayed from an icon in the notification area of the taskbar. If your application uses a `wxTaskBarIcon` you should call `useTaskBarIcon/1` to ensure that only one icon is shown in the notification area. Windows 10 displays all notifications as popup toasts. To suppress the additional icon in the notification area on Windows 10 and for toast notification support on Windows 8 it is recommended to call `mSWUseToasts/1` before showing the first notification message.

Par: The macOS implementation uses Notification Center to display native notifications. In order to use actions your notifications must use the alert style. This can be enabled by the user in system settings or by setting the `NSUserNotificationAlertStyle` value in Info.plist to `alert`. Please note that the user always has the option to change the notification style.

Since: 2.9.0

This class is derived (and can use functions) from: `wxEvtHandler`

wxWidgets docs: **wxNotificationMessage**

## Events

Event types emitted from this class: `notification_message_click`, `notification_message_dismissed`, `notification_message_action`

## Data Types

```
wxNotificationMessage() = wx:wx_object()
```

## Exports

```
new() -> wxNotificationMessage()
```

Default constructor, use `setParent/2`, `setTitle/2` and `setMessage/2` to initialize the object before showing it.

```
new(Title) -> wxNotificationMessage()
```

Types:

```
   Title = unicode:chardata()
```

```
new(Title, Options :: [Option]) -> wxNotificationMessage()
```

Types:

```
   Title = unicode:chardata()
   Option =
```

```
       {message, unicode:chardata()} |
       {parent, wxWindow:wxWindow()} |
       {flags, integer()}
```

Create a notification object with the given attributes.

See `setTitle/2`, `setMessage/2`, `setParent/2` and `setFlags/2` for the description of the corresponding parameters.

```
destroy(This :: wxNotificationMessage()) -> ok
```

Destructor does not hide the notification.

The notification can continue to be shown even after the C++ object was destroyed, call `close/1` explicitly if it needs to be hidden.

```
addAction(This, Actionid) -> boolean()
```

Types:

```
   This = wxNotificationMessage()
   Actionid = integer()
```

```
addAction(This, Actionid, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxNotificationMessage()
   Actionid = integer()
   Option = {label, unicode:chardata()}
```

Add an action to the notification.

If supported by the implementation this are usually buttons in the notification selectable by the user.

Return: false if the current implementation or OS version does not support actions in notifications.

Since: 3.1.0

```
close(This) -> boolean()
```

Types:

```
   This = wxNotificationMessage()
```

Hides the notification.

Returns true if it was hidden or false if it couldn't be done (e.g. on some systems automatically hidden notifications can't be hidden manually).

```
setFlags(This, Flags) -> ok
```

Types:

```
   This = wxNotificationMessage()
   Flags = integer()
```

This parameter can be currently used to specify the icon to show in the notification.

Valid values are `wxICON_INFORMATION`, `wxICON_WARNING` and `wxICON_ERROR` (notice that `wxICON_QUESTION` is not allowed here). Some implementations of this class may not support the icons.

See: `setIcon/2`

```
setIcon(This, Icon) -> ok
```

Types:

```
This = wxNotificationMessage()
Icon = wxIcon:wxIcon()
```

Specify a custom icon to be displayed in the notification.

Some implementations of this class may not support custom icons.

See: `setFlags/2`

Since: 3.1.0

```
setMessage(This, Message) -> ok
```

Types:

```
This = wxNotificationMessage()
Message = unicode:chardata()
```

Set the main text of the notification.

This should be a more detailed description than the title but still limited to reasonable length (not more than 256 characters).

```
setParent(This, Parent) -> ok
```

Types:

```
This = wxNotificationMessage()
Parent = wxWindow:wxWindow()
```

Set the parent for this notification: the notification will be associated with the top level parent of this window or, if this method is not called, with the main application window by default.

```
setTitle(This, Title) -> ok
```

Types:

```
This = wxNotificationMessage()
Title = unicode:chardata()
```

Set the title, it must be a concise string (not more than 64 characters), use `setMessage/2` to give the user more details.

```
show(This) -> boolean()
```

Types:

```
This = wxNotificationMessage()
```

```
show(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxNotificationMessage()
Option = {timeout, integer()}
```

Show the notification to the user and hides it after `timeout` seconds are elapsed.

Special values `Timeout_Auto` and `Timeout_Never` can be used here, notice that you shouldn't rely on `timeout` being exactly respected because the current platform may only support default timeout value and also because the user may be able to close the notification.

Note: When using native notifications in wxGTK, the timeout is ignored for the notifications with wxICON_WARNING or wxICON_ERROR flags, they always remain shown unless they're explicitly hidden by the user, i.e. behave as if Timeout_Auto were given.

Return: false if an error occurred.

```
useTaskBarIcon(Icon) -> wxTaskBarIcon:wxTaskBarIcon()
```

Types:

```
   Icon = wxTaskBarIcon:wxTaskBarIcon()
```

If the application already uses a wxTaskBarIcon, it should be connected to notifications by using this method.

This has no effect if toast notifications are used.

Return: the task bar icon which was used previously (may be NULL)

Only for:wxmsw

```
mSWUseToasts() -> boolean()

mSWUseToasts(Options :: [Option]) -> boolean()
```

Types:

```
   Option =
       {shortcutPath, unicode:chardata()} |
       {appId, unicode:chardata()}
```

Enables toast notifications available since Windows 8 and suppresses the additional icon in the notification area on Windows 10.

Toast notifications require a shortcut to the application in the start menu. The start menu shortcut needs to contain an Application User Model ID. It is recommended that the applications setup creates the shortcut and the application specifies the setup created shortcut in shortcutPath. A call to this method will verify (and if necessary modify) the shortcut before enabling toast notifications.

Return: false if toast notifications could not be enabled.

Only for:wxmsw

See: wxAppConsole::SetAppName() (not implemented in wx), wxAppConsole::SetVendorName() (not implemented in wx)

Since: 3.1.0

# wxNotifyEvent

Erlang module

This class is not used by the event handlers by itself, but is a base class for other event classes (such as `wxBookCtrlEvent`).

It (or an object of a derived class) is sent when the controls state is being changed and allows the program to `veto/1` this change if it wants to prevent it from happening.

See: `wxBookCtrlEvent`

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxNotifyEvent**

## Data Types

```
wxNotifyEvent() = wx:wx_object()
```

## Exports

```
allow(This) -> ok
```

Types:

```
This = wxNotifyEvent()
```

This is the opposite of `veto/1`: it explicitly allows the event to be processed.

For most events it is not necessary to call this method as the events are allowed anyhow but some are forbidden by default (this will be mentioned in the corresponding event description).

```
isAllowed(This) -> boolean()
```

Types:

```
This = wxNotifyEvent()
```

Returns true if the change is allowed (`veto/1` hasn't been called) or false otherwise (if it was).

```
veto(This) -> ok
```

Types:

```
This = wxNotifyEvent()
```

Prevents the change announced by this event from happening.

It is in general a good idea to notify the user about the reasons for vetoing the change because otherwise the applications behaviour (which just refuses to do what the user wants) might be quite surprising.

# wxOverlay

Erlang module

Creates an overlay over an existing window, allowing for manipulations like rubberbanding, etc. On wxOSX the overlay is implemented with native platform APIs, on the other platforms it is simulated using `wxMemoryDC`.

See: `wxDCOverlay`, `wxDC`

wxWidgets docs: **wxOverlay**

## Data Types

```
wxOverlay() = wx:wx_object()
```

## Exports

```
new() -> wxOverlay()
```

```
destroy(This :: wxOverlay()) -> ok
```

```
reset(This) -> ok
```

Types:

```
This = wxOverlay()
```

Clears the overlay without restoring the former state.

To be done, for example, when the window content has been changed and repainted.

# wxPageSetupDialog

Erlang module

This class represents the page setup common dialog.

The page setup dialog contains controls for paper size (letter, A4, A5 etc.), orientation (landscape or portrait), and, only under Windows currently, controls for setting left, top, right and bottom margin sizes in millimetres.

The exact appearance of this dialog varies among the platforms as a native dialog is used when available (currently the case for all major platforms).

When the dialog has been closed, you need to query the `wxPageSetupDialogData` object associated with the dialog.

Note that the OK and Cancel buttons do not destroy the dialog; this must be done by the application.

See: **Overview printing**, `wxPrintDialog`, `wxPageSetupDialogData`

wxWidgets docs: **wxPageSetupDialog**

## Data Types

```
wxPageSetupDialog() = wx:wx_object()
```

## Exports

```
new(Parent) -> wxPageSetupDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxPageSetupDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
Option = {data, wxPageSetupDialogData:wxPageSetupDialogData()}
```

Constructor.

Pass a parent window, and optionally a pointer to a block of page setup data, which will be copied to the print dialog's internal data.

```
destroy(This :: wxPageSetupDialog()) -> ok
```

Destructor.

```
getPageSetupData(This) ->
                    wxPageSetupDialogData:wxPageSetupDialogData()
```

Types:

```
This = wxPageSetupDialog()
```

Returns the `wxPageSetupDialogData` object associated with the dialog.

```
showModal(This) -> integer()
```

Types:

```
This = wxPageSetupDialog()
```

Shows the dialog, returning `wxID_OK` if the user pressed OK, and `wxID_CANCEL` otherwise.

# wxPageSetupDialogData

Erlang module

This class holds a variety of information related to `wxPageSetupDialog`.

It contains a `wxPrintData` member which is used to hold basic printer configuration data (as opposed to the user-interface configuration settings stored by `wxPageSetupDialogData`).

See: **Overview printing**, `wxPageSetupDialog`

wxWidgets docs: **wxPageSetupDialogData**

## Data Types

```
wxPageSetupDialogData() = wx:wx_object()
```

## Exports

```
new() -> wxPageSetupDialogData()
```

Default constructor.

```
new(PrintData) -> wxPageSetupDialogData()
```

Types:

```
   PrintData =
       wxPrintData:wxPrintData() |
       wxPageSetupDialogData:wxPageSetupDialogData()
```

Construct an object from a print data object.

```
destroy(This :: wxPageSetupDialogData()) -> ok
```

Destructor.

```
enableHelp(This, Flag) -> ok
```

Types:

```
   This = wxPageSetupDialogData()
   Flag = boolean()
```

Enables or disables the "Help" button (Windows only).

```
enableMargins(This, Flag) -> ok
```

Types:

```
   This = wxPageSetupDialogData()
   Flag = boolean()
```

Enables or disables the margin controls (Windows only).

```
enableOrientation(This, Flag) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Flag = boolean()
```

Enables or disables the orientation control (Windows only).

```
enablePaper(This, Flag) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Flag = boolean()
```

Enables or disables the paper size control (Windows only).

```
enablePrinter(This, Flag) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Flag = boolean()
```

Enables or disables the "Printer" button, which invokes a printer setup dialog.

```
getDefaultMinMargins(This) -> boolean()
```

Types:

```
This = wxPageSetupDialogData()
```

Returns true if the page setup dialog will take its minimum margin values from the currently selected printer properties (Windows only).

```
getEnableMargins(This) -> boolean()
```

Types:

```
This = wxPageSetupDialogData()
```

Returns true if the margin controls are enabled (Windows only).

```
getEnableOrientation(This) -> boolean()
```

Types:

```
This = wxPageSetupDialogData()
```

Returns true if the orientation control is enabled (Windows only).

```
getEnablePaper(This) -> boolean()
```

Types:

```
This = wxPageSetupDialogData()
```

Returns true if the paper size control is enabled (Windows only).

```
getEnablePrinter(This) -> boolean()
```

Types:

```
This = wxPageSetupDialogData()
```

Returns true if the printer setup button is enabled.

```
getEnableHelp(This) -> boolean()
```

Types:

```
This = wxPageSetupDialogData()
```

Returns true if the printer setup button is enabled.

```
getDefaultInfo(This) -> boolean()
```

Types:

```
This = wxPageSetupDialogData()
```

Returns true if the dialog will simply return default printer information (such as orientation) instead of showing a dialog (Windows only).

```
getMarginTopLeft(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxPageSetupDialogData()
```

Returns the left (x) and top (y) margins in millimetres.

```
getMarginBottomRight(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxPageSetupDialogData()
```

Returns the right (x) and bottom (y) margins in millimetres.

```
getMinMarginTopLeft(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxPageSetupDialogData()
```

Returns the left (x) and top (y) minimum margins the user can enter (Windows only).

Units are in millimetres.

```
getMinMarginBottomRight(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxPageSetupDialogData()
```

Returns the right (x) and bottom (y) minimum margins the user can enter (Windows only).

Units are in millimetres.

```
getPaperId(This) -> wx:wx_enum()
```

Types:

```
This = wxPageSetupDialogData()
```

Returns the paper id (stored in the internal `wxPrintData` object).

See: `wxPrintData:setPaperId/2`

```
getPaperSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxPageSetupDialogData()
```

Returns the paper size in millimetres.

```
getPrintData(This) -> wxPrintData:wxPrintData()
```

Types:

```
This = wxPageSetupDialogData()
```

```
isOk(This) -> boolean()
```

Types:

```
This = wxPageSetupDialogData()
```

Returns true if the print data associated with the dialog data is valid.

This can return false on Windows if the current printer is not set, for example. On all other platforms, it returns true.

```
setDefaultInfo(This, Flag) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Flag = boolean()
```

Pass true if the dialog will simply return default printer information (such as orientation) instead of showing a dialog (Windows only).

```
setDefaultMinMargins(This, Flag) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Flag = boolean()
```

Pass true if the page setup dialog will take its minimum margin values from the currently selected printer properties (Windows only).

Units are in millimetres.

```
setMarginTopLeft(This, Pt) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Pt = {X :: integer(), Y :: integer()}
```

Sets the left (x) and top (y) margins in millimetres.

```
setMarginBottomRight(This, Pt) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Pt = {X :: integer(), Y :: integer()}
```

Sets the right (x) and bottom (y) margins in millimetres.

```
setMinMarginTopLeft(This, Pt) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Pt = {X :: integer(), Y :: integer()}
```

Sets the left (x) and top (y) minimum margins the user can enter (Windows only).

Units are in millimetres.

```
setMinMarginBottomRight(This, Pt) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Pt = {X :: integer(), Y :: integer()}
```

Sets the right (x) and bottom (y) minimum margins the user can enter (Windows only).

Units are in millimetres.

```
setPaperId(This, Id) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Id = wx:wx_enum()
```

Sets the paper size id.

Calling this function overrides the explicit paper dimensions passed in `setPaperSize/2`.

See: `wxPrintData:setPaperId/2`

```
setPaperSize(This, Size) -> ok
```

Types:

```
This = wxPageSetupDialogData()
Size = {W :: integer(), H :: integer()}
```

Sets the paper size in millimetres.

If a corresponding paper id is found, it will be set in the internal `wxPrintData` object, otherwise the paper size overrides the paper id.

```
setPrintData(This, PrintData) -> ok
```

Types:

```
This = wxPageSetupDialogData()
PrintData = wxPrintData:wxPrintData()
```

Sets the print data associated with this object.

# wxPaintDC

Erlang module

A `wxPaintDC` must be constructed if an application wishes to paint on the client area of a window from within an EVT_PAINT() event handler. This should normally be constructed as a temporary stack object; don't store a `wxPaintDC` object. If you have an EVT_PAINT() handler, you `must` create a `wxPaintDC` object within it even if you don't actually use it.

Using `wxPaintDC` within your EVT_PAINT() handler is important because it automatically sets the clipping area to the damaged area of the window. Attempts to draw outside this area do not appear.

A `wxPaintDC` object is initialized to use the same font and colours as the window it is associated with.

See: `wxDC`, `wxClientDC`, `wxMemoryDC`, `wxWindowDC`, `wxScreenDC`

This class is derived (and can use functions) from: `wxWindowDC` `wxDC`

wxWidgets docs: **wxPaintDC**

## Data Types

```
wxPaintDC() = wx:wx_object()
```

## Exports

```
new(Window) -> wxPaintDC()
```

Types:

```
Window = wxWindow:wxWindow()
```

Constructor.

Pass a pointer to the window on which you wish to paint.

```
destroy(This :: wxPaintDC()) -> ok
```

Destroys the object.

# wxPaintEvent

Erlang module

A paint event is sent when a window's contents needs to be repainted.

The handler of this event must create a `wxPaintDC` object and use it for painting the window contents. For example:

Notice that you must `not` create other kinds of `wxDC` (e.g. `wxClientDC` or `wxWindowDC`) in EVT_PAINT handlers and also don't create `wxPaintDC` outside of this event handlers.

You can optimize painting by retrieving the rectangles that have been damaged and only repainting these. The rectangles are in terms of the client area, and are unscrolled, so you will need to do some calculations using the current view position to obtain logical, scrolled units. Here is an example of using the `wxRegionIterator` (not implemented in wx) class:

Remark: Please notice that in general it is impossible to change the drawing of a standard control (such as `wxButton`) and so you shouldn't attempt to handle paint events for them as even if it might work on some platforms, this is inherently not portable and won't work everywhere.

See: **Overview events**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxPaintEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxPaintEventType` to subscribe to events of this type.

## Data Types

```
wxPaintEvent() = wx:wx_object()
wxPaint() = #wxPaint{type = wxPaintEvent:wxPaintEventType()}
wxPaintEventType() = paint
```

# wxPalette

Erlang module

A palette is a table that maps pixel values to RGB colours. It allows the colours of a low-depth bitmap, for example, to be mapped to the available colours in a display. The notion of palettes is becoming more and more obsolete nowadays and only the MSW port is still using a native palette. All other ports use generic code which is basically just an array of colours.

It is likely that in the future the only use for palettes within wxWidgets will be for representing colour indices from images (such as GIF or PNG). The image handlers for these formats have been modified to create a palette if there is such information in the original image file (usually 256 or less colour images). See `wxImage` for more information.

Predefined objects (include wx.hrl): ?wxNullPalette

See: `wxDC:setPalette/2`, `wxBitmap`

wxWidgets docs: **wxPalette**

## Data Types

```
wxPalette() = wx:wx_object()
```

## Exports

```
new() -> wxPalette()
```

Default constructor.

```
new(Palette) -> wxPalette()
```

Types:

```
Palette = wxPalette()
```

Copy constructor, uses overview_refcount.

```
new(Red, Green, Blue) -> wxPalette()
```

Types:

```
Red = Green = Blue = binary()
```

Creates a palette from arrays of size `n`, one for each red, blue or green component.

See: `create/4`

```
destroy(This :: wxPalette()) -> ok
```

Destructor.

See: reference-counted object destruction

```
create(This, Red, Green, Blue) -> boolean()
```

Types:

```
This = wxPalette()
Red = Green = Blue = binary()
```

Creates a palette from arrays of size `n`, one for each red, blue or green component.

Return: true if the creation was successful, false otherwise.

See: `new/3`

```
getColoursCount(This) -> integer()
```

Types:

```
This = wxPalette()
```

Returns number of entries in palette.

```
getPixel(This, Red, Green, Blue) -> integer()
```

Types:

```
This = wxPalette()
Red = Green = Blue = integer()
```

Returns a pixel value (index into the palette) for the given RGB values.

Return: The nearest palette index or `wxNOT_FOUND` for unexpected errors.

See: `getRGB/2`

```
getRGB(This, Pixel) -> Result
```

Types:

```
Result =
    {Res :: boolean(),
     Red :: integer(),
     Green :: integer(),
     Blue :: integer()}
This = wxPalette()
Pixel = integer()
```

Returns RGB values for a given palette index.

Return: true if the operation was successful.

See: `getPixel/4`

```
ok(This) -> boolean()
```

Types:

```
This = wxPalette()
```

See: `isOk/1`.

```
isOk(This) -> boolean()
```

Types:

```
This = wxPalette()
```

Returns true if palette data is present.

# wxPaletteChangedEvent

Erlang module

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxPaletteChangedEvent**

## Data Types

```
wxPaletteChangedEvent() = wx:wx_object()
wxPaletteChanged() =
    #wxPaletteChanged{type =
                          wxPaletteChangedEvent:wxPaletteChangedEventType()}
wxPaletteChangedEventType() = palette_changed
```

## Exports

```
setChangedWindow(This, Win) -> ok
```

Types:

```
   This = wxPaletteChangedEvent()
   Win = wxWindow:wxWindow()
```

```
getChangedWindow(This) -> wxWindow:wxWindow()
```

Types:

```
   This = wxPaletteChangedEvent()
```

# wxPanel

Erlang module

A panel is a window on which controls are placed. It is usually placed within a frame. Its main feature over its parent class `wxWindow` is code for handling child windows and TAB traversal, which is implemented natively if possible (e.g. in wxGTK) or by wxWidgets itself otherwise.

Note: Tab traversal is implemented through an otherwise undocumented intermediate wxControlContainer class from which any class can derive in addition to the normal `wxWindow` base class. Please see and to find out how this is achieved.

Note: if not all characters are being intercepted by your OnKeyDown or OnChar handler, it may be because you are using the `wxTAB_TRAVERSAL` style, which grabs some keypresses for use by child controls.

Remark: By default, a panel has the same colouring as a dialog.

See: `wxDialog`

This class is derived (and can use functions) from: `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxPanel**

## Events

Event types emitted from this class: `navigation_key`

## Data Types

```
wxPanel() = wx:wx_object()
```

## Exports

```
new() -> wxPanel()
```

Default constructor.

```
new(Parent) -> wxPanel()
```

Types:

```
Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxPanel()
```

Types:

```
Parent = wxWindow:wxWindow()
Option =
    {winid, integer()} |
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Constructor.

See: `Create()` (not implemented in wx)

```
destroy(This :: wxPanel()) -> ok
```

Destructor.

Deletes any child windows before deleting the physical window.

```
initDialog(This) -> ok
```

Types:

```
This = wxPanel()
```

Sends a `wxInitDialogEvent`, which in turn transfers data to the dialog via validators.

See: `wxInitDialogEvent`

```
setFocusIgnoringChildren(This) -> ok
```

Types:

```
This = wxPanel()
```

In contrast to `wxWindow:setFocus/1` (see above) this will set the focus to the panel even if there are child windows in the panel.

This is only rarely needed.

# wxPasswordEntryDialog

Erlang module

This class represents a dialog that requests a one-line password string from the user.

It is implemented as a generic wxWidgets dialog.

See: **Overview cmndlg**

This class is derived (and can use functions) from: `wxTextEntryDialog` `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxPasswordEntryDialog**

## Data Types

```
wxPasswordEntryDialog() = wx:wx_object()
```

## Exports

```
new(Parent, Message) -> wxPasswordEntryDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Message = unicode:chardata()
```

```
new(Parent, Message, Options :: [Option]) ->
       wxPasswordEntryDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Message = unicode:chardata()
   Option =
       {caption, unicode:chardata()} |
       {value, unicode:chardata()} |
       {style, integer()} |
       {pos, {X :: integer(), Y :: integer()}}
```

Constructor.

Use `wxDialog:showModal/1` to show the dialog.

```
destroy(This :: wxPasswordEntryDialog()) -> ok
```

Destroys the object.

# wxPen

Erlang module

A pen is a drawing tool for drawing outlines. It is used for drawing lines and painting the outline of rectangles, ellipses, etc. It has a colour, a width and a style.

Note: On a monochrome display, wxWidgets shows all non-white pens as black.

Do not initialize objects on the stack before the program commences, since other required structures may not have been set up yet. Instead, define global pointers to objects and create them in `wxApp::OnInit()` (not implemented in wx) or when required.

An application may wish to dynamically create pens with different characteristics, and there is the consequent danger that a large number of duplicate pens will be created. Therefore an application may wish to get a pointer to a pen by using the global list of pens ?wxThePenList, and calling the member function `wxPenList::FindOrCreatePen()` (not implemented in wx). See `wxPenList` (not implemented in wx) for more info.

This class uses reference counting and copy-on-write internally so that assignments between two instances of this class are very cheap. You can therefore use actual objects instead of pointers without efficiency problems. If an instance of this class is changed it will create its own data internally so that other instances, which previously shared the data using the reference counting, are not affected.

Predefined objects (include wx.hrl):

See: `wxPenList` (not implemented in wx), `wxDC`, `wxDC:setPen/2`

wxWidgets docs: **wxPen**

## Data Types

```
wxPen() = wx:wx_object()
```

## Exports

```
new() -> wxPen()
```

Default constructor.

The pen will be uninitialised, and `isOk/1` will return false.

```
new(Colour) -> wxPen()
new(Pen) -> wxPen()
```

Types:

```
Pen = wxPen()
```

Copy constructor, uses overview_refcount.

```
new(Colour, Options :: [Option]) -> wxPen()
```

Types:

```
Colour = wx:wx_colour()
Option = {width, integer()} | {style, wx:wx_enum()}
```

Constructs a pen from a colour object, pen width and style.

Remark: Different versions of Windows and different versions of other platforms support very different subsets of the styles above so handle with care.

See: setStyle/2, setColour/4, setWidth/2

```
destroy(This :: wxPen()) -> ok
```

Destructor.

See: reference-counted object destruction

Remark: Although all remaining pens are deleted when the application exits, the application should try to clean up all pens itself. This is because wxWidgets cannot know if a pointer to the pen object is stored in an application data structure, and there is a risk of double deletion.

```
getCap(This) -> wx:wx_enum()
```

Types:

```
This = wxPen()
```

Returns the pen cap style, which may be one of wxCAP_ROUND, wxCAP_PROJECTING and wxCAP_BUTT.

The default is wxCAP_ROUND.

See: setCap/2

```
getColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxPen()
```

Returns a reference to the pen colour.

See: setColour/4

```
getJoin(This) -> wx:wx_enum()
```

Types:

```
This = wxPen()
```

Returns the pen join style, which may be one of wxJOIN_BEVEL, wxJOIN_ROUND and wxJOIN_MITER.

The default is wxJOIN_ROUND.

See: setJoin/2

```
getStyle(This) -> wx:wx_enum()
```

Types:

```
This = wxPen()
```

Returns the pen style.

See: new/2, setStyle/2

```
getWidth(This) -> integer()
```

Types:

```
This = wxPen()
```

Returns the pen width.

See: setWidth/2

```
isOk(This) -> boolean()
```

Types:

```
This = wxPen()
```

Returns true if the pen is initialised.

Notice that an uninitialized pen object can't be queried for any pen properties and all calls to the accessor methods on it will result in an assert failure.

```
setCap(This, CapStyle) -> ok
```

Types:

```
This = wxPen()
CapStyle = wx:wx_enum()
```

Sets the pen cap style, which may be one of `wxCAP_ROUND`, `wxCAP_PROJECTING` and `wxCAP_BUTT`.

The default is `wxCAP_ROUND`.

See: `getCap/1`

```
setColour(This, Colour) -> ok
```

Types:

```
This = wxPen()
Colour = wx:wx_colour()
```

The pen's colour is changed to the given colour.

See: `getColour/1`

```
setColour(This, Red, Green, Blue) -> ok
```

Types:

```
This = wxPen()
Red = Green = Blue = integer()
```

```
setJoin(This, Join_style) -> ok
```

Types:

```
This = wxPen()
Join_style = wx:wx_enum()
```

Sets the pen join style, which may be one of `wxJOIN_BEVEL`, `wxJOIN_ROUND` and `wxJOIN_MITER`.

The default is `wxJOIN_ROUND`.

See: `getJoin/1`

```
setStyle(This, Style) -> ok
```

Types:

```
This = wxPen()
Style = wx:wx_enum()
```

Set the pen style.

See: `new/2`

```
setWidth(This, Width) -> ok
```

Types:

```
This = wxPen()
Width = integer()
```

Sets the pen width.

See: `getWidth/1`

# wxPickerBase

Erlang module

Base abstract class for all pickers which support an auxiliary text control.

This class handles all positioning and sizing of the text control like a an horizontal `wxBoxSizer` would do, with the text control on the left of the picker button.

The proportion (see `wxSizer` documentation for more info about proportion values) of the picker control defaults to 1 when there isn't a text control associated (see `wxPB_USE_TEXTCTRL` style) and to 0 otherwise.

Styles

This class supports the following styles:

See: `wxColourPickerCtrl`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxPickerBase**

## Data Types

```
wxPickerBase() = wx:wx_object()
```

## Exports

```
setInternalMargin(This, Margin) -> ok
```

Types:

```
This = wxPickerBase()
Margin = integer()
```

Sets the margin (in pixel) between the picker and the text control.

This function can be used only when `hasTextCtrl/1` returns true.

```
getInternalMargin(This) -> integer()
```

Types:

```
This = wxPickerBase()
```

Returns the margin (in pixel) between the picker and the text control.

This function can be used only when `hasTextCtrl/1` returns true.

```
setTextCtrlProportion(This, Prop) -> ok
```

Types:

```
This = wxPickerBase()
Prop = integer()
```

Sets the proportion value of the text control.

Look at the detailed description of `wxPickerBase` for more info.

This function can be used only when `hasTextCtrl/1` returns true.

```
setPickerCtrlProportion(This, Prop) -> ok
```

Types:

```
This = wxPickerBase()
Prop = integer()
```

Sets the proportion value of the picker.

Look at the detailed description of `wxPickerBase` for more info.

```
getTextCtrlProportion(This) -> integer()
```

Types:

```
This = wxPickerBase()
```

Returns the proportion value of the text control.

This function can be used only when `hasTextCtrl/1` returns true.

```
getPickerCtrlProportion(This) -> integer()
```

Types:

```
This = wxPickerBase()
```

Returns the proportion value of the picker.

```
hasTextCtrl(This) -> boolean()
```

Types:

```
This = wxPickerBase()
```

Returns true if this window has a valid text control (i.e. if the `wxPB_USE_TEXTCTRL` style was given when creating this control).

```
getTextCtrl(This) -> wxTextCtrl:wxTextCtrl()
```

Types:

```
This = wxPickerBase()
```

Returns a pointer to the text control handled by this window or NULL if the `wxPB_USE_TEXTCTRL` style was not specified when this control was created.

Remark: The contents of the text control could be an invalid representation of the entity which can be chosen through the picker (e.g. when the user enters an invalid colour syntax because of a typo). Thus you should never parse the content of the textctrl to get the user's input; rather use the derived-class getter (e.g. `wxColourPickerCtrl:getColour/1`, `wxFilePickerCtrl:getPath/1`, etc).

```
isTextCtrlGrowable(This) -> boolean()
```

Types:

```
This = wxPickerBase()
```

Returns true if the text control is growable.

This function can be used only when `hasTextCtrl/1` returns true.

```
setPickerCtrlGrowable(This) -> ok
```

Types:

```
   This = wxPickerBase()
```

```
setPickerCtrlGrowable(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxPickerBase()
   Option = {grow, boolean()}
```

Sets the picker control as growable when `grow` is true.

```
setTextCtrlGrowable(This) -> ok
```

Types:

```
   This = wxPickerBase()
```

```
setTextCtrlGrowable(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxPickerBase()
   Option = {grow, boolean()}
```

Sets the text control as growable when `grow` is true.

This function can be used only when `hasTextCtrl/1` returns true.

```
isPickerCtrlGrowable(This) -> boolean()
```

Types:

```
   This = wxPickerBase()
```

Returns true if the picker control is growable.

# wxPopupTransientWindow

Erlang module

A `wxPopupWindow` which disappears automatically when the user clicks mouse outside it or if it loses focus in any other way.

This window can be useful for implementing custom combobox-like controls for example.

See: `wxPopupWindow`

This class is derived (and can use functions) from: `wxPopupWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxPopupTransientWindow**

## Data Types

```
wxPopupTransientWindow() = wx:wx_object()
```

## Exports

```
new() -> wxPopupTransientWindow()
```

Default constructor.

```
new(Parent) -> wxPopupTransientWindow()
```

Types:

```
Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxPopupTransientWindow()
```

Types:

```
Parent = wxWindow:wxWindow()
Option = {style, integer()}
```

Constructor.

```
popup(This) -> ok
```

Types:

```
This = wxPopupTransientWindow()
```

```
popup(This, Options :: [Option]) -> ok
```

Types:

```
This = wxPopupTransientWindow()
Option = {focus, wxWindow:wxWindow()}
```

Popup the window (this will show it too).

If `focus` is non-NULL, it will be kept focused while this window is shown if supported by the current platform, otherwise the popup itself will receive focus. In any case, the popup will disappear automatically if it loses focus because of a user action.

See: `dismiss/1`

```
dismiss(This) -> ok
```

Types:

```
   This = wxPopupTransientWindow()
```

Hide the window.

```
destroy(This :: wxPopupTransientWindow()) -> ok
```

Destroys the object.

# wxPopupWindow

Erlang module

A special kind of top level window used for popup menus, combobox popups and such.

Styles

This class supports the following styles:

See: `wxDialog`, `wxFrame`

This class is derived (and can use functions) from: `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxPopupWindow**

## Data Types

```
wxPopupWindow() = wx:wx_object()
```

## Exports

```
new() -> wxPopupWindow()
```

Default constructor.

```
new(Parent) -> wxPopupWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxPopupWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option = {flags, integer()}
```

Constructor.

```
create(This, Parent) -> boolean()
```

Types:

```
   This = wxPopupWindow()
   Parent = wxWindow:wxWindow()
```

```
create(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxPopupWindow()
   Parent = wxWindow:wxWindow()
   Option = {flags, integer()}
```

Create method for two-step creation.

```
position(This, PtOrigin, SizePopup) -> ok
```

Types:

```
This = wxPopupWindow()
PtOrigin = {X :: integer(), Y :: integer()}
SizePopup = {W :: integer(), H :: integer()}
```

Move the popup window to the right position, i.e. such that it is entirely visible.

The popup is positioned at ptOrigin + size if it opens below and to the right (default), at ptOrigin - sizePopup if it opens above and to the left etc.

```
destroy(This :: wxPopupWindow()) -> ok
```

Destroys the object.

# wxPostScriptDC

Erlang module

This defines the wxWidgets Encapsulated PostScript device context, which can write PostScript files on any platform. See `wxDC` for descriptions of the member functions.

Starting a document

Document should be started with call to `wxDC:startDoc/2` prior to calling any function to execute a drawing operation. However, some functions, like `wxDC:setFont/2`, may be legitimately called even before `wxDC:startDoc/2`.

This class is derived (and can use functions) from: `wxDC`

wxWidgets docs: **wxPostScriptDC**

## Data Types

```
wxPostScriptDC() = wx:wx_object()
```

## Exports

```
new() -> wxPostScriptDC()
```

```
new(PrintData) -> wxPostScriptDC()
```

Types:

```
PrintData = wxPrintData:wxPrintData()
```

Constructs a PostScript printer device context from a `wxPrintData` object.

```
destroy(This :: wxPostScriptDC()) -> ok
```

Destroys the object.

# wxPreviewCanvas

Erlang module

A preview canvas is the default canvas used by the print preview system to display the preview.

See: `wxPreviewFrame`, `wxPreviewControlBar`, `wxPrintPreview`

This class is derived (and can use functions) from: `wxScrolledWindow` `wxPanel` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxPreviewCanvas**

## Data Types

```
wxPreviewCanvas() = wx:wx_object()
```

# wxPreviewControlBar

Erlang module

This is the default implementation of the preview control bar, a panel with buttons and a zoom control.

You can derive a new class from this and override some or all member functions to change the behaviour and appearance; or you can leave it as it is.

See: `wxPreviewFrame`, `wxPreviewCanvas`, `wxPrintPreview`

This class is derived (and can use functions) from: `wxPanel` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxPreviewControlBar**

## Data Types

```
wxPreviewControlBar() = wx:wx_object()
```

## Exports

```
new(Preview, Buttons, Parent) -> wxPreviewControlBar()
```

Types:

```
   Preview = wxPrintPreview:wxPrintPreview()
   Buttons = integer()
   Parent = wxWindow:wxWindow()
```

```
new(Preview, Buttons, Parent, Options :: [Option]) ->
       wxPreviewControlBar()
```

Types:

```
   Preview = wxPrintPreview:wxPrintPreview()
   Buttons = integer()
   Parent = wxWindow:wxWindow()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructor.

The `buttons` parameter may be a combination of the following, using the bitwise 'or' operator:

```
destroy(This :: wxPreviewControlBar()) -> ok
```

Destructor.

```
createButtons(This) -> ok
```

Types:

```
   This = wxPreviewControlBar()
```

Creates buttons, according to value of the button style flags.

```
getPrintPreview(This) -> wxPrintPreview:wxPrintPreview()
```

Types:

```
This = wxPreviewControlBar()
```

Gets the print preview object associated with the control bar.

```
getZoomControl(This) -> integer()
```

Types:

```
This = wxPreviewControlBar()
```

Gets the current zoom setting in percent.

```
setZoomControl(This, Percent) -> ok
```

Types:

```
This = wxPreviewControlBar()
Percent = integer()
```

Sets the zoom control.

# wxPreviewFrame

Erlang module

This class provides the default method of managing the print preview interface. Member functions may be overridden to replace functionality, or the class may be used without derivation.

See: `wxPreviewCanvas`, `wxPreviewControlBar`, `wxPrintPreview`

This class is derived (and can use functions) from: `wxFrame` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxPreviewFrame**

## Data Types

```
wxPreviewFrame() = wx:wx_object()
```

## Exports

```
new(Preview, Parent) -> wxPreviewFrame()
```

Types:

```
Preview = wxPrintPreview:wxPrintPreview()
Parent = wxWindow:wxWindow()
```

```
new(Preview, Parent, Options :: [Option]) -> wxPreviewFrame()
```

Types:

```
Preview = wxPrintPreview:wxPrintPreview()
Parent = wxWindow:wxWindow()
Option =
    {title, unicode:chardata()} |
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Constructor.

Pass a print preview object plus other normal frame arguments. The print preview object will be destroyed by the frame when it closes.

```
destroy(This :: wxPreviewFrame()) -> ok
```

Destructor.

```
createControlBar(This) -> ok
```

Types:

```
This = wxPreviewFrame()
```

Creates a `wxPreviewControlBar`.

Override this function to allow a user-defined preview control bar object to be created.

```
createCanvas(This) -> ok
```

Types:

```
This = wxPreviewFrame()
```

Creates a `wxPreviewCanvas`.

Override this function to allow a user-defined preview canvas object to be created.

```
initialize(This) -> ok
```

Types:

```
This = wxPreviewFrame()
```

Initializes the frame elements and prepares for showing it.

Calling this method is equivalent to calling `InitializeWithModality()` (not implemented in wx) with wxPreviewFrame_AppModal argument, please see its documentation for more details.

Please notice that this function is virtual mostly for backwards compatibility only, there is no real need to override it as it's never called by wxWidgets itself.

```
onCloseWindow(This, Event) -> ok
```

Types:

```
This = wxPreviewFrame()
Event = wxCloseEvent:wxCloseEvent()
```

Enables any disabled frames in the application, and deletes the print preview object, implicitly deleting any printout objects associated with the print preview object.

# wxPrintData

Erlang module

This class holds a variety of information related to printers and printer device contexts. This class is used to create a `wxPrinterDC` (not implemented in wx) and a `wxPostScriptDC`. It is also used as a data member of `wxPrintDialogData` and `wxPageSetupDialogData`, as part of the mechanism for transferring data between the print dialogs and the application.

See: **Overview printing**, `wxPrintDialog`, `wxPageSetupDialog`, `wxPrintDialogData`, `wxPageSetupDialogData`, **Overview cmndlg**, `wxPrinterDC` (not implemented in wx), `wxPostScriptDC`

wxWidgets docs: **wxPrintData**

## Data Types

```
wxPrintData() = wx:wx_object()
```

## Exports

```
new() -> wxPrintData()
```

Default constructor.

```
new(Data) -> wxPrintData()
```

Types:

```
Data = wxPrintData()
```

Copy constructor.

```
destroy(This :: wxPrintData()) -> ok
```

Destructor.

```
getCollate(This) -> boolean()
```

Types:

```
This = wxPrintData()
```

Returns true if collation is on.

```
getBin(This) -> wx:wx_enum()
```

Types:

```
This = wxPrintData()
```

Returns the current bin (papersource).

By default, the system is left to select the bin (`wxPRINTBIN_DEFAULT` is returned).

See `setBin/2` for the full list of bin values.

```
getColour(This) -> boolean()
```

Types:

```
This = wxPrintData()
```

Returns true if colour printing is on.

```
getDuplex(This) -> wx:wx_enum()
```

Types:

```
This = wxPrintData()
```

Returns the duplex mode.

One of wxDUPLEX_SIMPLEX, wxDUPLEX_HORIZONTAL, wxDUPLEX_VERTICAL.

```
getNoCopies(This) -> integer()
```

Types:

```
This = wxPrintData()
```

Returns the number of copies requested by the user.

```
getOrientation(This) -> wx:wx_enum()
```

Types:

```
This = wxPrintData()
```

Gets the orientation.

This can be wxLANDSCAPE or wxPORTRAIT.

```
getPaperId(This) -> wx:wx_enum()
```

Types:

```
This = wxPrintData()
```

Returns the paper size id.

See: `setPaperId/2`

```
getPrinterName(This) -> unicode:charlist()
```

Types:

```
This = wxPrintData()
```

Returns the printer name.

If the printer name is the empty string, it indicates that the default printer should be used.

```
getQuality(This) -> integer()
```

Types:

```
This = wxPrintData()
```

Returns the current print quality.

This can be a positive integer, denoting the number of dots per inch, or one of the following identifiers:

On input you should pass one of these identifiers, but on return you may get back a positive integer indicating the current resolution setting.

```
isOk(This) -> boolean()
```

Types:

```
This = wxPrintData()
```

Returns true if the print data is valid for using in print dialogs.

This can return false on Windows if the current printer is not set, for example. On all other platforms, it returns true.

```
setBin(This, Flag) -> ok
```

Types:

```
This = wxPrintData()
Flag = wx:wx_enum()
```

Sets the current bin.

```
setCollate(This, Flag) -> ok
```

Types:

```
This = wxPrintData()
Flag = boolean()
```

Sets collation to on or off.

```
setColour(This, Flag) -> ok
```

Types:

```
This = wxPrintData()
Flag = boolean()
```

Sets colour printing on or off.

```
setDuplex(This, Mode) -> ok
```

Types:

```
This = wxPrintData()
Mode = wx:wx_enum()
```

Returns the duplex mode.

One of wxDUPLEX_SIMPLEX, wxDUPLEX_HORIZONTAL, wxDUPLEX_VERTICAL.

```
setNoCopies(This, N) -> ok
```

Types:

```
This = wxPrintData()
N = integer()
```

Sets the default number of copies to be printed out.

```
setOrientation(This, Orientation) -> ok
```

Types:

```
This = wxPrintData()
Orientation = wx:wx_enum()
```

Sets the orientation.

This can be wxLANDSCAPE or wxPORTRAIT.

```
setPaperId(This, PaperId) -> ok
```

Types:

```
This = wxPrintData()
PaperId = wx:wx_enum()
```

Sets the paper id.

This indicates the type of paper to be used. For a mapping between paper id, paper size and string name, see wxPrintPaperDatabase in `"paper.h"` (not yet documented).

See: `SetPaperSize()` (not implemented in wx)

```
setPrinterName(This, PrinterName) -> ok
```

Types:

```
This = wxPrintData()
PrinterName = unicode:chardata()
```

Sets the printer name.

This can be the empty string to indicate that the default printer should be used.

```
setQuality(This, Quality) -> ok
```

Types:

```
This = wxPrintData()
Quality = integer()
```

Sets the desired print quality.

This can be a positive integer, denoting the number of dots per inch, or one of the following identifiers:

On input you should pass one of these identifiers, but on return you may get back a positive integer indicating the current resolution setting.

# wxPrintDialog

Erlang module

This class represents the print and print setup common dialogs. You may obtain a `wxPrinterDC` (not implemented in wx) device context from a successfully dismissed print dialog.

See: **Overview printing**, **Overview cmndlg**

This class is derived (and can use functions) from: `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxPrintDialog**

## Data Types

```
wxPrintDialog() = wx:wx_object()
```

## Exports

```
new(Parent) -> wxPrintDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxPrintDialog()
new(Parent, Data) -> wxPrintDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
Data = wxPrintData:wxPrintData()
```

```
destroy(This :: wxPrintDialog()) -> ok
```

Destructor.

If `getPrintDC/1` has not been called, the device context obtained by the dialog (if any) will be deleted.

```
getPrintDialogData(This) -> wxPrintDialogData:wxPrintDialogData()
```

Types:

```
This = wxPrintDialog()
```

Returns the print dialog data associated with the print dialog.

```
getPrintDC(This) -> wxDC:wxDC()
```

Types:

```
This = wxPrintDialog()
```

Returns the device context created by the print dialog, if any.

When this function has been called, the ownership of the device context is transferred to the application, so it must then be deleted explicitly.

# wxPrintDialogData

Erlang module

This class holds information related to the visual characteristics of `wxPrintDialog`. It contains a `wxPrintData` object with underlying printing settings.

See: **Overview printing**, `wxPrintDialog`, **Overview cmndlg**

wxWidgets docs: **wxPrintDialogData**

## Data Types

```
wxPrintDialogData() = wx:wx_object()
```

## Exports

```
new() -> wxPrintDialogData()
```

Default constructor.

```
new(DialogData) -> wxPrintDialogData()
```

Types:

```
   DialogData =
       wxPrintDialogData:wxPrintDialogData() |
       wxPrintData:wxPrintData()
```

Copy constructor.

```
destroy(This :: wxPrintDialogData()) -> ok
```

Destructor.

```
enableHelp(This, Flag) -> ok
```

Types:

```
   This = wxPrintDialogData()
   Flag = boolean()
```

Enables or disables the "Help" button.

```
enablePageNumbers(This, Flag) -> ok
```

Types:

```
   This = wxPrintDialogData()
   Flag = boolean()
```

Enables or disables the "Page numbers" controls.

```
enablePrintToFile(This, Flag) -> ok
```

Types:

```
   This = wxPrintDialogData()
   Flag = boolean()
```

Enables or disables the "Print to file" checkbox.

```
enableSelection(This, Flag) -> ok
```

Types:

```
   This = wxPrintDialogData()
   Flag = boolean()
```

Enables or disables the "Selection" radio button.

```
getAllPages(This) -> boolean()
```

Types:

```
   This = wxPrintDialogData()
```

Returns true if the user requested that all pages be printed.

```
getCollate(This) -> boolean()
```

Types:

```
   This = wxPrintDialogData()
```

Returns true if the user requested that the document(s) be collated.

```
getFromPage(This) -> integer()
```

Types:

```
   This = wxPrintDialogData()
```

Returns the `from` page number, as entered by the user.

```
getMaxPage(This) -> integer()
```

Types:

```
   This = wxPrintDialogData()
```

Returns the `maximum` page number.

```
getMinPage(This) -> integer()
```

Types:

```
   This = wxPrintDialogData()
```

Returns the `minimum` page number.

```
getNoCopies(This) -> integer()
```

Types:

```
   This = wxPrintDialogData()
```

Returns the number of copies requested by the user.

```
getPrintData(This) -> wxPrintData:wxPrintData()
```

Types:

```
This = wxPrintDialogData()
```

Returns a reference to the internal `wxPrintData` object.

```
getPrintToFile(This) -> boolean()
```

Types:

```
This = wxPrintDialogData()
```

Returns true if the user has selected printing to a file.

```
getSelection(This) -> boolean()
```

Types:

```
This = wxPrintDialogData()
```

Returns true if the user requested that the selection be printed (where "selection" is a concept specific to the application).

```
getToPage(This) -> integer()
```

Types:

```
This = wxPrintDialogData()
```

Returns the `"print to"` page number, as entered by the user.

```
isOk(This) -> boolean()
```

Types:

```
This = wxPrintDialogData()
```

Returns true if the print data is valid for using in print dialogs.

This can return false on Windows if the current printer is not set, for example. On all other platforms, it returns true.

```
setCollate(This, Flag) -> ok
```

Types:

```
This = wxPrintDialogData()
Flag = boolean()
```

Sets the "Collate" checkbox to true or false.

```
setFromPage(This, Page) -> ok
```

Types:

```
This = wxPrintDialogData()
Page = integer()
```

Sets the `from` page number.

```
setMaxPage(This, Page) -> ok
```

Types:

```
This = wxPrintDialogData()
Page = integer()
```

Sets the `maximum` page number.

```
setMinPage(This, Page) -> ok
```

Types:

```
This = wxPrintDialogData()
Page = integer()
```

Sets the `minimum` page number.

```
setNoCopies(This, N) -> ok
```

Types:

```
This = wxPrintDialogData()
N = integer()
```

Sets the default number of copies the user has requested to be printed out.

```
setPrintData(This, PrintData) -> ok
```

Types:

```
This = wxPrintDialogData()
PrintData = wxPrintData:wxPrintData()
```

Sets the internal `wxPrintData`.

```
setPrintToFile(This, Flag) -> ok
```

Types:

```
This = wxPrintDialogData()
Flag = boolean()
```

Sets the "Print to file" checkbox to true or false.

```
setSelection(This, Flag) -> ok
```

Types:

```
This = wxPrintDialogData()
Flag = boolean()
```

Selects the "Selection" radio button.

The effect of printing the selection depends on how the application implements this command, if at all.

```
setToPage(This, Page) -> ok
```

Types:

```
This = wxPrintDialogData()
Page = integer()
```

Sets the `"print to"` page number.

# wxPrintPreview

Erlang module

Objects of this class manage the print preview process. The object is passed a `wxPrintout` object, and the `wxPrintPreview` object itself is passed to a `wxPreviewFrame` object. Previewing is started by initializing and showing the preview frame. Unlike `wxPrinter:print/4`, flow of control returns to the application immediately after the frame is shown.

Note: The preview shown is only exact on Windows. On other platforms, the `wxDC` used for preview is different from what is used for printing and the results may be significantly different, depending on how is the output created. In particular, printing code relying on `wxDC:getTextExtent/3` heavily (for example, `wxHtmlEasyPrinting` and other wxHTML classes do) is affected. It is recommended to use native preview functionality on platforms that offer it (macOS, GTK+).

See: **Overview printing**, `wxPrinterDC` (not implemented in wx), `wxPrintDialog`, `wxPrintout`, `wxPrinter`, `wxPreviewCanvas`, `wxPreviewControlBar`, `wxPreviewFrame`

wxWidgets docs: **wxPrintPreview**

## Data Types

```
wxPrintPreview() = wx:wx_object()
```

## Exports

```
new(Printout) -> wxPrintPreview()
```

Types:

```
   Printout = wxPrintout:wxPrintout()
```

```
new(Printout, Options :: [Option]) -> wxPrintPreview()
```

Types:

```
   Printout = wxPrintout:wxPrintout()
   Option =
       {printoutForPrinting, wxPrintout:wxPrintout()} |
       {data, wxPrintDialogData:wxPrintDialogData()}
```

Constructor.

Pass a printout object, an optional printout object to be used for actual printing, and the address of an optional block of printer data, which will be copied to the print preview object's print data.

If `printoutForPrinting` is non-NULL, a `"Print..."` button will be placed on the preview frame so that the user can print directly from the preview interface.

Remark: Do not explicitly delete the printout objects once this constructor has been called, since they will be deleted in the `wxPrintPreview` destructor. The same does not apply to the `data` argument.

Use `isOk/1` to check whether the `wxPrintPreview` object was created correctly.

```
new(Printout, PrintoutForPrinting, Data) -> wxPrintPreview()
```

Types:

```
   Printout = PrintoutForPrinting = wxPrintout:wxPrintout()
   Data = wxPrintData:wxPrintData()
```

```
destroy(This :: wxPrintPreview()) -> ok
```

Destructor.

Deletes both print preview objects, so do not destroy these objects in your application.

```
getCanvas(This) -> wxPreviewCanvas:wxPreviewCanvas()
```

Types:

```
   This = wxPrintPreview()
```

Gets the preview window used for displaying the print preview image.

```
getCurrentPage(This) -> integer()
```

Types:

```
   This = wxPrintPreview()
```

Gets the page currently being previewed.

```
getFrame(This) -> wxFrame:wxFrame()
```

Types:

```
   This = wxPrintPreview()
```

Gets the frame used for displaying the print preview canvas and control bar.

```
getMaxPage(This) -> integer()
```

Types:

```
   This = wxPrintPreview()
```

Returns the maximum page number.

```
getMinPage(This) -> integer()
```

Types:

```
   This = wxPrintPreview()
```

Returns the minimum page number.

```
getPrintout(This) -> wxPrintout:wxPrintout()
```

Types:

```
   This = wxPrintPreview()
```

Gets the preview printout object associated with the `wxPrintPreview` object.

```
getPrintoutForPrinting(This) -> wxPrintout:wxPrintout()
```

Types:

```
   This = wxPrintPreview()
```

Gets the printout object to be used for printing from within the preview interface, or NULL if none exists.

```
isOk(This) -> boolean()
```

Types:

```
This = wxPrintPreview()
```

Returns true if the `wxPrintPreview` is valid, false otherwise.

It could return false if there was a problem initializing the printer device context (current printer not set, for example).

```
paintPage(This, Canvas, Dc) -> boolean()
```

Types:

```
This = wxPrintPreview()
Canvas = wxPreviewCanvas:wxPreviewCanvas()
Dc = wxDC:wxDC()
```

This refreshes the preview window with the preview image.

It must be called from the preview window's OnPaint member.

The implementation simply blits the preview bitmap onto the canvas, creating a new preview bitmap if none exists.

```
print(This, Prompt) -> boolean()
```

Types:

```
This = wxPrintPreview()
Prompt = boolean()
```

Invokes the print process using the second `wxPrintout` object supplied in the `wxPrintPreview` constructor.

Will normally be called by the `Print`... panel item on the preview frame's control bar.

Returns false in case of error - call `wxPrinter:getLastError/0` to get detailed information about the kind of the error.

```
renderPage(This, PageNum) -> boolean()
```

Types:

```
This = wxPrintPreview()
PageNum = integer()
```

Renders a page into a `wxMemoryDC`.

Used internally by `wxPrintPreview`.

```
setCanvas(This, Window) -> ok
```

Types:

```
This = wxPrintPreview()
Window = wxPreviewCanvas:wxPreviewCanvas()
```

Sets the window to be used for displaying the print preview image.

```
setCurrentPage(This, PageNum) -> boolean()
```

Types:

```
This = wxPrintPreview()
PageNum = integer()
```

Sets the current page to be previewed.

```
setFrame(This, Frame) -> ok
```

Types:

```
This = wxPrintPreview()
Frame = wxFrame:wxFrame()
```

Sets the frame to be used for displaying the print preview canvas and control bar.

```
setPrintout(This, Printout) -> ok
```

Types:

```
This = wxPrintPreview()
Printout = wxPrintout:wxPrintout()
```

Associates a printout object with the `wxPrintPreview` object.

```
setZoom(This, Percent) -> ok
```

Types:

```
This = wxPrintPreview()
Percent = integer()
```

Sets the percentage preview zoom, and refreshes the preview canvas accordingly.

# wxPrinter

Erlang module

This class represents the Windows or PostScript printer, and is the vehicle through which printing may be launched by an application.

Printing can also be achieved through using of lower functions and classes, but this and associated classes provide a more convenient and general method of printing.

See: **Overview printing**, `wxPrinterDC` (not implemented in wx), `wxPrintDialog`, `wxPrintout`, `wxPrintPreview`

wxWidgets docs: **wxPrinter**

## Data Types

```
wxPrinter() = wx:wx_object()
```

## Exports

```
new() -> wxPrinter()
```

```
new(Options :: [Option]) -> wxPrinter()
```

Types:

```
Option = {data, wxPrintDialogData:wxPrintDialogData()}
```

Constructor.

Pass an optional pointer to a block of print dialog data, which will be copied to the printer object's local data.

See: `wxPrintDialogData`, `wxPrintData`

```
createAbortWindow(This, Parent, Printout) -> wxDialog:wxDialog()
```

Types:

```
This = wxPrinter()
Parent = wxWindow:wxWindow()
Printout = wxPrintout:wxPrintout()
```

Creates the default printing abort window, with a cancel button.

```
getAbort(This) -> boolean()
```

Types:

```
This = wxPrinter()
```

Returns true if the user has aborted the print job.

```
getLastError() -> wx:wx_enum()
```

Return last error.

Valid after calling `print/4`, `printDialog/2` or `wxPrintPreview:print/2`.

These functions set last error to `wxPRINTER_NO_ERROR` if no error happened.

Returned value is one of the following:

```
getPrintDialogData(This) -> wxPrintDialogData:wxPrintDialogData()
```

Types:

```
This = wxPrinter()
```

Returns the print data associated with the printer object.

```
print(This, Parent, Printout) -> boolean()
```

Types:

```
This = wxPrinter()
Parent = wxWindow:wxWindow()
Printout = wxPrintout:wxPrintout()
```

```
print(This, Parent, Printout, Options :: [Option]) -> boolean()
```

Types:

```
This = wxPrinter()
Parent = wxWindow:wxWindow()
Printout = wxPrintout:wxPrintout()
Option = {prompt, boolean()}
```

Starts the printing process.

Provide a parent window, a user-defined `wxPrintout` object which controls the printing of a document, and whether the print dialog should be invoked first.

`print/4` could return false if there was a problem initializing the printer device context (current printer not set, for example) or the user cancelled printing. Call `getLastError/0` to get detailed information about the kind of the error.

```
printDialog(This, Parent) -> wxDC:wxDC()
```

Types:

```
This = wxPrinter()
Parent = wxWindow:wxWindow()
```

Invokes the print dialog.

If successful (the user did not press Cancel and no error occurred), a suitable device context will be returned; otherwise NULL is returned; call `getLastError/0` to get detailed information about the kind of the error.

Remark: The application must delete this device context to avoid a memory leak.

```
reportError(This, Parent, Printout, Message) -> ok
```

Types:

```
This = wxPrinter()
Parent = wxWindow:wxWindow()
Printout = wxPrintout:wxPrintout()
Message = unicode:chardata()
```

Default error-reporting function.

```
setup(This, Parent) -> boolean()
```

Types:

```
This = wxPrinter()
Parent = wxWindow:wxWindow()
```

Invokes the print setup dialog.

Deprecated: The setup dialog is obsolete, though retained for backward compatibility.

```
destroy(This :: wxPrinter()) -> ok
```

Destroys the object.

# wxPrintout

Erlang module

This class encapsulates the functionality of printing out an application document.

A new class must be derived and members overridden to respond to calls such as `OnPrintPage()` (not implemented in wx) and `HasPage()` (not implemented in wx) and to render the print image onto an associated `wxDC`. Instances of this class are passed to `wxPrinter:print/4` or to a `wxPrintPreview` object to initiate printing or previewing.

Your derived `wxPrintout` is responsible for drawing both the preview image and the printed page. If your windows' drawing routines accept an arbitrary DC as an argument, you can re-use those routines within your `wxPrintout` subclass to draw the printout image. You may also add additional drawing elements within your `wxPrintout` subclass, like headers, footers, and/or page numbers. However, the image on the printed page will often differ from the image drawn on the screen, as will the print preview image - not just in the presence of headers and footers, but typically in scale. A high-resolution printer presents a much larger drawing surface (i.e., a higher-resolution DC); a zoomed-out preview image presents a much smaller drawing surface (lower-resolution DC). By using the routines FitThisSizeToXXX() and/or MapScreenSizeToXXX() within your `wxPrintout` subclass to set the user scale and origin of the associated DC, you can easily use a single drawing routine to draw on your application's windows, to create the print preview image, and to create the printed paper image, and achieve a common appearance to the preview image and the printed page.

See: **Overview printing**, `wxPrinterDC` (not implemented in wx), `wxPrintDialog`, `wxPageSetupDialog`, `wxPrinter`, `wxPrintPreview`

wxWidgets docs: **wxPrintout**

## Data Types

```
wxPrintout() = wx:wx_object()
```

## Exports

```
new(Title :: string(), OnPrintPage, Opts :: [Option]) ->
       wxPrintout:wxPrintout()
```

Types:

```
   OnPrintPage =
       fun((wxPrintout(), Page :: integer()) -> boolean())
   Option =
       {onPreparePrinting, fun((wxPrintout()) -> ok)} |
       {onBeginPrinting, fun((wxPrintout()) -> ok)} |
       {onEndPrinting, fun((wxPrintout()) -> ok)} |
       {onBeginDocument,
        fun((wxPrintout(),
             StartPage :: integer(),
             EndPage :: integer()) ->
                boolean())} |
       {onEndDocument, fun((wxPrintout()) -> ok)} |
       {hasPage, fun((wxPrintout(), Page :: integer()) -> ok)} |
       {getPageInfo,
        fun((wxPrintout()) ->
               {MinPage :: integer(),
                MaxPage :: integer(),
```

```
                PageFrom :: integer(),
                PageTo :: integer()})}
```

Constructor.

Creates a `wxPrintout` object with a callback fun and optionally other callback funs. The `This` argument is the `wxPrintout` object reference to this object

Notice: The callbacks may not call other processes.

```
destroy(This :: wxPrintout()) -> ok
```

Destructor.

```
getDC(This) -> wxDC:wxDC()
```

Types:

```
   This = wxPrintout()
```

Returns the device context associated with the printout (given to the printout at start of printing or previewing).

The application can use `getDC/1` to obtain a device context to draw on.

This will be a `wxPrinterDC` (not implemented in wx) if printing under Windows or Mac, a `wxPostScriptDC` if printing on other platforms, and a `wxMemoryDC` if previewing.

```
getPageSizeMM(This) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxPrintout()
```

Returns the size of the printer page in millimetres.

```
getPageSizePixels(This) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxPrintout()
```

Returns the size of the printer page in pixels, called the page rectangle.

The page rectangle has a top left corner at (0,0) and a bottom right corner at (w,h). These values may not be the same as the values returned from `wxDC:getSize/1`; if the printout is being used for previewing, a memory device context is used, which uses a bitmap size reflecting the current preview zoom. The application must take this discrepancy into account if previewing is to be supported.

```
getPaperRectPixels(This) ->
                      {X :: integer(),
                       Y :: integer(),
                       W :: integer(),
                       H :: integer()}
```

Types:

```
   This = wxPrintout()
```

Returns the rectangle that corresponds to the entire paper in pixels, called the paper rectangle.

This distinction between paper rectangle and page rectangle reflects the fact that most printers cannot print all the way to the edge of the paper. The page rectangle is a rectangle whose top left corner is at (0,0) and whose width and height are given by wxDC::GetPageSizePixels().

On MSW and Mac, the page rectangle gives the printable area of the paper, while the paper rectangle represents the entire paper, including non-printable borders. Thus, the rectangle returned by wxDC::GetPaperRectPixels() will have a top left corner whose coordinates are small negative numbers and the bottom right corner will have values somewhat larger than the width and height given by wxDC::GetPageSizePixels().

On other platforms and for PostScript printing, the paper is treated as if its entire area were printable, so this function will return the same rectangle as the page rectangle.

```
getPPIPrinter(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxPrintout()
```

Returns the number of pixels per logical inch of the printer device context.

Dividing the printer PPI by the screen PPI can give a suitable scaling factor for drawing text onto the printer.

Remember to multiply this by a scaling factor to take the preview DC size into account. Or you can just use the FitThisSizeToXXX() and MapScreenSizeToXXX routines below, which do most of the scaling calculations for you.

```
getPPIScreen(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxPrintout()
```

Returns the number of pixels per logical inch of the screen device context.

Dividing the printer PPI by the screen PPI can give a suitable scaling factor for drawing text onto the printer.

If you are doing your own scaling, remember to multiply this by a scaling factor to take the preview DC size into account.

```
getTitle(This) -> unicode:charlist()
```

Types:

```
This = wxPrintout()
```

Returns the title of the printout.

```
isPreview(This) -> boolean()
```

Types:

```
This = wxPrintout()
```

Returns true if the printout is currently being used for previewing.

See: `GetPreview()` (not implemented in wx)

```
fitThisSizeToPaper(This, ImageSize) -> ok
```

Types:

```
This = wxPrintout()
ImageSize = {W :: integer(), H :: integer()}
```

Set the user scale and device origin of the `wxDC` associated with this `wxPrintout` so that the given image size fits entirely within the paper and the origin is at the top left corner of the paper.

Use this if you're managing your own page margins.

Note: With most printers, the region around the edges of the paper are not printable so that the edges of the image could be cut off.

`fitThisSizeToPage(This, ImageSize) -> ok`

Types:

```
This = wxPrintout()
ImageSize = {W :: integer(), H :: integer()}
```

Set the user scale and device origin of the `wxDC` associated with this `wxPrintout` so that the given image size fits entirely within the page rectangle and the origin is at the top left corner of the page rectangle.

On MSW and Mac, the page rectangle is the printable area of the page. On other platforms and PostScript printing, the page rectangle is the entire paper.

Use this if you want your printed image as large as possible, but with the caveat that on some platforms, portions of the image might be cut off at the edges.

`fitThisSizeToPageMargins(This, ImageSize, PageSetupData) -> ok`

Types:

```
This = wxPrintout()
ImageSize = {W :: integer(), H :: integer()}
PageSetupData = wxPageSetupDialogData:wxPageSetupDialogData()
```

Set the user scale and device origin of the `wxDC` associated with this `wxPrintout` so that the given image size fits entirely within the page margins set in the given `wxPageSetupDialogData` object.

This function provides the greatest consistency across all platforms because it does not depend on having access to the printable area of the paper.

Remark: On Mac, the native `wxPageSetupDialog` does not let you set the page margins; you'll have to provide your own mechanism, or you can use the Mac-only class wxMacPageMarginsDialog.

`mapScreenSizeToPaper(This) -> ok`

Types:

```
This = wxPrintout()
```

Set the user scale and device origin of the `wxDC` associated with this `wxPrintout` so that the printed page matches the screen size as closely as possible and the logical origin is in the top left corner of the paper rectangle.

That is, a 100-pixel object on screen should appear at the same size on the printed page. (It will, of course, be larger or smaller in the preview image, depending on the zoom factor.)

Use this if you want WYSIWYG behaviour, e.g., in a text editor.

`mapScreenSizeToPage(This) -> ok`

Types:

```
This = wxPrintout()
```

This sets the user scale of the `wxDC` associated with this `wxPrintout` to the same scale as `mapScreenSizeToPaper/1` but sets the logical origin to the top left corner of the page rectangle.

`mapScreenSizeToPageMargins(This, PageSetupData) -> ok`

Types:

```
This = wxPrintout()
PageSetupData = wxPageSetupDialogData:wxPageSetupDialogData()
```

This sets the user scale of the `wxDC` associated with this `wxPrintout` to the same scale as `mapScreenSizeToPageMargins/2` but sets the logical origin to the top left corner of the page margins specified by the given `wxPageSetupDialogData` object.

```
mapScreenSizeToDevice(This) -> ok
```

Types:

```
This = wxPrintout()
```

Set the user scale and device origin of the `wxDC` associated with this `wxPrintout` so that one screen pixel maps to one device pixel on the DC.

That is, the user scale is set to (1,1) and the device origin is set to (0,0).

Use this if you want to do your own scaling prior to calling `wxDC` drawing calls, for example, if your underlying model is floating-point and you want to achieve maximum drawing precision on high-resolution printers.

You can use the GetLogicalXXXRect() routines below to obtain the paper rectangle, page rectangle, or page margins rectangle to perform your own scaling.

Note: While the underlying drawing model of macOS is floating-point, wxWidgets's drawing model scales from integer coordinates.

```
getLogicalPaperRect(This) ->
                       {X :: integer(),
                        Y :: integer(),
                        W :: integer(),
                        H :: integer()}
```

Types:

```
This = wxPrintout()
```

Return the rectangle corresponding to the paper in the associated `wxDC` 's logical coordinates for the current user scale and device origin.

```
getLogicalPageRect(This) ->
                      {X :: integer(),
                       Y :: integer(),
                       W :: integer(),
                       H :: integer()}
```

Types:

```
This = wxPrintout()
```

Return the rectangle corresponding to the page in the associated `wxDC` 's logical coordinates for the current user scale and device origin.

On MSW and Mac, this will be the printable area of the paper. On other platforms and PostScript printing, this will be the full paper rectangle.

```
getLogicalPageMarginsRect(This, PageSetupData) ->
                             {X :: integer(),
                              Y :: integer(),
                              W :: integer(),
```

```
                                H :: integer()}
```

Types:

```
This = wxPrintout()
PageSetupData = wxPageSetupDialogData:wxPageSetupDialogData()
```

Return the rectangle corresponding to the page margins specified by the given `wxPageSetupDialogData` object in the associated `wxDC`'s logical coordinates for the current user scale and device origin.

The page margins are specified with respect to the edges of the paper on all platforms.

```
setLogicalOrigin(This, X, Y) -> ok
```

Types:

```
This = wxPrintout()
X = Y = integer()
```

Set the device origin of the associated `wxDC` so that the current logical point becomes the new logical origin.

```
offsetLogicalOrigin(This, Xoff, Yoff) -> ok
```

Types:

```
This = wxPrintout()
Xoff = Yoff = integer()
```

Shift the device origin by an amount specified in logical coordinates.

# wxProgressDialog

Erlang module

If supported by the platform this class will provide the platform's native progress dialog, else it will simply be the `wxGenericProgressDialog` (not implemented in wx).

This class is derived (and can use functions) from: `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxProgressDialog**

## Data Types

```
wxProgressDialog() = wx:wx_object()
```

## Exports

```
new(Title, Message) -> wxProgressDialog()
```

Types:

```
   Title = Message = unicode:chardata()
```

```
new(Title, Message, Options :: [Option]) -> wxProgressDialog()
```

Types:

```
   Title = Message = unicode:chardata()
   Option =
       {maximum, integer()} |
       {parent, wxWindow:wxWindow()} |
       {style, integer()}
```

```
resume(This) -> ok
```

Types:

```
   This = wxProgressDialog()
```

Can be used to continue with the dialog, after the user had clicked the "Abort" button.

```
update(This, Value) -> boolean()
```

Types:

```
   This = wxProgressDialog()
   Value = integer()
```

```
update(This, Value, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxProgressDialog()
   Value = integer()
   Option = {newmsg, unicode:chardata()}
```

Updates the dialog, setting the progress bar to the new value and updating the message if new one is specified.

Returns true unless the "Cancel" button has been pressed.

If false is returned, the application can either immediately destroy the dialog or ask the user for the confirmation and if the abort is not confirmed the dialog may be resumed with `resume/1` function.

If `value` is the maximum value for the dialog, the behaviour of the function depends on whether `wxPD_AUTO_HIDE` was used when the dialog was created. If it was, the dialog is hidden and the function returns immediately. If it was not, the dialog becomes a modal dialog and waits for the user to dismiss it, meaning that this function does not return until this happens.

Notice that if `newmsg` is longer than the currently shown message, the dialog will be automatically made wider to account for it. However if the new message is shorter than the previous one, the dialog doesn't shrink back to avoid constant resizes if the message is changed often. To do this and fit the dialog to its current contents you may call `wxWindow:fit/1` explicitly. However the native MSW implementation of this class does make the dialog shorter if the new text has fewer lines of text than the old one, so it is recommended to keep the number of lines of text constant in order to avoid jarring dialog size changes. You may also want to make the initial message, specified when creating the dialog, wide enough to avoid having to resize the dialog later, e.g. by appending a long string of unbreakable spaces (`wxString` (not implemented in wx)(L'\u00a0', 100)) to it.

```
destroy(This :: wxProgressDialog()) -> ok
```

Destroys the object.

# wxQueryNewPaletteEvent

Erlang module

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxQueryNewPaletteEvent**

## Data Types

```
wxQueryNewPaletteEvent() = wx:wx_object()
wxQueryNewPalette() =
    #wxQueryNewPalette{type =
                           wxQueryNewPaletteEvent:wxQueryNewPaletteEventType()}
wxQueryNewPaletteEventType() = query_new_palette
```

## Exports

```
setPaletteRealized(This, Realized) -> ok
```

Types:

```
   This = wxQueryNewPaletteEvent()
   Realized = boolean()
```

```
getPaletteRealized(This) -> boolean()
```

Types:

```
   This = wxQueryNewPaletteEvent()
```

# wxRadioBox

Erlang module

A radio box item is used to select one of number of mutually exclusive choices. It is displayed as a vertical column or horizontal row of labelled buttons.

Styles

This class supports the following styles:

See: **Overview events**, `wxRadioButton`, `wxCheckBox`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxRadioBox**

## Events

Event types emitted from this class: `command_radiobox_selected`

## Data Types

```
wxRadioBox() = wx:wx_object()
```

## Exports

```
new(Parent, Id, Label, Pos, Size, Choices) -> wxRadioBox()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
   Pos = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Choices = [unicode:chardata()]
```

```
new(Parent, Id, Label, Pos, Size, Choices, Options :: [Option]) ->
       wxRadioBox()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
   Pos = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Choices = [unicode:chardata()]
   Option =
       {majorDim, integer()} |
       {style, integer()} |
       {val, wx:wx_object()}
```

Constructor, creating and showing a radiobox.

See: create/8, wxValidator (not implemented in wx)

```
destroy(This :: wxRadioBox()) -> ok
```

Destructor, destroying the radiobox item.

```
create(This, Parent, Id, Label, Pos, Size, Choices) -> boolean()
```

Types:

```
   This = wxRadioBox()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
   Pos = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Choices = [unicode:chardata()]
```

```
create(This, Parent, Id, Label, Pos, Size, Choices,
       Options :: [Option]) ->
          boolean()
```

Types:

```
   This = wxRadioBox()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
   Pos = {X :: integer(), Y :: integer()}
   Size = {W :: integer(), H :: integer()}
   Choices = [unicode:chardata()]
   Option =
       {majorDim, integer()} |
       {style, integer()} |
       {val, wx:wx_object()}
```

Creates the radiobox for two-step construction.

See new/7 for further details.

```
enable(This) -> boolean()
```

Types:

```
   This = wxRadioBox()
```

```
enable(This, N) -> boolean()
enable(This, N :: [Option]) -> boolean()
```

Types:

```
   This = wxRadioBox()
   Option = {enable, boolean()}
```

Enables or disables the radiobox.

See: wxWindow:enable/2

```
enable(This, N, Options :: [Option]) -> boolean()
```

Types:

```
This = wxRadioBox()
N = integer()
Option = {enable, boolean()}
```

Enables or disables an individual button in the radiobox.

See: `wxWindow:enable/2`

```
getSelection(This) -> integer()
```

Types:

```
This = wxRadioBox()
```

Returns the index of the selected item or `wxNOT_FOUND` if no item is selected.

Return: The position of the current selection.

Remark: This method can be used with single selection list boxes only, you should use `wxListBox:getSelections/1` for the list boxes with wxLB_MULTIPLE style.

See: `setSelection/2`, `wxControlWithItems:getStringSelection/1`

```
getString(This, N) -> unicode:charlist()
```

Types:

```
This = wxRadioBox()
N = integer()
```

Returns the label of the item with the given index.

Return: The label of the item or an empty string if the position was invalid.

```
setSelection(This, N) -> ok
```

Types:

```
This = wxRadioBox()
N = integer()
```

Sets the selection to the given item.

Notice that a radio box always has selection, so `n` must be valid here and passing `wxNOT_FOUND` is not allowed.

```
show(This, Item) -> boolean()
```

Types:

```
This = wxRadioBox()
Item = integer()
```

```
show(This, Item, Options :: [Option]) -> boolean()
```

Types:

```
This = wxRadioBox()
Item = integer()
Option = {show, boolean()}
```

Shows or hides individual buttons.

Return: true if the item has been shown or hidden or false if nothing was done because it already was in the requested state.

See: `show/3`

`getColumnCount(This) -> integer()`

Types:

```
This = wxRadioBox()
```

Returns the number of columns in the radiobox.

`getItemHelpText(This, Item) -> unicode:charlist()`

Types:

```
This = wxRadioBox()
Item = integer()
```

Returns the helptext associated with the specified `item` if any or `wxEmptyString`.

See: `setItemHelpText/3`

`getItemToolTip(This, Item) -> wxToolTip:wxToolTip()`

Types:

```
This = wxRadioBox()
Item = integer()
```

Returns the tooltip associated with the specified `item` if any or NULL.

See: `setItemToolTip/3`, `wxWindow:getToolTip/1`

`getItemFromPoint(This, Pt) -> integer()`

Types:

```
This = wxRadioBox()
Pt = {X :: integer(), Y :: integer()}
```

Returns a radio box item under the point, a zero-based item index, or `wxNOT_FOUND` if no item is under the point.

`getRowCount(This) -> integer()`

Types:

```
This = wxRadioBox()
```

Returns the number of rows in the radiobox.

`isItemEnabled(This, N) -> boolean()`

Types:

```
This = wxRadioBox()
N = integer()
```

Returns true if the item is enabled or false if it was disabled using `enable/3`.

This function is currently only implemented in wxMSW, wxGTK, wxQT and wxUniversal and always returns true in the other ports.

```
isItemShown(This, N) -> boolean()
```

Types:

```
This = wxRadioBox()
N = integer()
```

Returns true if the item is currently shown or false if it was hidden using `show/3`.

Note that this function returns true for an item which hadn't been hidden even if the entire radiobox is not currently shown.

This function is currently only implemented in wxMSW, wxGTK, wxQT and wxUniversal and always returns true in the other ports.

```
setItemHelpText(This, Item, Helptext) -> ok
```

Types:

```
This = wxRadioBox()
Item = integer()
Helptext = unicode:chardata()
```

Sets the helptext for an item.

Empty string erases any existing helptext.

See: `getItemHelpText/2`

```
setItemToolTip(This, Item, Text) -> ok
```

Types:

```
This = wxRadioBox()
Item = integer()
Text = unicode:chardata()
```

Sets the tooltip text for the specified item in the radio group.

This function is currently only implemented in wxMSW and wxGTK2 and does nothing in the other ports.

See: `getItemToolTip/2`, `wxWindow:setToolTip/2`

# wxRadioButton

Erlang module

A radio button item is a button which usually denotes one of several mutually exclusive options. It has a text label next to a (usually) round button.

You can create a group of mutually-exclusive radio buttons by specifying `wxRB_GROUP` for the first in the group. The group ends when another radio button group is created, or there are no more radio buttons.

Styles

This class supports the following styles:

See: **Overview events**, `wxRadioBox`, `wxCheckBox`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxRadioButton**

## Events

Event types emitted from this class: `command_radiobutton_selected`

## Data Types

```
wxRadioButton() = wx:wx_object()
```

## Exports

```
new() -> wxRadioButton()
```

Default constructor.

See: `create/5`, `wxValidator` (not implemented in wx)

```
new(Parent, Id, Label) -> wxRadioButton()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Label = unicode:chardata()
```

```
new(Parent, Id, Label, Options :: [Option]) -> wxRadioButton()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Label = unicode:chardata()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {validator, wx:wx_object()}
```

Constructor, creating and showing a radio button.

See: `create/5`, `wxValidator` (not implemented in wx)

```
destroy(This :: wxRadioButton()) -> ok
```

Destructor, destroying the radio button item.

```
create(This, Parent, Id, Label) -> boolean()
```

Types:

```
   This = wxRadioButton()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
```

```
create(This, Parent, Id, Label, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxRadioButton()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Creates the choice for two-step construction.

See `new/4` for further details.

```
getValue(This) -> boolean()
```

Types:

```
   This = wxRadioButton()
```

Returns true if the radio button is checked, false otherwise.

```
setValue(This, Value) -> ok
```

Types:

```
   This = wxRadioButton()
   Value = boolean()
```

Sets the radio button to checked or unchecked status.

This does not cause a `wxEVT_RADIOBUTTON` event to get emitted.

If the radio button belongs to a radio group exactly one button in the group may be checked and so this method can be only called with `value` set to true. To uncheck a radio button in a group you must check another button in the same group.

Note: Under MSW, the focused radio button is always selected, i.e. its value is true. And, conversely, calling `SetValue(true)` will also set focus to the radio button if the focus had previously been on another radio button in the same group - as otherwise setting it on wouldn't work.

# wxRegion

Erlang module

A `wxRegion` represents a simple or complex region on a device context or window.

This class uses reference counting and copy-on-write internally so that assignments between two instances of this class are very cheap. You can therefore use actual objects instead of pointers without efficiency problems. If an instance of this class is changed it will create its own data internally so that other instances, which previously shared the data using the reference counting, are not affected.

Predefined objects (include wx.hrl):

See: `wxRegionIterator` (not implemented in wx)

wxWidgets docs: **wxRegion**

## Data Types

```
wxRegion() = wx:wx_object()
```

## Exports

```
new() -> wxRegion()
```

Default constructor.

This constructor creates an invalid, or null, object, i.e. calling IsOk() on it returns false and `isEmpty/1` returns true.

```
new(Rect) -> wxRegion()
new(Bmp) -> wxRegion()
```

Types:

```
Bmp = wxBitmap:wxBitmap()
```

Constructs a region using a bitmap.

See `union/5` for more details.

```
new(TopLeft, BottomRight) -> wxRegion()
```

Types:

```
TopLeft = BottomRight = {X :: integer(), Y :: integer()}
```

Constructs a rectangular region from the top left point and the bottom right point.

```
new(X, Y, Width, Height) -> wxRegion()
```

Types:

```
X = Y = Width = Height = integer()
```

Constructs a rectangular region with the given position and size.

```
destroy(This :: wxRegion()) -> ok
```

Destructor.

See reference-counted object destruction for more info.

```
clear(This) -> ok
```

Types:

```
This = wxRegion()
```

Clears the current region.

The object becomes invalid, or null, after being cleared.

```
contains(This, Pt) -> wx:wx_enum()
contains(This, Rect) -> wx:wx_enum()
```

Types:

```
This = wxRegion()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
```

Returns a value indicating whether the given rectangle is contained within the region.

This method always returns `wxOutRegion` for an invalid region but may, nevertheless, be safely called in this case.

Return: One of ?wxOutRegion, ?wxPartRegion or ?wxInRegion.

Note: On Windows, only ?wxOutRegion and ?wxInRegion are returned; a value ?wxInRegion then indicates that all or some part of the region is contained in this region.

```
contains(This, X, Y) -> wx:wx_enum()
```

Types:

```
This = wxRegion()
X = Y = integer()
```

Returns a value indicating whether the given point is contained within the region.

This method always returns `wxOutRegion` for an invalid region but may, nevertheless, be safely called in this case.

Return: The return value is one of `wxOutRegion` and `wxInRegion`.

```
contains(This, X, Y, Width, Height) -> wx:wx_enum()
```

Types:

```
This = wxRegion()
X = Y = Width = Height = integer()
```

Returns a value indicating whether the given rectangle is contained within the region.

This method always returns `wxOutRegion` for an invalid region but may, nevertheless, be safely called in this case.

Return: One of ?wxOutRegion, ?wxPartRegion or ?wxInRegion.

Note: On Windows, only ?wxOutRegion and ?wxInRegion are returned; a value ?wxInRegion then indicates that all or some part of the region is contained in this region.

```
convertToBitmap(This) -> wxBitmap:wxBitmap()
```

Types:

```
   This = wxRegion()
```

Convert the region to a black and white bitmap with the white pixels being inside the region.

This method can't be used for invalid region.

```
getBox(This) ->
          {X :: integer(),
           Y :: integer(),
           W :: integer(),
           H :: integer()}
```

Types:

```
   This = wxRegion()
```

```
intersect(This, Rect) -> boolean()
intersect(This, Region) -> boolean()
```

Types:

```
   This = Region = wxRegion()
```

Finds the intersection of this region and another region.

This method always fails, i.e. returns false, if this region is invalid but may nevertheless be safely used even in this case.

Return: true if successful, false otherwise.

Remark: Creates the intersection of the two regions, that is, the parts which are in both regions. The result is stored in this region.

```
intersect(This, X, Y, Width, Height) -> boolean()
```

Types:

```
   This = wxRegion()
   X = Y = Width = Height = integer()
```

Finds the intersection of this region and another, rectangular region, specified using position and size.

This method always fails, i.e. returns false, if this region is invalid but may nevertheless be safely used even in this case.

Return: true if successful, false otherwise.

Remark: Creates the intersection of the two regions, that is, the parts which are in both regions. The result is stored in this region.

```
isEmpty(This) -> boolean()
```

Types:

```
   This = wxRegion()
```

Returns true if the region is empty, false otherwise.

Always returns true if the region is invalid.

```
subtract(This, Rect) -> boolean()
subtract(This, Region) -> boolean()
```

Types:

```
   This = Region = wxRegion()
```

Subtracts a region from this region.

This method always fails, i.e. returns false, if this region is invalid but may nevertheless be safely used even in this case.

Return: true if successful, false otherwise.

Remark: This operation combines the parts of 'this' region that are not part of the second region. The result is stored in this region.

```
offset(This, Pt) -> boolean()
```

Types:

```
   This = wxRegion()
   Pt = {X :: integer(), Y :: integer()}
```

```
offset(This, X, Y) -> boolean()
```

Types:

```
   This = wxRegion()
   X = Y = integer()
```

Moves the region by the specified offsets in horizontal and vertical directions.

This method can't be called if the region is invalid as it doesn't make sense to offset it then. Attempts to do it will result in assert failure.

Return: true if successful, false otherwise (the region is unchanged then).

```
union(This, Region) -> boolean()
union(This, Rect) -> boolean()
```

Types:

```
   This = wxRegion()
   Rect =
       {X :: integer(),
        Y :: integer(),
        W :: integer(),
        H :: integer()}
```

Finds the union of this region and another, rectangular region.

This method can be used even if this region is invalid and has the natural behaviour in this case, i.e. makes this region equal to the given rectangle.

Return: true if successful, false otherwise.

Remark: This operation creates a region that combines all of this region and the second region. The result is stored in this region.

```
union(This, Bmp, TransColour) -> boolean()
```

Types:

```
   This = wxRegion()
   Bmp = wxBitmap:wxBitmap()
   TransColour = wx:wx_colour()

union(This, Bmp, TransColour, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxRegion()
   Bmp = wxBitmap:wxBitmap()
   TransColour = wx:wx_colour()
   Option = {tolerance, integer()}
```

Finds the union of this region and the non-transparent pixels of a bitmap.

Colour to be treated as transparent is specified in the `transColour` argument, along with an optional colour tolerance value.

Return: true if successful, false otherwise.

Remark: This operation creates a region that combines all of this region and the second region. The result is stored in this region.

```
union(This, X, Y, Width, Height) -> boolean()
```

Types:

```
   This = wxRegion()
   X = Y = Width = Height = integer()
```

Finds the union of this region and another, rectangular region, specified using position and size.

This method can be used even if this region is invalid and has the natural behaviour in this case, i.e. makes this region equal to the given rectangle.

Return: true if successful, false otherwise.

Remark: This operation creates a region that combines all of this region and the second region. The result is stored in this region.

```
'Xor'(This, Rect) -> boolean()
'Xor'(This, Region) -> boolean()
```

Types:

```
   This = Region = wxRegion()
```

Finds the Xor of this region and another region.

This method can be used even if this region is invalid and has the natural behaviour in this case, i.e. makes this region equal to the given `region`.

Return: true if successful, false otherwise.

Remark: This operation creates a region that combines all of this region and the second region, except for any overlapping areas. The result is stored in this region.

```
'Xor'(This, X, Y, Width, Height) -> boolean()
```

Types:

```
This = wxRegion()
X = Y = Width = Height = integer()
```

Finds the Xor of this region and another, rectangular region, specified using position and size.

This method can be used even if this region is invalid and has the natural behaviour in this case, i.e. makes this region equal to the given rectangle.

Return: true if successful, false otherwise.

Remark: This operation creates a region that combines all of this region and the second region, except for any overlapping areas. The result is stored in this region.

# wxSashEvent

Erlang module

A sash event is sent when the sash of a `wxSashWindow` has been dragged by the user.

Remark: When a sash belonging to a sash window is dragged by the user, and then released, this event is sent to the window, where it may be processed by an event table entry in a derived class, a plug-in event handler or an ancestor class. Note that the `wxSashWindow` doesn't change the window's size itself. It relies on the application's event handler to do that. This is because the application may have to handle other consequences of the resize, or it may wish to veto it altogether. The event handler should look at the drag rectangle: see `getDragRect/1` to see what the new size of the window would be if the resize were to be applied. It should also call `getDragStatus/1` to see whether the drag was OK or out of the current allowed range.

See: `wxSashWindow`, **Overview events**

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxSashEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxSashEventType` to subscribe to events of this type.

## Data Types

```
wxSashEvent() = wx:wx_object()
wxSash() =
    #wxSash{type = wxSashEvent:wxSashEventType(),
            edge = wx:wx_enum(),
            dragRect =
                {X :: integer(),
                 Y :: integer(),
                 W :: integer(),
                 H :: integer()},
            dragStatus = wx:wx_enum()}
wxSashEventType() = sash_dragged
```

## Exports

```
getEdge(This) -> wx:wx_enum()
```

Types:

```
   This = wxSashEvent()
```

Returns the dragged edge.

The return value is one of wxSASH_TOP, wxSASH_RIGHT, wxSASH_BOTTOM, wxSASH_LEFT.

```
getDragRect(This) ->
               {X :: integer(),
                Y :: integer(),
                W :: integer(),
```

```
                H :: integer()}
```

Types:

```
   This = wxSashEvent()
```

Returns the rectangle representing the new size the window would be if the resize was applied.

It is up to the application to set the window size if required.

```
getDragStatus(This) -> wx:wx_enum()
```

Types:

```
   This = wxSashEvent()
```

Returns the status of the sash: one of wxSASH_STATUS_OK, wxSASH_STATUS_OUT_OF_RANGE.

If the drag caused the notional bounding box of the window to flip over, for example, the drag will be out of rage.

# wxSashLayoutWindow

Erlang module

wxSashLayoutWindow responds to OnCalculateLayout events generated by wxLayoutAlgorithm. It allows the application to use simple accessors to specify how the window should be laid out, rather than having to respond to events.

The fact that the class derives from wxSashWindow allows sashes to be used if required, to allow the windows to be user-resizable.

The documentation for wxLayoutAlgorithm explains the purpose of this class in more detail.

For the window styles see wxSashWindow.

This class handles the EVT_QUERY_LAYOUT_INFO and EVT_CALCULATE_LAYOUT events for you. However, if you use sashes, see wxSashWindow for relevant event information. See also wxLayoutAlgorithm for information about the layout events.

See: wxLayoutAlgorithm, wxSashWindow, **Overview events**

This class is derived (and can use functions) from: wxSashWindow wxWindow wxEvtHandler

wxWidgets docs: **wxSashLayoutWindow**

## Data Types

```
wxSashLayoutWindow() = wx:wx_object()
```

## Exports

```
new() -> wxSashLayoutWindow()
```

Default ctor.

```
new(Parent) -> wxSashLayoutWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxSashLayoutWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructs a sash layout window, which can be a child of a frame, dialog or any other non-control window.

```
create(This, Parent) -> boolean()
```

Types:

```
   This = wxSashLayoutWindow()
   Parent = wxWindow:wxWindow()

create(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxSashLayoutWindow()
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Initializes a sash layout window, which can be a child of a frame, dialog or any other non-control window.

```
getAlignment(This) -> wx:wx_enum()
```

Types:

```
   This = wxSashLayoutWindow()
```

Returns the alignment of the window: one of wxLAYOUT_TOP, wxLAYOUT_LEFT, wxLAYOUT_RIGHT, wxLAYOUT_BOTTOM.

```
getOrientation(This) -> wx:wx_enum()
```

Types:

```
   This = wxSashLayoutWindow()
```

Returns the orientation of the window: one of wxLAYOUT_HORIZONTAL, wxLAYOUT_VERTICAL.

```
setAlignment(This, Alignment) -> ok
```

Types:

```
   This = wxSashLayoutWindow()
   Alignment = wx:wx_enum()
```

Sets the alignment of the window (which edge of the available parent client area the window is attached to).

`alignment` is one of wxLAYOUT_TOP, wxLAYOUT_LEFT, wxLAYOUT_RIGHT, wxLAYOUT_BOTTOM.

```
setDefaultSize(This, Size) -> ok
```

Types:

```
   This = wxSashLayoutWindow()
   Size = {W :: integer(), H :: integer()}
```

Sets the default dimensions of the window.

The dimension other than the orientation will be fixed to this value, and the orientation dimension will be ignored and the window stretched to fit the available space.

```
setOrientation(This, Orientation) -> ok
```

Types:

```
This = wxSashLayoutWindow()
Orientation = wx:wx_enum()
```

Sets the orientation of the window (the direction the window will stretch in, to fill the available parent client area). `orientation` is one of wxLAYOUT_HORIZONTAL, wxLAYOUT_VERTICAL.

```
destroy(This :: wxSashLayoutWindow()) -> ok
```

Destroys the object.

# wxSashWindow

Erlang module

wxSashWindow allows any of its edges to have a sash which can be dragged to resize the window. The actual content window will be created by the application as a child of wxSashWindow.

The window (or an ancestor) will be notified of a drag via a wxSashEvent notification.

Styles

This class supports the following styles:

See: wxSashEvent, wxSashLayoutWindow, **Overview events**

This class is derived (and can use functions) from: wxWindow wxEvtHandler

wxWidgets docs: **wxSashWindow**

## Events

Event types emitted from this class: sash_dragged

## Data Types

```
wxSashWindow() = wx:wx_object()
```

## Exports

```
new() -> wxSashWindow()
```

Default ctor.

```
new(Parent) -> wxSashWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxSashWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructs a sash window, which can be a child of a frame, dialog or any other non-control window.

```
destroy(This :: wxSashWindow()) -> ok
```

Destructor.

```
getSashVisible(This, Edge) -> boolean()
```

Types:

```
This = wxSashWindow()
Edge = wx:wx_enum()
```

Returns true if a sash is visible on the given edge, false otherwise.

See: `setSashVisible/3`

```
getMaximumSizeX(This) -> integer()
```

Types:

```
This = wxSashWindow()
```

Gets the maximum window size in the x direction.

```
getMaximumSizeY(This) -> integer()
```

Types:

```
This = wxSashWindow()
```

Gets the maximum window size in the y direction.

```
getMinimumSizeX(This) -> integer()
```

Types:

```
This = wxSashWindow()
```

Gets the minimum window size in the x direction.

```
getMinimumSizeY(This) -> integer()
```

Types:

```
This = wxSashWindow()
```

Gets the minimum window size in the y direction.

```
setMaximumSizeX(This, Min) -> ok
```

Types:

```
This = wxSashWindow()
Min = integer()
```

Sets the maximum window size in the x direction.

```
setMaximumSizeY(This, Min) -> ok
```

Types:

```
This = wxSashWindow()
Min = integer()
```

Sets the maximum window size in the y direction.

```
setMinimumSizeX(This, Min) -> ok
```

Types:

```
This = wxSashWindow()
Min = integer()
```

Sets the minimum window size in the x direction.

```
setMinimumSizeY(This, Min) -> ok
```

Types:

```
This = wxSashWindow()
Min = integer()
```

Sets the minimum window size in the y direction.

```
setSashVisible(This, Edge, Visible) -> ok
```

Types:

```
This = wxSashWindow()
Edge = wx:wx_enum()
Visible = boolean()
```

Call this function to make a sash visible or invisible on a particular edge.

See: `getSashVisible/2`

# wxScreenDC

Erlang module

A `wxScreenDC` can be used to paint on the screen. This should normally be constructed as a temporary stack object; don't store a `wxScreenDC` object.

When using multiple monitors, `wxScreenDC` corresponds to the entire virtual screen composed of all of them. Notice that coordinates on `wxScreenDC` can be negative in this case, see `wxDisplay:getGeometry/1` for more.

See: `wxDC`, `wxMemoryDC`, `wxPaintDC`, `wxClientDC`, `wxWindowDC`

This class is derived (and can use functions) from: `wxDC`

wxWidgets docs: **wxScreenDC**

## Data Types

```
wxScreenDC() = wx:wx_object()
```

## Exports

```
new() -> wxScreenDC()
```

Constructor.

```
destroy(This :: wxScreenDC()) -> ok
```

Destroys the object.

# wxScrollBar

Erlang module

A `wxScrollBar` is a control that represents a horizontal or vertical scrollbar.

It is distinct from the two scrollbars that some windows provide automatically, but the two types of scrollbar share the way events are received.

Remark: A scrollbar has the following main attributes: range, thumb size, page size, and position. The range is the total number of units associated with the view represented by the scrollbar. For a table with 15 columns, the range would be 15. The thumb size is the number of units that are currently visible. For the table example, the window might be sized so that only 5 columns are currently visible, in which case the application would set the thumb size to 5. When the thumb size becomes the same as or greater than the range, the scrollbar will be automatically hidden on most platforms. The page size is the number of units that the scrollbar should scroll by, when 'paging' through the data. This value is normally the same as the thumb size length, because it is natural to assume that the visible window size defines a page. The scrollbar position is the current thumb position. Most applications will find it convenient to provide a function called AdjustScrollbars() which can be called initially, from an OnSize event handler, and whenever the application data changes in size. It will adjust the view, object and page size according to the size of the window and the size of the data.

Styles

This class supports the following styles:

The difference between EVT_SCROLL_THUMBRELEASE and EVT_SCROLL_CHANGED

The EVT_SCROLL_THUMBRELEASE event is only emitted when actually dragging the thumb using the mouse and releasing it (This EVT_SCROLL_THUMBRELEASE event is also followed by an EVT_SCROLL_CHANGED event).

The EVT_SCROLL_CHANGED event also occurs when using the keyboard to change the thumb position, and when clicking next to the thumb (In all these cases the EVT_SCROLL_THUMBRELEASE event does not happen).

In short, the EVT_SCROLL_CHANGED event is triggered when scrolling/moving has finished independently of the way it had started. Please see the page_samples_widgets ("Slider" page) to see the difference between EVT_SCROLL_THUMBRELEASE and EVT_SCROLL_CHANGED in action.

See: **Overview scrolling**, **Overview events**, `wxScrolled` (not implemented in wx)

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxScrollBar**

## Events

Event types emitted from this class: `scroll_top`, `scroll_bottom`, `scroll_lineup`, `scroll_linedown`, `scroll_pageup`, `scroll_pagedown`, `scroll_thumbtrack`, `scroll_thumbrelease`, `scroll_changed`, `scroll_top`, `scroll_bottom`, `scroll_lineup`, `scroll_linedown`, `scroll_pageup`, `scroll_pagedown`, `scroll_thumbtrack`, `scroll_thumbrelease`, `scroll_changed`

## Data Types

```
wxScrollBar() = wx:wx_object()
```

## Exports

```
new() -> wxScrollBar()
```

Default constructor.

```
new(Parent, Id) -> wxScrollBar()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxScrollBar()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Constructor, creating and showing a scrollbar.

See: `create/4`, `wxValidator` (not implemented in wx)

```
destroy(This :: wxScrollBar()) -> ok
```

Destructor, destroying the scrollbar.

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = wxScrollBar()
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxScrollBar()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
```

```
{validator, wx:wx_object()}
```

Scrollbar creation function called by the scrollbar constructor.

See `new/3` for details.

```
getRange(This) -> integer()
```

Types:

```
This = wxScrollBar()
```

Returns the length of the scrollbar.

See: `setScrollbar/6`

```
getPageSize(This) -> integer()
```

Types:

```
This = wxScrollBar()
```

Returns the page size of the scrollbar.

This is the number of scroll units that will be scrolled when the user pages up or down. Often it is the same as the thumb size.

See: `setScrollbar/6`

```
getThumbPosition(This) -> integer()
```

Types:

```
This = wxScrollBar()
```

Returns the current position of the scrollbar thumb.

See: `setThumbPosition/2`

```
getThumbSize(This) -> integer()
```

Types:

```
This = wxScrollBar()
```

Returns the thumb or 'view' size.

See: `setScrollbar/6`

```
setThumbPosition(This, ViewStart) -> ok
```

Types:

```
This = wxScrollBar()
ViewStart = integer()
```

Sets the position of the scrollbar.

See: `getThumbPosition/1`

```
setScrollbar(This, Position, ThumbSize, Range, PageSize) -> ok
```

Types:

```
   This = wxScrollBar()
   Position = ThumbSize = Range = PageSize = integer()

setScrollbar(This, Position, ThumbSize, Range, PageSize,
             Options :: [Option]) ->
                ok
```

Types:

```
   This = wxScrollBar()
   Position = ThumbSize = Range = PageSize = integer()
   Option = {refresh, boolean()}
```

Sets the scrollbar properties.

Remark: Let's say you wish to display 50 lines of text, using the same font. The window is sized so that you can only see 16 lines at a time. You would use: The page size is 1 less than the thumb size so that the last line of the previous page will be visible on the next page, to help orient the user. Note that with the window at this size, the thumb position can never go above 50 minus 16, or 34. You can determine how many lines are currently visible by dividing the current view size by the character height in pixels. When defining your own scrollbar behaviour, you will always need to recalculate the scrollbar settings when the window size changes. You could therefore put your scrollbar calculations and `setScrollbar/6` call into a function named AdjustScrollbars, which can be called initially and also from a `wxSizeEvent` event handler function.

# wxScrollEvent

Erlang module

A scroll event holds information about events sent from stand-alone scrollbars (see `wxScrollBar`) and sliders (see `wxSlider`).

Note that scrolled windows send the `wxScrollWinEvent` which does not derive from `wxCommandEvent`, but from `wxEvent` directly - don't confuse these two kinds of events and use the event table macros mentioned below only for the scrollbar-like controls.

The difference between EVT_SCROLL_THUMBRELEASE and EVT_SCROLL_CHANGED

The EVT_SCROLL_THUMBRELEASE event is only emitted when actually dragging the thumb using the mouse and releasing it (This EVT_SCROLL_THUMBRELEASE event is also followed by an EVT_SCROLL_CHANGED event).

The EVT_SCROLL_CHANGED event also occurs when using the keyboard to change the thumb position, and when clicking next to the thumb (In all these cases the EVT_SCROLL_THUMBRELEASE event does not happen).

In short, the EVT_SCROLL_CHANGED event is triggered when scrolling/ moving has finished independently of the way it had started. Please see the page_samples_widgets ("Slider" page) to see the difference between EVT_SCROLL_THUMBRELEASE and EVT_SCROLL_CHANGED in action.

Remark: Note that unless specifying a scroll control identifier, you will need to test for scrollbar orientation with `getOrientation/1`, since horizontal and vertical scroll events are processed using the same event handler.

See: `wxScrollBar`, `wxSlider`, `wxSpinButton`, `wxScrollWinEvent`, **Overview events**

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxScrollEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxScrollEventType` to subscribe to events of this type.

## Data Types

```
wxScrollEvent() = wx:wx_object()
wxScroll() =
    #wxScroll{type = wxScrollEvent:wxScrollEventType(),
              commandInt = integer(),
              extraLong = integer()}
wxScrollEventType() =
    scroll_top | scroll_bottom | scroll_lineup | scroll_linedown |
    scroll_pageup | scroll_pagedown | scroll_thumbtrack |
    scroll_thumbrelease | scroll_changed
```

## Exports

```
getOrientation(This) -> integer()
```

Types:

```
   This = wxScrollEvent()
```

Returns wxHORIZONTAL or wxVERTICAL, depending on the orientation of the scrollbar.

```
getPosition(This) -> integer()
```

Types:

```
   This = wxScrollEvent()
```

Returns the position of the scrollbar.

# wxScrollWinEvent

Erlang module

A scroll event holds information about events sent from scrolling windows.

Note that you can use the EVT_SCROLLWIN* macros for intercepting scroll window events from the receiving window.

See: `wxScrollEvent`, **Overview events**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxScrollWinEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxScrollWinEventType` to subscribe to events of this type.

## Data Types

```
wxScrollWinEvent() = wx:wx_object()
wxScrollWin() =
    #wxScrollWin{type = wxScrollWinEvent:wxScrollWinEventType(),
                 commandInt = integer(),
                 extraLong = integer()}
wxScrollWinEventType() =
    scrollwin_top | scrollwin_bottom | scrollwin_lineup |
    scrollwin_linedown | scrollwin_pageup | scrollwin_pagedown |
    scrollwin_thumbtrack | scrollwin_thumbrelease
```

## Exports

```
getOrientation(This) -> integer()
```

Types:

```
   This = wxScrollWinEvent()
```

Returns wxHORIZONTAL or wxVERTICAL, depending on the orientation of the scrollbar.

```
getPosition(This) -> integer()
```

Types:

```
   This = wxScrollWinEvent()
```

Returns the position of the scrollbar for the thumb track and release events.

Note that this field can't be used for the other events, you need to query the window itself for the current position in that case.

# wxScrolledWindow

Erlang module

There are two commonly used (but not the only possible!) specializations of this class:

Note: See `wxScrolled::Create()` (not implemented in wx) if you want to use `wxScrolled` (not implemented in wx) with a custom class.

Starting from version 2.4 of wxWidgets, there are several ways to use a ?wxScrolledWindow (and now `wxScrolled` (not implemented in wx)). In particular, there are three ways to set the size of the scrolling area:

One way is to set the scrollbars directly using a call to `setScrollbars/6`. This is the way it used to be in any previous version of wxWidgets and it will be kept for backwards compatibility.

An additional method of manual control, which requires a little less computation of your own, is to set the total size of the scrolling area by calling either `wxWindow:setVirtualSize/3`, or `wxWindow:fitInside/1`, and setting the scrolling increments for it by calling `setScrollRate/3`. Scrolling in some orientation is enabled by setting a non-zero increment for it.

The most automatic and newest way is to simply let sizers determine the scrolling area. This is now the default when you set an interior sizer into a `wxScrolled` (not implemented in wx) with `wxWindow:setSizer/3`. The scrolling area will be set to the size requested by the sizer and the scrollbars will be assigned for each orientation according to the need for them and the scrolling increment set by `setScrollRate/3`. As above, scrolling is only enabled in orientations with a non-zero increment. You can influence the minimum size of the scrolled area controlled by a sizer by calling wxWindow::SetVirtualSizeHints(). (Calling `setScrollbars/6` has analogous effects in wxWidgets 2.4 - in later versions it may not continue to override the sizer.)

Note that if maximum size hints are still supported by wxWindow::SetVirtualSizeHints(), use them at your own dire risk. They may or may not have been removed for 2.4, but it really only makes sense to set minimum size hints here. We should probably replace wxWindow::SetVirtualSizeHints() with wxWindow::SetMinVirtualSize() or similar and remove it entirely in future.

As with all windows, an application can draw onto a `wxScrolled` (not implemented in wx) using a device context.

You have the option of handling the OnPaint handler or overriding the `wxScrolled::OnDraw()` (not implemented in wx) function, which is passed a pre-scrolled device context (prepared by `doPrepareDC/2`).

If you don't wish to calculate your own scrolling, you must call `doPrepareDC/2` when not drawing from within `OnDraw()` (not implemented in wx), to set the device origin for the device context according to the current scroll position.

A `wxScrolled` (not implemented in wx) will normally scroll itself and therefore its child windows as well. It might however be desired to scroll a different window than itself: e.g. when designing a spreadsheet, you will normally only have to scroll the (usually white) cell area, whereas the (usually grey) label area will scroll very differently. For this special purpose, you can call `setTargetWindow/2` which means that pressing the scrollbars will scroll a different window.

Note that the underlying system knows nothing about scrolling coordinates, so that all system functions (mouse events, expose events, refresh calls etc) as well as the position of subwindows are relative to the "physical" origin of the scrolled window. If the user insert a child window at position (10,10) and scrolls the window down 100 pixels (moving the child window out of the visible area), the child window will report a position of (10,-90).

Styles

This class supports the following styles:

Note: Don't confuse wxScrollWinEvents generated by this class with `wxScrollEvent` objects generated by `wxScrollBar` and `wxSlider`.

Remark: Use `wxScrolled` (not implemented in wx) for applications where the user scrolls by a fixed amount, and where a 'page' can be interpreted to be the current visible portion of the window. For more sophisticated applications, use the `wxScrolled` (not implemented in wx) implementation as a guide to build your own scroll behaviour or use `wxVScrolledWindow` (not implemented in wx) or its variants.

Since: The `wxScrolled` (not implemented in wx) template exists since version 2.9.0. In older versions, only ? wxScrolledWindow (equivalent of wxScrolled<wxPanel>) was available.

See: `wxScrollBar`, `wxClientDC`, `wxPaintDC`, `wxVScrolledWindow` (not implemented in wx), `wxHScrolledWindow` (not implemented in wx), `wxHVScrolledWindow` (not implemented in wx)

This class is derived (and can use functions) from: `wxPanel` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxScrolledWindow**

## Events

Event types emitted from this class: `scrollwin_top`, `scrollwin_bottom`, `scrollwin_lineup`, `scrollwin_linedown`, `scrollwin_pageup`, `scrollwin_pagedown`, `scrollwin_thumbtrack`, `scrollwin_thumbrelease`

## Data Types

```
wxScrolledWindow() = wx:wx_object()
```

## Exports

```
new() -> wxScrolledWindow()
```

Default constructor.

```
new(Parent) -> wxScrolledWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxScrolledWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {winid, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructor.

Remark: The window is initially created without visible scrollbars. Call `setScrollbars/6` to specify how big the virtual window size should be.

```
calcScrolledPosition(This, Pt) -> {X :: integer(), Y :: integer()}
```

Types:

```
   This = wxScrolledWindow()
   Pt = {X :: integer(), Y :: integer()}

calcScrolledPosition(This, X, Y) ->
                        {Xx :: integer(), Yy :: integer()}
```

Types:

```
   This = wxScrolledWindow()
   X = Y = integer()
```

Translates the logical coordinates to the device ones.

For example, if a window is scrolled 10 pixels to the bottom, the device coordinates of the origin are (0, 0) (as always), but the logical coordinates are (0, 10) and so the call to CalcScrolledPosition(0, 10, xx, yy) will return 0 in yy.

See: `calcUnscrolledPosition/3`

```
calcUnscrolledPosition(This, Pt) ->
                          {X :: integer(), Y :: integer()}
```

Types:

```
   This = wxScrolledWindow()
   Pt = {X :: integer(), Y :: integer()}

calcUnscrolledPosition(This, X, Y) ->
                          {Xx :: integer(), Yy :: integer()}
```

Types:

```
   This = wxScrolledWindow()
   X = Y = integer()
```

Translates the device coordinates to the logical ones.

For example, if a window is scrolled 10 pixels to the bottom, the device coordinates of the origin are (0, 0) (as always), but the logical coordinates are (0, 10) and so the call to CalcUnscrolledPosition(0, 0, xx, yy) will return 10 in yy.

See: `calcScrolledPosition/3`

```
enableScrolling(This, XScrolling, YScrolling) -> ok
```

Types:

```
   This = wxScrolledWindow()
   XScrolling = YScrolling = boolean()
```

Enable or disable use of `wxWindow:scrollWindow/4` for scrolling.

By default, when a scrolled window is logically scrolled, `wxWindow:scrollWindow/4` is called on the underlying window which scrolls the window contents and only invalidates the part of the window newly brought into view. If false is passed as an argument, then this "physical scrolling" is disabled and the window is entirely invalidated whenever it is scrolled by calling `wxWindow:refresh/2`.

It should be rarely necessary to disable physical scrolling, so this method shouldn't be called in normal circumstances.

```
getScrollPixelsPerUnit(This) ->
                          {XUnit :: integer(), YUnit :: integer()}
```

Types:

```
This = wxScrolledWindow()
```

Get the number of pixels per scroll unit (line), in each direction, as set by `setScrollbars/6`.

A value of zero indicates no scrolling in that direction.

See: `setScrollbars/6`, `wxWindow:getVirtualSize/1`

```
getViewStart(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxScrolledWindow()
```

This is a simple overload of GetViewStart(int*,int*); see that function for more info.

```
doPrepareDC(This, Dc) -> ok
```

Types:

```
This = wxScrolledWindow()
Dc = wxDC:wxDC()
```

Call this function to prepare the device context for drawing a scrolled image.

It sets the device origin according to the current scroll position. `doPrepareDC/2` is called automatically within the default `wxEVT_PAINT` event handler, so your `OnDraw()` (not implemented in wx) override will be passed an already 'pre-scrolled' device context. However, if you wish to draw from outside of `OnDraw()` (not implemented in wx) (e.g. from your own `wxEVT_PAINT` handler), you must call this function yourself.

For example:

Notice that the function sets the origin by moving it relatively to the current origin position, so you shouldn't change the origin before calling `doPrepareDC/2` or, if you do, reset it to (0, 0) later. If you call `doPrepareDC/2` immediately after device context creation, as in the example above, this problem doesn't arise, of course, so it is customary to do it like this.

```
prepareDC(This, Dc) -> ok
```

Types:

```
This = wxScrolledWindow()
Dc = wxDC:wxDC()
```

This function is for backwards compatibility only and simply calls `doPrepareDC/2` now.

Notice that it is not called by the default paint event handle (`doPrepareDC/2` is), so overriding this method in your derived class is useless.

```
scroll(This, Pt) -> ok
```

Types:

```
This = wxScrolledWindow()
Pt = {X :: integer(), Y :: integer()}
```

This is an overload of `scroll/3`; see that function for more info.

```
scroll(This, X, Y) -> ok
```

Types:

```
This = wxScrolledWindow()
X = Y = integer()
```

Scrolls a window so the view start is at the given point.

Remark: The positions are in scroll units, not pixels, so to convert to pixels you will have to multiply by the number of pixels per scroll increment. If either parameter is ?wxDefaultCoord (-1), that position will be ignored (no change in that direction).

See: `setScrollbars/6`, `getScrollPixelsPerUnit/1`

```
setScrollbars(This, PixelsPerUnitX, PixelsPerUnitY, NoUnitsX,
              NoUnitsY) ->
                 ok
```

Types:

```
This = wxScrolledWindow()
PixelsPerUnitX = PixelsPerUnitY = NoUnitsX = NoUnitsY = integer()
```

```
setScrollbars(This, PixelsPerUnitX, PixelsPerUnitY, NoUnitsX,
              NoUnitsY,
              Options :: [Option]) ->
                 ok
```

Types:

```
This = wxScrolledWindow()
PixelsPerUnitX = PixelsPerUnitY = NoUnitsX = NoUnitsY = integer()
Option =
    {xPos, integer()} | {yPos, integer()} | {noRefresh, boolean()}
```

Sets up vertical and/or horizontal scrollbars.

The first pair of parameters give the number of pixels per 'scroll step', i.e. amount moved when the up or down scroll arrows are pressed. The second pair gives the length of scrollbar in scroll steps, which sets the size of the virtual window.

`xPos` and `yPos` optionally specify a position to scroll to immediately.

For example, the following gives a window horizontal and vertical scrollbars with 20 pixels per scroll step, and a size of 50 steps (1000 pixels) in each direction:

`wxScrolled` (not implemented in wx) manages the page size itself, using the current client window size as the page size.

Note that for more sophisticated scrolling applications, for example where scroll steps may be variable according to the position in the document, it will be necessary to derive a new class from `wxWindow`, overriding OnSize() and adjusting the scrollbars appropriately.

See: `wxWindow:setVirtualSize/3`

```
setScrollRate(This, Xstep, Ystep) -> ok
```

Types:

```
This = wxScrolledWindow()
Xstep = Ystep = integer()
```

Set the horizontal and vertical scrolling increment only.

See the pixelsPerUnit parameter in `setScrollbars/6`.

`setTargetWindow(This, Window) -> ok`

Types:

```
This = wxScrolledWindow()
Window = wxWindow:wxWindow()
```

Call this function to tell `wxScrolled` (not implemented in wx) to perform the actual scrolling on a different window (and not on itself).

This method is useful when only a part of the window should be scrolled. A typical example is a control consisting of a fixed header and the scrollable contents window: the scrollbars are attached to the main window itself, hence it, and not the contents window must be derived from `wxScrolled` (not implemented in wx), but only the contents window scrolls when the scrollbars are used. To implement such setup, you need to call this method with the contents window as argument.

Notice that if this method is used, `GetSizeAvailableForScrollTarget()` (not implemented in wx) method must be overridden.

`destroy(This :: wxScrolledWindow()) -> ok`

Destroys the object.

# wxSetCursorEvent

Erlang module

A `wxSetCursorEvent` is generated from `wxWindow` when the mouse cursor is about to be set as a result of mouse motion.

This event gives the application the chance to perform specific mouse cursor processing based on the current position of the mouse within the window. Use `setCursor/2` to specify the cursor you want to be displayed.

See: `wx_misc:setCursor/1`, `wxWindow:setCursor/2`

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxSetCursorEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxSetCursorEventType` to subscribe to events of this type.

## Data Types

```
wxSetCursorEvent() = wx:wx_object()
wxSetCursor() =
    #wxSetCursor{type = wxSetCursorEvent:wxSetCursorEventType(),
                 x = integer(),
                 y = integer(),
                 cursor = wxCursor:wxCursor()}
wxSetCursorEventType() = set_cursor
```

## Exports

```
getCursor(This) -> wxCursor:wxCursor()
```

Types:

```
   This = wxSetCursorEvent()
```

Returns a reference to the cursor specified by this event.

```
getX(This) -> integer()
```

Types:

```
   This = wxSetCursorEvent()
```

Returns the X coordinate of the mouse in client coordinates.

```
getY(This) -> integer()
```

Types:

```
   This = wxSetCursorEvent()
```

Returns the Y coordinate of the mouse in client coordinates.

```
hasCursor(This) -> boolean()
```

Types:

```
This = wxSetCursorEvent()
```

Returns true if the cursor specified by this event is a valid cursor.

Remark: You cannot specify wxNullCursor with this event, as it is not considered a valid cursor.

```
setCursor(This, Cursor) -> ok
```

Types:

```
This = wxSetCursorEvent()
Cursor = wxCursor:wxCursor()
```

Sets the cursor associated with this event.

# wxShowEvent

Erlang module

An event being sent when the window is shown or hidden. The event is triggered by calls to `wxWindow:show/2`, and any user action showing a previously hidden window or vice versa (if allowed by the current platform and/or window manager). Notice that the event is not triggered when the application is iconized (minimized) or restored under wxMSW.

See: **Overview events**, `wxWindow:show/2`, `wxWindow:isShown/1`

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxShowEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxShowEventType` to subscribe to events of this type.

## Data Types

```
wxShowEvent() = wx:wx_object()
wxShow() =
    #wxShow{type = wxShowEvent:wxShowEventType(),
            show = boolean()}
wxShowEventType() = show
```

## Exports

```
setShow(This, Show) -> ok
```

Types:

```
   This = wxShowEvent()
   Show = boolean()
```

Set whether the windows was shown or hidden.

```
isShown(This) -> boolean()
```

Types:

```
   This = wxShowEvent()
```

Return true if the window has been shown, false if it has been hidden.

# wxSingleChoiceDialog

Erlang module

This class represents a dialog that shows a list of strings, and allows the user to select one. Double-clicking on a list item is equivalent to single-clicking and then pressing OK.

Styles

This class supports the following styles:

See: **Overview cmndlg**, `wxMultiChoiceDialog`

This class is derived (and can use functions) from: `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxSingleChoiceDialog**

## Data Types

```
wxSingleChoiceDialog() = wx:wx_object()
```

## Exports

```
new(Parent, Message, Caption, Choices) -> wxSingleChoiceDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Message = Caption = unicode:chardata()
   Choices = [unicode:chardata()]
```

```
new(Parent, Message, Caption, Choices, Options :: [Option]) ->
       wxSingleChoiceDialog()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Message = Caption = unicode:chardata()
   Choices = [unicode:chardata()]
   Option =
       {style, integer()} | {pos, {X :: integer(), Y :: integer()}}
```

Constructor, taking an array of `wxString` (not implemented in wx) choices and optional client data.

Remark: Use `wxDialog:showModal/1` to show the dialog.

```
getSelection(This) -> integer()
```

Types:

```
   This = wxSingleChoiceDialog()
```

Returns the index of selected item.

```
getStringSelection(This) -> unicode:charlist()
```

Types:

```
   This = wxSingleChoiceDialog()
```

Returns the selected string.

```
setSelection(This, Selection) -> ok
```

Types:

```
This = wxSingleChoiceDialog()
Selection = integer()
```

Sets the index of the initially selected item.

```
destroy(This :: wxSingleChoiceDialog()) -> ok
```

Destroys the object.

# wxSizeEvent

Erlang module

A size event holds information about size change events of `wxWindow`.

The EVT_SIZE handler function will be called when the window has been resized.

You may wish to use this for frames to resize their child windows as appropriate.

Note that the size passed is of the whole window: call `wxWindow:getClientSize/1` for the area which may be used by the application.

When a window is resized, usually only a small part of the window is damaged and you may only need to repaint that area. However, if your drawing depends on the size of the window, you may need to clear the DC explicitly and repaint the whole window. In which case, you may need to call `wxWindow:refresh/2` to invalidate the entire window.

`Important` : Sizers ( see overview_sizer ) rely on size events to function correctly. Therefore, in a sizer-based layout, do not forget to call Skip on all size events you catch (and don't catch size events at all when you don't need to).

See: {Width,Height}, **Overview events**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxSizeEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxSizeEventType` to subscribe to events of this type.

## Data Types

```
wxSizeEvent() = wx:wx_object()
wxSize() =
    #wxSize{type = wxSizeEvent:wxSizeEventType(),
            size = {W :: integer(), H :: integer()},
            rect =
                {X :: integer(),
                 Y :: integer(),
                 W :: integer(),
                 H :: integer()}}
wxSizeEventType() = size
```

## Exports

```
getSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxSizeEvent()
```

Returns the entire size of the window generating the size change event.

This is the new total size of the window, i.e. the same size as would be returned by `wxWindow:getSize/1` if it were called now. Use `wxWindow:getClientSize/1` if you catch this event in a top level window such as `wxFrame` to find the size available for the window contents.

```
getRect(This) ->
```

```
           {X :: integer(),
            Y :: integer(),
            W :: integer(),
            H :: integer()}
```

Types:

```
   This = wxSizeEvent()
```

# wxSizer

Erlang module

`wxSizer` is the abstract base class used for laying out subwindows in a window. You cannot use `wxSizer` directly; instead, you will have to use one of the sizer classes derived from it. Currently there are `wxBoxSizer`, `wxStaticBoxSizer`, `wxGridSizer`, `wxFlexGridSizer`, `wxWrapSizer` (not implemented in wx) and `wxGridBagSizer`.

The layout algorithm used by sizers in wxWidgets is closely related to layout in other GUI toolkits, such as Java's AWT, the GTK toolkit or the Qt toolkit. It is based upon the idea of the individual subwindows reporting their minimal required size and their ability to get stretched if the size of the parent window has changed.

This will most often mean that the programmer does not set the original size of a dialog in the beginning, rather the dialog will be assigned a sizer and this sizer will be queried about the recommended size. The sizer in turn will query its children, which can be normal windows, empty space or other sizers, so that a hierarchy of sizers can be constructed. Note that `wxSizer` does not derive from `wxWindow` and thus does not interfere with tab ordering and requires very little resources compared to a real window on screen.

What makes sizers so well fitted for use in wxWidgets is the fact that every control reports its own minimal size and the algorithm can handle differences in font sizes or different window (dialog item) sizes on different platforms without problems. If e.g. the standard font as well as the overall design of Motif widgets requires more space than on Windows, the initial dialog size will automatically be bigger on Motif than on Windows.

Sizers may also be used to control the layout of custom drawn items on the window. The `add/4`, `insert/5`, and `prepend/4` functions return a pointer to the newly added `wxSizerItem`. Just add empty space of the desired size and attributes, and then use the `wxSizerItem:getRect/1` method to determine where the drawing operations should take place.

Please notice that sizers, like child windows, are owned by the library and will be deleted by it which implies that they must be allocated on the heap. However if you create a sizer and do not add it to another sizer or window, the library wouldn't be able to delete such an orphan sizer and in this, and only this, case it should be deleted explicitly.

wxSizer flags

The "flag" argument accepted by `wxSizerItem` constructors and other functions, e.g. `add/4`, is an OR-combination of the following flags. Two main behaviours are defined using these flags. One is the border around a window: the border parameter determines the border width whereas the flags given here determine which side(s) of the item that the border will be added. The other flags determine how the sizer item behaves when the space allotted to the sizer changes, and is somewhat dependent on the specific kind of sizer used.

See: **Overview sizer**

wxWidgets docs: **wxSizer**

## Data Types

```
wxSizer() = wx:wx_object()
```

## Exports

```
add(This, Window) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Window = wxWindow:wxWindow() | wxSizer:wxSizer()

add(This, Width, Height) -> wxSizerItem:wxSizerItem()
add(This, Window, Flags) -> wxSizerItem:wxSizerItem()
add(This, Window, Height :: [Option]) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Window = wxWindow:wxWindow() | wxSizer:wxSizer()
   Option =
       {proportion, integer()} |
       {flag, integer()} |
       {border, integer()} |
       {userData, wx:wx_object()}
```

Appends a child to the sizer.

`wxSizer` itself is an abstract class, but the parameters are equivalent in the derived classes that you will instantiate to use it so they are described here:

```
add(This, Width, Height, Options :: [Option]) ->
       wxSizerItem:wxSizerItem()
add(This, Width, Height, Flags) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Width = Height = integer()
   Flags = wxSizerFlags:wxSizerFlags()
```

Appends a spacer child to the sizer.

```
addSpacer(This, Size) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Size = integer()
```

This base function adds non-stretchable space to both the horizontal and vertical orientation of the sizer.

More readable way of calling:

See: `addSpacer/2`

```
addStretchSpacer(This) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()

addStretchSpacer(This, Options :: [Option]) ->
                    wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Option = {prop, integer()}
```

Adds stretchable space to the sizer.

More readable way of calling:

```
calcMin(This) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxSizer()
```

This method is abstract and has to be overwritten by any derived class.

Here, the sizer will do the actual calculation of its children's minimal sizes.

```
clear(This) -> ok
```

Types:

```
   This = wxSizer()
```

```
clear(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxSizer()
   Option = {delete_windows, boolean()}
```

Detaches all children from the sizer.

If `delete_windows` is true then child windows will also be deleted.

Notice that child sizers are always deleted, as a general consequence of the principle that sizers own their sizer children, but don't own their window children (because they are already owned by their parent windows).

```
detach(This, Window) -> boolean()
detach(This, Index) -> boolean()
```

Types:

```
   This = wxSizer()
   Index = integer()
```

Detach a item at position `index` from the sizer without destroying it.

This method does not cause any layout or resizing to take place, call `layout/1` to update the layout "on screen" after detaching a child from the sizer. Returns true if the child item was found and detached, false otherwise.

See: `remove/2`

```
fit(This, Window) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxSizer()
   Window = wxWindow:wxWindow()
```

Tell the sizer to resize the `window` so that its client area matches the sizer's minimal size (`ComputeFittingClientSize()` (not implemented in wx) is called to determine it).

This is commonly done in the constructor of the window itself, see sample in the description of `wxBoxSizer`.

Return: The new window size.

See: ComputeFittingClientSize() (not implemented in wx), ComputeFittingWindowSize() (not implemented in wx)

```
setVirtualSizeHints(This, Window) -> ok
```

Types:

```
This = wxSizer()
Window = wxWindow:wxWindow()
```

See: fitInside/2.

```
fitInside(This, Window) -> ok
```

Types:

```
This = wxSizer()
Window = wxWindow:wxWindow()
```

Tell the sizer to resize the virtual size of the window to match the sizer's minimal size.

This will not alter the on screen size of the window, but may cause the addition/removal/alteration of scrollbars required to view the virtual area in windows which manage it.

See: wxScrolledWindow:setScrollbars/6, setVirtualSizeHints/2

```
getChildren(This) -> [wxSizerItem:wxSizerItem()]
```

Types:

```
This = wxSizer()
```

```
getItem(This, Window) -> wxSizerItem:wxSizerItem()
getItem(This, Index) -> wxSizerItem:wxSizerItem()
```

Types:

```
This = wxSizer()
Index = integer()
```

Finds wxSizerItem which is located in the sizer at position index.

Use parameter recursive to search in subsizers too. Returns pointer to item or NULL.

```
getItem(This, Window, Options :: [Option]) ->
           wxSizerItem:wxSizerItem()
```

Types:

```
This = wxSizer()
Window = wxWindow:wxWindow() | wxSizer:wxSizer()
Option = {recursive, boolean()}
```

Finds wxSizerItem which holds the given window.

Use parameter recursive to search in subsizers too. Returns pointer to item or NULL.

```
getSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxSizer()
```

Returns the current size of the sizer.

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxSizer()
```

Returns the current position of the sizer.

```
getMinSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxSizer()
```

Returns the minimal size of the sizer.

This is either the combined minimal size of all the children and their borders or the minimal size set by `setMinSize/3`, depending on which is bigger. Note that the returned value is client size, not window size. In particular, if you use the value to set toplevel window's minimal or actual size, use `wxWindow::SetMinClientSize()` (not implemented in wx) or `wxWindow:setClientSize/3`, not `wxWindow:setMinSize/2` or `wxWindow:setSize/6`.

```
hide(This, Window) -> boolean()
hide(This, Index) -> boolean()
```

Types:

```
This = wxSizer()
Index = integer()
```

Hides the item at position `index`.

To make a sizer item disappear, use `hide/3` followed by `layout/1`.

Use parameter `recursive` to hide elements found in subsizers. Returns true if the child item was found, false otherwise.

See: `isShown/2`, `show/3`

```
hide(This, Window, Options :: [Option]) -> boolean()
```

Types:

```
This = wxSizer()
Window = wxWindow:wxWindow() | wxSizer:wxSizer()
Option = {recursive, boolean()}
```

Hides the child `window`.

To make a sizer item disappear, use `hide/3` followed by `layout/1`.

Use parameter `recursive` to hide elements found in subsizers. Returns true if the child item was found, false otherwise.

See: `isShown/2`, `show/3`

```
insert(This, Index, Item) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Index = integer()
   Item = wxSizerItem:wxSizerItem()

insert(This, Index, Width, Height) -> wxSizerItem:wxSizerItem()
insert(This, Index, Window, Flags) -> wxSizerItem:wxSizerItem()
insert(This, Index, Window, Height :: [Option]) ->
          wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Index = integer()
   Window = wxWindow:wxWindow() | wxSizer:wxSizer()
   Option =
       {proportion, integer()} |
       {flag, integer()} |
       {border, integer()} |
       {userData, wx:wx_object()}
```

Insert a child into the sizer before any existing item at `index`.

See `add/4` for the meaning of the other parameters.

```
insert(This, Index, Width, Height, Options :: [Option]) ->
          wxSizerItem:wxSizerItem()
insert(This, Index, Width, Height, Flags) ->
          wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Index = Width = Height = integer()
   Flags = wxSizerFlags:wxSizerFlags()
```

Insert a child into the sizer before any existing item at `index`.

See `add/4` for the meaning of the other parameters.

```
insertSpacer(This, Index, Size) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Index = Size = integer()
```

Inserts non-stretchable space to the sizer.

More readable way of calling wxSizer::Insert(index, size, size).

```
insertStretchSpacer(This, Index) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Index = integer()

insertStretchSpacer(This, Index, Options :: [Option]) ->
```

```
                        wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Index = integer()
   Option = {prop, integer()}
```

Inserts stretchable space to the sizer.

More readable way of calling wxSizer::Insert(0, 0, prop).

```
isShown(This, Window) -> boolean()
isShown(This, Index) -> boolean()
```

Types:

```
   This = wxSizer()
   Index = integer()
```

Returns true if the item at `index` is shown.

See: `hide/3`, `show/3`, `wxSizerItem:isShown/1`

```
recalcSizes(This) -> ok
```

Types:

```
   This = wxSizer()
```

See: `layout/1`.

```
layout(This) -> ok
```

Types:

```
   This = wxSizer()
```

Call this to force layout of the children anew, e.g. after having added a child to or removed a child (window, other sizer or space) from the sizer while keeping the current dimension.

```
prepend(This, Item) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Item = wxSizerItem:wxSizerItem()
```

```
prepend(This, Width, Height) -> wxSizerItem:wxSizerItem()
prepend(This, Window, Flags) -> wxSizerItem:wxSizerItem()
prepend(This, Window, Height :: [Option]) ->
          wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Window = wxWindow:wxWindow() | wxSizer:wxSizer()
   Option =
       {proportion, integer()} |
       {flag, integer()} |
       {border, integer()} |
```

```
       {userData, wx:wx_object()}
```

Same as `add/4`, but prepends the items to the beginning of the list of items (windows, subsizers or spaces) owned by this sizer.

```
prepend(This, Width, Height, Options :: [Option]) ->
           wxSizerItem:wxSizerItem()
prepend(This, Width, Height, Flags) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Width = Height = integer()
   Flags = wxSizerFlags:wxSizerFlags()
```

Same as `add/4`, but prepends the items to the beginning of the list of items (windows, subsizers or spaces) owned by this sizer.

```
prependSpacer(This, Size) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Size = integer()
```

Prepends non-stretchable space to the sizer.

More readable way of calling wxSizer::Prepend(size, size, 0).

```
prependStretchSpacer(This) -> wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
```

```
prependStretchSpacer(This, Options :: [Option]) ->
                        wxSizerItem:wxSizerItem()
```

Types:

```
   This = wxSizer()
   Option = {prop, integer()}
```

Prepends stretchable space to the sizer.

More readable way of calling wxSizer::Prepend(0, 0, prop).

```
remove(This, Index) -> boolean()
remove(This, Sizer) -> boolean()
```

Types:

```
   This = Sizer = wxSizer()
```

Removes a sizer child from the sizer and destroys it.

Note: This method does not cause any layout or resizing to take place, call `layout/1` to update the layout "on screen" after removing a child from the sizer.

Return: true if the child item was found and removed, false otherwise.

```
replace(This, Oldwin, Newwin) -> boolean()
replace(This, Index, Newitem) -> boolean()
```

Types:

```
This = wxSizer()
Index = integer()
Newitem = wxSizerItem:wxSizerItem()
```

Detaches the given item at position `index` from the sizer and replaces it with the given `wxSizerItem` `newitem`.

The detached child is deleted `only` if it is a sizer or a spacer (but not if it is a `wxWindow` because windows are owned by their parent window, not the sizer).

This method does not cause any layout or resizing to take place, call `layout/1` to update the layout "on screen" after replacing a child from the sizer.

Returns true if the child item was found and removed, false otherwise.

```
replace(This, Oldwin, Newwin, Options :: [Option]) -> boolean()
```

Types:

```
This = wxSizer()
Oldwin = Newwin = wxWindow:wxWindow() | wxSizer:wxSizer()
Option = {recursive, boolean()}
```

Detaches the given `oldwin` from the sizer and replaces it with the given `newwin`.

The detached child window is `not` deleted (because windows are owned by their parent window, not the sizer).

Use parameter `recursive` to search the given element recursively in subsizers.

This method does not cause any layout or resizing to take place, call `layout/1` to update the layout "on screen" after replacing a child from the sizer.

Returns true if the child item was found and removed, false otherwise.

```
setDimension(This, Pos, Size) -> ok
```

Types:

```
This = wxSizer()
Pos = {X :: integer(), Y :: integer()}
Size = {W :: integer(), H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setDimension(This, X, Y, Width, Height) -> ok
```

Types:

```
This = wxSizer()
X = Y = Width = Height = integer()
```

Call this to force the sizer to take the given dimension and thus force the items owned by the sizer to resize themselves according to the rules defined by the parameter in the `add/4` and `prepend/4` methods.

```
setMinSize(This, Size) -> ok
```

Types:

```
This = wxSizer()
Size = {W :: integer(), H :: integer()}
```

Call this to give the sizer a minimal size.

Normally, the sizer will calculate its minimal size based purely on how much space its children need. After calling this method `getMinSize/1` will return either the minimal size as requested by its children or the minimal size set here, depending on which is bigger.

```
setMinSize(This, Width, Height) -> ok
```

Types:

```
This = wxSizer()
Width = Height = integer()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setItemMinSize(This, Window, Size) -> boolean()
setItemMinSize(This, Index, Size) -> boolean()
```

Types:

```
This = wxSizer()
Index = integer()
Size = {W :: integer(), H :: integer()}
```

```
setItemMinSize(This, Window, Width, Height) -> boolean()
setItemMinSize(This, Index, Width, Height) -> boolean()
```

Types:

```
This = wxSizer()
Index = Width = Height = integer()
```

```
setSizeHints(This, Window) -> ok
```

Types:

```
This = wxSizer()
Window = wxWindow:wxWindow()
```

This method first calls `fit/2` and then `setSizeHints/2` on the `window` passed to it.

This only makes sense when `window` is actually a `wxTopLevelWindow` such as a `wxFrame` or a `wxDialog`, since SetSizeHints only has any effect in these classes. It does nothing in normal windows or controls.

This method is implicitly used by `wxWindow:setSizerAndFit/3` which is commonly invoked in the constructor of a toplevel window itself (see the sample in the description of `wxBoxSizer`) if the toplevel window is resizable.

```
show(This, Window) -> boolean()
show(This, Index) -> boolean()
show(This, Show) -> ok
```

Types:

```
   This = wxSizer()
   Show = boolean()
```

```
show(This, Window, Options :: [Option]) -> boolean()
show(This, Index, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxSizer()
   Index = integer()
   Option = {show, boolean()}
```

Shows the item at `index`.

To make a sizer item disappear or reappear, use `show/3` followed by `layout/1`.

Returns true if the child item was found, false otherwise.

See: `hide/3`, `isShown/2`

```
showItems(This, Show) -> ok
```

Types:

```
   This = wxSizer()
   Show = boolean()
```

Show or hide all items managed by the sizer.

# wxSizerFlags

Erlang module

Container for sizer items flags providing readable names for them.

Normally, when you add an item to a sizer via `wxSizer:add/4`, you have to specify a lot of flags and parameters which can be unwieldy. This is where `wxSizerFlags` comes in: it allows you to specify all parameters using the named methods instead. For example, instead of

you can now write

This is more readable and also allows you to create `wxSizerFlags` objects which can be reused for several sizer items.

Note that by specification, all methods of `wxSizerFlags` return the `wxSizerFlags` object itself to allowing chaining multiple methods calls like in the examples above.

See: `wxSizer`

wxWidgets docs: **wxSizerFlags**

## Data Types

```
wxSizerFlags() = wx:wx_object()
```

## Exports

```
new() -> wxSizerFlags()
```

```
new(Options :: [Option]) -> wxSizerFlags()
```

Types:

```
Option = {proportion, integer()}
```

Creates the `wxSizer` with the proportion specified by `proportion`.

```
align(This, Alignment) -> wxSizerFlags()
```

Types:

```
This = wxSizerFlags()
Alignment = integer()
```

Sets the alignment of this `wxSizerFlags` to `align`.

This method replaces the previously set alignment with the specified one.

See: `Top()` (not implemented in wx), `left/1`, `right/1`, `Bottom()` (not implemented in wx), `centre/1`

```
border(This) -> wxSizerFlags()
```

Types:

```
This = wxSizerFlags()
```

```
border(This, Options :: [Option]) -> wxSizerFlags()
```

Types:

```
This = wxSizerFlags()
Option = {direction, integer()}
```

Sets the `wxSizerFlags` to have a border with size as returned by `GetDefaultBorder()` (not implemented in wx).

```
border(This, Direction, Borderinpixels) -> wxSizerFlags()
```

Types:

```
This = wxSizerFlags()
Direction = Borderinpixels = integer()
```

Sets the `wxSizerFlags` to have a border of a number of pixels specified by `borderinpixels` with the directions specified by `direction`.

Prefer to use the overload below or `DoubleBorder()` (not implemented in wx) or `TripleBorder()` (not implemented in wx) versions instead of hard-coding the border value in pixels to avoid too small borders on devices with high DPI displays.

```
centre(This) -> wxSizerFlags()
```

Types:

```
This = wxSizerFlags()
```

See: `center/1`.

```
center(This) -> wxSizerFlags()
```

Types:

```
This = wxSizerFlags()
```

Sets the object of the `wxSizerFlags` to center itself in the area it is given.

```
expand(This) -> wxSizerFlags()
```

Types:

```
This = wxSizerFlags()
```

Sets the object of the `wxSizerFlags` to expand to fill as much area as it can.

```
left(This) -> wxSizerFlags()
```

Types:

```
This = wxSizerFlags()
```

Aligns the object to the left, similar for `Align(wxALIGN_LEFT)`.

Unlike `align/2`, this method doesn't change the vertical alignment of the item.

```
proportion(This, Proportion) -> wxSizerFlags()
```

Types:

```
This = wxSizerFlags()
Proportion = integer()
```

Sets the proportion of this `wxSizerFlags` to `proportion`.

```
right(This) -> wxSizerFlags()
```

Types:

```
This = wxSizerFlags()
```

Aligns the object to the right, similar for `Align(wxALIGN_RIGHT)`.

Unlike `align/2`, this method doesn't change the vertical alignment of the item.

```
destroy(This :: wxSizerFlags()) -> ok
```

Destroys the object.

# wxSizerItem

Erlang module

The `wxSizerItem` class is used to track the position, size and other attributes of each item managed by a `wxSizer`.

It is not usually necessary to use this class because the sizer elements can also be identified by their positions or window or sizer pointers but sometimes it may be more convenient to use it directly.

wxWidgets docs: **wxSizerItem**

## Data Types

```
wxSizerItem() = wx:wx_object()
```

## Exports

```
new(Window) -> wxSizerItem()
```

Types:

```
   Window = wxWindow:wxWindow() | wxSizer:wxSizer()
```

```
new(Width, Height) -> wxSizerItem()
new(Window, Flags) -> wxSizerItem()
new(Window, Height :: [Option]) -> wxSizerItem()
```

Types:

```
   Window = wxWindow:wxWindow() | wxSizer:wxSizer()
   Option =
       {proportion, integer()} |
       {flag, integer()} |
       {border, integer()} |
       {userData, wx:wx_object()}
```

```
new(Width, Height, Options :: [Option]) -> wxSizerItem()
```

Types:

```
   Width = Height = integer()
   Option =
       {proportion, integer()} |
       {flag, integer()} |
       {border, integer()} |
       {userData, wx:wx_object()}
```

Construct a sizer item for tracking a spacer.

```
destroy(This :: wxSizerItem()) -> ok
```

Deletes the user data and subsizer, if any.

```
calcMin(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxSizerItem()
```

Calculates the minimum desired size for the item, including any space needed by borders.

```
deleteWindows(This) -> ok
```

Types:

```
This = wxSizerItem()
```

Destroy the window or the windows in a subsizer, depending on the type of item.

```
detachSizer(This) -> ok
```

Types:

```
This = wxSizerItem()
```

Enable deleting the SizerItem without destroying the contained sizer.

```
getBorder(This) -> integer()
```

Types:

```
This = wxSizerItem()
```

Return the border attribute.

```
getFlag(This) -> integer()
```

Types:

```
This = wxSizerItem()
```

Return the flags attribute.

See `wxSizer flags list` (not implemented in wx) for details.

```
getMinSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxSizerItem()
```

Get the minimum size needed for the item.

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxSizerItem()
```

What is the current position of the item, as set in the last Layout.

```
getProportion(This) -> integer()
```

Types:

```
This = wxSizerItem()
```

Get the proportion item attribute.

```
getRatio(This) -> number()
```

Types:

```
   This = wxSizerItem()
```

Get the ratio item attribute.

```
getRect(This) ->
           {X :: integer(),
            Y :: integer(),
            W :: integer(),
            H :: integer()}
```

Types:

```
   This = wxSizerItem()
```

Get the rectangle of the item on the parent window, excluding borders.

```
getSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxSizerItem()
```

Get the current size of the item, as set in the last Layout.

```
getSizer(This) -> wxSizer:wxSizer()
```

Types:

```
   This = wxSizerItem()
```

If this item is tracking a sizer, return it.

NULL otherwise.

```
getSpacer(This) -> {W :: integer(), H :: integer()}
```

Types:

```
   This = wxSizerItem()
```

If this item is tracking a spacer, return its size.

```
getUserData(This) -> wx:wx_object()
```

Types:

```
   This = wxSizerItem()
```

Get the userData item attribute.

```
getWindow(This) -> wxWindow:wxWindow()
```

Types:

```
   This = wxSizerItem()
```

If this item is tracking a window then return it.

NULL otherwise.

```
isSizer(This) -> boolean()
```

Types:

```
   This = wxSizerItem()
```

Is this item a sizer?

```
isShown(This) -> boolean()
```

Types:

```
This = wxSizerItem()
```

Returns true if this item is a window or a spacer and it is shown or if this item is a sizer and not all of its elements are hidden.

In other words, for sizer items, all of the child elements must be hidden for the sizer itself to be considered hidden.

As an exception, if the `wxRESERVE_SPACE_EVEN_IF_HIDDEN` flag was used for this sizer item, then `isShown/1` always returns true for it (see `wxSizerFlags::ReserveSpaceEvenIfHidden()` (not implemented in wx)).

```
isSpacer(This) -> boolean()
```

Types:

```
This = wxSizerItem()
```

Is this item a spacer?

```
isWindow(This) -> boolean()
```

Types:

```
This = wxSizerItem()
```

Is this item a window?

```
setBorder(This, Border) -> ok
```

Types:

```
This = wxSizerItem()
Border = integer()
```

Set the border item attribute.

```
setDimension(This, Pos, Size) -> ok
```

Types:

```
This = wxSizerItem()
Pos = {X :: integer(), Y :: integer()}
Size = {W :: integer(), H :: integer()}
```

Set the position and size of the space allocated to the sizer, and adjust the position and size of the item to be within that space taking alignment and borders into account.

```
setFlag(This, Flag) -> ok
```

Types:

```
This = wxSizerItem()
Flag = integer()
```

Set the flag item attribute.

```
setInitSize(This, X, Y) -> ok
```

Types:

```
This = wxSizerItem()
X = Y = integer()
```

Sets the minimum size to be allocated for this item.

This is identical to `setMinSize/3`, prefer to use the other function, as its name is more clear.

```
setMinSize(This, Size) -> ok
```

Types:

```
This = wxSizerItem()
Size = {W :: integer(), H :: integer()}
```

Sets the minimum size to be allocated for this item.

If this item is a window, the `size` is also passed to `wxWindow:setMinSize/2`.

```
setMinSize(This, X, Y) -> ok
```

Types:

```
This = wxSizerItem()
X = Y = integer()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setProportion(This, Proportion) -> ok
```

Types:

```
This = wxSizerItem()
Proportion = integer()
```

Set the proportion item attribute.

```
setRatio(This, Ratio) -> ok
setRatio(This, Size) -> ok
```

Types:

```
This = wxSizerItem()
Size = {W :: integer(), H :: integer()}
```

```
setRatio(This, Width, Height) -> ok
```

Types:

```
This = wxSizerItem()
Width = Height = integer()
```

Set the ratio item attribute.

```
assignSizer(This, Sizer) -> ok
```

Types:

```
This = wxSizerItem()
Sizer = wxSizer:wxSizer()
```

Set the sizer tracked by this item.

Old sizer, if any, is deleted.

```
assignSpacer(This, Size) -> ok
```

Types:

```
This = wxSizerItem()
Size = {W :: integer(), H :: integer()}
```

Set the size of the spacer tracked by this item.

Old spacer, if any, is deleted.

```
assignSpacer(This, W, H) -> ok
```

Types:

```
This = wxSizerItem()
W = H = integer()
```

```
assignWindow(This, Window) -> ok
```

Types:

```
This = wxSizerItem()
Window = wxWindow:wxWindow()
```

Set the window to be tracked by this item.

Note: This is a low-level method which is dangerous if used incorrectly, avoid using it if possible, i.e. if higher level methods such as `wxSizer:replace/4` can be used instead.

If the sizer item previously contained a window, it is dissociated from the sizer containing this sizer item (if any), but this object doesn't have the pointer to the containing sizer and so it's the caller's responsibility to call `wxWindow:setContainingSizer/2` on `window`. Failure to do this can result in memory corruption when the window is destroyed later, so it is crucial to not forget to do it.

Also note that the previously contained window is `not` deleted, so it's also the callers responsibility to do it, if necessary.

```
show(This, Show) -> ok
```

Types:

```
This = wxSizerItem()
Show = boolean()
```

Set the show item attribute, which sizers use to determine if the item is to be made part of the layout or not.

If the item is tracking a window then it is shown or hidden as needed.

# wxSlider

Erlang module

A slider is a control with a handle which can be pulled back and forth to change the value.

On Windows, the track bar control is used.

On GTK+, tick marks are only available for version 2.16 and later.

Slider generates the same events as `wxScrollBar` but in practice the most convenient way to process `wxSlider` updates is by handling the slider-specific `wxEVT_SLIDER` event which carries `wxCommandEvent` containing just the latest slider position.

Styles

This class supports the following styles:

The difference between EVT_SCROLL_THUMBRELEASE and EVT_SCROLL_CHANGED

The EVT_SCROLL_THUMBRELEASE event is only emitted when actually dragging the thumb using the mouse and releasing it (This EVT_SCROLL_THUMBRELEASE event is also followed by an EVT_SCROLL_CHANGED event).

The EVT_SCROLL_CHANGED event also occurs when using the keyboard to change the thumb position, and when clicking next to the thumb (In all these cases the EVT_SCROLL_THUMBRELEASE event does not happen). In short, the EVT_SCROLL_CHANGED event is triggered when scrolling/ moving has finished independently of the way it had started. Please see the page_samples_widgets ("Slider" page) to see the difference between EVT_SCROLL_THUMBRELEASE and EVT_SCROLL_CHANGED in action.

See: **Overview events**, `wxScrollBar`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxSlider**

## Events

Event types emitted from this class: `scroll_top`, `scroll_bottom`, `scroll_lineup`, `scroll_linedown`, `scroll_pageup`, `scroll_pagedown`, `scroll_thumbtrack`, `scroll_thumbrelease`, `scroll_changed`, `scroll_top`, `scroll_bottom`, `scroll_lineup`, `scroll_linedown`, `scroll_pageup`, `scroll_pagedown`, `scroll_thumbtrack`, `scroll_thumbrelease`, `scroll_changed`, `command_slider_updated`

## Data Types

```
wxSlider() = wx:wx_object()
```

## Exports

```
new() -> wxSlider()
```

Default constructor.

```
new(Parent, Id, Value, MinValue, MaxValue) -> wxSlider()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = Value = MinValue = MaxValue = integer()

new(Parent, Id, Value, MinValue, MaxValue, Options :: [Option]) ->
       wxSlider()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = Value = MinValue = MaxValue = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Constructor, creating and showing a slider.

See: `create/7`, `wxValidator` (not implemented in wx)

```
destroy(This :: wxSlider()) -> ok
```

Destructor, destroying the slider.

```
create(This, Parent, Id, Value, MinValue, MaxValue) -> boolean()
```

Types:

```
   This = wxSlider()
   Parent = wxWindow:wxWindow()
   Id = Value = MinValue = MaxValue = integer()

create(This, Parent, Id, Value, MinValue, MaxValue,
       Options :: [Option]) ->
          boolean()
```

Types:

```
   This = wxSlider()
   Parent = wxWindow:wxWindow()
   Id = Value = MinValue = MaxValue = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Used for two-step slider construction.

See `new/6` for further details.

```
getLineSize(This) -> integer()
```

Types:

```
This = wxSlider()
```

Returns the line size.

See: `setLineSize/2`

```
getMax(This) -> integer()
```

Types:

```
This = wxSlider()
```

Gets the maximum slider value.

See: `getMin/1`, `setRange/3`

```
getMin(This) -> integer()
```

Types:

```
This = wxSlider()
```

Gets the minimum slider value.

See: `getMin/1`, `setRange/3`

```
getPageSize(This) -> integer()
```

Types:

```
This = wxSlider()
```

Returns the page size.

See: `setPageSize/2`

```
getThumbLength(This) -> integer()
```

Types:

```
This = wxSlider()
```

Returns the thumb length.

Only for:wxmsw

See: `setThumbLength/2`

```
getValue(This) -> integer()
```

Types:

```
This = wxSlider()
```

Gets the current slider value.

See: `getMin/1`, `getMax/1`, `setValue/2`

```
setLineSize(This, LineSize) -> ok
```

Types:

```
This = wxSlider()
LineSize = integer()
```

Sets the line size for the slider.

See: `getLineSize/1`

```
setPageSize(This, PageSize) -> ok
```

Types:

```
This = wxSlider()
PageSize = integer()
```

Sets the page size for the slider.

See: `getPageSize/1`

```
setRange(This, MinValue, MaxValue) -> ok
```

Types:

```
This = wxSlider()
MinValue = MaxValue = integer()
```

Sets the minimum and maximum slider values.

See: `getMin/1`, `getMax/1`

```
setThumbLength(This, Len) -> ok
```

Types:

```
This = wxSlider()
Len = integer()
```

Sets the slider thumb length.

Only for:wxmsw

See: `getThumbLength/1`

```
setValue(This, Value) -> ok
```

Types:

```
This = wxSlider()
Value = integer()
```

Sets the slider position.

# wxSpinButton

Erlang module

A `wxSpinButton` has two small up and down (or left and right) arrow buttons.

It is often used next to a text control for increment and decrementing a value. Portable programs should try to use `wxSpinCtrl` instead as `wxSpinButton` is not implemented for all platforms but `wxSpinCtrl` is as it degenerates to a simple `wxTextCtrl` on such platforms.

Note: the range supported by this control (and `wxSpinCtrl`) depends on the platform but is at least `-0x8000` to `0x7fff`. Under GTK and Win32 with sufficiently new version of `comctrl32.dll` (at least 4.71 is required, 5.80 is recommended) the full 32 bit range is supported.

Styles

This class supports the following styles:

See: `wxSpinCtrl`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxSpinButton**

## Events

Event types emitted from this class: `spin`, `spin_up`, `spin_down`

## Data Types

```
wxSpinButton() = wx:wx_object()
```

## Exports

```
new() -> wxSpinButton()
```

Default constructor.

```
new(Parent) -> wxSpinButton()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxSpinButton()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructor, creating and showing a spin button.

See: `create/3`

```
destroy(This :: wxSpinButton()) -> ok
```

Destructor, destroys the spin button control.

```
create(This, Parent) -> boolean()
```

Types:

```
   This = wxSpinButton()
   Parent = wxWindow:wxWindow()
```

```
create(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxSpinButton()
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Scrollbar creation function called by the spin button constructor.

See `new/2` for details.

```
getMax(This) -> integer()
```

Types:

```
   This = wxSpinButton()
```

Returns the maximum permissible value.

See: `setRange/3`

```
getMin(This) -> integer()
```

Types:

```
   This = wxSpinButton()
```

Returns the minimum permissible value.

See: `setRange/3`

```
getValue(This) -> integer()
```

Types:

```
   This = wxSpinButton()
```

Returns the current spin button value.

See: `setValue/2`

```
setRange(This, Min, Max) -> ok
```

Types:

```
   This = wxSpinButton()
   Min = Max = integer()
```

Sets the range of the spin button.

In portable code, `min` should be less than or equal to `max`. In wxMSW it is possible to specify minimum greater than maximum and the native control supports the same range as if they were reversed, but swaps the meaning of up and down arrows, however this dubious feature is not supported on other platforms.

See: `getMin/1`, `getMax/1`

```
setValue(This, Value) -> ok
```

Types:

```
This = wxSpinButton()
Value = integer()
```

Sets the value of the spin button.

# wxSpinCtrl

Erlang module

`wxSpinCtrl` combines `wxTextCtrl` and `wxSpinButton` in one control.

Styles

This class supports the following styles:

See: `wxSpinButton`, `wxSpinCtrlDouble` (not implemented in wx), `wxControl`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxSpinCtrl**

## Events

Event types emitted from this class: `command_spinctrl_updated`

## Data Types

```
wxSpinCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxSpinCtrl()
```

Default constructor.

```
new(Parent) -> wxSpinCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxSpinCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {value, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {min, integer()} |
       {max, integer()} |
       {initial, integer()}
```

Constructor, creating and showing a spin control.

If `value` is non-empty, it will be shown in the text entry part of the control and if it has numeric value, the initial numeric value of the control, as returned by `getValue/1` will also be determined by it instead of by `initial`. Hence, it only makes sense to specify `initial` if `value` is an empty string or is not convertible to a number, otherwise `initial` is simply ignored and the number specified by `value` is used.

See: `create/3`

```
create(This, Parent) -> boolean()
```

Types:

```
   This = wxSpinCtrl()
   Parent = wxWindow:wxWindow()
```

```
create(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxSpinCtrl()
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {value, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {min, integer()} |
       {max, integer()} |
       {initial, integer()}
```

Creation function called by the spin control constructor.

See `new/2` for details.

```
setValue(This, Value) -> ok
setValue(This, Text) -> ok
```

Types:

```
   This = wxSpinCtrl()
   Text = unicode:chardata()
```

Sets the value of the spin control.

It is recommended to use the overload taking an integer value instead.

Notice that, unlike `wxTextCtrl:setValue/2`, but like most of the other setter methods in wxWidgets, calling this method does not generate any events as events are only generated for the user actions.

```
getValue(This) -> integer()
```

Types:

```
   This = wxSpinCtrl()
```

Gets the value of the spin control.

```
setRange(This, MinVal, MaxVal) -> ok
```

Types:

```
   This = wxSpinCtrl()
   MinVal = MaxVal = integer()
```

Sets range of allowable values.

Notice that calling this method may change the value of the control if it's not inside the new valid range, e.g. it will become `minVal` if it is less than it now. However no `wxEVT_SPINCTRL` event is generated, even if it the value does change.

Note: Setting a range including negative values is silently ignored if current base is set to 16.

```
setSelection(This, From, To) -> ok
```

Types:

```
This = wxSpinCtrl()
From = To = integer()
```

Select the text in the text part of the control between positions `from` (inclusive) and `to` (exclusive).

This is similar to `wxTextCtrl:setSelection/3`.

Note: this is currently only implemented for Windows and generic versions of the control.

```
getMin(This) -> integer()
```

Types:

```
This = wxSpinCtrl()
```

Gets minimal allowable value.

```
getMax(This) -> integer()
```

Types:

```
This = wxSpinCtrl()
```

Gets maximal allowable value.

```
destroy(This :: wxSpinCtrl()) -> ok
```

Destroys the object.

# wxSpinEvent

Erlang module

This event class is used for the events generated by `wxSpinButton` and `wxSpinCtrl`.

See: `wxSpinButton`, and, `wxSpinCtrl`

This class is derived (and can use functions) from: `wxNotifyEvent` `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxSpinEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxSpinEventType` to subscribe to events of this type.

## Data Types

```
wxSpinEvent() = wx:wx_object()
wxSpin() =
    #wxSpin{type = wxSpinEvent:wxSpinEventType(),
            commandInt = integer()}
wxSpinEventType() =
    command_spinctrl_updated | spin_up | spin_down | spin
```

## Exports

```
getPosition(This) -> integer()
```

Types:

```
   This = wxSpinEvent()
```

Retrieve the current spin button or control value.

```
setPosition(This, Pos) -> ok
```

Types:

```
   This = wxSpinEvent()
   Pos = integer()
```

Set the value associated with the event.

# wxSplashScreen

Erlang module

`wxSplashScreen` shows a window with a thin border, displaying a bitmap describing your application.

Show it in application initialisation, and then either explicitly destroy it or let it time-out.

Example usage:

This class is derived (and can use functions) from: `wxFrame` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxSplashScreen**

## Data Types

```
wxSplashScreen() = wx:wx_object()
```

## Exports

```
new(Bitmap, SplashStyle, Milliseconds, Parent, Id) ->
       wxSplashScreen()
```

Types:

```
   Bitmap = wxBitmap:wxBitmap()
   SplashStyle = Milliseconds = integer()
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
new(Bitmap, SplashStyle, Milliseconds, Parent, Id,
    Options :: [Option]) ->
       wxSplashScreen()
```

Types:

```
   Bitmap = wxBitmap:wxBitmap()
   SplashStyle = Milliseconds = integer()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Construct the splash screen passing a bitmap, a style, a timeout, a window id, optional position and size, and a window style.

`splashStyle` is a bitlist of some of the following:

`milliseconds` is the timeout in milliseconds.

```
destroy(This :: wxSplashScreen()) -> ok
```

Destroys the splash screen.

```
getSplashStyle(This) -> integer()
```

Types:

```
This = wxSplashScreen()
```

Returns the splash style (see `new/6` for details).

```
getTimeout(This) -> integer()
```

Types:

```
This = wxSplashScreen()
```

Returns the timeout in milliseconds.

# wxSplitterEvent

Erlang module

This class represents the events generated by a splitter control.

Also there is only one event class, the data associated to the different events is not the same and so not all accessor functions may be called for each event. The documentation mentions the kind of event(s) for which the given accessor function makes sense: calling it for other types of events will result in assert failure (in debug mode) and will return meaningless results.

See: `wxSplitterWindow`, **Overview events**

This class is derived (and can use functions) from: `wxNotifyEvent` `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxSplitterEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxSplitterEventType` to subscribe to events of this type.

## Data Types

```
wxSplitterEvent() = wx:wx_object()
wxSplitter() =
    #wxSplitter{type = wxSplitterEvent:wxSplitterEventType()}
wxSplitterEventType() =
    command_splitter_sash_pos_changed |
    command_splitter_sash_pos_changing |
    command_splitter_doubleclicked | command_splitter_unsplit
```

## Exports

```
getSashPosition(This) -> integer()
```

Types:

```
   This = wxSplitterEvent()
```

Returns the new sash position.

May only be called while processing `wxEVT_SPLITTER_SASH_POS_CHANGING` and `wxEVT_SPLITTER_SASH_POS_CHANGED` events.

```
getX(This) -> integer()
```

Types:

```
   This = wxSplitterEvent()
```

Returns the x coordinate of the double-click point.

May only be called while processing `wxEVT_SPLITTER_DOUBLECLICKED` events.

```
getY(This) -> integer()
```

Types:

```
This = wxSplitterEvent()
```

Returns the y coordinate of the double-click point.

May only be called while processing `wxEVT_SPLITTER_DOUBLECLICKED` events.

```
getWindowBeingRemoved(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxSplitterEvent()
```

Returns a pointer to the window being removed when a splitter window is unsplit.

May only be called while processing `wxEVT_SPLITTER_UNSPLIT` events.

```
setSashPosition(This, Pos) -> ok
```

Types:

```
This = wxSplitterEvent()
Pos = integer()
```

In the case of `wxEVT_SPLITTER_SASH_POS_CHANGED` events, sets the new sash position.

In the case of `wxEVT_SPLITTER_SASH_POS_CHANGING` events, sets the new tracking bar position so visual feedback during dragging will represent that change that will actually take place. Set to -1 from the event handler code to prevent repositioning.

May only be called while processing `wxEVT_SPLITTER_SASH_POS_CHANGING` and `wxEVT_SPLITTER_SASH_POS_CHANGED` events.

# wxSplitterWindow

Erlang module

This class manages up to two subwindows. The current view can be split into two programmatically (perhaps from a menu command), and unsplit either programmatically or via the `wxSplitterWindow` user interface.

Styles

This class supports the following styles:

See: `wxSplitterEvent`, **Overview splitterwindow**

This class is derived (and can use functions) from: `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxSplitterWindow**

## Events

Event types emitted from this class: `command_splitter_sash_pos_changing`, `command_splitter_sash_pos_changed`, `command_splitter_unsplit`

## Data Types

```
wxSplitterWindow() = wx:wx_object()
```

## Exports

```
new() -> wxSplitterWindow()
```

Default constructor.

```
new(Parent) -> wxSplitterWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxSplitterWindow()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructor for creating the window.

Remark: After using this constructor, you must create either one or two subwindows with the splitter window as parent, and then call one of `initialize/2`, `splitVertically/4` and `splitHorizontally/4` in order to set the pane(s). You can create two windows, with one hidden when not being shown; or you can create and delete the second pane on demand.

See: `initialize/2`, `splitVertically/4`, `splitHorizontally/4`, `create/3`

```
destroy(This :: wxSplitterWindow()) -> ok
```

Destroys the `wxSplitterWindow` and its children.

```
create(This, Parent) -> boolean()
```

Types:

```
   This = wxSplitterWindow()
   Parent = wxWindow:wxWindow()
```

```
create(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxSplitterWindow()
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Creation function, for two-step construction.

See `new/2` for details.

```
getMinimumPaneSize(This) -> integer()
```

Types:

```
   This = wxSplitterWindow()
```

Returns the current minimum pane size (defaults to zero).

See: `setMinimumPaneSize/2`

```
getSashGravity(This) -> number()
```

Types:

```
   This = wxSplitterWindow()
```

Returns the current sash gravity.

See: `setSashGravity/2`

```
getSashPosition(This) -> integer()
```

Types:

```
   This = wxSplitterWindow()
```

Returns the current sash position.

See: `setSashPosition/3`

```
getSplitMode(This) -> wx:wx_enum()
```

Types:

```
   This = wxSplitterWindow()
```

Gets the split mode.

See: `setSplitMode/2`, `splitVertically/4`, `splitHorizontally/4`

```
getWindow1(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxSplitterWindow()
```

Returns the left/top or only pane.

```
getWindow2(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxSplitterWindow()
```

Returns the right/bottom pane.

```
initialize(This, Window) -> ok
```

Types:

```
This = wxSplitterWindow()
Window = wxWindow:wxWindow()
```

Initializes the splitter window to have one pane.

The child window is shown if it is currently hidden.

Remark: This should be called if you wish to initially view only a single pane in the splitter window.

See: `splitVertically/4`, `splitHorizontally/4`

```
isSplit(This) -> boolean()
```

Types:

```
This = wxSplitterWindow()
```

Returns true if the window is split, false otherwise.

```
replaceWindow(This, WinOld, WinNew) -> boolean()
```

Types:

```
This = wxSplitterWindow()
WinOld = WinNew = wxWindow:wxWindow()
```

This function replaces one of the windows managed by the `wxSplitterWindow` with another one.

It is in general better to use it instead of calling `unsplit/2` and then resplitting the window back because it will provoke much less flicker (if any). It is valid to call this function whether the splitter has two windows or only one.

Both parameters should be non-NULL and `winOld` must specify one of the windows managed by the splitter. If the parameters are incorrect or the window couldn't be replaced, false is returned. Otherwise the function will return true, but please notice that it will not delete the replaced window and you may wish to do it yourself.

See: `getMinimumPaneSize/1`

```
setSashGravity(This, Gravity) -> ok
```

Types:

```
This = wxSplitterWindow()
Gravity = number()
```

Sets the sash gravity.

Remark: Gravity is real factor which controls position of sash while resizing `wxSplitterWindow`. Gravity tells `wxSplitterWindow` how much will left/top window grow while resizing. Example values:

Notice that when sash gravity for a newly created splitter window, it is often necessary to explicitly set the splitter size using `wxWindow:setSize/6` to ensure that is big enough for its initial sash position. Otherwise, i.e. if the window is created with the default tiny size and only resized to its correct size later, the initial sash position will be affected by the gravity and typically result in sash being at the rightmost position for the gravity of 1. See the example code creating `wxSplitterWindow` in the splitter sample for more details.

See: `getSashGravity/1`

```
setSashPosition(This, Position) -> ok
```

Types:

```
   This = wxSplitterWindow()
   Position = integer()
```

```
setSashPosition(This, Position, Options :: [Option]) -> ok
```

Types:

```
   This = wxSplitterWindow()
   Position = integer()
   Option = {redraw, boolean()}
```

Sets the sash position.

Remark: Does not currently check for an out-of-range value.

See: `getSashPosition/1`

```
setMinimumPaneSize(This, PaneSize) -> ok
```

Types:

```
   This = wxSplitterWindow()
   PaneSize = integer()
```

Sets the minimum pane size.

Remark: The default minimum pane size is zero, which means that either pane can be reduced to zero by dragging the sash, thus removing one of the panes. To prevent this behaviour (and veto out-of-range sash dragging), set a minimum size, for example 20 pixels. If the wxSP_PERMIT_UNSPLIT style is used when a splitter window is created, the window may be unsplit even if minimum size is non-zero.

See: `getMinimumPaneSize/1`

```
setSplitMode(This, Mode) -> ok
```

Types:

```
   This = wxSplitterWindow()
   Mode = integer()
```

Sets the split mode.

Remark: Only sets the internal variable; does not update the display.

See: `getSplitMode/1`, `splitVertically/4`, `splitHorizontally/4`

```
splitHorizontally(This, Window1, Window2) -> boolean()
```

Types:

```
   This = wxSplitterWindow()
   Window1 = Window2 = wxWindow:wxWindow()
```

```
splitHorizontally(This, Window1, Window2, Options :: [Option]) ->
                     boolean()
```

Types:

```
   This = wxSplitterWindow()
   Window1 = Window2 = wxWindow:wxWindow()
   Option = {sashPosition, integer()}
```

Initializes the top and bottom panes of the splitter window.

The child windows are shown if they are currently hidden.

Return: true if successful, false otherwise (the window was already split).

Remark: This should be called if you wish to initially view two panes. It can also be called at any subsequent time, but the application should check that the window is not currently split using `isSplit/1`.

See: `splitVertically/4`, `isSplit/1`, `unsplit/2`

```
splitVertically(This, Window1, Window2) -> boolean()
```

Types:

```
   This = wxSplitterWindow()
   Window1 = Window2 = wxWindow:wxWindow()
```

```
splitVertically(This, Window1, Window2, Options :: [Option]) ->
                   boolean()
```

Types:

```
   This = wxSplitterWindow()
   Window1 = Window2 = wxWindow:wxWindow()
   Option = {sashPosition, integer()}
```

Initializes the left and right panes of the splitter window.

The child windows are shown if they are currently hidden.

Return: true if successful, false otherwise (the window was already split).

Remark: This should be called if you wish to initially view two panes. It can also be called at any subsequent time, but the application should check that the window is not currently split using `isSplit/1`.

See: `splitHorizontally/4`, `isSplit/1`, `unsplit/2`

```
unsplit(This) -> boolean()
```

Types:

```
   This = wxSplitterWindow()
```

```
unsplit(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxSplitterWindow()
Option = {toRemove, wxWindow:wxWindow()}
```

Unsplits the window.

Return: true if successful, false otherwise (the window was not split).

Remark: This call will not actually delete the pane being removed; it calls `OnUnsplit()` (not implemented in wx) which can be overridden for the desired behaviour. By default, the pane being removed is hidden.

See: `splitHorizontally/4`, `splitVertically/4`, `isSplit/1`, `OnUnsplit()` (not implemented in wx)

`updateSize(This) -> ok`

Types:

```
This = wxSplitterWindow()
```

Causes any pending sizing of the sash and child panes to take place immediately.

Such resizing normally takes place in idle time, in order to wait for layout to be completed. However, this can cause unacceptable flicker as the panes are resized after the window has been shown. To work around this, you can perform window layout (for example by sending a size event to the parent window), and then call this function, before showing the top-level window.

# wxStaticBitmap

Erlang module

A static bitmap control displays a bitmap. Native implementations on some platforms are only meant for display of the small icons in the dialog boxes.

If you want to display larger images portably, you may use generic implementation wxGenericStaticBitmap declared in <wx/generic/statbmpg.h>.

Notice that for the best results, the size of the control should be the same as the size of the image displayed in it, as happens by default if it's not resized explicitly. Otherwise, behaviour depends on the platform: under MSW, the bitmap is drawn centred inside the control, while elsewhere it is drawn at the origin of the control. You can use `SetScaleMode()` (not implemented in wx) to control how the image is scaled inside the control.

See: `wxBitmap`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxStaticBitmap**

## Data Types

```
wxStaticBitmap() = wx:wx_object()
```

## Exports

```
new() -> wxStaticBitmap()
```

Default constructor.

```
new(Parent, Id, Label) -> wxStaticBitmap()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Label = wxBitmap:wxBitmap()
```

```
new(Parent, Id, Label, Options :: [Option]) -> wxStaticBitmap()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Label = wxBitmap:wxBitmap()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Constructor, creating and showing a static bitmap control.

See: `create/5`

```
create(This, Parent, Id, Label) -> boolean()
```

Types:

```
   This = wxStaticBitmap()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = wxBitmap:wxBitmap()

create(This, Parent, Id, Label, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxStaticBitmap()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = wxBitmap:wxBitmap()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Creation function, for two-step construction.

For details see `new/4`.

```
getBitmap(This) -> wxBitmap:wxBitmap()
```

Types:

```
   This = wxStaticBitmap()
```

Returns the bitmap currently used in the control.

Notice that this method can be called even if `SetIcon()` (not implemented in wx) had been used.

See: `setBitmap/2`

```
setBitmap(This, Label) -> ok
```

Types:

```
   This = wxStaticBitmap()
   Label = wxBitmap:wxBitmap()
```

Sets the bitmap label.

See: `getBitmap/1`

```
destroy(This :: wxStaticBitmap()) -> ok
```

Destroys the object.

# wxStaticBox

Erlang module

A static box is a rectangle drawn around other windows to denote a logical grouping of items.

Note that while the previous versions required that windows appearing inside a static box be created as its siblings (i.e. use the same parent as the static box itself), since wxWidgets 2.9.1 it is also possible to create them as children of `wxStaticBox` itself and you are actually encouraged to do it like this if compatibility with the previous versions is not important.

So the new recommended way to create static box is:

While the compatible - and now deprecated - way is

Also note that there is a specialized `wxSizer` class (`wxStaticBoxSizer`) which can be used as an easier way to pack items into a static box.

See: `wxStaticText`, `wxStaticBoxSizer`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxStaticBox**

## Data Types

```
wxStaticBox() = wx:wx_object()
```

## Exports

```
new() -> wxStaticBox()
```

Default constructor.

```
new(Parent, Id, Label) -> wxStaticBox()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Label = unicode:chardata()
```

```
new(Parent, Id, Label, Options :: [Option]) -> wxStaticBox()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Label = unicode:chardata()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Constructor, creating and showing a static box.

See: `create/5`

```
destroy(This :: wxStaticBox()) -> ok
```

Constructor for a static box using the given window as label.

This constructor takes a pointer to an arbitrary window (although usually a `wxCheckBox` or a `wxRadioButton`) instead of just the usual text label and puts this window at the top of the box at the place where the label would be shown.

The `label` window must be a non-null, fully created window and will become a child of this `wxStaticBox`, i.e. it will be owned by this control and will be deleted when the `wxStaticBox` itself is deleted.

An example of creating a `wxStaticBox` with window as a label:

Currently this constructor is only available in wxGTK and wxMSW, use `wxHAS_WINDOW_LABEL_IN_STATIC_BOX` to check whether it can be used at compile-time.

Since: 3.1.1 Destructor, destroying the group box.

```
create(This, Parent, Id, Label) -> boolean()
```

Types:

```
   This = wxStaticBox()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
```

```
create(This, Parent, Id, Label, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxStaticBox()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Creates the static box for two-step construction.

See `new/4` for further details.

# wxStaticBoxSizer

Erlang module

`wxStaticBoxSizer` is a sizer derived from `wxBoxSizer` but adds a static box around the sizer.

The static box may be either created independently or the sizer may create it itself as a convenience. In any case, the sizer owns the `wxStaticBox` control and will delete it in the `wxStaticBoxSizer` destructor.

Note that since wxWidgets 2.9.1 you are encouraged to create the windows which are added to `wxStaticBoxSizer` as children of `wxStaticBox` itself, see this class documentation for more details.

Example of use of this class:

See: `wxSizer`, `wxStaticBox`, `wxBoxSizer`, **Overview sizer**

This class is derived (and can use functions) from: `wxBoxSizer` `wxSizer`

wxWidgets docs: **wxStaticBoxSizer**

## Data Types

```
wxStaticBoxSizer() = wx:wx_object()
```

## Exports

```
new(Orient, Parent) -> wxStaticBoxSizer()
new(Box, Orient) -> wxStaticBoxSizer()
```

Types:

```
Box = wxStaticBox:wxStaticBox()
Orient = integer()
```

This constructor uses an already existing static box.

```
new(Orient, Parent, Options :: [Option]) -> wxStaticBoxSizer()
```

Types:

```
Orient = integer()
Parent = wxWindow:wxWindow()
Option = {label, unicode:chardata()}
```

This constructor creates a new static box with the given label and parent window.

```
getStaticBox(This) -> wxStaticBox:wxStaticBox()
```

Types:

```
This = wxStaticBoxSizer()
```

Returns the static box associated with the sizer.

```
destroy(This :: wxStaticBoxSizer()) -> ok
```

Destroys the object.

# wxStaticLine

Erlang module

A static line is just a line which may be used in a dialog to separate the groups of controls.

The line may be only vertical or horizontal. Moreover, not all ports (notably not wxGTK) support specifying the transversal direction of the line (e.g. height for a horizontal line) so for maximal portability you should specify it as wxDefaultCoord.

Styles

This class supports the following styles:

See: `wxStaticBox`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxStaticLine**

## Data Types

```
wxStaticLine() = wx:wx_object()
```

## Exports

```
new() -> wxStaticLine()
```

Default constructor.

```
new(Parent) -> wxStaticLine()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxStaticLine()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructor, creating and showing a static line.

See: `create/3`

```
create(This, Parent) -> boolean()
```

Types:

```
   This = wxStaticLine()
   Parent = wxWindow:wxWindow()

create(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxStaticLine()
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Creates the static line for two-step construction.

See `new/2` for further details.

```
isVertical(This) -> boolean()
```

Types:

```
   This = wxStaticLine()
```

Returns true if the line is vertical, false if horizontal.

```
getDefaultSize() -> integer()
```

This static function returns the size which will be given to the smaller dimension of the static line, i.e. its height for a horizontal line or its width for a vertical one.

```
destroy(This :: wxStaticLine()) -> ok
```

Destroys the object.

# wxStaticText

Erlang module

A static text control displays one or more lines of read-only text. `wxStaticText` supports the three classic text alignments, label ellipsization i.e. replacing parts of the text with the ellipsis ("...") if the label doesn't fit into the provided space and also formatting markup with `wxControl::SetLabelMarkup()` (not implemented in wx).

Styles

This class supports the following styles:

See: `wxStaticBitmap`, `wxStaticBox`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxStaticText**

## Data Types

```
wxStaticText() = wx:wx_object()
```

## Exports

```
new() -> wxStaticText()
```

Default constructor.

```
new(Parent, Id, Label) -> wxStaticText()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
```

```
new(Parent, Id, Label, Options :: [Option]) -> wxStaticText()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructor, creating and showing a text control.

See: `create/5`

```
create(This, Parent, Id, Label) -> boolean()
```

Types:

```
   This = wxStaticText()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()

create(This, Parent, Id, Label, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxStaticText()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Creation function, for two-step construction.

For details see `new/4`.

```
getLabel(This) -> unicode:charlist()
```

Types:

```
   This = wxStaticText()
```

Returns the control's label, as it was passed to `wxControl:setLabel/2`.

Note that the returned string may contains mnemonics ("&" characters) if they were passed to the `wxControl:setLabel/2` function; use `GetLabelText()` (not implemented in wx) if they are undesired.

Also note that the returned string is always the string which was passed to `wxControl:setLabel/2` but may be different from the string passed to `SetLabelText()` (not implemented in wx) (since this last one escapes mnemonic characters).

```
setLabel(This, Label) -> ok
```

Types:

```
   This = wxStaticText()
   Label = unicode:chardata()
```

Change the label shown in the control.

Notice that since wxWidgets 3.1.1 this function is guaranteed not to do anything if the label didn't really change, so there is no benefit to checking if the new label is different from the current one in the application code.

See: `wxControl:setLabel/2`

```
wrap(This, Width) -> ok
```

Types:

```
   This = wxStaticText()
   Width = integer()
```

This functions wraps the controls label so that each of its lines becomes at most `width` pixels wide if possible (the lines are broken at words boundaries so it might not be the case if words are too long).

If `width` is negative, no wrapping is done. Note that this width is not necessarily the total width of the control, since a few pixels for the border (depending on the controls border style) may be added.

Since: 2.6.2

`destroy(This :: wxStaticText()) -> ok`

Destroys the object.

# wxStatusBar

Erlang module

A status bar is a narrow window that can be placed along the bottom of a frame to give small amounts of status information. It can contain one or more fields, one or more of which can be variable length according to the size of the window.

`wxStatusBar` also maintains an independent stack of status texts for each field (see `pushStatusText/3` and `popStatusText/2`).

Note that in `wxStatusBar` context, the terms `pane` and `field` are synonyms.

Styles

This class supports the following styles:

Remark: It is possible to create controls and other windows on the status bar. Position these windows from an OnSize() event handler.

Remark: Notice that only the first 127 characters of a string will be shown in status bar fields under Windows if a proper manifest indicating that the program uses version 6 of common controls library is not used. This is a limitation of the native control on these platforms.

See: `wxStatusBarPane` (not implemented in wx), `wxFrame`, **Examples**

This class is derived (and can use functions) from: `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxStatusBar**

## Data Types

```
wxStatusBar() = wx:wx_object()
```

## Exports

```
new() -> wxStatusBar()
```

Default ctor.

```
new(Parent) -> wxStatusBar()
```

Types:

```
Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxStatusBar()
```

Types:

```
Parent = wxWindow:wxWindow()
Option = {winid, integer()} | {style, integer()}
```

Constructor, creating the window.

See: `create/3`

```
destroy(This :: wxStatusBar()) -> ok
```

Destructor.

```
create(This, Parent) -> boolean()
```
Types:
```
   This = wxStatusBar()
   Parent = wxWindow:wxWindow()
```

```
create(This, Parent, Options :: [Option]) -> boolean()
```
Types:
```
   This = wxStatusBar()
   Parent = wxWindow:wxWindow()
   Option = {winid, integer()} | {style, integer()}
```
Creates the window, for two-step construction.

See `new/2` for details.

```
getFieldRect(This, I) -> Result
```
Types:
```
   Result =
       {Res :: boolean(),
        Rect ::
            {X :: integer(),
             Y :: integer(),
             W :: integer(),
             H :: integer()}}
   This = wxStatusBar()
   I = integer()
```
Returns the size and position of a field's internal bounding rectangle.

Return: true if the field index is valid, false otherwise.

See: {X,Y,W,H}

```
getFieldsCount(This) -> integer()
```
Types:
```
   This = wxStatusBar()
```
Returns the number of fields in the status bar.

```
getStatusText(This) -> unicode:charlist()
```
Types:
```
   This = wxStatusBar()
```

```
getStatusText(This, Options :: [Option]) -> unicode:charlist()
```
Types:
```
   This = wxStatusBar()
   Option = {number, integer()}
```
Returns the string associated with a status bar field.

Return: The status field string if the field is valid, otherwise the empty string.

See: setStatusText/3

```
popStatusText(This) -> ok
```

Types:

```
   This = wxStatusBar()
```

```
popStatusText(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxStatusBar()
   Option = {number, integer()}
```

Restores the text to the value it had before the last call to pushStatusText/3.

Notice that if setStatusText/3 had been called in the meanwhile, popStatusText/2 will not change the text, i.e. it does not override explicit changes to status text but only restores the saved text if it hadn't been changed since.

See: pushStatusText/3

```
pushStatusText(This, String) -> ok
```

Types:

```
   This = wxStatusBar()
   String = unicode:chardata()
```

```
pushStatusText(This, String, Options :: [Option]) -> ok
```

Types:

```
   This = wxStatusBar()
   String = unicode:chardata()
   Option = {number, integer()}
```

Saves the current field text in a per-field stack, and sets the field text to the string passed as argument.

See: popStatusText/2

```
setFieldsCount(This, Number) -> ok
```

Types:

```
   This = wxStatusBar()
   Number = integer()
```

```
setFieldsCount(This, Number, Options :: [Option]) -> ok
```

Types:

```
   This = wxStatusBar()
   Number = integer()
   Option = {widths, [integer()]}
```

Sets the number of fields, and optionally the field widths.

```
setMinHeight(This, Height) -> ok
```

Types:

```
This = wxStatusBar()
Height = integer()
```

Sets the minimal possible height for the status bar.

The real height may be bigger than the height specified here depending on the size of the font used by the status bar.

```
setStatusText(This, Text) -> ok
```

Types:

```
This = wxStatusBar()
Text = unicode:chardata()
```

```
setStatusText(This, Text, Options :: [Option]) -> ok
```

Types:

```
This = wxStatusBar()
Text = unicode:chardata()
Option = {number, integer()}
```

Sets the status text for the `i-th` field.

The given text will replace the current text. The display of the status bar is updated immediately, so there is no need to call `wxWindow:update/1` after calling this function.

Note that if `pushStatusText/3` had been called before the new text will also replace the last saved value to make sure that the next call to `popStatusText/2` doesn't restore the old value, which was overwritten by the call to this function.

See: `getStatusText/2`, `wxFrame:setStatusText/3`

```
setStatusWidths(This, Widths_field) -> ok
```

Types:

```
This = wxStatusBar()
Widths_field = [integer()]
```

Sets the widths of the fields in the status line.

There are two types of fields: `fixed` widths and `variable` width fields. For the fixed width fields you should specify their (constant) width in pixels. For the variable width fields, specify a negative number which indicates how the field should expand: the space left for all variable width fields is divided between them according to the absolute value of this number. A variable width field with width of -2 gets twice as much of it as a field with width -1 and so on.

For example, to create one fixed width field of width 100 in the right part of the status bar and two more fields which get 66% and 33% of the remaining space correspondingly, you should use an array containing -2, -1 and 100.

Remark: The widths of the variable fields are calculated from the total width of all fields, minus the sum of widths of the non-variable fields, divided by the number of variable fields.

See: `setFieldsCount/3`, `wxFrame:setStatusWidths/2`

```
setStatusStyles(This, Styles) -> ok
```

Types:

```
This = wxStatusBar()
Styles = [integer()]
```

Sets the styles of the fields in the status line which can make fields appear flat or raised instead of the standard sunken 3D border.

# wxStdDialogButtonSizer

Erlang module

This class creates button layouts which conform to the standard button spacing and ordering defined by the platform or toolkit's user interface guidelines (if such things exist). By using this class, you can ensure that all your standard dialogs look correct on all major platforms. Currently it conforms to the Windows, GTK+ and macOS human interface guidelines.

When there aren't interface guidelines defined for a particular platform or toolkit, `wxStdDialogButtonSizer` reverts to the Windows implementation.

To use this class, first add buttons to the sizer by calling `addButton/2` (or `setAffirmativeButton/2`, `setNegativeButton/2` or `setCancelButton/2`) and then call Realize in order to create the actual button layout used. Other than these special operations, this sizer works like any other sizer.

If you add a button with wxID_SAVE, on macOS the button will be renamed to "Save" and the wxID_NO button will be renamed to "Don't Save" in accordance with the macOS Human Interface Guidelines.

See: `wxSizer`, **Overview sizer**, `wxDialog:createButtonSizer/2`

This class is derived (and can use functions) from: `wxBoxSizer` `wxSizer`

wxWidgets docs: **wxStdDialogButtonSizer**

## Data Types

```
wxStdDialogButtonSizer() = wx:wx_object()
```

## Exports

```
new() -> wxStdDialogButtonSizer()
```

Constructor for a `wxStdDialogButtonSizer`.

```
addButton(This, Button) -> ok
```

Types:

```
This = wxStdDialogButtonSizer()
Button = wxButton:wxButton()
```

Adds a button to the `wxStdDialogButtonSizer`.

The `button` must have one of the following identifiers:

```
realize(This) -> ok
```

Types:

```
This = wxStdDialogButtonSizer()
```

Rearranges the buttons and applies proper spacing between buttons to make them match the platform or toolkit's interface guidelines.

```
setAffirmativeButton(This, Button) -> ok
```

Types:

```
This = wxStdDialogButtonSizer()
Button = wxButton:wxButton()
```

Sets the affirmative button for the sizer.

This allows you to use identifiers other than the standard identifiers outlined above.

```
setCancelButton(This, Button) -> ok
```

Types:

```
This = wxStdDialogButtonSizer()
Button = wxButton:wxButton()
```

Sets the cancel button for the sizer.

This allows you to use identifiers other than the standard identifiers outlined above.

```
setNegativeButton(This, Button) -> ok
```

Types:

```
This = wxStdDialogButtonSizer()
Button = wxButton:wxButton()
```

Sets the negative button for the sizer.

This allows you to use identifiers other than the standard identifiers outlined above.

```
destroy(This :: wxStdDialogButtonSizer()) -> ok
```

Destroys the object.

# wxStyledTextCtrl

Erlang module

A wxWidgets implementation of the Scintilla source code editing component.

As well as features found in standard text editing components, Scintilla includes features especially useful when editing and debugging source code. These include support for syntax styling, error indicators, code completion and call tips.

The selection margin can contain markers like those used in debuggers to indicate breakpoints and the current line. Styling choices are more open than with many editors, allowing the use of proportional fonts, bold and italics, multiple foreground and background colours and multiple fonts.

`wxStyledTextCtrl` is a 1 to 1 mapping of "raw" scintilla interface, whose documentation can be found in the Scintilla website (**http://www.scintilla.org/**).

Please see `wxStyledTextEvent` for the documentation of all event types you can use with `wxStyledTextCtrl`.

Index of the member groups

Links for quick access to the various categories of `wxStyledTextCtrl` functions:

See: `wxStyledTextEvent`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxStyledTextCtrl**

## Data Types

```
wxStyledTextCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxStyledTextCtrl()
```

Default ctor.

```
new(Parent) -> wxStyledTextCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxStyledTextCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Ctor.

```
destroy(This :: wxStyledTextCtrl()) -> ok
```

Destructor.

```
create(This, Parent) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
   Parent = wxWindow:wxWindow()
```

```
create(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Create the UI elements for a STC that was created with the default ctor.

(For 2-phase create.)

```
addText(This, Text) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Text = unicode:chardata()
```

Add text to the document at current position.

```
insertText(This, Pos, Text) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Pos = integer()
   Text = unicode:chardata()
```

Insert string at a position.

```
clearAll(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Delete all text in the document.

```
clearDocumentStyle(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Set all style bytes to 0, remove all folding information.

```
getLength(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Returns the number of bytes in the document.

```
getCharAt(This, Pos) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Pos = integer()
```

Returns the character byte at the position.

```
getCurrentPos(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Returns the position of the caret.

```
getAnchor(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Returns the position of the opposite end of the selection to the caret.

```
getStyleAt(This, Pos) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Pos = integer()
```

Returns the style byte at the position.

```
redo(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Redoes the next action on the undo history.

```
setUndoCollection(This, CollectUndo) -> ok
```

Types:

```
This = wxStyledTextCtrl()
CollectUndo = boolean()
```

Choose between collecting actions into the undo history and discarding them.

```
selectAll(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Select all the text in the document.

```
setSavePoint(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Remember the current position in the undo history as the position at which the document was saved.

```
canRedo(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Are there any redoable actions in the undo history?

```
markerLineFromHandle(This, MarkerHandle) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
MarkerHandle = integer()
```

Retrieve the line number at which a particular marker is located.

```
markerDeleteHandle(This, MarkerHandle) -> ok
```

Types:

```
This = wxStyledTextCtrl()
MarkerHandle = integer()
```

Delete a marker.

```
getUndoCollection(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Is undo history being collected?

```
getViewWhiteSpace(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Are white space characters currently visible? Returns one of wxSTC_WS_* constants.

```
setViewWhiteSpace(This, ViewWS) -> ok
```

Types:

```
This = wxStyledTextCtrl()
ViewWS = integer()
```

Make white space characters invisible, always visible or visible outside indentation.

The input should be one of the ?wxSTC_WS_* constants.

```
positionFromPoint(This, Pt) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Pt = {X :: integer(), Y :: integer()}
```

Find the position from a point within the window.

```
positionFromPointClose(This, X, Y) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   X = Y = integer()
```

Find the position from a point within the window but return wxSTC_INVALID_POSITION if not close to text.

```
gotoLine(This, Line) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Line = integer()
```

Set caret to start of a line and ensure it is visible.

```
gotoPos(This, Caret) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Caret = integer()
```

Set caret to a position and ensure it is visible.

```
setAnchor(This, Anchor) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Anchor = integer()
```

Set the selection anchor to a position.

The anchor is the opposite end of the selection from the caret.

```
getCurLine(This) -> Result
```

Types:

```
   Result = {Res :: unicode:charlist(), LinePos :: integer()}
   This = wxStyledTextCtrl()
```

Retrieve the text of the line containing the caret.

linePos can optionally be passed in to receive the index of the caret on the line.

```
getEndStyled(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the position of the last correctly styled character.

```
convertEOLs(This, EolMode) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   EolMode = integer()
```

Convert all line endings in the document to one mode.

```
getEOLMode(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the current end of line mode - one of wxSTC_EOL_CRLF, wxSTC_EOL_CR, or wxSTC_EOL_LF.

```
setEOLMode(This, EolMode) -> ok
```

Types:

```
This = wxStyledTextCtrl()
EolMode = integer()
```

Set the current end of line mode.

The input should be one of the ?wxSTC_EOL_* constants.

```
startStyling(This, Start) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Start = integer()
```

Set the current styling position to start.

```
setStyling(This, Length, Style) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Length = Style = integer()
```

Change style from current styling position for length characters to a style and move the current styling position to after this newly styled segment.

```
getBufferedDraw(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Is drawing done first into a buffer or direct to the screen?

```
setBufferedDraw(This, Buffered) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Buffered = boolean()
```

If drawing is buffered then each line of text is drawn into a bitmap buffer before drawing it to the screen to avoid flicker.

```
setTabWidth(This, TabWidth) -> ok
```

Types:

```
This = wxStyledTextCtrl()
TabWidth = integer()
```

Change the visible size of a tab to be a multiple of the width of a space character.

```
getTabWidth(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the visible size of a tab.

```
setCodePage(This, CodePage) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   CodePage = integer()
```

Set the code page used to interpret the bytes of the document as characters.

```
markerDefine(This, MarkerNumber, MarkerSymbol) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   MarkerNumber = MarkerSymbol = integer()
```

```
markerDefine(This, MarkerNumber, MarkerSymbol,
             Options :: [Option]) ->
                ok
```

Types:

```
   This = wxStyledTextCtrl()
   MarkerNumber = MarkerSymbol = integer()
   Option =
       {foreground, wx:wx_colour()} | {background, wx:wx_colour()}
```

Set the symbol used for a particular marker number, and optionally the fore and background colours.

The second argument should be one of the ?wxSTC_MARK_* constants.

```
markerSetForeground(This, MarkerNumber, Fore) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   MarkerNumber = integer()
   Fore = wx:wx_colour()
```

Set the foreground colour used for a particular marker number.

```
markerSetBackground(This, MarkerNumber, Back) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   MarkerNumber = integer()
   Back = wx:wx_colour()
```

Set the background colour used for a particular marker number.

```
markerAdd(This, Line, MarkerNumber) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Line = MarkerNumber = integer()
```

Add a marker to a line, returning an ID which can be used to find or delete the marker.

```
markerDelete(This, Line, MarkerNumber) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Line = MarkerNumber = integer()
```

Delete a marker from a line.

```
markerDeleteAll(This, MarkerNumber) -> ok
```

Types:

```
This = wxStyledTextCtrl()
MarkerNumber = integer()
```

Delete all markers with a particular number from all lines.

```
markerGet(This, Line) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Get a bit mask of all the markers set on a line.

```
markerNext(This, LineStart, MarkerMask) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
LineStart = MarkerMask = integer()
```

Find the next line at or after lineStart that includes a marker in mask.

Return -1 when no more lines.

```
markerPrevious(This, LineStart, MarkerMask) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
LineStart = MarkerMask = integer()
```

Find the previous line before lineStart that includes a marker in mask.

```
markerDefineBitmap(This, MarkerNumber, Bmp) -> ok
```

Types:

```
This = wxStyledTextCtrl()
MarkerNumber = integer()
Bmp = wxBitmap:wxBitmap()
```

Define a marker with a `wxBitmap`.

```
markerAddSet(This, Line, MarkerSet) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Line = MarkerSet = integer()
```

Add a set of markers to a line.

```
markerSetAlpha(This, MarkerNumber, Alpha) -> ok
```

Types:

```
This = wxStyledTextCtrl()
MarkerNumber = Alpha = integer()
```

Set the alpha used for a marker that is drawn in the text area, not the margin.

```
setMarginType(This, Margin, MarginType) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Margin = MarginType = integer()
```

Set a margin to be either numeric or symbolic.

The second argument should be one of the ?wxSTC_MARGIN_* constants.

```
getMarginType(This, Margin) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Margin = integer()
```

Retrieve the type of a margin.

The return value will be one of the ?wxSTC_MARGIN_* constants.

```
setMarginWidth(This, Margin, PixelWidth) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Margin = PixelWidth = integer()
```

Set the width of a margin to a width expressed in pixels.

```
getMarginWidth(This, Margin) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Margin = integer()
```

Retrieve the width of a margin in pixels.

```
setMarginMask(This, Margin, Mask) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Margin = Mask = integer()
```

Set a mask that determines which markers are displayed in a margin.

```
getMarginMask(This, Margin) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   Margin = integer()
```

Retrieve the marker mask of a margin.

```
setMarginSensitive(This, Margin, Sensitive) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Margin = integer()
   Sensitive = boolean()
```

Make a margin sensitive or insensitive to mouse clicks.

```
getMarginSensitive(This, Margin) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
   Margin = integer()
```

Retrieve the mouse click sensitivity of a margin.

```
styleClearAll(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Clear all the styles and make equivalent to the global default style.

```
styleSetForeground(This, Style, Fore) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Style = integer()
   Fore = wx:wx_colour()
```

Set the foreground colour of a style.

```
styleSetBackground(This, Style, Back) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Style = integer()
   Back = wx:wx_colour()
```

Set the background colour of a style.

```
styleSetBold(This, Style, Bold) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Style = integer()
   Bold = boolean()
```

Set a style to be bold or not.

```
styleSetItalic(This, Style, Italic) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Style = integer()
   Italic = boolean()
```

Set a style to be italic or not.

```
styleSetSize(This, Style, SizePoints) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Style = SizePoints = integer()
```

Set the size of characters of a style.

```
styleSetFaceName(This, Style, FontName) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Style = integer()
   FontName = unicode:chardata()
```

Set the font of a style.

```
styleSetEOLFilled(This, Style, EolFilled) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Style = integer()
   EolFilled = boolean()
```

Set a style to have its end of line filled or not.

```
styleResetDefault(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Reset the default style to its state at startup.

```
styleSetUnderline(This, Style, Underline) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Style = integer()
Underline = boolean()
```

Set a style to be underlined or not.

```
styleSetCase(This, Style, CaseVisible) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Style = CaseVisible = integer()
```

Set a style to be mixed case, or to force upper or lower case.

The second argument should be one of the ?wxSTC_CASE_* constants.

```
styleSetHotSpot(This, Style, Hotspot) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Style = integer()
Hotspot = boolean()
```

Set a style to be a hotspot or not.

```
setSelForeground(This, UseSetting, Fore) -> ok
```

Types:

```
This = wxStyledTextCtrl()
UseSetting = boolean()
Fore = wx:wx_colour()
```

Set the foreground colour of the main and additional selections and whether to use this setting.

```
setSelBackground(This, UseSetting, Back) -> ok
```

Types:

```
This = wxStyledTextCtrl()
UseSetting = boolean()
Back = wx:wx_colour()
```

Set the background colour of the main and additional selections and whether to use this setting.

```
getSelAlpha(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the alpha of the selection.

```
setSelAlpha(This, Alpha) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Alpha = integer()
```

Set the alpha of the selection.

```
setCaretForeground(This, Fore) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Fore = wx:wx_colour()
```

Set the foreground colour of the caret.

```
cmdKeyAssign(This, Key, Modifiers, Cmd) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Key = Modifiers = Cmd = integer()
```

When key+modifier combination keyDefinition is pressed perform sciCommand.

The second argument should be a bit list containing one or more of the ?wxSTC_KEYMOD_* constants and the third argument should be one of the ?wxSTC_CMD_* constants.

```
cmdKeyClear(This, Key, Modifiers) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Key = Modifiers = integer()
```

When key+modifier combination keyDefinition is pressed do nothing.

The second argument should be a bit list containing one or more of the ?wxSTC_KEYMOD_* constants.

```
cmdKeyClearAll(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Drop all key mappings.

```
setStyleBytes(This, Length) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Length = integer()
```

Set the styles for a segment of the document.

```
styleSetVisible(This, Style, Visible) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Style = integer()
Visible = boolean()
```

Set a style to be visible or not.

```
getCaretPeriod(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the time in milliseconds that the caret is on and off.

```
setCaretPeriod(This, PeriodMilliseconds) -> ok
```

Types:

```
This = wxStyledTextCtrl()
PeriodMilliseconds = integer()
```

Get the time in milliseconds that the caret is on and off.

0 = steady on.

```
setWordChars(This, Characters) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Characters = unicode:chardata()
```

Set the set of characters making up words for when moving or selecting by word.

First sets defaults like SetCharsDefault.

```
beginUndoAction(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Start a sequence of actions that is undone and redone as a unit.

May be nested.

```
endUndoAction(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

End a sequence of actions that is undone and redone as a unit.

```
indicatorSetStyle(This, Indicator, IndicatorStyle) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Indicator = IndicatorStyle = integer()
```

Set an indicator to plain, squiggle or TT.

The second argument should be one of the ?wxSTC_INDIC_* constants.

```
indicatorGetStyle(This, Indicator) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Indicator = integer()
```

Retrieve the style of an indicator.

The return value will be one of the ?wxSTC_INDIC_* constants.

```
indicatorSetForeground(This, Indicator, Fore) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Indicator = integer()
   Fore = wx:wx_colour()
```

Set the foreground colour of an indicator.

```
indicatorGetForeground(This, Indicator) -> wx:wx_colour4()
```

Types:

```
   This = wxStyledTextCtrl()
   Indicator = integer()
```

Retrieve the foreground colour of an indicator.

```
setWhitespaceForeground(This, UseSetting, Fore) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   UseSetting = boolean()
   Fore = wx:wx_colour()
```

Set the foreground colour of all whitespace and whether to use this setting.

```
setWhitespaceBackground(This, UseSetting, Back) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   UseSetting = boolean()
   Back = wx:wx_colour()
```

Set the background colour of all whitespace and whether to use this setting.

```
getStyleBits(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve number of bits in style bytes used to hold the lexical state.

Deprecated:

```
setLineState(This, Line, State) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Line = State = integer()
```

Used to hold extra styling information for each line.

```
getLineState(This, Line) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Retrieve the extra styling information for a line.

```
getMaxLineState(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the last line number that has line state.

```
getCaretLineVisible(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Is the background of the line containing the caret in a different colour?

```
setCaretLineVisible(This, Show) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Show = boolean()
```

Display the background of the line containing the caret in a different colour.

```
getCaretLineBackground(This) -> wx:wx_colour4()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the colour of the background of the line containing the caret.

```
setCaretLineBackground(This, Back) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Back = wx:wx_colour()
```

Set the colour of the background of the line containing the caret.

```
autoCompShow(This, LengthEntered, ItemList) -> ok
```

Types:

```
This = wxStyledTextCtrl()
LengthEntered = integer()
ItemList = unicode:chardata()
```

Display a auto-completion list.

The lengthEntered parameter indicates how many characters before the caret should be used to provide context.

```
autoCompCancel(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Remove the auto-completion list from the screen.

```
autoCompActive(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Is there an auto-completion list visible?

```
autoCompPosStart(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the position of the caret when the auto-completion list was displayed.

```
autoCompComplete(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

User has selected an item so remove the list and insert the selection.

```
autoCompStops(This, CharacterSet) -> ok
```

Types:

```
This = wxStyledTextCtrl()
CharacterSet = unicode:chardata()
```

Define a set of character that when typed cancel the auto-completion list.

```
autoCompSetSeparator(This, SeparatorCharacter) -> ok
```

Types:

```
This = wxStyledTextCtrl()
SeparatorCharacter = integer()
```

Change the separator character in the string setting up an auto-completion list.

Default is space but can be changed if items contain space.

```
autoCompGetSeparator(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the auto-completion list separator character.

```
autoCompSelect(This, Select) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Select = unicode:chardata()
```

Select the item in the auto-completion list that starts with a string.

```
autoCompSetCancelAtStart(This, Cancel) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Cancel = boolean()
```

Should the auto-completion list be cancelled if the user backspaces to a position before where the box was created.

```
autoCompGetCancelAtStart(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve whether auto-completion cancelled by backspacing before start.

```
autoCompSetFillUps(This, CharacterSet) -> ok
```

Types:

```
This = wxStyledTextCtrl()
CharacterSet = unicode:chardata()
```

Define a set of characters that when typed will cause the autocompletion to choose the selected item.

```
autoCompSetChooseSingle(This, ChooseSingle) -> ok
```

Types:

```
This = wxStyledTextCtrl()
ChooseSingle = boolean()
```

Should a single item auto-completion list automatically choose the item.

```
autoCompGetChooseSingle(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve whether a single item auto-completion list automatically choose the item.

```
autoCompSetIgnoreCase(This, IgnoreCase) -> ok
```

Types:

```
This = wxStyledTextCtrl()
IgnoreCase = boolean()
```

Set whether case is significant when performing auto-completion searches.

```
autoCompGetIgnoreCase(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve state of ignore case flag.

```
userListShow(This, ListType, ItemList) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   ListType = integer()
   ItemList = unicode:chardata()
```

Display a list of strings and send notification when user chooses one.

```
autoCompSetAutoHide(This, AutoHide) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   AutoHide = boolean()
```

Set whether or not autocompletion is hidden automatically when nothing matches.

```
autoCompGetAutoHide(This) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve whether or not autocompletion is hidden automatically when nothing matches.

```
autoCompSetDropRestOfWord(This, DropRestOfWord) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   DropRestOfWord = boolean()
```

Set whether or not autocompletion deletes any word characters after the inserted text upon completion.

```
autoCompGetDropRestOfWord(This) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve whether or not autocompletion deletes any word characters after the inserted text upon completion.

```
registerImage(This, Type, Bmp) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Type = integer()
   Bmp = wxBitmap:wxBitmap()
```

Register an image for use in autocompletion lists.

```
clearRegisteredImages(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Clear all the registered images.

```
autoCompGetTypeSeparator(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the auto-completion list type-separator character.

```
autoCompSetTypeSeparator(This, SeparatorCharacter) -> ok
```

Types:

```
This = wxStyledTextCtrl()
SeparatorCharacter = integer()
```

Change the type-separator character in the string setting up an auto-completion list.

Default is '?' but can be changed if items contain '?'.

```
autoCompSetMaxWidth(This, CharacterCount) -> ok
```

Types:

```
This = wxStyledTextCtrl()
CharacterCount = integer()
```

Set the maximum width, in characters, of auto-completion and user lists.

Set to 0 to autosize to fit longest item, which is the default.

```
autoCompGetMaxWidth(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the maximum width, in characters, of auto-completion and user lists.

```
autoCompSetMaxHeight(This, RowCount) -> ok
```

Types:

```
This = wxStyledTextCtrl()
RowCount = integer()
```

Set the maximum height, in rows, of auto-completion and user lists.

The default is 5 rows.

```
autoCompGetMaxHeight(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Set the maximum height, in rows, of auto-completion and user lists.

```
setIndent(This, IndentSize) -> ok
```

Types:

```
This = wxStyledTextCtrl()
IndentSize = integer()
```

Set the number of spaces used for one level of indentation.

```
getIndent(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve indentation size.

```
setUseTabs(This, UseTabs) -> ok
```

Types:

```
This = wxStyledTextCtrl()
UseTabs = boolean()
```

Indentation will only use space characters if useTabs is false, otherwise it will use a combination of tabs and spaces.

```
getUseTabs(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve whether tabs will be used in indentation.

```
setLineIndentation(This, Line, Indentation) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Line = Indentation = integer()
```

Change the indentation of a line to a number of columns.

```
getLineIndentation(This, Line) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Retrieve the number of columns that a line is indented.

```
getLineIndentPosition(This, Line) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Retrieve the position before the first non indentation character on a line.

```
getColumn(This, Pos) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Pos = integer()
```

Retrieve the column number of a position, taking tab width into account.

```
setUseHorizontalScrollBar(This, Visible) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Visible = boolean()
```

Show or hide the horizontal scroll bar.

```
getUseHorizontalScrollBar(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Is the horizontal scroll bar visible?

```
setIndentationGuides(This, IndentView) -> ok
```

Types:

```
This = wxStyledTextCtrl()
IndentView = integer()
```

Show or hide indentation guides.

The input should be one of the ?wxSTC_IV_* constants.

```
getIndentationGuides(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Are the indentation guides visible?

The return value will be one of the ?wxSTC_IV_* constants.

```
setHighlightGuide(This, Column) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Column = integer()
```

Set the highlighted indentation guide column.

0 = no highlighted guide.

```
getHighlightGuide(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the highlighted indentation guide column.

```
getLineEndPosition(This, Line) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Get the position after the last visible characters on a line.

```
getCodePage(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the code page used to interpret the bytes of the document as characters.

```
getCaretForeground(This) -> wx:wx_colour4()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the foreground colour of the caret.

```
getReadOnly(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

In read-only mode?

```
setCurrentPos(This, Caret) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Caret = integer()
```

Sets the position of the caret.

```
setSelectionStart(This, Anchor) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Anchor = integer()
```

Sets the position that starts the selection - this becomes the anchor.

```
getSelectionStart(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Returns the position at the start of the selection.

```
setSelectionEnd(This, Caret) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Caret = integer()
```

Sets the position that ends the selection - this becomes the caret.

```
getSelectionEnd(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Returns the position at the end of the selection.

```
setPrintMagnification(This, Magnification) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Magnification = integer()
```

Sets the print magnification added to the point size of each style for printing.

```
getPrintMagnification(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Returns the print magnification.

```
setPrintColourMode(This, Mode) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Mode = integer()
```

Modify colours when printing for clearer printed text.

The input should be one of the ?wxSTC_PRINT_* constants.

```
getPrintColourMode(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Returns the print colour mode.

The return value will be one of the ?wxSTC_PRINT_* constants.

```
findText(This, MinPos, MaxPos, Text) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   MinPos = MaxPos = integer()
   Text = unicode:chardata()
```

```
findText(This, MinPos, MaxPos, Text, Options :: [Option]) ->
            integer()
```

Types:

```
   This = wxStyledTextCtrl()
   MinPos = MaxPos = integer()
   Text = unicode:chardata()
   Option = {flags, integer()}
```

`Find some text in the document. @param minPos The position (starting from zero) in the document at which to begin the search @param maxPos The last position (starting from zero) in the document to which the search will be restricted. @param text The text to search for. @param flags (Optional) The search flags. This should be a bit list containing one or more of the @link wxStyledTextCtrl::wxSTC_FIND_WHOLEWORD wxSTC_FIND_* @endlink constants.`

Return: The position (starting from zero) in the document at which the text was found or wxSTC_INVALID_POSITION if the search fails.

Remark: A backwards search can be performed by setting minPos to be greater than maxPos.

```
formatRange(This, DoDraw, StartPos, EndPos, Draw, Target,
            RenderRect, PageRect) ->
```

```
               integer()
```

Types:

```
   This = wxStyledTextCtrl()
   DoDraw = boolean()
   StartPos = EndPos = integer()
   Draw = Target = wxDC:wxDC()
   RenderRect = PageRect =
       {X :: integer(),
        Y :: integer(),
        W :: integer(),
        H :: integer()}
```

On Windows, will draw the document into a display context such as a printer.

```
getFirstVisibleLine(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the display line at the top of the display.

```
getLine(This, Line) -> unicode:charlist()
```

Types:

```
   This = wxStyledTextCtrl()
   Line = integer()
```

Retrieve the contents of a line.

```
getLineCount(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Returns the number of lines in the document.

There is always at least one.

```
setMarginLeft(This, PixelWidth) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   PixelWidth = integer()
```

Sets the size in pixels of the left margin.

```
getMarginLeft(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Returns the size in pixels of the left margin.

```
setMarginRight(This, PixelWidth) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   PixelWidth = integer()
```

Sets the size in pixels of the right margin.

```
getMarginRight(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Returns the size in pixels of the right margin.

```
getModify(This) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
```

Is the document different from when it was last saved?

```
setSelection(This, From, To) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   From = To = integer()
```

Selects the text starting at the first position up to (but not including) the character at the last position.

If both parameters are equal to -1 all text in the control is selected.

Notice that the insertion point will be moved to `from` by this function.

See: `selectAll/1`

```
getSelectedText(This) -> unicode:charlist()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the selected text.

```
getTextRange(This, StartPos, EndPos) -> unicode:charlist()
```

Types:

```
   This = wxStyledTextCtrl()
   StartPos = EndPos = integer()
```

Retrieve a range of text.

```
hideSelection(This, Hide) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Hide = boolean()
```

Draw the selection in normal style or with selection highlighted.

```
lineFromPosition(This, Pos) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   Pos = integer()
```

Retrieve the line containing a position.

```
positionFromLine(This, Line) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   Line = integer()
```

Retrieve the position at the start of a line.

```
lineScroll(This, Columns, Lines) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Columns = Lines = integer()
```

Scroll horizontally and vertically.

```
ensureCaretVisible(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Ensure the caret is visible.

```
replaceSelection(This, Text) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Text = unicode:chardata()
```

Replace the selected text with the argument text.

```
setReadOnly(This, ReadOnly) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   ReadOnly = boolean()
```

Set to read only or read write.

```
canPaste(This) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
```

Will a paste succeed?

```
canUndo(This) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
```

Are there any undoable actions in the undo history?

```
emptyUndoBuffer(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Delete the undo history.

```
undo(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Undo one action in the undo history.

```
cut(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Cut the selection to the clipboard.

```
copy(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Copy the selection to the clipboard.

```
paste(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Paste the contents of the clipboard into the document replacing the selection.

```
clear(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Clear the selection.

```
setText(This, Text) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Text = unicode:chardata()
```

Replace the contents of the document with the argument text.

```
getText(This) -> unicode:charlist()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve all the text in the document.

```
getTextLength(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the number of characters in the document.

```
getOvertype(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Returns true if overtype mode is active otherwise false is returned.

```
setCaretWidth(This, PixelWidth) -> ok
```

Types:

```
This = wxStyledTextCtrl()
PixelWidth = integer()
```

Set the width of the insert mode caret.

```
getCaretWidth(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Returns the width of the insert mode caret.

```
setTargetStart(This, Start) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Start = integer()
```

Sets the position that starts the target which is used for updating the document without affecting the scroll position.

```
getTargetStart(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the position that starts the target.

```
setTargetEnd(This, End) -> ok
```

Types:

```
This = wxStyledTextCtrl()
End = integer()
```

Sets the position that ends the target which is used for updating the document without affecting the scroll position.

```
getTargetEnd(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the position that ends the target.

```
replaceTarget(This, Text) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   Text = unicode:chardata()
```

Replace the target text with the argument text.

Text is counted so it can contain NULs. Returns the length of the replacement text.

```
searchInTarget(This, Text) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   Text = unicode:chardata()
```

Search for a counted string in the target and set the target to the found range.

Text is counted so it can contain NULs. Returns length of range or -1 for failure in which case target is not moved.

```
setSearchFlags(This, SearchFlags) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   SearchFlags = integer()
```

Set the search flags used by SearchInTarget.

The input should be a bit list containing one or more of the ?wxSTC_FIND_* constants.

```
getSearchFlags(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Get the search flags used by SearchInTarget.

The return value will be a bit list containing one or more of the ?wxSTC_FIND_* constants.

```
callTipShow(This, Pos, Definition) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Pos = integer()
   Definition = unicode:chardata()
```

Show a call tip containing a definition near position pos.

```
callTipCancel(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Remove the call tip from the screen.

```
callTipActive(This) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
```

Is there an active call tip?

```
callTipPosAtStart(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the position where the caret was before displaying the call tip.

Since: 3.1.0

```
callTipSetHighlight(This, HighlightStart, HighlightEnd) -> ok
```

Types:

```
This = wxStyledTextCtrl()
HighlightStart = HighlightEnd = integer()
```

Highlight a segment of the definition.

```
callTipSetBackground(This, Back) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Back = wx:wx_colour()
```

Set the background colour for the call tip.

```
callTipSetForeground(This, Fore) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Fore = wx:wx_colour()
```

Set the foreground colour for the call tip.

```
callTipSetForegroundHighlight(This, Fore) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Fore = wx:wx_colour()
```

Set the foreground colour for the highlighted part of the call tip.

```
callTipUseStyle(This, TabSize) -> ok
```

Types:

```
This = wxStyledTextCtrl()
TabSize = integer()
```

Enable use of wxSTC_STYLE_CALLTIP and set call tip tab size in pixels.

```
visibleFromDocLine(This, DocLine) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
DocLine = integer()
```

Find the display line of a document line taking hidden lines into account.

```
docLineFromVisible(This, DisplayLine) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
DisplayLine = integer()
```

Find the document line of a display line taking hidden lines into account.

```
wrapCount(This, DocLine) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
DocLine = integer()
```

The number of display lines needed to wrap a document line.

```
setFoldLevel(This, Line, Level) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Line = Level = integer()
```

Set the fold level of a line.

This encodes an integer level along with flags indicating whether the line is a header and whether it is effectively white space.

```
getFoldLevel(This, Line) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Retrieve the fold level of a line.

```
getLastChild(This, Line, Level) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Line = Level = integer()
```

Find the last child line of a header line.

```
getFoldParent(This, Line) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Find the parent line of a child line.

```
showLines(This, LineStart, LineEnd) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   LineStart = LineEnd = integer()
```

Make a range of lines visible.

```
hideLines(This, LineStart, LineEnd) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   LineStart = LineEnd = integer()
```

Make a range of lines invisible.

```
getLineVisible(This, Line) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
   Line = integer()
```

Is a line visible?

```
setFoldExpanded(This, Line, Expanded) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Line = integer()
   Expanded = boolean()
```

Show the children of a header line.

```
getFoldExpanded(This, Line) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
   Line = integer()
```

Is a header line expanded?

```
toggleFold(This, Line) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Line = integer()
```

Switch a header line between expanded and contracted.

```
ensureVisible(This, Line) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Line = integer()
```

Ensure a particular line is visible by expanding any header line hiding it.

```
setFoldFlags(This, Flags) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Flags = integer()
```

Set some style options for folding.

The second argument should be a bit list containing one or more of the ?wxSTC_FOLDFLAG_* constants.

```
ensureVisibleEnforcePolicy(This, Line) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Ensure a particular line is visible by expanding any header line hiding it.

Use the currently set visibility policy to determine which range to display.

```
setTabIndents(This, TabIndents) -> ok
```

Types:

```
This = wxStyledTextCtrl()
TabIndents = boolean()
```

Sets whether a tab pressed when caret is within indentation indents.

```
getTabIndents(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Does a tab pressed when caret is within indentation indent?

```
setBackSpaceUnIndents(This, BsUnIndents) -> ok
```

Types:

```
This = wxStyledTextCtrl()
BsUnIndents = boolean()
```

Sets whether a backspace pressed when caret is within indentation unindents.

```
getBackSpaceUnIndents(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Does a backspace pressed when caret is within indentation unindent?

```
setMouseDwellTime(This, PeriodMilliseconds) -> ok
```

Types:

```
This = wxStyledTextCtrl()
PeriodMilliseconds = integer()
```

Sets the time the mouse must sit still to generate a mouse dwell event.

The input should be a time in milliseconds or wxSTC_TIME_FOREVER.

```
getMouseDwellTime(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the time the mouse must sit still to generate a mouse dwell event.

The return value will be a time in milliseconds or wxSTC_TIME_FOREVER.

```
wordStartPosition(This, Pos, OnlyWordCharacters) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   Pos = integer()
   OnlyWordCharacters = boolean()
```

Get position of start of word.

```
wordEndPosition(This, Pos, OnlyWordCharacters) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   Pos = integer()
   OnlyWordCharacters = boolean()
```

Get position of end of word.

```
setWrapMode(This, WrapMode) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   WrapMode = integer()
```

Sets whether text is word wrapped.

The input should be one of the ?wxSTC_WRAP_* constants.

```
getWrapMode(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve whether text is word wrapped.

The return value will be one of the ?wxSTC_WRAP_* constants.

```
setWrapVisualFlags(This, WrapVisualFlags) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   WrapVisualFlags = integer()
```

Set the display mode of visual flags for wrapped lines.

The input should be a bit list containing one or more of the ?wxSTC_WRAPVISUALFLAG_* constants.

```
getWrapVisualFlags(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the display mode of visual flags for wrapped lines.

The return value will be a bit list containing one or more of the ?wxSTC_WRAPVISUALFLAG_* constants.

```
setWrapVisualFlagsLocation(This, WrapVisualFlagsLocation) -> ok
```

Types:

```
This = wxStyledTextCtrl()
WrapVisualFlagsLocation = integer()
```

Set the location of visual flags for wrapped lines.

The input should be a bit list containing one or more of the ?wxSTC_WRAPVISUALFLAGLOC_* constants.

```
getWrapVisualFlagsLocation(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the location of visual flags for wrapped lines.

The return value will be a bit list containing one or more of the ?wxSTC_WRAPVISUALFLAGLOC_* constants.

```
setWrapStartIndent(This, Indent) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Indent = integer()
```

Set the start indent for wrapped lines.

```
getWrapStartIndent(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the start indent for wrapped lines.

```
setLayoutCache(This, CacheMode) -> ok
```

Types:

```
This = wxStyledTextCtrl()
CacheMode = integer()
```

Sets the degree of caching of layout information.

The input should be one of the ?wxSTC_CACHE_* constants.

```
getLayoutCache(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the degree of caching of layout information.

The return value will be one of the ?wxSTC_CACHE_* constants.

```
setScrollWidth(This, PixelWidth) -> ok
```

Types:

```
This = wxStyledTextCtrl()
PixelWidth = integer()
```

Sets the document width assumed for scrolling.

```
getScrollWidth(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the document width assumed for scrolling.

```
textWidth(This, Style, Text) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Style = integer()
Text = unicode:chardata()
```

Measure the pixel width of some text in a particular style.

Does not handle tab or control characters.

```
getEndAtLastLine(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve whether the maximum scroll position has the last line at the bottom of the view.

```
textHeight(This, Line) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Retrieve the height of a particular line of text in pixels.

```
setUseVerticalScrollBar(This, Visible) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Visible = boolean()
```

Show or hide the vertical scroll bar.

```
getUseVerticalScrollBar(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Is the vertical scroll bar visible?

```
appendText(This, Text) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Text = unicode:chardata()
```

Append a string to the end of the document without changing the selection.

```
getTwoPhaseDraw(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Is drawing done in two phases with backgrounds drawn before foregrounds?

```
setTwoPhaseDraw(This, TwoPhase) -> ok
```

Types:

```
This = wxStyledTextCtrl()
TwoPhase = boolean()
```

In twoPhaseDraw mode, drawing is performed in two phases, first the background and then the foreground.

This avoids chopping off characters that overlap the next run.

```
targetFromSelection(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Make the target range start and end be the same as the selection range start and end.

```
linesJoin(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Join the lines in the target.

```
linesSplit(This, PixelWidth) -> ok
```

Types:

```
This = wxStyledTextCtrl()
PixelWidth = integer()
```

Split the lines in the target into lines that are less wide than pixelWidth where possible.

```
setFoldMarginColour(This, UseSetting, Back) -> ok
```

Types:

```
This = wxStyledTextCtrl()
UseSetting = boolean()
Back = wx:wx_colour()
```

Set one of the colours used as a chequerboard pattern in the fold margin.

```
setFoldMarginHiColour(This, UseSetting, Fore) -> ok
```

Types:

```
This = wxStyledTextCtrl()
UseSetting = boolean()
Fore = wx:wx_colour()
```

Set the other colour used as a chequerboard pattern in the fold margin.

`lineDown(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret down one line.

`lineDownExtend(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret down one line extending selection to new caret position.

`lineUp(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret up one line.

`lineUpExtend(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret up one line extending selection to new caret position.

`charLeft(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret left one character.

`charLeftExtend(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret left one character extending selection to new caret position.

`charRight(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret right one character.

`charRightExtend(This) -> ok`

Types:

```
   This = wxStyledTextCtrl()
```

Move caret right one character extending selection to new caret position.

```
wordLeft(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Move caret left one word.

```
wordLeftExtend(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Move caret left one word extending selection to new caret position.

```
wordRight(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Move caret right one word.

```
wordRightExtend(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Move caret right one word extending selection to new caret position.

```
home(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Move caret to first position on line.

```
homeExtend(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Move caret to first position on line extending selection to new caret position.

```
lineEnd(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Move caret to last position on line.

```
lineEndExtend(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Move caret to last position on line extending selection to new caret position.

```
documentStart(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to first position in document.

```
documentStartExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to first position in document extending selection to new caret position.

```
documentEnd(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to last position in document.

```
documentEndExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to last position in document extending selection to new caret position.

```
pageUp(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret one page up.

```
pageUpExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret one page up extending selection to new caret position.

```
pageDown(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret one page down.

```
pageDownExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret one page down extending selection to new caret position.

```
editToggleOvertype(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Switch from insert to overtype mode or the reverse.

```
cancel(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Cancel any modes such as call tip or auto-completion list display.

```
deleteBack(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Delete the selection or if no selection, the character before the caret.

```
tab(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

If selection is empty or all on one line replace the selection with a tab character.

If more than one line selected, indent the lines.

```
backTab(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Dedent the selected lines.

```
newLine(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Insert a new line, may use a CRLF, CR or LF depending on EOL mode.

```
formFeed(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Insert a Form Feed character.

```
vCHome(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to before first visible character on line.

If already there move to first character on line.

```
vCHomeExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Like VCHome but extending selection to new caret position.

`zoomIn(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Magnify the displayed text by increasing the sizes by 1 point.

`zoomOut(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Make the displayed text smaller by decreasing the sizes by 1 point.

`delWordLeft(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Delete the word to the left of the caret.

`delWordRight(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Delete the word to the right of the caret.

`lineCut(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Cut the line containing the caret.

`lineDelete(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Delete the line containing the caret.

`lineTranspose(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Switch the current line with the previous.

`lineDuplicate(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Duplicate the current line.

```
lowerCase(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Transform the selection to lower case.

```
upperCase(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Transform the selection to upper case.

```
lineScrollDown(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Scroll the document down, keeping the caret visible.

```
lineScrollUp(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Scroll the document up, keeping the caret visible.

```
deleteBackNotLine(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Delete the selection or if no selection, the character before the caret.

Will not delete the character before at the start of a line.

```
homeDisplay(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to first position on display line.

```
homeDisplayExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to first position on display line extending selection to new caret position.

```
lineEndDisplay(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to last position on display line.

```
lineEndDisplayExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to last position on display line extending selection to new caret position.

### homeWrapExtend(This) -> ok

Types:

```
This = wxStyledTextCtrl()
```

Like HomeExtend but when word-wrap is enabled extends first to start of display line HomeDisplayExtend, then to start of document line HomeExtend.

### lineEndWrap(This) -> ok

Types:

```
This = wxStyledTextCtrl()
```

Like LineEnd but when word-wrap is enabled goes first to end of display line LineEndDisplay, then to start of document line LineEnd.

### lineEndWrapExtend(This) -> ok

Types:

```
This = wxStyledTextCtrl()
```

Like LineEndExtend but when word-wrap is enabled extends first to end of display line LineEndDisplayExtend, then to start of document line LineEndExtend.

### vCHomeWrap(This) -> ok

Types:

```
This = wxStyledTextCtrl()
```

Like VCHome but when word-wrap is enabled goes first to start of display line VCHomeDisplay, then behaves like VCHome.

### vCHomeWrapExtend(This) -> ok

Types:

```
This = wxStyledTextCtrl()
```

Like VCHomeExtend but when word-wrap is enabled extends first to start of display line VCHomeDisplayExtend, then behaves like VCHomeExtend.

### lineCopy(This) -> ok

Types:

```
This = wxStyledTextCtrl()
```

Copy the line containing the caret.

### moveCaretInsideView(This) -> ok

Types:

```
This = wxStyledTextCtrl()
```

Move the caret inside current view if it's not there already.

```
lineLength(This, Line) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   Line = integer()
```

How many characters are on a line, including end of line characters?

```
braceHighlight(This, PosA, PosB) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   PosA = PosB = integer()
```

Highlight the characters at two positions.

```
braceBadLight(This, Pos) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Pos = integer()
```

Highlight the character at a position indicating there is no matching brace.

```
braceMatch(This, Pos) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   Pos = integer()
```

Find the position of a matching brace or wxSTC_INVALID_POSITION if no match.

```
getViewEOL(This) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
```

Are the end of line characters visible?

```
setViewEOL(This, Visible) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Visible = boolean()
```

Make the end of line characters visible or invisible.

```
setModEventMask(This, EventMask) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   EventMask = integer()
```

Set which document modification events are sent to the container.

The input should be a bit list containing one or more of the ?wxSTC_MOD_* constants, the ?wxSTC_PERFORMED_* constants, wxSTC_STARTACTION, wxSTC_MULTILINEUNDOREDO, wxSTC_MULTISTEPUNDOREDO, and

wxSTC_LASTSTEPINUNDOREDO. The input can also be wxSTC_MODEVENTMASKALL to indicate that all changes should generate events.

```
getEdgeColumn(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the column number which text should be kept within.

```
setEdgeColumn(This, Column) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Column = integer()
```

Set the column number of the edge.

If text goes past the edge then it is highlighted.

```
setEdgeMode(This, EdgeMode) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   EdgeMode = integer()
```

The edge may be displayed by a line (wxSTC_EDGE_LINE/wxSTC_EDGE_MULTILINE) or by highlighting text that goes beyond it (wxSTC_EDGE_BACKGROUND) or not displayed at all (wxSTC_EDGE_NONE).

The input should be one of the ?wxSTC_EDGE_* constants.

```
getEdgeMode(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the edge highlight mode.

The return value will be one of the ?wxSTC_EDGE_* constants.

```
getEdgeColour(This) -> wx:wx_colour4()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the colour used in edge indication.

```
setEdgeColour(This, EdgeColour) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   EdgeColour = wx:wx_colour()
```

Change the colour used in edge indication.

```
searchAnchor(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Sets the current caret position to be the search anchor.

```
searchNext(This, SearchFlags, Text) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
SearchFlags = integer()
Text = unicode:chardata()
```

Find some text starting at the search anchor.

Does not ensure the selection is visible.

```
searchPrev(This, SearchFlags, Text) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
SearchFlags = integer()
Text = unicode:chardata()
```

Find some text starting at the search anchor and moving backwards.

Does not ensure the selection is visible.

```
linesOnScreen(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieves the number of lines completely visible.

```
usePopUp(This, PopUpMode) -> ok
```

Types:

```
This = wxStyledTextCtrl()
PopUpMode = integer()
```

Set whether a pop up menu is displayed automatically when the user presses the wrong mouse button on certain areas.

The input should be one of the ?wxSTC_POPUP_* constants.

Remark: When `wxContextMenuEvent` is used to create a custom popup menu, this function should be called with wxSTC_POPUP_NEVER. Otherwise the default menu will be shown instead of the custom one.

```
selectionIsRectangle(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Is the selection rectangular? The alternative is the more common stream selection.

```
setZoom(This, ZoomInPoints) -> ok
```

Types:

```
This = wxStyledTextCtrl()
ZoomInPoints = integer()
```

Set the zoom level.

This number of points is added to the size of all fonts. It may be positive to magnify or negative to reduce.

```
getZoom(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the zoom level.

```
getModEventMask(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get which document modification events are sent to the container.

The return value will wxSTC_MODEVENTMASKALL if all changes generate events. Otherwise it will be a bit list containing one or more of the ?wxSTC_MOD_* constants, the ?wxSTC_PERFORMED_* constants, wxSTC_STARTACTION, wxSTC_MULTILINEUNDOREDO, wxSTC_MULTISTEPUNDOREDO, and wxSTC_LASTSTEPINUNDOREDO.

```
setSTCFocus(This, Focus) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Focus = boolean()
```

Change internal focus flag.

```
getSTCFocus(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Get internal focus flag.

```
setStatus(This, Status) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Status = integer()
```

Change error status - 0 = OK.

The input should be one of the ?wxSTC_STATUS_* constants.

```
getStatus(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get error status.

The return value will be one of the ?wxSTC_STATUS_* constants.

```
setMouseDownCaptures(This, Captures) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Captures = boolean()
```

Set whether the mouse is captured when its button is pressed.

```
getMouseDownCaptures(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Get whether mouse gets captured.

```
setSTCCursor(This, CursorType) -> ok
```

Types:

```
This = wxStyledTextCtrl()
CursorType = integer()
```

Sets the cursor to one of the wxSTC_CURSOR* values.

```
getSTCCursor(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get cursor type.

The return value will be one of the ?wxSTC_CURSOR* constants.

```
setControlCharSymbol(This, Symbol) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Symbol = integer()
```

Change the way control characters are displayed: If symbol is < 32, keep the drawn way, else, use the given character.

```
getControlCharSymbol(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the way control characters are displayed.

```
wordPartLeft(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move to the previous change in capitalisation.

```
wordPartLeftExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move to the previous change in capitalisation extending selection to new caret position.

```
wordPartRight(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move to the change next in capitalisation.

```
wordPartRightExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move to the next change in capitalisation extending selection to new caret position.

```
setVisiblePolicy(This, VisiblePolicy, VisibleSlop) -> ok
```

Types:

```
This = wxStyledTextCtrl()
VisiblePolicy = VisibleSlop = integer()
```

Set the way the display area is determined when a particular line is to be moved to by Find, FindNext, GotoLine, etc.

The first argument should be a bit list containing one or more of the ?wxSTC_VISIBLE_* constants.

```
delLineLeft(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Delete back from the current position to the start of the line.

```
delLineRight(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Delete forwards from the current position to the end of the line.

```
getXOffset(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get the xOffset (ie, horizontal scroll position).

```
chooseCaretX(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Set the last x chosen value to be the caret x position.

```
setXCaretPolicy(This, CaretPolicy, CaretSlop) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   CaretPolicy = CaretSlop = integer()
```

Set the way the caret is kept visible when going sideways.

The exclusion zone is given in pixels.

The first argument should be a bit list containing one or more of the ?wxSTC_CARET_* constants.

```
setYCaretPolicy(This, CaretPolicy, CaretSlop) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   CaretPolicy = CaretSlop = integer()
```

Set the way the line the caret is on is kept visible.

The exclusion zone is given in lines.

The first argument should be a bit list containing one or more of the ?wxSTC_CARET_* constants.

```
getPrintWrapMode(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Is printing line wrapped?

The return value will be one of the ?wxSTC_WRAP_* constants.

```
setHotspotActiveForeground(This, UseSetting, Fore) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   UseSetting = boolean()
   Fore = wx:wx_colour()
```

Set a fore colour for active hotspots.

```
setHotspotActiveBackground(This, UseSetting, Back) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   UseSetting = boolean()
   Back = wx:wx_colour()
```

Set a back colour for active hotspots.

```
setHotspotActiveUnderline(This, Underline) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Underline = boolean()
```

Enable / Disable underlining active hotspots.

```
setHotspotSingleLine(This, SingleLine) -> ok
```

Types:

```
This = wxStyledTextCtrl()
SingleLine = boolean()
```

Limit hotspots to single line so hotspots on two lines don't merge.

```
paraDownExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Extend selection down one paragraph (delimited by empty lines).

```
paraUp(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret up one paragraph (delimited by empty lines).

```
paraUpExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Extend selection up one paragraph (delimited by empty lines).

```
positionBefore(This, Pos) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Pos = integer()
```

Given a valid document position, return the previous position taking code page into account.

Returns 0 if passed 0.

```
positionAfter(This, Pos) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
Pos = integer()
```

Given a valid document position, return the next position taking code page into account.

Maximum value returned is the last position in the document.

```
copyRange(This, Start, End) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Start = End = integer()
```

Copy a range of text to the clipboard.

Positions are clipped into the document.

```
copyText(This, Length, Text) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Length = integer()
Text = unicode:chardata()
```

Copy argument text to the clipboard.

`setSelectionMode(This, SelectionMode) -> ok`

Types:

```
This = wxStyledTextCtrl()
SelectionMode = integer()
```

Set the selection mode to stream (wxSTC_SEL_STREAM) or rectangular (wxSTC_SEL_RECTANGLE/ wxSTC_SEL_THIN) or by lines (wxSTC_SEL_LINES).

`getSelectionMode(This) -> integer()`

Types:

```
This = wxStyledTextCtrl()
```

Get the mode of the current selection.

The return value will be one of the ?wxSTC_SEL_* constants.

`lineDownRectExtend(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret down one line, extending rectangular selection to new caret position.

`lineUpRectExtend(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret up one line, extending rectangular selection to new caret position.

`charLeftRectExtend(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret left one character, extending rectangular selection to new caret position.

`charRightRectExtend(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret right one character, extending rectangular selection to new caret position.

`homeRectExtend(This) -> ok`

Types:

```
This = wxStyledTextCtrl()
```

Move caret to first position on line, extending rectangular selection to new caret position.

```
vCHomeRectExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to before first visible character on line.

If already there move to first character on line. In either case, extend rectangular selection to new caret position.

```
lineEndRectExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to last position on line, extending rectangular selection to new caret position.

```
pageUpRectExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret one page up, extending rectangular selection to new caret position.

```
pageDownRectExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret one page down, extending rectangular selection to new caret position.

```
stutteredPageUp(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to top of page, or one page up if already at top of page.

```
stutteredPageUpExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to top of page, or one page up if already at top of page, extending selection to new caret position.

```
stutteredPageDown(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to bottom of page, or one page down if already at bottom of page.

```
stutteredPageDownExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret to bottom of page, or one page down if already at bottom of page, extending selection to new caret position.

```
wordLeftEnd(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret left one word, position cursor at end of word.

```
wordLeftEndExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret left one word, position cursor at end of word, extending selection to new caret position.

```
wordRightEnd(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret right one word, position cursor at end of word.

```
wordRightEndExtend(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Move caret right one word, position cursor at end of word, extending selection to new caret position.

```
setWhitespaceChars(This, Characters) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Characters = unicode:chardata()
```

Set the set of characters making up whitespace for when moving or selecting by word.

Should be called after SetWordChars.

```
setCharsDefault(This) -> ok
```

Types:

```
This = wxStyledTextCtrl()
```

Reset the set of characters for whitespace and word characters to the defaults.

```
autoCompGetCurrent(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Get currently selected item position in the auto-completion list.

```
allocate(This, Bytes) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Bytes = integer()
```

Enlarge the document to a particular size of text bytes.

```
findColumn(This, Line, Column) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
   Line = Column = integer()
```

Find the position of a column on a line taking into account tabs and multi-byte characters.

If beyond end of line, return line end position.

```
getCaretSticky(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Can the caret preferred x position only be changed by explicit movement commands?

The return value will be one of the ?wxSTC_CARETSTICKY_* constants.

```
setCaretSticky(This, UseCaretStickyBehaviour) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   UseCaretStickyBehaviour = integer()
```

Stop the caret preferred x position changing when the user types.

The input should be one of the ?wxSTC_CARETSTICKY_* constants.

```
toggleCaretSticky(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Switch between sticky and non-sticky: meant to be bound to a key.

```
setPasteConvertEndings(This, Convert) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Convert = boolean()
```

Enable/Disable convert-on-paste for line endings.

```
getPasteConvertEndings(This) -> boolean()
```

Types:

```
   This = wxStyledTextCtrl()
```

Get convert-on-paste setting.

```
selectionDuplicate(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Duplicate the selection.

If selection empty duplicate the line containing the caret.

```
setCaretLineBackAlpha(This, Alpha) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Alpha = integer()
```

Set background alpha of the caret line.

```
getCaretLineBackAlpha(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Get the background alpha of the caret line.

```
startRecord(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Start notifying the container of all key presses and commands.

```
stopRecord(This) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
```

Stop notifying the container of all key presses and commands.

```
setLexer(This, Lexer) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Lexer = integer()
```

Set the lexing language of the document.

The input should be one of the ?wxSTC_LEX_* constants.

```
getLexer(This) -> integer()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the lexing language of the document.

The return value will be one of the ?wxSTC_LEX_* constants.

```
colourise(This, Start, End) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Start = End = integer()
```

Colourise a segment of the document using the current lexing language.

```
setProperty(This, Key, Value) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Key = Value = unicode:chardata()
```

Set up a value that may be used by a lexer for some optional feature.

```
setKeyWords(This, KeyWordSet, KeyWords) -> ok
```

Types:

```
This = wxStyledTextCtrl()
KeyWordSet = integer()
KeyWords = unicode:chardata()
```

Set up the key words used by the lexer.

```
setLexerLanguage(This, Language) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Language = unicode:chardata()
```

Set the lexing language of the document based on string name.

```
getProperty(This, Key) -> unicode:charlist()
```

Types:

```
This = wxStyledTextCtrl()
Key = unicode:chardata()
```

Retrieve a "property" value previously set with SetProperty.

```
getStyleBitsNeeded(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Retrieve the number of bits the current lexer needs for styling.

Deprecated:

```
getCurrentLine(This) -> integer()
```

Types:

```
This = wxStyledTextCtrl()
```

Returns the line number of the line with the caret.

```
styleSetSpec(This, StyleNum, Spec) -> ok
```

Types:

```
This = wxStyledTextCtrl()
StyleNum = integer()
Spec = unicode:chardata()
```

Extract style settings from a spec-string which is composed of one or more of the following comma separated elements:

bold turns on bold italic turns on italics fore:[name or #RRGGBB] sets the foreground colour back:[name or #RRGGBB] sets the background colour face:[facename] sets the font face name to use size:[num] sets the font size in points eol turns on eol filling underline turns on underlining

```
styleSetFont(This, StyleNum, Font) -> ok
```

Types:

```
This = wxStyledTextCtrl()
StyleNum = integer()
Font = wxFont:wxFont()
```

Set style size, face, bold, italic, and underline attributes from a `wxFont`'s attributes.

```
styleSetFontAttr(This, StyleNum, Size, FaceName, Bold, Italic,
                 Underline) ->
                    ok
```

Types:

```
This = wxStyledTextCtrl()
StyleNum = Size = integer()
FaceName = unicode:chardata()
Bold = Italic = Underline = boolean()
```

```
styleSetFontAttr(This, StyleNum, Size, FaceName, Bold, Italic,
                 Underline,
                 Options :: [Option]) ->
                    ok
```

Types:

```
This = wxStyledTextCtrl()
StyleNum = Size = integer()
FaceName = unicode:chardata()
Bold = Italic = Underline = boolean()
Option = {encoding, wx:wx_enum()}
```

Set all font style attributes at once.

```
styleSetCharacterSet(This, Style, CharacterSet) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Style = CharacterSet = integer()
```

Set the character set of the font in a style.

Converts the Scintilla character set values to a wxFontEncoding.

```
styleSetFontEncoding(This, Style, Encoding) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Style = integer()
Encoding = wx:wx_enum()
```

Set the font encoding to be used by a style.

```
cmdKeyExecute(This, Cmd) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Cmd = integer()
```

Perform one of the operations defined by the wxSTC_CMD_* constants.

```
setMargins(This, Left, Right) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Left = Right = integer()
```

Set the left and right margin in the edit area, measured in pixels.

```
getSelection(This) -> {From :: integer(), To :: integer()}
```

Types:

```
This = wxStyledTextCtrl()
```

Gets the current selection span.

If the returned values are equal, there was no selection. Please note that the indices returned may be used with the other `wxTextCtrl` methods but don't necessarily represent the correct indices into the string returned by `wxComboBox:getValue/1` for multiline controls under Windows (at least,) you should use `wxTextCtrl:getStringSelection/1` to get the selected text.

```
pointFromPosition(This, Pos) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxStyledTextCtrl()
Pos = integer()
```

Retrieve the point in the window where a position is displayed.

```
scrollToLine(This, Line) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Scroll enough to make the given line visible.

```
scrollToColumn(This, Column) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Column = integer()
```

Scroll enough to make the given column visible.

```
setVScrollBar(This, Bar) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Bar = wxScrollBar:wxScrollBar()
```

Set the vertical scrollbar to use instead of the one that's built-in.

```
setHScrollBar(This, Bar) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Bar = wxScrollBar:wxScrollBar()
```

Set the horizontal scrollbar to use instead of the one that's built-in.

```
getLastKeydownProcessed(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Can be used to prevent the EVT_CHAR handler from adding the char.

```
setLastKeydownProcessed(This, Val) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Val = boolean()
```

Returns the line number of the line with the caret.

```
saveFile(This, Filename) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
Filename = unicode:chardata()
```

Write the contents of the editor to filename.

```
loadFile(This, Filename) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
Filename = unicode:chardata()
```

Load the contents of filename into the editor.

```
doDragOver(This, X, Y, DefaultRes) -> wx:wx_enum()
```

Types:

```
This = wxStyledTextCtrl()
X = Y = integer()
DefaultRes = wx:wx_enum()
```

Allow for simulating a DnD DragOver.

```
doDropText(This, X, Y, Data) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
X = Y = integer()
Data = unicode:chardata()
```

Allow for simulating a DnD DropText.

```
getUseAntiAliasing(This) -> boolean()
```

Types:

```
This = wxStyledTextCtrl()
```

Returns the current UseAntiAliasing setting.

```
addTextRaw(This, Text) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Text = binary()
```

```
addTextRaw(This, Text, Options :: [Option]) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Text = binary()
Option = {length, integer()}
```

Add text to the document at current position.

```
insertTextRaw(This, Pos, Text) -> ok
```

Types:

```
This = wxStyledTextCtrl()
Pos = integer()
Text = binary()
```

Insert string at a position.

```
getCurLineRaw(This) -> Result
```

Types:

```
Result = {Res :: binary(), LinePos :: integer()}
This = wxStyledTextCtrl()
```

Retrieve the text of the line containing the caret.

Returns the index of the caret on the line.

```
getLineRaw(This, Line) -> binary()
```

Types:

```
This = wxStyledTextCtrl()
Line = integer()
```

Retrieve the contents of a line.

```
getSelectedTextRaw(This) -> binary()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve the selected text.

```
getTextRangeRaw(This, StartPos, EndPos) -> binary()
```

Types:

```
   This = wxStyledTextCtrl()
   StartPos = EndPos = integer()
```

Retrieve a range of text.

```
setTextRaw(This, Text) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Text = binary()
```

Replace the contents of the document with the argument text.

```
getTextRaw(This) -> binary()
```

Types:

```
   This = wxStyledTextCtrl()
```

Retrieve all the text in the document.

```
appendTextRaw(This, Text) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Text = binary()
```

```
appendTextRaw(This, Text, Options :: [Option]) -> ok
```

Types:

```
   This = wxStyledTextCtrl()
   Text = binary()
   Option = {length, integer()}
```

Append a string to the end of the document without changing the selection.

# wxStyledTextEvent

Erlang module

The type of events sent from `wxStyledTextCtrl`.

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxStyledTextEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxStyledTextEventType` to subscribe to events of this type.

## Data Types

```
wxStyledTextEvent() = wx:wx_object()
wxStyledText() =
    #wxStyledText{type =
                      wxStyledTextEvent:wxStyledTextEventType(),
                  position = integer(),
                  key = integer(),
                  modifiers = integer(),
                  modificationType = integer(),
                  text = unicode:chardata(),
                  length = integer(),
                  linesAdded = integer(),
                  line = integer(),
                  foldLevelNow = integer(),
                  foldLevelPrev = integer(),
                  margin = integer(),
                  message = integer(),
                  wParam = integer(),
                  lParam = integer(),
                  listType = integer(),
                  x = integer(),
                  y = integer(),
                  dragText = unicode:chardata(),
                  dragAllowMove = boolean(),
                  dragResult = wx:wx_enum()}
wxStyledTextEventType() =
    stc_autocomp_cancelled | stc_autocomp_char_deleted |
    stc_autocomp_selection | stc_calltip_click | stc_change |
    stc_charadded | stc_do_drop | stc_doubleclick |
    stc_drag_over | stc_dwellend | stc_dwellstart |
    stc_hotspot_click | stc_hotspot_dclick |
    stc_hotspot_release_click | stc_indicator_click |
    stc_indicator_release | stc_macrorecord | stc_marginclick |
    stc_modified | stc_needshown | stc_painted |
    stc_romodifyattempt | stc_savepointleft |
    stc_savepointreached | stc_start_drag | stc_styleneeded |
```

```
    stc_updateui | stc_userlistselection | stc_zoom
```

## Exports

`getPosition(This) -> integer()`

Types:

```
   This = wxStyledTextEvent()
```

Returns the zero-based text position associated this event.

This method is valid for the following event types:

`getKey(This) -> integer()`

Types:

```
   This = wxStyledTextEvent()
```

Returns the key code of the key that generated this event.

This method is valid for the following event types:

`getModifiers(This) -> integer()`

Types:

```
   This = wxStyledTextEvent()
```

Returns the modifiers of the key press or mouse click for this event.

The returned value is a bit list that may contain one or more of the following values:

In addition, the value can be checked for equality with ?wxSTC_KEYMOD_NORM to test if no modifiers are present.

This method is valid for the following event types:

`getModificationType(This) -> integer()`

Types:

```
   This = wxStyledTextEvent()
```

Returns the modification type for this event.

The modification type is a bit list that describes the change that generated this event. It may contain one or more of the following values:

This method is valid for `wxEVT_STC_MODIFIED` events.

`getText(This) -> unicode:charlist()`

Types:

```
   This = wxStyledTextEvent()
```

Deprecated: Use `wxCommandEvent:getString/1` instead.

`getLength(This) -> integer()`

Types:

```
   This = wxStyledTextEvent()
```

Returns the length (number of characters) of this event.

This method is valid for `wxEVT_STC_MODIFIED` and `wxEVT_STC_NEEDSHOWN` events.

```
getLinesAdded(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns the number of lines added or deleted with this event.

This method is valid for `wxEVT_STC_MODIFIED` events when the result of `getModificationType/1` includes ?wxSTC_MOD_INSERTTEXT or ?wxSTC_MOD_DELETETEXT.

```
getLine(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns zero-based line number for this event.

This method is valid for `wxEVT_STC_DOUBLECLICK` and `wxEVT_STC_MODIFIED` events.

```
getFoldLevelNow(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns the current fold level for the line.

This method is valid for `wxEVT_STC_MODIFIED` events when the result of `getModificationType/1` includes ?wxSTC_MOD_CHANGEFOLD.

```
getFoldLevelPrev(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns previous fold level for the line.

This method is valid for `wxEVT_STC_MODIFIED` events when the result of `getModificationType/1` includes ?wxSTC_MOD_CHANGEFOLD.

```
getMargin(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns the zero-based index of the margin that generated this event.

This method is valid for `wxEVT_STC_MARGINCLICK` and `wxEVT_STC_MARGIN_RIGHT_CLICK` events.

```
getMessage(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns a message number while a macro is being recorded.

Many of the `wxStyledTextCtrl` methods such as `wxStyledTextCtrl:insertText/3` and `wxStyledTextCtrl:paste/1` have an event number associated with them. This method returns that number while a macro is being recorded so that the macro can be played back later.

This method is valid for `wxEVT_STC_MACRORECORD` events.

```
getWParam(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns value of the WParam field for this event.

This method is valid for `wxEVT_STC_MACRORECORD` events.

```
getLParam(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns the value of the LParam field for this event.

This method is valid for `wxEVT_STC_MACRORECORD` events.

```
getListType(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns the list type for this event.

The list type is an integer passed to a list when it is created with the `wxStyledTextCtrl:userListShow/3` method and can be used to distinguish lists if more than one is used.

This method is valid for `wxEVT_STC_AUTOCOMP_SELECTION_CHANGE` and `wxEVT_STC_USERLISTSELECTION` events.

```
getX(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns the X coordinate of the mouse for this event.

This method is valid for the following event types:

```
getY(This) -> integer()
```

Types:

```
This = wxStyledTextEvent()
```

Returns the Y coordinate of the mouse for this event.

This method is valid for the following event types:

```
getDragText(This) -> unicode:charlist()
```

Types:

```
This = wxStyledTextEvent()
```

Deprecated: Use `wxCommandEvent:getString/1` instead.

```
getDragAllowMove(This) -> boolean()
```

Types:

```
This = wxStyledTextEvent()
```

```
getDragResult(This) -> wx:wx_enum()
```

Types:

```
This = wxStyledTextEvent()
```

Returns drag result for this event.

This method is valid for `wxEVT_STC_DRAG_OVER` and `wxEVT_STC_DO_DROP` events.

```
getShift(This) -> boolean()
```

Types:

```
This = wxStyledTextEvent()
```

Returns true if the Shift key is pressed.

This method is valid for the following event types:

```
getControl(This) -> boolean()
```

Types:

```
This = wxStyledTextEvent()
```

Returns true if the Control key is pressed.

This method is valid for the following event types:

```
getAlt(This) -> boolean()
```

Types:

```
This = wxStyledTextEvent()
```

Returns true if the Alt key is pressed.

This method is valid for the following event types:

# wxSysColourChangedEvent

Erlang module

This class is used for system colour change events, which are generated when the user changes the colour settings using the control panel. This is only appropriate under Windows.

Remark: The default event handler for this event propagates the event to child windows, since Windows only sends the events to top-level windows. If intercepting this event for a top-level window, remember to call the base class handler, or to pass the event on to the window's children explicitly.

See: **Overview events**

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxSysColourChangedEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxSysColourChangedEventType` to subscribe to events of this type.

## Data Types

```
wxSysColourChangedEvent() = wx:wx_object()
wxSysColourChanged() =
    #wxSysColourChanged{type =
                           wxSysColourChangedEvent:wxSysColourChangedEventType()}
wxSysColourChangedEventType() = sys_colour_changed
```

# wxSystemOptions

Erlang module

`wxSystemOptions` stores option/value pairs that wxWidgets itself or applications can use to alter behaviour at run-time. It can be used to optimize behaviour that doesn't deserve a distinct API, but is still important to be able to configure.

System options can be set by the program itself using `setOption/2` method and they also can be set from the program environment by defining an environment variable `wx_option` to set the given option for all wxWidgets applications or `wx_appname_option` to set it just for the application with the given name (as returned by `wxApp::GetAppName()` (not implemented in wx)). Notice that any characters not allowed in the environment variables names, such as periods and dashes, should be replaced with underscores. E.g. to define a system option "foo-bar" you need to define the environment variable "wx_foo_bar".

The program may use system options for its own needs but they are mostly used to control the behaviour of wxWidgets library itself.

These options are currently recognised by wxWidgets:

All platforms

Windows

GTK+

Mac

Motif

The compile-time option to include or exclude this functionality is wxUSE_SYSTEM_OPTIONS.

See: `wxSystemSettings`

wxWidgets docs: **wxSystemOptions**

## Data Types

```
wxSystemOptions() = wx:wx_object()
```

## Exports

```
getOption(Name) -> unicode:charlist()
```

Types:

```
Name = unicode:chardata()
```

Gets an option.

The function is case-insensitive to `name`. Returns empty string if the option hasn't been set.

See: `setOption/2`, `getOptionInt/1`, `hasOption/1`

```
getOptionInt(Name) -> integer()
```

Types:

```
Name = unicode:chardata()
```

Gets an option as an integer.

The function is case-insensitive to `name`. If the option hasn't been set, this function returns 0.

See: `setOption/2`, `getOption/1`, `hasOption/1`

```
hasOption(Name) -> boolean()
```

Types:

```
Name = unicode:chardata()
```

Returns true if the given option is present.

The function is case-insensitive to `name`.

See: `setOption/2`, `getOption/1`, `getOptionInt/1`

```
isFalse(Name) -> boolean()
```

Types:

```
Name = unicode:chardata()
```

Returns true if the option with the given `name` had been set to 0 value.

This is mostly useful for boolean options for which you can't use `GetOptionInt(name)` == 0 as this would also be true if the option hadn't been set at all.

```
setOption(Name, Value) -> ok
setOption(Name, Value) -> ok
```

Types:

```
Name = Value = unicode:chardata()
```

Sets an option.

The function is case-insensitive to `name`.

# wxSystemSettings

Erlang module

`wxSystemSettings` allows the application to ask for details about the system.

This can include settings such as standard colours, fonts, and user interface element sizes.

See: `wxFont`, `wx_color()`, `wxSystemOptions`

wxWidgets docs: **wxSystemSettings**

## Data Types

```
wxSystemSettings() = wx:wx_object()
```

## Exports

```
getColour(Index) -> wx:wx_colour4()
```

Types:

```
Index = wx:wx_enum()
```

Returns a system colour.

Return: The returned colour is always valid.

```
getFont(Index) -> wxFont:wxFont()
```

Types:

```
Index = wx:wx_enum()
```

Returns a system font.

Return: The returned font is always valid.

```
getMetric(Index) -> integer()
```

Types:

```
Index = wx:wx_enum()
```

```
getMetric(Index, Options :: [Option]) -> integer()
```

Types:

```
Index = wx:wx_enum()
Option = {win, wxWindow:wxWindow()}
```

Returns the value of a system metric, or -1 if the metric is not supported on the current system.

The value of `win` determines if the metric returned is a global value or a `wxWindow` based value, in which case it might determine the widget, the display the window is on, or something similar. The window given should be as close to the metric as possible (e.g. a `wxTopLevelWindow` in case of the wxSYS_CAPTION_Y metric).

`index` can be one of the ?wxSystemMetric enum values.

`win` is a pointer to the window for which the metric is requested. Specifying the `win` parameter is encouraged, because some metrics on some ports are not supported without one,or they might be capable of reporting better values if given one. If a window does not make sense for a metric, one should still be given, as for example it might determine which displays cursor width is requested with wxSYS_CURSOR_X.

```
getScreenType() -> wx:wx_enum()
```

Returns the screen type.

The return value is one of the ?wxSystemScreenType enum values.

# wxTaskBarIcon

Erlang module

This class represents a taskbar icon. A taskbar icon is an icon that appears in the 'system tray' and responds to mouse clicks, optionally with a tooltip above it to help provide information.

X Window System Note

Under X Window System, the window manager must support either the "System Tray Protocol" (see **http://freedesktop.org/wiki/Specifications/systemtray-spec**) by freedesktop.org (WMs used by modern desktop environments such as GNOME >= 2, KDE >= 3 and XFCE >= 4 all do) or the older methods used in GNOME 1.2 and KDE 1 and 2.

If it doesn't, the icon will appear as a toplevel window on user's desktop. Because not all window managers have system tray, there's no guarantee that `wxTaskBarIcon` will work correctly under X Window System and so the applications should use it only as an optional component of their user interface. The user should be required to explicitly enable the taskbar icon on Unix, it shouldn't be on by default.

This class is derived (and can use functions) from: `wxEvtHandler`

wxWidgets docs: **wxTaskBarIcon**

## Events

Event types emitted from this class: `taskbar_move`, `taskbar_left_down`, `taskbar_left_up`, `taskbar_right_down`, `taskbar_right_up`, `taskbar_left_dclick`, `taskbar_right_dclick`

## Data Types

```
wxTaskBarIcon() = wx:wx_object()
```

## Exports

```
new(Options :: [Option]) -> wxTaskBarIcon()
```

Types:

```
   Option =
       {iconType, wx:wx_enum()} |
       {createPopupMenu, fun(() -> wxMenu:wxMenu())}
```

Default constructor.

The iconType is only applicable on wxOSX/Cocoa.

```
destroy(This :: wxTaskBarIcon()) -> ok
```

Destroys the `wxTaskBarIcon` object, removing the icon if not already removed.

```
popupMenu(This, Menu) -> boolean()
```

Types:

```
   This = wxTaskBarIcon()
   Menu = wxMenu:wxMenu()
```

Pops up a menu at the current mouse position.

The events can be handled by a class derived from `wxTaskBarIcon`.

Note: It is recommended to override `CreatePopupMenu()` (not implemented in wx) callback instead of calling this method from event handler, because some ports (e.g. wxCocoa) may not implement `popupMenu/2` and mouse click events at all.

```
removeIcon(This) -> boolean()
```

Types:

```
   This = wxTaskBarIcon()
```

Removes the icon previously set with `setIcon/3`.

```
setIcon(This, Icon) -> boolean()
```

Types:

```
   This = wxTaskBarIcon()
   Icon = wxIcon:wxIcon()
```

```
setIcon(This, Icon, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxTaskBarIcon()
   Icon = wxIcon:wxIcon()
   Option = {tooltip, unicode:chardata()}
```

Sets the icon, and optional tooltip text.

# wxTaskBarIconEvent

Erlang module

The event class used by `wxTaskBarIcon`. For a list of the event macros meant to be used with `wxTaskBarIconEvent`, please look at `wxTaskBarIcon` description.

This class is derived (and can use functions) from: `wxEvent`

wxWidgets docs: **wxTaskBarIconEvent**

## Data Types

```
wxTaskBarIconEvent() = wx:wx_object()
wxTaskBarIcon() =
    #wxTaskBarIcon{type =
                       wxTaskBarIconEvent:wxTaskBarIconEventType()}
wxTaskBarIconEventType() =
    taskbar_move | taskbar_left_down | taskbar_left_up |
    taskbar_right_down | taskbar_right_up | taskbar_left_dclick |
    taskbar_right_dclick
```

# wxTextAttr

Erlang module

wxTextAttr represents the character and paragraph attributes, or style, for a range of text in a wxTextCtrl or wxRichTextCtrl (not implemented in wx).

When setting up a wxTextAttr object, pass a bitlist mask to setFlags/2 to indicate which style elements should be changed. As a convenience, when you call a setter such as SetFont, the relevant bit will be set.

See: wxTextCtrl, wxRichTextCtrl (not implemented in wx)

wxWidgets docs: **wxTextAttr**

## Data Types

```
wxTextAttr() = wx:wx_object()
```

## Exports

```
new() -> wxTextAttr()
```

Constructors.

```
new(ColText) -> wxTextAttr()
new(Attr) -> wxTextAttr()
```

Types:

```
   Attr = wxTextAttr()
```

```
new(ColText, Options :: [Option]) -> wxTextAttr()
```

Types:

```
   ColText = wx:wx_colour()
   Option =
       {colBack, wx:wx_colour()} |
       {font, wxFont:wxFont()} |
       {alignment, wx:wx_enum()}
```

```
getAlignment(This) -> wx:wx_enum()
```

Types:

```
   This = wxTextAttr()
```

Returns the alignment flags.

See ?wxTextAttrAlignment for a list of available styles.

```
getBackgroundColour(This) -> wx:wx_colour4()
```

Types:

```
   This = wxTextAttr()
```

Returns the background colour.

```
getFont(This) -> wxFont:wxFont()
```

Types:

```
This = wxTextAttr()
```

Creates and returns a font specified by the font attributes in the `wxTextAttr` object.

Note that `wxTextAttr` does not store a `wxFont` object, so this is only a temporary font.

For greater efficiency, access the font attributes directly.

```
getFontEncoding(This) -> wx:wx_enum()
```

Types:

```
This = wxTextAttr()
```

Returns the font encoding.

```
getFontFaceName(This) -> unicode:charlist()
```

Types:

```
This = wxTextAttr()
```

Returns the font face name.

```
getFontSize(This) -> integer()
```

Types:

```
This = wxTextAttr()
```

Returns the font size in points.

```
getFontStyle(This) -> wx:wx_enum()
```

Types:

```
This = wxTextAttr()
```

Returns the font style.

```
getFontUnderlined(This) -> boolean()
```

Types:

```
This = wxTextAttr()
```

Returns true if the font is underlined.

```
getFontWeight(This) -> wx:wx_enum()
```

Types:

```
This = wxTextAttr()
```

Returns the font weight.

```
getLeftIndent(This) -> integer()
```

Types:

```
This = wxTextAttr()
```

Returns the left indent in tenths of a millimetre.

```
getLeftSubIndent(This) -> integer()
```

Types:

```
This = wxTextAttr()
```

Returns the left sub-indent in tenths of a millimetre.

```
getRightIndent(This) -> integer()
```

Types:

```
This = wxTextAttr()
```

Returns the right indent in tenths of a millimeter.

```
getTabs(This) -> [integer()]
```

Types:

```
This = wxTextAttr()
```

Returns an array of tab stops, each expressed in tenths of a millimeter.

Each stop is measured from the left margin and therefore each value must be larger than the last.

```
getTextColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxTextAttr()
```

Returns the text foreground colour.

```
hasBackgroundColour(This) -> boolean()
```

Types:

```
This = wxTextAttr()
```

Returns true if the attribute object specifies a background colour.

```
hasFont(This) -> boolean()
```

Types:

```
This = wxTextAttr()
```

Returns true if the attribute object specifies any font attributes.

```
hasTextColour(This) -> boolean()
```

Types:

```
This = wxTextAttr()
```

Returns true if the attribute object specifies a text foreground colour.

```
getFlags(This) -> integer()
```

Types:

```
This = wxTextAttr()
```

Returns flags indicating which attributes are applicable.

See `setFlags/2` for a list of available flags.

```
isDefault(This) -> boolean()
```

Types:

```
This = wxTextAttr()
```

Returns false if we have any attributes set, true otherwise.

```
setAlignment(This, Alignment) -> ok
```

Types:

```
This = wxTextAttr()
Alignment = wx:wx_enum()
```

Sets the paragraph alignment.

See ?wxTextAttrAlignment enumeration values.

Of these, wxTEXT_ALIGNMENT_JUSTIFIED is unimplemented. In future justification may be supported when printing or previewing, only.

```
setBackgroundColour(This, ColBack) -> ok
```

Types:

```
This = wxTextAttr()
ColBack = wx:wx_colour()
```

Sets the background colour.

```
setFlags(This, Flags) -> ok
```

Types:

```
This = wxTextAttr()
Flags = integer()
```

Sets the flags determining which styles are being specified.

The ?wxTextAttrFlags values can be passed in a bitlist.

```
setFont(This, Font) -> ok
```

Types:

```
This = wxTextAttr()
Font = wxFont:wxFont()
```

```
setFont(This, Font, Options :: [Option]) -> ok
```

Types:

```
This = wxTextAttr()
Font = wxFont:wxFont()
Option = {flags, integer()}
```

Sets the attributes for the given font.

Note that `wxTextAttr` does not store an actual `wxFont` object.

```
setFontEncoding(This, Encoding) -> ok
```

Types:

```
   This = wxTextAttr()
   Encoding = wx:wx_enum()
```

Sets the font encoding.

```
setFontFaceName(This, FaceName) -> ok
```

Types:

```
   This = wxTextAttr()
   FaceName = unicode:chardata()
```

Sets the font face name.

```
setFontFamily(This, Family) -> ok
```

Types:

```
   This = wxTextAttr()
   Family = wx:wx_enum()
```

Sets the font family.

```
setFontSize(This, PointSize) -> ok
```

Types:

```
   This = wxTextAttr()
   PointSize = integer()
```

Sets the font size in points.

```
setFontPointSize(This, PointSize) -> ok
```

Types:

```
   This = wxTextAttr()
   PointSize = integer()
```

Sets the font size in points.

```
setFontPixelSize(This, PixelSize) -> ok
```

Types:

```
   This = wxTextAttr()
   PixelSize = integer()
```

Sets the font size in pixels.

```
setFontStyle(This, FontStyle) -> ok
```

Types:

```
   This = wxTextAttr()
   FontStyle = wx:wx_enum()
```

Sets the font style (normal, italic or slanted).

```
setFontUnderlined(This, Underlined) -> ok
```

Types:

```
This = wxTextAttr()
Underlined = boolean()
```

Sets the font underlining (solid line, text colour).

```
setFontWeight(This, FontWeight) -> ok
```

Types:

```
This = wxTextAttr()
FontWeight = wx:wx_enum()
```

Sets the font weight.

```
setLeftIndent(This, Indent) -> ok
```

Types:

```
This = wxTextAttr()
Indent = integer()
```

```
setLeftIndent(This, Indent, Options :: [Option]) -> ok
```

Types:

```
This = wxTextAttr()
Indent = integer()
Option = {subIndent, integer()}
```

Sets the left indent and left subindent in tenths of a millimetre.

The sub-indent is an offset from the left of the paragraph, and is used for all but the first line in a paragraph.

A positive value will cause the first line to appear to the left of the subsequent lines, and a negative value will cause the first line to be indented relative to the subsequent lines.

`wxRichTextBuffer` (not implemented in wx) uses indentation to render a bulleted item. The left indent is the distance between the margin and the bullet. The content of the paragraph, including the first line, starts at leftMargin + leftSubIndent. So the distance between the left edge of the bullet and the left of the actual paragraph is leftSubIndent.

```
setRightIndent(This, Indent) -> ok
```

Types:

```
This = wxTextAttr()
Indent = integer()
```

Sets the right indent in tenths of a millimetre.

```
setTabs(This, Tabs) -> ok
```

Types:

```
This = wxTextAttr()
Tabs = [integer()]
```

Sets the tab stops, expressed in tenths of a millimetre.

Each stop is measured from the left margin and therefore each value must be larger than the last.

```
setTextColour(This, ColText) -> ok
```

Types:

```
This = wxTextAttr()
ColText = wx:wx_colour()
```

Sets the text foreground colour.

```
destroy(This :: wxTextAttr()) -> ok
```

Destroys the object.

# wxTextCtrl

Erlang module

A text control allows text to be displayed and edited.

It may be single line or multi-line. Notice that a lot of methods of the text controls are found in the base `wxTextEntry` (not implemented in wx) class which is a common base class for `wxTextCtrl` and other controls using a single line text entry field (e.g. `wxComboBox`).

Styles

This class supports the following styles:

wxTextCtrl Text Format

The multiline text controls always store the text as a sequence of lines separated by `'\n'` characters, i.e. in the Unix text format even on non-Unix platforms. This allows the user code to ignore the differences between the platforms but at a price: the indices in the control such as those returned by `getInsertionPoint/1` or `getSelection/1` can `not` be used as indices into the string returned by `getValue/1` as they're going to be slightly off for platforms using `"\\r\\n"` as separator (as Windows does).

Instead, if you need to obtain a substring between the 2 indices obtained from the control with the help of the functions mentioned above, you should use `getRange/3`. And the indices themselves can only be passed to other methods, for example `setInsertionPoint/2` or `setSelection/3`.

To summarize: never use the indices returned by (multiline) `wxTextCtrl` as indices into the string it contains, but only as arguments to be passed back to the other `wxTextCtrl` methods. This problem doesn't arise for single-line platforms however where the indices in the control do correspond to the positions in the value string.

wxTextCtrl Positions and Coordinates

It is possible to use either linear positions, i.e. roughly (but `not` always exactly, as explained in the previous section) the index of the character in the text contained in the control or X-Y coordinates, i.e. column and line of the character when working with this class and it provides the functions `positionToXY/2` and `xYToPosition/3` to convert between the two.

Additionally, a position in the control can be converted to its coordinates in pixels using `PositionToCoords()` (not implemented in wx) which can be useful to e.g. show a popup menu near the given character. And, in the other direction, `HitTest()` (not implemented in wx) can be used to find the character under, or near, the given pixel coordinates.

To be more precise, positions actually refer to the gaps between characters and not the characters themselves. Thus, position 0 is the one before the very first character in the control and so is a valid position even when the control is empty. And if the control contains a single character, it has two valid positions: 0 before this character and 1 - after it. This, when the documentation of various functions mentions "invalid position", it doesn't consider the position just after the last character of the line to be invalid, only the positions beyond that one (e.g. 2 and greater in the single character example) are actually invalid.

wxTextCtrl Styles.

Multi-line text controls support styling, i.e. provide a possibility to set colours and font for individual characters in it (note that under Windows `wxTE_RICH` style is required for style support). To use the styles you can either call `setDefaultStyle/2` before inserting the text or call `setStyle/4` later to change the style of the text already in the control (the first solution is much more efficient).

In either case, if the style doesn't specify some of the attributes (for example you only want to set the text colour but without changing the font nor the text background), the values of the default style will be used for them. If there is no default style, the attributes of the text control itself are used.

So the following code correctly describes what it does: the second call to `setDefaultStyle/2` doesn't change the text foreground colour (which stays red) while the last one doesn't change the background colour (which stays grey):

wxTextCtrl and C++ Streams

This class multiply-inherits from `std::streambuf` (except for some really old compilers using non-standard iostream library), allowing code such as the following:

Note that even if your build of wxWidgets doesn't support this (the symbol `wxHAS_TEXT_WINDOW_STREAM` has value of 0 then) you can still use `wxTextCtrl` itself in a stream-like manner:

However the possibility to create a `std::ostream` associated with `wxTextCtrl` may be useful if you need to redirect the output of a function taking a `std::ostream` as parameter to a text control.

Another commonly requested need is to redirect `std::cout` to the text control. This may be done in the following way:

But wxWidgets provides a convenient class to make it even simpler so instead you may just do

See `wxStreamToTextRedirector` (not implemented in wx) for more details.

Event Handling.

The following commands are processed by default event handlers in `wxTextCtrl`: `wxID_CUT`, `wxID_COPY`, `wxID_PASTE`, `wxID_UNDO`, `wxID_REDO`. The associated UI update events are also processed automatically, when the control has the focus.

See: `create/4`, `wxValidator` (not implemented in wx)

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxTextCtrl**

## Events

Event types emitted from this class: `command_text_updated`, `command_text_enter`, `text_maxlen`

## Data Types

```
wxTextCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxTextCtrl()
```

Default ctor.

```
new(Parent, Id) -> wxTextCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxTextCtrl()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {value, unicode:chardata()} |
```

```
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Constructor, creating and showing a text control.

Remark: The horizontal scrollbar (wxHSCROLL style flag) will only be created for multi-line text controls. Without a horizontal scrollbar, text lines that don't fit in the control's size will be wrapped (but no newline character is inserted). Single line controls don't have a horizontal scrollbar, the text is automatically scrolled so that the insertion point is always visible.

See: `create/4`, `wxValidator` (not implemented in wx)

```
destroy(This :: wxTextCtrl()) -> ok
```

Destructor, destroying the text control.

```
appendText(This, Text) -> ok
```

Types:

```
   This = wxTextCtrl()
   Text = unicode:chardata()
```

Appends the text to the end of the text control.

Remark: After the text is appended, the insertion point will be at the end of the text control. If this behaviour is not desired, the programmer should use `getInsertionPoint/1` and `setInsertionPoint/2`.

See: `writeText/2`

```
canCopy(This) -> boolean()
```

Types:

```
   This = wxTextCtrl()
```

Returns true if the selection can be copied to the clipboard.

```
canCut(This) -> boolean()
```

Types:

```
   This = wxTextCtrl()
```

Returns true if the selection can be cut to the clipboard.

```
canPaste(This) -> boolean()
```

Types:

```
   This = wxTextCtrl()
```

Returns true if the contents of the clipboard can be pasted into the text control.

On some platforms (Motif, GTK) this is an approximation and returns true if the control is editable, false otherwise.

```
canRedo(This) -> boolean()
```

Types:

```
   This = wxTextCtrl()
```

Returns true if there is a redo facility available and the last operation can be redone.

```
canUndo(This) -> boolean()
```
Types:
```
   This = wxTextCtrl()
```
Returns true if there is an undo facility available and the last operation can be undone.

```
clear(This) -> ok
```
Types:
```
   This = wxTextCtrl()
```
Clears the text in the control.

Note that this function will generate a `wxEVT_TEXT` event, i.e. its effect is identical to calling `SetValue`("").

```
copy(This) -> ok
```
Types:
```
   This = wxTextCtrl()
```
Copies the selected text to the clipboard.

```
create(This, Parent, Id) -> boolean()
```
Types:
```
   This = wxTextCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
create(This, Parent, Id, Options :: [Option]) -> boolean()
```
Types:
```
   This = wxTextCtrl()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {value, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```
Creates the text control for two-step construction.

This method should be called if the default constructor was used for the control creation. Its parameters have the same meaning as for the non-default constructor.

```
cut(This) -> ok
```
Types:
```
   This = wxTextCtrl()
```
Copies the selected text to the clipboard and removes it from the control.

```
discardEdits(This) -> ok
```
Types:

```
This = wxTextCtrl()
```

Resets the internal modified flag as if the current changes had been saved.

```
changeValue(This, Value) -> ok
```

Types:

```
This = wxTextCtrl()
Value = unicode:chardata()
```

Sets the new text control value.

It also marks the control as not-modified which means that IsModified() would return false immediately after the call to `changeValue/2`.

The insertion point is set to the start of the control (i.e. position 0) by this function.

This functions does not generate the `wxEVT_TEXT` event but otherwise is identical to `setValue/2`.

See overview_events_prog for more information.

Since: 2.7.1

```
emulateKeyPress(This, Event) -> boolean()
```

Types:

```
This = wxTextCtrl()
Event = wxKeyEvent:wxKeyEvent()
```

This function inserts into the control the character which would have been inserted if the given key event had occurred in the text control.

The `event` object should be the same as the one passed to `EVT_KEY_DOWN` handler previously by wxWidgets. Please note that this function doesn't currently work correctly for all keys under any platform but MSW.

Return: true if the event resulted in a change to the control, false otherwise.

```
getDefaultStyle(This) -> wxTextAttr:wxTextAttr()
```

Types:

```
This = wxTextCtrl()
```

Returns the style currently used for the new text.

See: `setDefaultStyle/2`

```
getInsertionPoint(This) -> integer()
```

Types:

```
This = wxTextCtrl()
```

Returns the insertion point, or cursor, position.

This is defined as the zero based index of the character position to the right of the insertion point. For example, if the insertion point is at the end of the single-line text control, it is equal to `getLastPosition/1`.

Notice that insertion position is, in general, different from the index of the character the cursor position at in the string returned by `getValue/1`. While this is always the case for the single line controls, multi-line controls can use two characters `"\\r\\n"` as line separator (this is notably the case under MSW) meaning that indices in the control and its string value are offset by 1 for every line.

Hence to correctly get the character at the current cursor position, taking into account that there can be none if the cursor is at the end of the string, you could do the following:

```
getLastPosition(This) -> integer()
```

Types:

```
This = wxTextCtrl()
```

Returns the zero based index of the last position in the text control, which is equal to the number of characters in the control.

```
getLineLength(This, LineNo) -> integer()
```

Types:

```
This = wxTextCtrl()
LineNo = integer()
```

Gets the length of the specified line, not including any trailing newline character(s).

Return: The length of the line, or -1 if `lineNo` was invalid.

```
getLineText(This, LineNo) -> unicode:charlist()
```

Types:

```
This = wxTextCtrl()
LineNo = integer()
```

Returns the contents of a given line in the text control, not including any trailing newline character(s).

Return: The contents of the line.

```
getNumberOfLines(This) -> integer()
```

Types:

```
This = wxTextCtrl()
```

Returns the number of lines in the text control buffer.

The returned number is the number of logical lines, i.e. just the count of the number of newline characters in the control + 1, for wxGTK and wxOSX/Cocoa ports while it is the number of physical lines, i.e. the count of lines actually shown in the control, in wxMSW. Because of this discrepancy, it is not recommended to use this function.

Remark: Note that even empty text controls have one line (where the insertion point is), so `getNumberOfLines/1` never returns 0.

```
getRange(This, From, To) -> unicode:charlist()
```

Types:

```
This = wxTextCtrl()
From = To = integer()
```

Returns the string containing the text starting in the positions `from` and up to `to` in the control.

The positions must have been returned by another `wxTextCtrl` method. Please note that the positions in a multiline `wxTextCtrl` do `not` correspond to the indices in the string returned by `getValue/1` because of the different new line representations (`CR` or `CR` LF) and so this method should be used to obtain the correct results instead of extracting parts of the entire value. It may also be more efficient, especially if the control contains a lot of data.

```
getSelection(This) -> {From :: integer(), To :: integer()}
```

Types:

```
This = wxTextCtrl()
```

Gets the current selection span.

If the returned values are equal, there was no selection. Please note that the indices returned may be used with the other `wxTextCtrl` methods but don't necessarily represent the correct indices into the string returned by `getValue/1` for multiline controls under Windows (at least,) you should use `getStringSelection/1` to get the selected text.

```
getStringSelection(This) -> unicode:charlist()
```

Types:

```
This = wxTextCtrl()
```

Gets the text currently selected in the control.

If there is no selection, the returned string is empty.

```
getStyle(This, Position, Style) -> boolean()
```

Types:

```
This = wxTextCtrl()
Position = integer()
Style = wxTextAttr:wxTextAttr()
```

Returns the style at this position in the text control.

Not all platforms support this function.

Return: true on success, false if an error occurred (this may also mean that the styles are not supported under this platform).

See: `setStyle/4`, `wxTextAttr`

```
getValue(This) -> unicode:charlist()
```

Types:

```
This = wxTextCtrl()
```

Gets the contents of the control.

Notice that for a multiline text control, the lines will be separated by (Unix-style) `\n` characters, even under Windows where they are separated by a `\r\n` sequence in the native control.

```
isEditable(This) -> boolean()
```

Types:

```
This = wxTextCtrl()
```

Returns true if the controls contents may be edited by user (note that it always can be changed by the program).

In other words, this functions returns true if the control hasn't been put in read-only mode by a previous call to `setEditable/2`.

```
isModified(This) -> boolean()
```

Types:

```
This = wxTextCtrl()
```

Returns true if the text has been modified by user.

Note that calling `setValue/2` doesn't make the control modified.

See: `markDirty/1`

```
isMultiLine(This) -> boolean()
```

Types:

```
This = wxTextCtrl()
```

Returns true if this is a multi line edit control and false otherwise.

See: `isSingleLine/1`

```
isSingleLine(This) -> boolean()
```

Types:

```
This = wxTextCtrl()
```

Returns true if this is a single line edit control and false otherwise.

See: `isSingleLine/1`, `isMultiLine/1`

```
loadFile(This, Filename) -> boolean()
```

Types:

```
This = wxTextCtrl()
Filename = unicode:chardata()
```

```
loadFile(This, Filename, Options :: [Option]) -> boolean()
```

Types:

```
This = wxTextCtrl()
Filename = unicode:chardata()
Option = {fileType, integer()}
```

Loads and displays the named file, if it exists.

Return: true if successful, false otherwise.

```
markDirty(This) -> ok
```

Types:

```
This = wxTextCtrl()
```

Mark text as modified (dirty).

See: `isModified/1`

```
paste(This) -> ok
```

Types:

```
This = wxTextCtrl()
```

Pastes text from the clipboard to the text item.

```
positionToXY(This, Pos) -> Result
```

Types:

```
Result = {Res :: boolean(), X :: integer(), Y :: integer()}
This = wxTextCtrl()
Pos = integer()
```

Converts given position to a zero-based column, line number pair.

Return: true on success, false on failure (most likely due to a too large position parameter).

See: `xYToPosition/3`

```
redo(This) -> ok
```

Types:

```
This = wxTextCtrl()
```

If there is a redo facility and the last operation can be redone, redoes the last operation.

Does nothing if there is no redo facility.

```
remove(This, From, To) -> ok
```

Types:

```
This = wxTextCtrl()
From = To = integer()
```

Removes the text starting at the first given position up to (but not including) the character at the last position.

This function puts the current insertion point position at `to` as a side effect.

```
replace(This, From, To, Value) -> ok
```

Types:

```
This = wxTextCtrl()
From = To = integer()
Value = unicode:chardata()
```

Replaces the text starting at the first position up to (but not including) the character at the last position with the given text.

This function puts the current insertion point position at `to` as a side effect.

```
saveFile(This) -> boolean()
```

Types:

```
This = wxTextCtrl()
```

```
saveFile(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxTextCtrl()
Option = {file, unicode:chardata()} | {fileType, integer()}
```

Saves the contents of the control in a text file.

Return: true if the operation was successful, false otherwise.

```
setDefaultStyle(This, Style) -> boolean()
```

Types:

```
This = wxTextCtrl()
Style = wxTextAttr:wxTextAttr()
```

Changes the default style to use for the new text which is going to be added to the control.

This applies both to the text added programmatically using `writeText/2` or `appendText/2` and to the text entered by the user interactively.

If either of the font, foreground, or background colour is not set in `style`, the values of the previous default style are used for them. If the previous default style didn't set them neither, the global font or colours of the text control itself are used as fall back.

However if the `style` parameter is the default `wxTextAttr`, then the default style is just reset (instead of being combined with the new style which wouldn't change it at all).

Return: true on success, false if an error occurred (this may also mean that the styles are not supported under this platform).

See: `getDefaultStyle/1`

```
setEditable(This, Editable) -> ok
```

Types:

```
This = wxTextCtrl()
Editable = boolean()
```

Makes the text item editable or read-only, overriding the `wxTE_READONLY` flag.

See: `isEditable/1`

```
setInsertionPoint(This, Pos) -> ok
```

Types:

```
This = wxTextCtrl()
Pos = integer()
```

Sets the insertion point at the given position.

```
setInsertionPointEnd(This) -> ok
```

Types:

```
This = wxTextCtrl()
```

Sets the insertion point at the end of the text control.

This is equivalent to calling `setInsertionPoint/2` with `getLastPosition/1` argument.

```
setMaxLength(This, Len) -> ok
```

Types:

```
This = wxTextCtrl()
Len = integer()
```

This function sets the maximum number of characters the user can enter into the control.

In other words, it allows limiting the text value length to `len` not counting the terminating `NUL` character.

If len is 0, the previously set max length limit, if any, is discarded and the user may enter as much text as the underlying native text control widget supports (typically at least 32Kb). If the user tries to enter more characters into the text control when it already is filled up to the maximal length, a wxEVT_TEXT_MAXLEN event is sent to notify the program about it (giving it the possibility to show an explanatory message, for example) and the extra input is discarded.

Note that in wxGTK this function may only be used with single line text controls.

```
setSelection(This, From, To) -> ok
```

Types:

```
This = wxTextCtrl()
From = To = integer()
```

Selects the text starting at the first position up to (but not including) the character at the last position.

If both parameters are equal to -1 all text in the control is selected.

Notice that the insertion point will be moved to from by this function.

See: SelectAll() (not implemented in wx)

```
setStyle(This, Start, End, Style) -> boolean()
```

Types:

```
This = wxTextCtrl()
Start = End = integer()
Style = wxTextAttr:wxTextAttr()
```

Changes the style of the given range.

If any attribute within style is not set, the corresponding attribute from getDefaultStyle/1 is used.

Return: true on success, false if an error occurred (this may also mean that the styles are not supported under this platform).

See: getStyle/3, wxTextAttr

```
setValue(This, Value) -> ok
```

Types:

```
This = wxTextCtrl()
Value = unicode:chardata()
```

Sets the new text control value.

It also marks the control as not-modified which means that IsModified() would return false immediately after the call to setValue/2.

The insertion point is set to the start of the control (i.e. position 0) by this function unless the control value doesn't change at all, in which case the insertion point is left at its original position.

Note that, unlike most other functions changing the controls values, this function generates a wxEVT_TEXT event. To avoid this you can use changeValue/2 instead.

```
showPosition(This, Pos) -> ok
```

Types:

```
This = wxTextCtrl()
Pos = integer()
```

Makes the line containing the given position visible.

```
undo(This) -> ok
```

Types:

```
This = wxTextCtrl()
```

If there is an undo facility and the last operation can be undone, undoes the last operation.

Does nothing if there is no undo facility.

```
writeText(This, Text) -> ok
```

Types:

```
This = wxTextCtrl()
Text = unicode:chardata()
```

Writes the text into the text control at the current insertion position.

Remark: Newlines in the text string are the only control characters allowed, and they will cause appropriate line breaks. See operator<<() and `appendText/2` for more convenient ways of writing to the window. After the write operation, the insertion point will be at the end of the inserted text, so subsequent write operations will be appended. To append text after the user may have interacted with the control, call `setInsertionPointEnd/1` before writing.

```
xYToPosition(This, X, Y) -> integer()
```

Types:

```
This = wxTextCtrl()
X = Y = integer()
```

Converts the given zero based column and line number to a position.

Return: The position value, or -1 if x or y was invalid.

# wxTextDataObject

Erlang module

`wxTextDataObject` is a specialization of `wxDataObjectSimple` (not implemented in wx) for text data. It can be used without change to paste data into the `wxClipboard` or a `wxDropSource` (not implemented in wx). A user may wish to derive a new class from this class for providing text on-demand in order to minimize memory consumption when offering data in several formats, such as plain text and RTF because by default the text is stored in a string in this class, but it might as well be generated when requested. For this, `getTextLength/1` and `getText/1` will have to be overridden.

Note that if you already have the text inside a string, you will not achieve any efficiency gain by overriding these functions because copying wxStrings is already a very efficient operation (data is not actually copied because wxStrings are reference counted).

See: **Overview dnd**, `wxDataObject`, `wxDataObjectSimple` (not implemented in wx), `wxFileDataObject`, `wxBitmapDataObject`

This class is derived (and can use functions) from: `wxDataObject`

wxWidgets docs: **wxTextDataObject**

## Data Types

```
wxTextDataObject() = wx:wx_object()
```

## Exports

```
new() -> wxTextDataObject()
```

```
new(Options :: [Option]) -> wxTextDataObject()
```

Types:

```
Option = {text, unicode:chardata()}
```

Constructor, may be used to initialise the text (otherwise `setText/2` should be used later).

```
getTextLength(This) -> integer()
```

Types:

```
This = wxTextDataObject()
```

Returns the data size.

By default, returns the size of the text data set in the constructor or using `setText/2`. This can be overridden to provide text size data on-demand. It is recommended to return the text length plus 1 for a trailing zero, but this is not strictly required.

```
getText(This) -> unicode:charlist()
```

Types:

```
This = wxTextDataObject()
```

Returns the text associated with the data object.

You may wish to override this method when offering data on-demand, but this is not required by wxWidgets' internals. Use this method to get data in text form from the `wxClipboard`.

```
setText(This, StrText) -> ok
```

Types:

```
This = wxTextDataObject()
StrText = unicode:chardata()
```

Sets the text associated with the data object.

This method is called when the data object receives the data and, by default, copies the text into the member variable. If you want to process the text on the fly you may wish to override this function.

```
destroy(This :: wxTextDataObject()) -> ok
```

Destroys the object.

# wxTextEntryDialog

Erlang module

This class represents a dialog that requests a one-line text string from the user. It is implemented as a generic wxWidgets dialog.

See: **Overview cmndlg**

This class is derived (and can use functions) from: `wxDialog` `wxTopLevelWindow` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxTextEntryDialog**

## Data Types

```
wxTextEntryDialog() = wx:wx_object()
```

## Exports

```
new() -> wxTextEntryDialog()
```

Default constructor.

Call `Create()` (not implemented in wx) to really create the dialog later.

Since: 2.9.5

```
new(Parent, Message) -> wxTextEntryDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
Message = unicode:chardata()
```

```
new(Parent, Message, Options :: [Option]) -> wxTextEntryDialog()
```

Types:

```
Parent = wxWindow:wxWindow()
Message = unicode:chardata()
Option =
    {caption, unicode:chardata()} |
    {value, unicode:chardata()} |
    {style, integer()} |
    {pos, {X :: integer(), Y :: integer()}}
```

Constructor.

Use `wxDialog:showModal/1` to show the dialog.

See `Create()` (not implemented in wx) method for parameter description.

```
destroy(This :: wxTextEntryDialog()) -> ok
```

Destructor.

```
getValue(This) -> unicode:charlist()
```

Types:

```
This = wxTextEntryDialog()
```

Returns the text that the user has entered if the user has pressed OK, or the original value if the user has pressed Cancel.

```
setValue(This, Value) -> ok
```

Types:

```
This = wxTextEntryDialog()
Value = unicode:chardata()
```

Sets the default text value.

# wxToggleButton

Erlang module

wxToggleButton is a button that stays pressed when clicked by the user. In other words, it is similar to wxCheckBox in functionality but looks like a wxButton.

Since wxWidgets version 2.9.0 this control emits an update UI event.

You can see wxToggleButton in action in page_samples_widgets.

See: wxCheckBox, wxButton, wxBitmapToggleButton (not implemented in wx)

This class is derived (and can use functions) from: wxControl wxWindow wxEvtHandler

wxWidgets docs: **wxToggleButton**

## Events

Event types emitted from this class: command_togglebutton_clicked

## Data Types

```
wxToggleButton() = wx:wx_object()
```

## Exports

```
new() -> wxToggleButton()
```

Default constructor.

```
new(Parent, Id, Label) -> wxToggleButton()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Label = unicode:chardata()
```

```
new(Parent, Id, Label, Options :: [Option]) -> wxToggleButton()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Label = unicode:chardata()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {validator, wx:wx_object()}
```

Constructor, creating and showing a toggle button.

See: create/5, wxValidator (not implemented in wx)

```
destroy(This :: wxToggleButton()) -> ok
```

Destructor, destroying the toggle button.

```
create(This, Parent, Id, Label) -> boolean()
```

Types:

```
   This = wxToggleButton()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
```

```
create(This, Parent, Id, Label, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxToggleButton()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Label = unicode:chardata()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Creates the toggle button for two-step construction.

See `new/4` for details.

```
getValue(This) -> boolean()
```

Types:

```
   This = wxToggleButton()
```

Gets the state of the toggle button.

Return: Returns true if it is pressed, false otherwise.

```
setValue(This, State) -> ok
```

Types:

```
   This = wxToggleButton()
   State = boolean()
```

Sets the toggle button to the given state.

This does not cause a `EVT_TOGGLEBUTTON` event to be emitted.

# wxToolBar

Erlang module

A toolbar is a bar of buttons and/or other controls usually placed below the menu bar in a `wxFrame`.

You may create a toolbar that is managed by a frame calling `wxFrame:createToolBar/2`. Under Pocket PC, you should always use this function for creating the toolbar to be managed by the frame, so that wxWidgets can use a combined menubar and toolbar. Where you manage your own toolbars, create `wxToolBar` as usual.

There are several different types of tools you can add to a toolbar. These types are controlled by the ?wxItemKind enumeration.

Note that many methods in `wxToolBar` such as `addTool/6` return a `wxToolBarToolBase*` object. This should be regarded as an opaque handle representing the newly added toolbar item, providing access to its id and position within the toolbar. Changes to the item's state should be made through calls to `wxToolBar` methods, for example `enableTool/3`. Calls to `wxToolBarToolBase` (not implemented in wx) methods (undocumented by purpose) will not change the visible state of the item within the tool bar.

After you have added all the tools you need, you must call `realize/1` to effectively construct and display the toolbar.

`wxMSW note`: Note that under wxMSW toolbar paints tools to reflect system-wide colours. If you use more than 16 colours in your tool bitmaps, you may wish to suppress this behaviour, otherwise system colours in your bitmaps will inadvertently be mapped to system colours. To do this, set the msw.remap system option before creating the toolbar: If you wish to use 32-bit images (which include an alpha channel for transparency) use: Then colour remapping is switched off, and a transparent background used. But only use this option under Windows XP with true colour:

Styles

This class supports the following styles:

See: **Overview toolbar**

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxToolBar**

## Events

Event types emitted from this class: `command_tool_rclicked`, `command_tool_enter`, `tool_dropdown`

## Data Types

```
wxToolBar() = wx:wx_object()
```

## Exports

```
addControl(This, Control) -> wx:wx_object()
```

Types:

```
   This = wxToolBar()
   Control = wxControl:wxControl()
```

```
addControl(This, Control, Options :: [Option]) -> wx:wx_object()
```

Types:

```
This = wxToolBar()
Control = wxControl:wxControl()
Option = {label, unicode:chardata()}
```

Adds any control to the toolbar, typically e.g. a `wxComboBox`.

Remark: wxMac: labels are only displayed if wxWidgets is built with `wxMAC_USE_NATIVE_TOOLBAR` set to 1

```
addSeparator(This) -> wx:wx_object()
```

Types:

```
This = wxToolBar()
```

Adds a separator for spacing groups of tools.

Notice that the separator uses the look appropriate for the current platform so it can be a vertical line (MSW, some versions of GTK) or just an empty space or something else.

See: `addTool/6`, `setToolSeparation/2`, `addStretchableSpace/1`

```
addTool(This, Tool) -> wx:wx_object()
```

Types:

```
This = wxToolBar()
Tool = wx:wx_object()
```

Adds a tool to the toolbar.

Remark: After you have added tools to a toolbar, you must call `realize/1` in order to have the tools appear.

See: `addSeparator/1`, `addCheckTool/5`, `addRadioTool/5`, `insertTool/6`, `deleteTool/2`, `realize/1`, `SetDropdownMenu()` (not implemented in wx)

```
addTool(This, ToolId, Label, Bitmap) -> wx:wx_object()
```

Types:

```
This = wxToolBar()
ToolId = integer()
Label = unicode:chardata()
Bitmap = wxBitmap:wxBitmap()
```

```
addTool(This, ToolId, Label, Bitmap, BmpDisabled) ->
          wx:wx_object()
addTool(This, ToolId, Label, Bitmap, BmpDisabled :: [Option]) ->
          wx:wx_object()
```

Types:

```
This = wxToolBar()
ToolId = integer()
Label = unicode:chardata()
Bitmap = wxBitmap:wxBitmap()
Option = {shortHelp, unicode:chardata()} | {kind, wx:wx_enum()}
```

Adds a tool to the toolbar.

This most commonly used version has fewer parameters than the full version below which specifies the more rarely used button features.

Remark: After you have added tools to a toolbar, you must call `realize/1` in order to have the tools appear.

See: `addSeparator/1`, `addCheckTool/5`, `addRadioTool/5`, `insertTool/6`, `deleteTool/2`, `realize/1`, `SetDropdownMenu()` (not implemented in wx)

```
addTool(This, ToolId, Label, Bitmap, BmpDisabled,
        Options :: [Option]) ->
           wx:wx_object()
```

Types:

```
   This = wxToolBar()
   ToolId = integer()
   Label = unicode:chardata()
   Bitmap = BmpDisabled = wxBitmap:wxBitmap()
   Option =
       {kind, wx:wx_enum()} |
       {shortHelp, unicode:chardata()} |
       {longHelp, unicode:chardata()} |
       {data, wx:wx_object()}
```

Adds a tool to the toolbar.

Remark: After you have added tools to a toolbar, you must call `realize/1` in order to have the tools appear.

See: `addSeparator/1`, `addCheckTool/5`, `addRadioTool/5`, `insertTool/6`, `deleteTool/2`, `realize/1`, `SetDropdownMenu()` (not implemented in wx)

```
addCheckTool(This, ToolId, Label, Bitmap1) -> wx:wx_object()
```

Types:

```
   This = wxToolBar()
   ToolId = integer()
   Label = unicode:chardata()
   Bitmap1 = wxBitmap:wxBitmap()
```

```
addCheckTool(This, ToolId, Label, Bitmap1, Options :: [Option]) ->
               wx:wx_object()
```

Types:

```
   This = wxToolBar()
   ToolId = integer()
   Label = unicode:chardata()
   Bitmap1 = wxBitmap:wxBitmap()
   Option =
       {bmpDisabled, wxBitmap:wxBitmap()} |
       {shortHelp, unicode:chardata()} |
       {longHelp, unicode:chardata()} |
       {data, wx:wx_object()}
```

Adds a new check (or toggle) tool to the toolbar.

The parameters are the same as in `addTool/6`.

See: `addTool/6`

```
addRadioTool(This, ToolId, Label, Bitmap1) -> wx:wx_object()
```

Types:

```
   This = wxToolBar()
   ToolId = integer()
   Label = unicode:chardata()
   Bitmap1 = wxBitmap:wxBitmap()
```

```
addRadioTool(This, ToolId, Label, Bitmap1, Options :: [Option]) ->
               wx:wx_object()
```

Types:

```
   This = wxToolBar()
   ToolId = integer()
   Label = unicode:chardata()
   Bitmap1 = wxBitmap:wxBitmap()
   Option =
       {bmpDisabled, wxBitmap:wxBitmap()} |
       {shortHelp, unicode:chardata()} |
       {longHelp, unicode:chardata()} |
       {data, wx:wx_object()}
```

Adds a new radio tool to the toolbar.

Consecutive radio tools form a radio group such that exactly one button in the group is pressed at any moment, in other words whenever a button in the group is pressed the previously pressed button is automatically released. You should avoid having the radio groups of only one element as it would be impossible for the user to use such button.

By default, the first button in the radio group is initially pressed, the others are not.

See: `addTool/6`

```
addStretchableSpace(This) -> wx:wx_object()
```

Types:

```
   This = wxToolBar()
```

Adds a stretchable space to the toolbar.

Any space not taken up by the fixed items (all items except for stretchable spaces) is distributed in equal measure between the stretchable spaces in the toolbar. The most common use for this method is to add a single stretchable space before the items which should be right-aligned in the toolbar, but more exotic possibilities are possible, e.g. a stretchable space may be added in the beginning and the end of the toolbar to centre all toolbar items.

See: `addTool/6`, `addSeparator/1`, `insertStretchableSpace/2`

Since: 2.9.1

```
insertStretchableSpace(This, Pos) -> wx:wx_object()
```

Types:

```
   This = wxToolBar()
   Pos = integer()
```

Inserts a stretchable space at the given position.

See `addStretchableSpace/1` for details about stretchable spaces.

See: insertTool/6, insertSeparator/2

Since: 2.9.1

```
deleteTool(This, ToolId) -> boolean()
```

Types:

```
This = wxToolBar()
ToolId = integer()
```

Removes the specified tool from the toolbar and deletes it.

If you don't want to delete the tool, but just to remove it from the toolbar (to possibly add it back later), you may use removeTool/2 instead.

Note: It is unnecessary to call realize/1 for the change to take place, it will happen immediately.

Return: true if the tool was deleted, false otherwise.

See: deleteToolByPos/2

```
deleteToolByPos(This, Pos) -> boolean()
```

Types:

```
This = wxToolBar()
Pos = integer()
```

This function behaves like deleteTool/2 but it deletes the tool at the specified position and not the one with the given id.

```
enableTool(This, ToolId, Enable) -> ok
```

Types:

```
This = wxToolBar()
ToolId = integer()
Enable = boolean()
```

Enables or disables the tool.

Remark: Some implementations will change the visible state of the tool to indicate that it is disabled.

See: getToolEnabled/2, toggleTool/3

```
findById(This, Id) -> wx:wx_object()
```

Types:

```
This = wxToolBar()
Id = integer()
```

Returns a pointer to the tool identified by id or NULL if no corresponding tool is found.

```
findControl(This, Id) -> wxControl:wxControl()
```

Types:

```
This = wxToolBar()
Id = integer()
```

Returns a pointer to the control identified by id or NULL if no corresponding control is found.

```
findToolForPosition(This, X, Y) -> wx:wx_object()
```

Types:

```
This = wxToolBar()
X = Y = integer()
```

Finds a tool for the given mouse position.

Return: A pointer to a tool if a tool is found, or NULL otherwise.

Remark: Currently not implemented in wxGTK (always returns NULL there).

```
getToolSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxToolBar()
```

Returns the size of a whole button, which is usually larger than a tool bitmap because of added 3D effects.

See: `setToolBitmapSize/2`, `getToolBitmapSize/1`

```
getToolBitmapSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxToolBar()
```

Returns the size of bitmap that the toolbar expects to have.

The default bitmap size is platform-dependent: for example, it is 16*15 for MSW and 24*24 for GTK. This size does `not` necessarily indicate the best size to use for the toolbars on the given platform, for this you should use `wxArtProvider::GetNativeSizeHint(wxART_TOOLBAR)` but in any case, as the bitmap size is deduced automatically from the size of the bitmaps associated with the tools added to the toolbar, it is usually unnecessary to call `setToolBitmapSize/2` explicitly.

Remark: Note that this is the size of the bitmap you pass to `addTool/6`, and not the eventual size of the tool button.

See: `setToolBitmapSize/2`, `getToolSize/1`

```
getMargins(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxToolBar()
```

Returns the left/right and top/bottom margins, which are also used for inter-toolspacing.

See: `setMargins/3`

```
getToolEnabled(This, ToolId) -> boolean()
```

Types:

```
This = wxToolBar()
ToolId = integer()
```

Called to determine whether a tool is enabled (responds to user input).

Return: true if the tool is enabled, false otherwise.

See: `enableTool/3`

```
getToolLongHelp(This, ToolId) -> unicode:charlist()
```

Types:

```
This = wxToolBar()
ToolId = integer()
```

Returns the long help for the given tool.

See: `setToolLongHelp/3`, `setToolShortHelp/3`

`getToolPacking(This) -> integer()`

Types:

```
This = wxToolBar()
```

Returns the value used for packing tools.

See: `setToolPacking/2`

`getToolPos(This, ToolId) -> integer()`

Types:

```
This = wxToolBar()
ToolId = integer()
```

Returns the tool position in the toolbar, or `wxNOT_FOUND` if the tool is not found.

`getToolSeparation(This) -> integer()`

Types:

```
This = wxToolBar()
```

Returns the default separator size.

See: `setToolSeparation/2`

`getToolShortHelp(This, ToolId) -> unicode:charlist()`

Types:

```
This = wxToolBar()
ToolId = integer()
```

Returns the short help for the given tool.

See: `getToolLongHelp/2`, `setToolShortHelp/3`

`getToolState(This, ToolId) -> boolean()`

Types:

```
This = wxToolBar()
ToolId = integer()
```

Gets the on/off state of a toggle tool.

Return: true if the tool is toggled on, false otherwise.

See: `toggleTool/3`

`insertControl(This, Pos, Control) -> wx:wx_object()`

Types:

```
   This = wxToolBar()
   Pos = integer()
   Control = wxControl:wxControl()

insertControl(This, Pos, Control, Options :: [Option]) ->
                 wx:wx_object()
```

Types:

```
   This = wxToolBar()
   Pos = integer()
   Control = wxControl:wxControl()
   Option = {label, unicode:chardata()}
```

Inserts the control into the toolbar at the given position.

You must call `realize/1` for the change to take place.

See: `addControl/3`, `insertTool/6`

```
insertSeparator(This, Pos) -> wx:wx_object()
```

Types:

```
   This = wxToolBar()
   Pos = integer()
```

Inserts the separator into the toolbar at the given position.

You must call `realize/1` for the change to take place.

See: `addSeparator/1`, `insertTool/6`

```
insertTool(This, Pos, Tool) -> wx:wx_object()
```

Types:

```
   This = wxToolBar()
   Pos = integer()
   Tool = wx:wx_object()

insertTool(This, Pos, ToolId, Label, Bitmap) -> wx:wx_object()
```

Types:

```
   This = wxToolBar()
   Pos = ToolId = integer()
   Label = unicode:chardata()
   Bitmap = wxBitmap:wxBitmap()

insertTool(This, Pos, ToolId, Label, Bitmap, Options :: [Option]) ->
              wx:wx_object()
```

Types:

```
This = wxToolBar()
Pos = ToolId = integer()
Label = unicode:chardata()
Bitmap = wxBitmap:wxBitmap()
Option =
    {bmpDisabled, wxBitmap:wxBitmap()} |
    {kind, wx:wx_enum()} |
    {shortHelp, unicode:chardata()} |
    {longHelp, unicode:chardata()} |
    {clientData, wx:wx_object()}
```

Inserts the tool with the specified attributes into the toolbar at the given position.

You must call `realize/1` for the change to take place.

See: `addTool/6`, `insertControl/4`, `insertSeparator/2`

Return: The newly inserted tool or NULL on failure. Notice that with the overload taking `tool` parameter the caller is responsible for deleting the tool in the latter case.

```
realize(This) -> boolean()
```

Types:

```
This = wxToolBar()
```

This function should be called after you have added tools.

```
removeTool(This, Id) -> wx:wx_object()
```

Types:

```
This = wxToolBar()
Id = integer()
```

Removes the given tool from the toolbar but doesn't delete it.

This allows inserting/adding this tool back to this (or another) toolbar later.

Note: It is unnecessary to call `realize/1` for the change to take place, it will happen immediately.

See: `deleteTool/2`

```
setMargins(This, X, Y) -> ok
```

Types:

```
This = wxToolBar()
X = Y = integer()
```

Set the values to be used as margins for the toolbar.

Remark: This must be called before the tools are added if absolute positioning is to be used, and the default (zero-size) margins are to be overridden.

See: `getMargins/1`

```
setToolBitmapSize(This, Size) -> ok
```

Types:

```
This = wxToolBar()
Size = {W :: integer(), H :: integer()}
```

Sets the default size of each tool bitmap.

The default bitmap size is 16 by 15 pixels.

Remark: This should be called to tell the toolbar what the tool bitmap size is. Call it before you add tools.

See: `getToolBitmapSize/1`, `getToolSize/1`

```
setToolLongHelp(This, ToolId, HelpString) -> ok
```

Types:

```
This = wxToolBar()
ToolId = integer()
HelpString = unicode:chardata()
```

Sets the long help for the given tool.

Remark: You might use the long help for displaying the tool purpose on the status line.

See: `getToolLongHelp/2`, `setToolShortHelp/3`

```
setToolPacking(This, Packing) -> ok
```

Types:

```
This = wxToolBar()
Packing = integer()
```

Sets the value used for spacing tools.

The default value is 1.

Remark: The packing is used for spacing in the vertical direction if the toolbar is horizontal, and for spacing in the horizontal direction if the toolbar is vertical.

See: `getToolPacking/1`

```
setToolShortHelp(This, ToolId, HelpString) -> ok
```

Types:

```
This = wxToolBar()
ToolId = integer()
HelpString = unicode:chardata()
```

Sets the short help for the given tool.

Remark: An application might use short help for identifying the tool purpose in a tooltip.

See: `getToolShortHelp/2`, `setToolLongHelp/3`

```
setToolSeparation(This, Separation) -> ok
```

Types:

```
This = wxToolBar()
Separation = integer()
```

Sets the default separator size.

The default value is 5.

See: addSeparator/1

```
toggleTool(This, ToolId, Toggle) -> ok
```

Types:

```
This = wxToolBar()
ToolId = integer()
Toggle = boolean()
```

Toggles a tool on or off.

This does not cause any event to get emitted.

Remark: Only applies to a tool that has been specified as a toggle tool.

# wxToolTip

Erlang module

This class holds information about a tooltip associated with a window (see `wxWindow:setToolTip/2`).

The four static methods, `enable/1`, `setDelay/1` `wxToolTip::SetAutoPop()` (not implemented in wx) and `wxToolTip::SetReshow()` (not implemented in wx) can be used to globally alter tooltips behaviour.

wxWidgets docs: **wxToolTip**

## Data Types

```
wxToolTip() = wx:wx_object()
```

## Exports

```
enable(Flag) -> ok
```

Types:

```
Flag = boolean()
```

Enable or disable tooltips globally.

Note: May not be supported on all platforms (eg. wxCocoa).

```
setDelay(Msecs) -> ok
```

Types:

```
Msecs = integer()
```

Set the delay after which the tooltip appears.

Note: May not be supported on all platforms.

```
new(Tip) -> wxToolTip()
```

Types:

```
Tip = unicode:chardata()
```

Constructor.

```
setTip(This, Tip) -> ok
```

Types:

```
This = wxToolTip()
Tip = unicode:chardata()
```

Set the tooltip text.

```
getTip(This) -> unicode:charlist()
```

Types:

```
This = wxToolTip()
```

Get the tooltip text.

```
getWindow(This) -> wxWindow:wxWindow()
```

Types:

```
   This = wxToolTip()
```

Get the associated window.

```
destroy(This :: wxToolTip()) -> ok
```

Destroys the object.

# wxToolbook

Erlang module

wxToolbook is a class similar to wxNotebook but which uses a wxToolBar to show the labels instead of the tabs.

There is no documentation for this class yet but its usage is identical to wxNotebook (except for the features clearly related to tabs only), so please refer to that class documentation for now. You can also use the page_samples_notebook to see wxToolbook in action.

One feature of this class not supported by wxBookCtrlBase is the support for disabling some of the pages, see EnablePage() (not implemented in wx).

Styles

This class supports the following styles:

See: **Overview bookctrl**, wxBookCtrlBase, wxNotebook, **Examples**

This class is derived (and can use functions) from: wxBookCtrlBase wxControl wxWindow wxEvtHandler

wxWidgets docs: **wxToolbook**

## Events

Event types emitted from this class: toolbook_page_changed, toolbook_page_changing

## Data Types

```
wxToolbook() = wx:wx_object()
```

## Exports

```
new() -> wxToolbook()
```

Constructs a choicebook control.

```
new(Parent, Id) -> wxToolbook()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxToolbook()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

```
addPage(This, Page, Text) -> boolean()
```

Types:

```
   This = wxToolbook()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()

addPage(This, Page, Text, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxToolbook()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
   Option = {bSelect, boolean()} | {imageId, integer()}
```

Adds a new page.

The page must have the book control itself as the parent and must not have been added to this control previously.

The call to this function will generate the page changing and page changed events if `select` is true, but not when inserting the very first page (as there is no previous page selection to switch from in this case and so it wouldn't make sense to e.g. veto such event).

Return: true if successful, false otherwise.

Remark: Do not delete the page, it will be deleted by the book control.

See: `insertPage/5`

```
advanceSelection(This) -> ok
```

Types:

```
   This = wxToolbook()

advanceSelection(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxToolbook()
   Option = {forward, boolean()}
```

Cycles through the tabs.

The call to this function generates the page changing events.

```
assignImageList(This, ImageList) -> ok
```

Types:

```
   This = wxToolbook()
   ImageList = wxImageList:wxImageList()
```

Sets the image list for the page control and takes ownership of the list.

See: `wxImageList`, `setImageList/2`

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = wxToolbook()
   Parent = wxWindow:wxWindow()
   Id = integer()

create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxToolbook()
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Create the tool book control that has already been constructed with the default constructor.

```
deleteAllPages(This) -> boolean()
```

Types:

```
   This = wxToolbook()
```

Deletes all pages.

```
getCurrentPage(This) -> wxWindow:wxWindow()
```

Types:

```
   This = wxToolbook()
```

Returns the currently selected page or NULL.

```
getImageList(This) -> wxImageList:wxImageList()
```

Types:

```
   This = wxToolbook()
```

Returns the associated image list, may be NULL.

See: `wxImageList`, `setImageList/2`

```
getPage(This, Page) -> wxWindow:wxWindow()
```

Types:

```
   This = wxToolbook()
   Page = integer()
```

Returns the window at the given page position.

```
getPageCount(This) -> integer()
```

Types:

```
   This = wxToolbook()
```

Returns the number of pages in the control.

```
getPageImage(This, NPage) -> integer()
```

Types:

```
This = wxToolbook()
NPage = integer()
```

Returns the image index for the given page.

```
getPageText(This, NPage) -> unicode:charlist()
```

Types:

```
This = wxToolbook()
NPage = integer()
```

Returns the string for the given page.

```
getSelection(This) -> integer()
```

Types:

```
This = wxToolbook()
```

Returns the currently selected page, or `wxNOT_FOUND` if none was selected.

Note that this method may return either the previously or newly selected page when called from the `EVT_BOOKCTRL_PAGE_CHANGED` handler depending on the platform and so `wxBookCtrlEvent:getSelection/1` should be used instead in this case.

```
hitTest(This, Pt) -> Result
```

Types:

```
Result = {Res :: integer(), Flags :: integer()}
This = wxToolbook()
Pt = {X :: integer(), Y :: integer()}
```

Returns the index of the tab at the specified position or `wxNOT_FOUND` if none.

If `flags` parameter is non-NULL, the position of the point inside the tab is returned as well.

Return: Returns the zero-based tab index or `wxNOT_FOUND` if there is no tab at the specified position.

```
insertPage(This, Index, Page, Text) -> boolean()
```

Types:

```
This = wxToolbook()
Index = integer()
Page = wxWindow:wxWindow()
Text = unicode:chardata()
```

```
insertPage(This, Index, Page, Text, Options :: [Option]) ->
              boolean()
```

Types:

```
This = wxToolbook()
Index = integer()
Page = wxWindow:wxWindow()
Text = unicode:chardata()
Option = {bSelect, boolean()} | {imageId, integer()}
```

Inserts a new page at the specified position.

Return: true if successful, false otherwise.

Remark: Do not delete the page, it will be deleted by the book control.

See: `addPage/4`

```
setImageList(This, ImageList) -> ok
```

Types:

```
This = wxToolbook()
ImageList = wxImageList:wxImageList()
```

Sets the image list to use.

It does not take ownership of the image list, you must delete it yourself.

See: `wxImageList`, `assignImageList/2`

```
setPageSize(This, Size) -> ok
```

Types:

```
This = wxToolbook()
Size = {W :: integer(), H :: integer()}
```

Sets the width and height of the pages.

Note: This method is currently not implemented for wxGTK.

```
setPageImage(This, Page, Image) -> boolean()
```

Types:

```
This = wxToolbook()
Page = Image = integer()
```

Sets the image index for the given page.

`image` is an index into the image list which was set with `setImageList/2`.

```
setPageText(This, Page, Text) -> boolean()
```

Types:

```
This = wxToolbook()
Page = integer()
Text = unicode:chardata()
```

Sets the text for the given page.

```
setSelection(This, Page) -> integer()
```

Types:

```
This = wxToolbook()
Page = integer()
```

Sets the selection to the given page, returning the previous selection.

Notice that the call to this function generates the page changing events, use the `changeSelection/2` function if you don't want these events to be generated.

See: `getSelection/1`

```
changeSelection(This, Page) -> integer()
```

Types:

```
This = wxToolbook()
Page = integer()
```

Changes the selection to the given page, returning the previous selection.

This function behaves as `setSelection/2` but does `not` generate the page changing events.

See overview_events_prog for more information.

```
destroy(This :: wxToolbook()) -> ok
```

Destroys the object.

# wxTopLevelWindow

Erlang module

`wxTopLevelWindow` is a common base class for `wxDialog` and `wxFrame`. It is an abstract base class meaning that you never work with objects of this class directly, but all of its methods are also applicable for the two classes above.

Note that the instances of `wxTopLevelWindow` are managed by wxWidgets in the internal top level window list.

See: `wxDialog`, `wxFrame`

This class is derived (and can use functions) from: `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxTopLevelWindow**

## Events

Event types emitted from this class: `maximize`, `move`, `show`

## Data Types

```
wxTopLevelWindow() = wx:wx_object()
```

## Exports

```
getIcon(This) -> wxIcon:wxIcon()
```

Types:

```
This = wxTopLevelWindow()
```

Returns the standard icon of the window.

The icon will be invalid if it hadn't been previously set by `setIcon/2`.

See: `getIcons/1`

```
getIcons(This) -> wxIconBundle:wxIconBundle()
```

Types:

```
This = wxTopLevelWindow()
```

Returns all icons associated with the window, there will be none of them if neither `setIcon/2` nor `setIcons/2` had been called before.

Use `getIcon/1` to get the main icon of the window.

See: `wxIconBundle`

```
getTitle(This) -> unicode:charlist()
```

Types:

```
This = wxTopLevelWindow()
```

Gets a string containing the window title.

See: `setTitle/2`

```
isActive(This) -> boolean()
```

Types:

```
This = wxTopLevelWindow()
```

Returns true if this window is currently active, i.e. if the user is currently working with it.

```
iconize(This) -> ok
```

Types:

```
This = wxTopLevelWindow()
```

```
iconize(This, Options :: [Option]) -> ok
```

Types:

```
This = wxTopLevelWindow()
Option = {iconize, boolean()}
```

Iconizes or restores the window.

Note that in wxGTK the change to the window state is not immediate, i.e. `isIconized/1` will typically return false right after a call to `iconize/2` and its return value will only change after the control flow returns to the event loop and the notification about the window being really iconized is received.

See: `isIconized/1`, `Restore()` (not implemented in wx), (), `wxIconizeEvent`

```
isFullScreen(This) -> boolean()
```

Types:

```
This = wxTopLevelWindow()
```

Returns true if the window is in fullscreen mode.

See: `showFullScreen/3`

```
isIconized(This) -> boolean()
```

Types:

```
This = wxTopLevelWindow()
```

Returns true if the window is iconized.

```
isMaximized(This) -> boolean()
```

Types:

```
This = wxTopLevelWindow()
```

Returns true if the window is maximized.

```
maximize(This) -> ok
```

Types:

```
This = wxTopLevelWindow()
```

```
maximize(This, Options :: [Option]) -> ok
```

Types:

```
This = wxTopLevelWindow()
Option = {maximize, boolean()}
```

Maximizes or restores the window.

Note that, just as with `iconize/2`, the change to the window state is not immediate in at least wxGTK port.

See: `Restore()` (not implemented in wx), `iconize/2`

```
requestUserAttention(This) -> ok
```

Types:

```
This = wxTopLevelWindow()
```

```
requestUserAttention(This, Options :: [Option]) -> ok
```

Types:

```
This = wxTopLevelWindow()
Option = {flags, integer()}
```

Use a system-dependent way to attract users attention to the window when it is in background.

`flags` may have the value of either `?wxUSER_ATTENTION_INFO` (default) or `?wxUSER_ATTENTION_ERROR` which results in a more drastic action. When in doubt, use the default value.

Note: This function should normally be only used when the application is not already in foreground.

This function is currently implemented for Win32 where it flashes the window icon in the taskbar, and for wxGTK with task bars supporting it.

```
setIcon(This, Icon) -> ok
```

Types:

```
This = wxTopLevelWindow()
Icon = wxIcon:wxIcon()
```

Sets the icon for this window.

Remark: The window takes a 'copy' of `icon`, but since it uses reference counting, the copy is very quick. It is safe to delete `icon` after calling this function.

Note: In wxMSW, `icon` must be either 16x16 or 32x32 icon.

See: `wxIcon`, `setIcons/2`

```
setIcons(This, Icons) -> ok
```

Types:

```
This = wxTopLevelWindow()
Icons = wxIconBundle:wxIconBundle()
```

Sets several icons of different sizes for this window: this allows using different icons for different situations (e.g.

task switching bar, taskbar, window title bar) instead of scaling, with possibly bad looking results, the only icon set by `setIcon/2`.

Note: In wxMSW, `icons` must contain a 16x16 or 32x32 icon, preferably both.

See: `wxIconBundle`

```
centerOnScreen(This) -> ok
```

Types:

```
This = wxTopLevelWindow()
```

```
centreOnScreen(This) -> ok
```

Types:

```
   This = wxTopLevelWindow()
```

```
centerOnScreen(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxTopLevelWindow()
   Option = {dir, integer()}
```

See: `centreOnScreen/2`.

```
centreOnScreen(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxTopLevelWindow()
   Option = {dir, integer()}
```

Centres the window on screen.

See: `wxWindow:centreOnParent/2`

```
setShape(This, Region) -> boolean()
```

Types:

```
   This = wxTopLevelWindow()
   Region = wxRegion:wxRegion() | wxGraphicsPath:wxGraphicsPath()
```

If the platform supports it, sets the shape of the window to that depicted by `region`.

The system will not display or respond to any mouse event for the pixels that lie outside of the region. To reset the window to the normal rectangular shape simply call `setShape/2` again with an empty `wxRegion`. Returns true if the operation is successful.

This method is available in this class only since wxWidgets 2.9.3, previous versions only provided it in `wxTopLevelWindow`.

Note that windows with non default shape have a fixed size and can't be resized using `wxWindow:setSize/6`.

```
setTitle(This, Title) -> ok
```

Types:

```
   This = wxTopLevelWindow()
   Title = unicode:chardata()
```

Sets the window title.

See: `getTitle/1`

```
showFullScreen(This, Show) -> boolean()
```

Types:

```
   This = wxTopLevelWindow()
   Show = boolean()
```

```
showFullScreen(This, Show, Options :: [Option]) -> boolean()
```

Types:

```
This = wxTopLevelWindow()
Show = boolean()
Option = {style, integer()}
```

Depending on the value of `show` parameter the window is either shown full screen or restored to its normal state.

`style` is a bit list containing some or all of the following values, which indicate what elements of the window to hide in full-screen mode:

This function has not been tested with MDI frames.

Note: Showing a window full screen also actually `wxWindow:show/2`s the window if it isn't shown.

See: `EnableFullScreenView()` (not implemented in wx), `isFullScreen/1`

# wxTreeCtrl

Erlang module

A tree control presents information as a hierarchy, with items that may be expanded to show further items. Items in a tree control are referenced by `wxTreeItemId` (not implemented in wx) handles, which may be tested for validity by calling `wxTreeItemId::IsOk()` (not implemented in wx).

A similar control with a fully native implementation for GTK+ and macOS as well is `wxDataViewTreeCtrl` (not implemented in wx).

To intercept events from a tree control, use the event table macros described in `wxTreeEvent`.

Styles

This class supports the following styles:

See also overview_windowstyles.

`Win32 notes:`

`wxTreeCtrl` class uses the standard common treeview control under Win32 implemented in the system library comctl32.dll. Some versions of this library are known to have bugs with handling the tree control colours: the usual symptom is that the expanded items leave black (or otherwise incorrectly coloured) background behind them, especially for the controls using non-default background colour. The recommended solution is to upgrade the comctl32.dll to a newer version: see **http://www.microsoft.com/downloads/details.aspx?familyid=cb2cf3a2-8025-4e8f-8511-9b476a8d35d2**

See: `wxDataViewTreeCtrl` (not implemented in wx), `wxTreeEvent`, `wxTreeItemData` (not implemented in wx), **Overview treectrl**, `wxListBox`, `wxListCtrl`, `wxImageList`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxTreeCtrl**

## Events

Event types emitted from this class: `command_tree_begin_drag`, `command_tree_begin_rdrag`, `command_tree_end_drag`, `command_tree_begin_label_edit`, `command_tree_end_label_edit`, `command_tree_delete_item`, `command_tree_get_info`, `command_tree_set_info`, `command_tree_item_activated`, `command_tree_item_collapsed`, `command_tree_item_collapsing`, `command_tree_item_expanded`, `command_tree_item_expanding`, `command_tree_item_right_click`, `command_tree_item_middle_click`, `command_tree_sel_changed`, `command_tree_sel_changing`, `command_tree_key_down`, `command_tree_item_gettooltip`, `command_tree_item_menu`, `command_tree_state_image_click`

## Data Types

```
wxTreeCtrl() = wx:wx_object()
```

## Exports

```
new() -> wxTreeCtrl()
```

Default Constructor.

```
new(Parent) -> wxTreeCtrl()
```

Types:

```
Parent = wxWindow:wxWindow()
```

```
new(Parent, Options :: [Option]) -> wxTreeCtrl()
```

Types:

```
Parent = wxWindow:wxWindow()
Option =
    {id, integer()} |
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()} |
    {validator, wx:wx_object()}
```

Constructor, creating and showing a tree control.

See: `create/3`, `wxValidator` (not implemented in wx)

```
destroy(This :: wxTreeCtrl()) -> ok
```

Destructor, destroying the tree control.

```
addRoot(This, Text) -> integer()
```

Types:

```
This = wxTreeCtrl()
Text = unicode:chardata()
```

```
addRoot(This, Text, Options :: [Option]) -> integer()
```

Types:

```
This = wxTreeCtrl()
Text = unicode:chardata()
Option =
    {image, integer()} |
    {selectedImage, integer()} |
    {data, term()}
```

Adds the root node to the tree, returning the new item.

The `image` and `selImage` parameters are an index within the normal image list specifying the image to use for unselected and selected items, respectively. If `image` > -1 and `selImage` is -1, the same image is used for both selected and unselected items.

```
appendItem(This, Parent, Text) -> integer()
```

Types:

```
This = wxTreeCtrl()
Parent = integer()
Text = unicode:chardata()
```

```
appendItem(This, Parent, Text, Options :: [Option]) -> integer()
```

Types:

```
This = wxTreeCtrl()
Parent = integer()
Text = unicode:chardata()
Option =
    {image, integer()} |
    {selectedImage, integer()} |
    {data, term()}
```

Appends an item to the end of the branch identified by `parent`, return a new item id.

The `image` and `selImage` parameters are an index within the normal image list specifying the image to use for unselected and selected items, respectively. If `image` > -1 and `selImage` is -1, the same image is used for both selected and unselected items.

### assignImageList(This, ImageList) -> ok

Types:

```
This = wxTreeCtrl()
ImageList = wxImageList:wxImageList()
```

Sets the normal image list.

The image list assigned with this method will be automatically deleted by `wxTreeCtrl` as appropriate (i.e. it takes ownership of the list).

See: `setImageList/2`

### assignStateImageList(This, ImageList) -> ok

Types:

```
This = wxTreeCtrl()
ImageList = wxImageList:wxImageList()
```

Sets the state image list.

Image list assigned with this method will be automatically deleted by `wxTreeCtrl` as appropriate (i.e. it takes ownership of the list).

See: `setStateImageList/2`

### collapse(This, Item) -> ok

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Collapses the given item.

### collapseAndReset(This, Item) -> ok

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Collapses the given item and removes all children.

```
create(This, Parent) -> boolean()
```

Types:

```
   This = wxTreeCtrl()
   Parent = wxWindow:wxWindow()
```

```
create(This, Parent, Options :: [Option]) -> boolean()
```

Types:

```
   This = wxTreeCtrl()
   Parent = wxWindow:wxWindow()
   Option =
       {id, integer()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()} |
       {validator, wx:wx_object()}
```

Creates the tree control.

See `new/2` for further details.

```
delete(This, Item) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Deletes the specified item.

A `EVT_TREE_DELETE_ITEM` event will be generated.

This function may cause a subsequent call to `getNextChild/3` to fail.

```
deleteAllItems(This) -> ok
```

Types:

```
   This = wxTreeCtrl()
```

Deletes all items in the control.

This function generates `wxEVT_TREE_DELETE_ITEM` events for each item being deleted, including the root one if it is shown, i.e. unless wxTR_HIDE_ROOT style is used.

```
deleteChildren(This, Item) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Deletes all children of the given item (but not the item itself).

A `wxEVT_TREE_DELETE_ITEM` event will be generated for every item being deleted.

If you have called `setItemHasChildren/3`, you may need to call it again since `deleteChildren/2` does not automatically clear the setting.

```
editLabel(This, Item) -> wxTextCtrl:wxTextCtrl()
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Starts editing the label of the given `item`.

This function generates a `EVT_TREE_BEGIN_LABEL_EDIT` event which can be vetoed so that no text control will appear for in-place editing.

If the user changed the label (i.e. s/he does not press ESC or leave the text control without changes, a `EVT_TREE_END_LABEL_EDIT` event will be sent which can be vetoed as well.

See: `EndEditLabel()` (not implemented in wx), `wxTreeEvent`

```
ensureVisible(This, Item) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Scrolls and/or expands items to ensure that the given item is visible.

This method can be used, and will work, even while the window is frozen (see `wxWindow:freeze/1`).

```
expand(This, Item) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Expands the given item.

```
getBoundingRect(This, Item) -> Result
```

Types:

```
   Result =
       {Res :: boolean(),
        Rect ::
            {X :: integer(),
             Y :: integer(),
             W :: integer(),
             H :: integer()}}
   This = wxTreeCtrl()
   Item = integer()
```

```
getBoundingRect(This, Item, Options :: [Option]) -> Result
```

Types:

```
   Result =
       {Res :: boolean(),
        Rect ::
            {X :: integer(),
             Y :: integer(),
             W :: integer(),
```

```
           H :: integer()}}
   This = wxTreeCtrl()
   Item = integer()
   Option = {textOnly, boolean()}
```

Retrieves the rectangle bounding the `item`.

If `textOnly` is true, only the rectangle around the item's label will be returned, otherwise the item's image is also taken into account.

The return value is true if the rectangle was successfully retrieved or false if it was not (in this case `rect` is not changed) - for example, if the item is currently invisible.

Notice that the rectangle coordinates are logical, not physical ones. So, for example, the x coordinate may be negative if the tree has a horizontal scrollbar and its position is not 0.

```
getChildrenCount(This, Item) -> integer()
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

```
getChildrenCount(This, Item, Options :: [Option]) -> integer()
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
   Option = {recursively, boolean()}
```

Returns the number of items in the branch.

If `recursively` is true, returns the total number of descendants, otherwise only one level of children is counted.

```
getCount(This) -> integer()
```

Types:

```
   This = wxTreeCtrl()
```

Returns the number of items in the control.

```
getEditControl(This) -> wxTextCtrl:wxTextCtrl()
```

Types:

```
   This = wxTreeCtrl()
```

Returns the edit control being currently used to edit a label.

Returns NULL if no label is being edited.

Note: This is currently only implemented for wxMSW.

```
getFirstChild(This, Item) -> Result
```

Types:

```
Result = {Res :: integer(), Cookie :: integer()}
This = wxTreeCtrl()
Item = integer()
```

Returns the first child; call `getNextChild/3` for the next child.

For this enumeration function you must pass in a 'cookie' parameter which is opaque for the application but is necessary for the library to make these functions reentrant (i.e. allow more than one enumeration on one and the same object simultaneously). The cookie passed to `getFirstChild/2` and `getNextChild/3` should be the same variable.

Returns an invalid tree item (i.e. `wxTreeItemId::IsOk()` (not implemented in wx) returns false) if there are no further children.

See: `getNextChild/3`, `getNextSibling/2`

```
getNextChild(This, Item, Cookie) -> Result
```

Types:

```
Result = {Res :: integer(), Cookie :: integer()}
This = wxTreeCtrl()
Item = Cookie = integer()
```

Returns the next child; call `getFirstChild/2` for the first child.

For this enumeration function you must pass in a 'cookie' parameter which is opaque for the application but is necessary for the library to make these functions reentrant (i.e. allow more than one enumeration on one and the same object simultaneously). The cookie passed to `getFirstChild/2` and `getNextChild/3` should be the same.

Returns an invalid tree item if there are no further children.

See: `getFirstChild/2`

```
getFirstVisibleItem(This) -> integer()
```

Types:

```
This = wxTreeCtrl()
```

Returns the first visible item.

```
getImageList(This) -> wxImageList:wxImageList()
```

Types:

```
This = wxTreeCtrl()
```

Returns the normal image list.

```
getIndent(This) -> integer()
```

Types:

```
This = wxTreeCtrl()
```

Returns the current tree control indentation.

```
getItemBackgroundColour(This, Item) -> wx:wx_colour4()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the background colour of the item.

```
getItemData(This, Item) -> term()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the tree item data associated with the item.

See: `wxTreeItemData` (not implemented in wx)

```
getItemFont(This, Item) -> wxFont:wxFont()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the font of the item label.

If the font hadn't been explicitly set for the specified `item` with `setItemFont/3`, returns an invalid ?wxNullFont font. `wxWindow:getFont/1` can be used to retrieve the global tree control font used for the items without any specific font.

```
getItemImage(This, Item) -> integer()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

```
getItemImage(This, Item, Options :: [Option]) -> integer()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
Option = {which, wx:wx_enum()}
```

Gets the specified item image.

The value of `which` may be:

```
getItemText(This, Item) -> unicode:charlist()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the item label.

```
getItemTextColour(This, Item) -> wx:wx_colour4()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the colour of the item label.

```
getLastChild(This, Item) -> integer()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the last child of the item (or an invalid tree item if this item has no children).

See: `getFirstChild/2`, `getNextSibling/2`, `getLastChild/2`

```
getNextSibling(This, Item) -> integer()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the next sibling of the specified item; call `getPrevSibling/2` for the previous sibling.

Returns an invalid tree item if there are no further siblings.

See: `getPrevSibling/2`

```
getNextVisible(This, Item) -> integer()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the next visible item or an invalid item if this item is the last visible one.

Note: The `item` itself must be visible.

```
getItemParent(This, Item) -> integer()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the item's parent.

```
getPrevSibling(This, Item) -> integer()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the previous sibling of the specified item; call `getNextSibling/2` for the next sibling.

Returns an invalid tree item if there are no further children.

See: `getNextSibling/2`

```
getPrevVisible(This, Item) -> integer()
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Returns the previous visible item or an invalid item if this item is the first visible one.

Note: The `item` itself must be visible.

```
getRootItem(This) -> integer()
```

Types:

```
This = wxTreeCtrl()
```

Returns the root item for the tree control.

```
getSelection(This) -> integer()
```

Types:

```
This = wxTreeCtrl()
```

Returns the selection, or an invalid item if there is no selection.

This function only works with the controls without `wxTR_MULTIPLE` style, use `getSelections/1` for the controls which do have this style or, if a single item is wanted, use `GetFocusedItem()` (not implemented in wx).

```
getSelections(This) -> Result
```

Types:

```
Result = {Res :: integer(), Selection :: [integer()]}
This = wxTreeCtrl()
```

Fills the array of tree items passed in with the currently selected items.

This function can be called only if the control has the `wxTR_MULTIPLE` style.

Returns the number of selected items.

```
getStateImageList(This) -> wxImageList:wxImageList()
```

Types:

```
This = wxTreeCtrl()
```

Returns the state image list (from which application-defined state images are taken).

```
hitTest(This, Point) -> Result
```

Types:

```
Result = {Res :: integer(), Flags :: integer()}
This = wxTreeCtrl()
Point = {X :: integer(), Y :: integer()}
```

Calculates which (if any) item is under the given `point`, returning the tree item id at this point plus extra information `flags`.

`flags` is a bitlist of the following:

```
insertItem(This, Parent, Previous, Text) -> integer()
```

Types:

```
   This = wxTreeCtrl()
   Parent = Previous = integer()
   Text = unicode:chardata()

insertItem(This, Parent, Previous, Text, Options :: [Option]) ->
              integer()
```

Types:

```
   This = wxTreeCtrl()
   Parent = Previous = integer()
   Text = unicode:chardata()
   Option =
       {image, integer()} | {selImage, integer()} | {data, term()}
```

Inserts an item after a given one (`previous`).

The `image` and `selImage` parameters are an index within the normal image list specifying the image to use for unselected and selected items, respectively. If `image` > -1 and `selImage` is -1, the same image is used for both selected and unselected items.

```
isBold(This, Item) -> boolean()
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Returns true if the given item is in bold state.

See: `setItemBold/3`

```
isExpanded(This, Item) -> boolean()
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Returns true if the item is expanded (only makes sense if it has children).

```
isSelected(This, Item) -> boolean()
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Returns true if the item is selected.

```
isVisible(This, Item) -> boolean()
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Returns true if the item is visible on the screen.

```
itemHasChildren(This, Item) -> boolean()
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Returns true if the item has children.

```
isTreeItemIdOk(Item) -> boolean()
```

Types:

```
   Item = integer()
```

Returns true if the item is valid.

```
prependItem(This, Parent, Text) -> integer()
```

Types:

```
   This = wxTreeCtrl()
   Parent = integer()
   Text = unicode:chardata()
```

```
prependItem(This, Parent, Text, Options :: [Option]) -> integer()
```

Types:

```
   This = wxTreeCtrl()
   Parent = integer()
   Text = unicode:chardata()
   Option =
       {image, integer()} |
       {selectedImage, integer()} |
       {data, term()}
```

Appends an item as the first child of `parent`, return a new item id.

The `image` and `selImage` parameters are an index within the normal image list specifying the image to use for unselected and selected items, respectively. If `image` > -1 and `selImage` is -1, the same image is used for both selected and unselected items.

```
scrollTo(This, Item) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
```

Scrolls the specified item into view.

Note that this method doesn't work while the window is frozen (See `wxWindow:freeze/1`), at least under MSW.

See: `ensureVisible/2`

```
selectItem(This, Item) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()

selectItem(This, Item, Options :: [Option]) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
   Option = {select, boolean()}
```

Selects the given item.

In multiple selection controls, can be also used to deselect a currently selected item if the value of `select` is false.

Notice that calling this method will generate `wxEVT_TREE_SEL_CHANGING` and `wxEVT_TREE_SEL_CHANGED` events and that the change could be vetoed by the former event handler.

```
setIndent(This, Indent) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Indent = integer()
```

Sets the indentation for the tree control.

```
setImageList(This, ImageList) -> ok
```

Types:

```
   This = wxTreeCtrl()
   ImageList = wxImageList:wxImageList()
```

Sets the normal image list.

The image list assigned with this method will `not` be deleted by `wxTreeCtrl`'s destructor, you must delete it yourself.

See: `assignImageList/2`

```
setItemBackgroundColour(This, Item, Col) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
   Col = wx:wx_colour()
```

Sets the colour of the item's background.

```
setItemBold(This, Item) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()

setItemBold(This, Item, Options :: [Option]) -> ok
```

Types:

```
This = wxTreeCtrl()
Item = integer()
Option = {bold, boolean()}
```

Makes item appear in bold font if `bold` parameter is true or resets it to the normal state.

See: `isBold/2`

```
setItemData(This, Item, Data) -> ok
```

Types:

```
This = wxTreeCtrl()
Item = integer()
Data = term()
```

Sets the item client data.

Notice that the client data previously associated with the `item` (if any) is `not` freed by this function and so calling this function multiple times for the same item will result in memory leaks unless you delete the old item data pointer yourself.

```
setItemDropHighlight(This, Item) -> ok
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

```
setItemDropHighlight(This, Item, Options :: [Option]) -> ok
```

Types:

```
This = wxTreeCtrl()
Item = integer()
Option = {highlight, boolean()}
```

Gives the item the visual feedback for Drag'n'Drop actions, which is useful if something is dragged from the outside onto the tree control (as opposed to a DnD operation within the tree control, which already is implemented internally).

```
setItemFont(This, Item, Font) -> ok
```

Types:

```
This = wxTreeCtrl()
Item = integer()
Font = wxFont:wxFont()
```

Sets the item's font.

All items in the tree should have the same height to avoid text clipping, so the fonts height should be the same for all of them, although font attributes may vary.

See: `setItemBold/3`

```
setItemHasChildren(This, Item) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()

setItemHasChildren(This, Item, Options :: [Option]) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
   Option = {has, boolean()}
```

Force appearance of the button next to the item.

This is useful to allow the user to expand the items which don't have any children now, but instead adding them only when needed, thus minimizing memory usage and loading time.

```
setItemImage(This, Item, Image) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = Image = integer()

setItemImage(This, Item, Image, Options :: [Option]) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = Image = integer()
   Option = {which, wx:wx_enum()}
```

Sets the specified item's image.

See `getItemImage/3` for the description of the `which` parameter.

```
setItemText(This, Item, Text) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
   Text = unicode:chardata()
```

Sets the item label.

```
setItemTextColour(This, Item, Col) -> ok
```

Types:

```
   This = wxTreeCtrl()
   Item = integer()
   Col = wx:wx_colour()
```

Sets the colour of the item's text.

```
setStateImageList(This, ImageList) -> ok
```

Types:

```
This = wxTreeCtrl()
ImageList = wxImageList:wxImageList()
```

Sets the state image list (from which application-defined state images are taken).

Image list assigned with this method will `not` be deleted by `wxTreeCtrl`'s destructor, you must delete it yourself.

See: `assignStateImageList/2`

`setWindowStyle(This, Styles) -> ok`

Types:

```
This = wxTreeCtrl()
Styles = integer()
```

Sets the mode flags associated with the display of the tree control.

The new mode takes effect immediately.

Note: Generic only; MSW ignores changes.

`sortChildren(This, Item) -> ok`

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Sorts the children of the given item using `OnCompareItems()` (not implemented in wx).

You should override that method to change the sort order (the default is ascending case-sensitive alphabetical order).

See: `wxTreeItemData` (not implemented in wx), `OnCompareItems()` (not implemented in wx)

`toggle(This, Item) -> ok`

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Toggles the given item between collapsed and expanded states.

`toggleItemSelection(This, Item) -> ok`

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Toggles the given item between selected and unselected states.

For multiselection controls only.

`unselect(This) -> ok`

Types:

```
This = wxTreeCtrl()
```

Removes the selection from the currently selected item (if any).

`unselectAll(This) -> ok`

Types:

```
This = wxTreeCtrl()
```

This function either behaves the same as `unselect/1` if the control doesn't have `wxTR_MULTIPLE` style, or removes the selection from all items if it does have this style.

```
unselectItem(This, Item) -> ok
```

Types:

```
This = wxTreeCtrl()
Item = integer()
```

Unselects the given item.

This works in multiselection controls only.

# wxTreeEvent

Erlang module

A tree event holds information about events associated with `wxTreeCtrl` objects.

To process input from a tree control, use these event handler macros to direct input to member functions that take a `wxTreeEvent` argument.

See: `wxTreeCtrl`

This class is derived (and can use functions) from: `wxNotifyEvent` `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxTreeEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxTreeEventType` to subscribe to events of this type.

## Data Types

```
wxTreeEvent() = wx:wx_object()
wxTree() =
    #wxTree{type = wxTreeEvent:wxTreeEventType(),
            item = integer(),
            itemOld = integer(),
            pointDrag = {X :: integer(), Y :: integer()}}
wxTreeEventType() =
    command_tree_begin_drag | command_tree_begin_rdrag |
    command_tree_begin_label_edit | command_tree_end_label_edit |
    command_tree_delete_item | command_tree_get_info |
    command_tree_set_info | command_tree_item_expanded |
    command_tree_item_expanding | command_tree_item_collapsed |
    command_tree_item_collapsing | command_tree_sel_changed |
    command_tree_sel_changing | command_tree_key_down |
    command_tree_item_activated | command_tree_item_right_click |
    command_tree_item_middle_click | command_tree_end_drag |
    command_tree_state_image_click |
    command_tree_item_gettooltip | command_tree_item_menu |
    dirctrl_selectionchanged | dirctrl_fileactivated
```

## Exports

```
getKeyCode(This) -> integer()
```

Types:

```
   This = wxTreeEvent()
```

Returns the key code if the event is a key event.

Use `getKeyEvent/1` to get the values of the modifier keys for this event (i.e. Shift or Ctrl).

```
getItem(This) -> integer()
```

Types:

```
This = wxTreeEvent()
```

Returns the item (valid for all events).

```
getKeyEvent(This) -> wxKeyEvent:wxKeyEvent()
```

Types:

```
This = wxTreeEvent()
```

Returns the key event for EVT_TREE_KEY_DOWN events.

```
getLabel(This) -> unicode:charlist()
```

Types:

```
This = wxTreeEvent()
```

Returns the label if the event is a begin or end edit label event.

```
getOldItem(This) -> integer()
```

Types:

```
This = wxTreeEvent()
```

Returns the old item index (valid for EVT_TREE_SEL_CHANGING and EVT_TREE_SEL_CHANGED events).

```
getPoint(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxTreeEvent()
```

Returns the position of the mouse pointer if the event is a drag or menu-context event.

In both cases the position is in client coordinates - i.e. relative to the wxTreeCtrl window (so that you can pass it directly to e.g. wxWindow:popupMenu/4).

```
isEditCancelled(This) -> boolean()
```

Types:

```
This = wxTreeEvent()
```

Returns true if the label edit was cancelled.

This should be called from within an EVT_TREE_END_LABEL_EDIT handler.

```
setToolTip(This, Tooltip) -> ok
```

Types:

```
This = wxTreeEvent()
Tooltip = unicode:chardata()
```

Set the tooltip for the item (valid for EVT_TREE_ITEM_GETTOOLTIP events).

Windows only.

# wxTreebook

Erlang module

This class is an extension of the `wxNotebook` class that allows a tree structured set of pages to be shown in a control. A classic example is a netscape preferences dialog that shows a tree of preference sections on the left and select section page on the right.

To use the class simply create it and populate with pages using `insertPage/5`, `insertSubPage/5`, `addPage/4`, `AddSubPage()` (not implemented in wx).

If your tree is no more than 1 level in depth then you could simply use `addPage/4` and `AddSubPage()` (not implemented in wx) to sequentially populate your tree by adding at every step a page or a subpage to the end of the tree.

See: ?wxBookCtrl, `wxBookCtrlEvent`, `wxNotebook`, `wxTreeCtrl`, `wxImageList`, **Overview bookctrl**, **Examples**

This class is derived (and can use functions) from: `wxBookCtrlBase` `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxTreebook**

## Events

Event types emitted from this class: `treebook_page_changed`, `treebook_page_changing`

## Data Types

```
wxTreebook() = wx:wx_object()
```

## Exports

```
new() -> wxTreebook()
```

Default constructor.

```
new(Parent, Id) -> wxTreebook()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxTreebook()
```

Types:

```
Parent = wxWindow:wxWindow()
Id = integer()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Creates an empty `wxTreebook`.

```
destroy(This :: wxTreebook()) -> ok
```

Destroys the `wxTreebook` object.

Also deletes all the pages owned by the control (inserted previously into it).

```
addPage(This, Page, Text) -> boolean()
```
Types:
```
   This = wxTreebook()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
```

```
addPage(This, Page, Text, Options :: [Option]) -> boolean()
```
Types:
```
   This = wxTreebook()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
   Option = {bSelect, boolean()} | {imageId, integer()}
```
Adds a new page.

The page is placed at the topmost level after all other pages. NULL could be specified for page to create an empty page.

```
advanceSelection(This) -> ok
```
Types:
```
   This = wxTreebook()
```

```
advanceSelection(This, Options :: [Option]) -> ok
```
Types:
```
   This = wxTreebook()
   Option = {forward, boolean()}
```
Cycles through the tabs.

The call to this function generates the page changing events.

```
assignImageList(This, ImageList) -> ok
```
Types:
```
   This = wxTreebook()
   ImageList = wxImageList:wxImageList()
```
Sets the image list for the page control and takes ownership of the list.

See: `wxImageList`, `setImageList/2`

```
create(This, Parent, Id) -> boolean()
```
Types:
```
   This = wxTreebook()
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
create(This, Parent, Id, Options :: [Option]) -> boolean()
```
Types:

```
This = wxTreebook()
Parent = wxWindow:wxWindow()
Id = integer()
Option =
    {pos, {X :: integer(), Y :: integer()}} |
    {size, {W :: integer(), H :: integer()}} |
    {style, integer()}
```

Creates a treebook control.

See `new/3` for the description of the parameters.

```
deleteAllPages(This) -> boolean()
```

Types:

```
This = wxTreebook()
```

Deletes all pages.

```
getCurrentPage(This) -> wxWindow:wxWindow()
```

Types:

```
This = wxTreebook()
```

Returns the currently selected page or NULL.

```
getImageList(This) -> wxImageList:wxImageList()
```

Types:

```
This = wxTreebook()
```

Returns the associated image list, may be NULL.

See: `wxImageList`, `setImageList/2`

```
getPage(This, Page) -> wxWindow:wxWindow()
```

Types:

```
This = wxTreebook()
Page = integer()
```

Returns the window at the given page position.

```
getPageCount(This) -> integer()
```

Types:

```
This = wxTreebook()
```

Returns the number of pages in the control.

```
getPageImage(This, NPage) -> integer()
```

Types:

```
This = wxTreebook()
NPage = integer()
```

Returns the image index for the given page.

```
getPageText(This, NPage) -> unicode:charlist()
```

Types:

```
This = wxTreebook()
NPage = integer()
```

Returns the string for the given page.

```
getSelection(This) -> integer()
```

Types:

```
This = wxTreebook()
```

Returns the currently selected page, or `wxNOT_FOUND` if none was selected.

Note: This method may return either the previously or newly selected page when called from the EVT_TREEBOOK_PAGE_CHANGED() handler depending on the platform and so `wxBookCtrlEvent:getSelection/1` should be used instead in this case.

```
expandNode(This, PageId) -> boolean()
```

Types:

```
This = wxTreebook()
PageId = integer()
```

```
expandNode(This, PageId, Options :: [Option]) -> boolean()
```

Types:

```
This = wxTreebook()
PageId = integer()
Option = {expand, boolean()}
```

Expands (collapses) the `pageId` node.

Returns the previous state. May generate page changing events (if selected page is under the collapsed branch, then its parent is autoselected).

```
isNodeExpanded(This, PageId) -> boolean()
```

Types:

```
This = wxTreebook()
PageId = integer()
```

Returns true if the page represented by `pageId` is expanded.

```
hitTest(This, Pt) -> Result
```

Types:

```
Result = {Res :: integer(), Flags :: integer()}
This = wxTreebook()
Pt = {X :: integer(), Y :: integer()}
```

Returns the index of the tab at the specified position or `wxNOT_FOUND` if none.

If `flags` parameter is non-NULL, the position of the point inside the tab is returned as well.

Return: Returns the zero-based tab index or `wxNOT_FOUND` if there is no tab at the specified position.

```
insertPage(This, PagePos, Page, Text) -> boolean()
```

Types:

```
   This = wxTreebook()
   PagePos = integer()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
```

```
insertPage(This, PagePos, Page, Text, Options :: [Option]) ->
              boolean()
```

Types:

```
   This = wxTreebook()
   PagePos = integer()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
   Option = {bSelect, boolean()} | {imageId, integer()}
```

Inserts a new page just before the page indicated by `pagePos`.

The new page is placed before `pagePos` page and on the same level. NULL could be specified for page to create an empty page.

```
insertSubPage(This, PagePos, Page, Text) -> boolean()
```

Types:

```
   This = wxTreebook()
   PagePos = integer()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
```

```
insertSubPage(This, PagePos, Page, Text, Options :: [Option]) ->
                 boolean()
```

Types:

```
   This = wxTreebook()
   PagePos = integer()
   Page = wxWindow:wxWindow()
   Text = unicode:chardata()
   Option = {bSelect, boolean()} | {imageId, integer()}
```

Inserts a sub page under the specified page.

NULL could be specified for page to create an empty page.

```
setImageList(This, ImageList) -> ok
```

Types:

```
   This = wxTreebook()
   ImageList = wxImageList:wxImageList()
```

Sets the image list to use.

It does not take ownership of the image list, you must delete it yourself.

See: wxImageList, assignImageList/2

setPageSize(This, Size) -> ok

Types:

```
This = wxTreebook()
Size = {W :: integer(), H :: integer()}
```

Sets the width and height of the pages.

Note: This method is currently not implemented for wxGTK.

setPageImage(This, Page, Image) -> boolean()

Types:

```
This = wxTreebook()
Page = Image = integer()
```

Sets the image index for the given page.

image is an index into the image list which was set with setImageList/2.

setPageText(This, Page, Text) -> boolean()

Types:

```
This = wxTreebook()
Page = integer()
Text = unicode:chardata()
```

Sets the text for the given page.

setSelection(This, Page) -> integer()

Types:

```
This = wxTreebook()
Page = integer()
```

Sets the selection to the given page, returning the previous selection.

Notice that the call to this function generates the page changing events, use the changeSelection/2 function if you don't want these events to be generated.

See: wxBookCtrlBase:getSelection/1

changeSelection(This, Page) -> integer()

Types:

```
This = wxTreebook()
Page = integer()
```

Changes the selection to the given page, returning the previous selection.

This function behaves as setSelection/2 but does not generate the page changing events.

See overview_events_prog for more information.

# wxUpdateUIEvent

Erlang module

This class is used for pseudo-events which are called by wxWidgets to give an application the chance to update various user interface elements.

Without update UI events, an application has to work hard to check/uncheck, enable/disable, show/hide, and set the text for elements such as menu items and toolbar buttons. The code for doing this has to be mixed up with the code that is invoked when an action is invoked for a menu item or button.

With update UI events, you define an event handler to look at the state of the application and change UI elements accordingly. wxWidgets will call your member functions in idle time, so you don't have to worry where to call this code.

In addition to being a clearer and more declarative method, it also means you don't have to worry whether you're updating a toolbar or menubar identifier. The same handler can update a menu item and toolbar button, if the identifier is the same. Instead of directly manipulating the menu or button, you call functions in the event object, such as `check/2`. wxWidgets will determine whether such a call has been made, and which UI element to update.

These events will work for popup menus as well as menubars. Just before a menu is popped up, `wxMenu::UpdateUI` (not implemented in wx) is called to process any UI events for the window that owns the menu.

If you find that the overhead of UI update processing is affecting your application, you can do one or both of the following:

Note that although events are sent in idle time, defining a `wxIdleEvent` handler for a window does not affect this because the events are sent from `wxWindow::OnInternalIdle` (not implemented in wx) which is always called in idle time.

wxWidgets tries to optimize update events on some platforms. On Windows and GTK+, events for menubar items are only sent when the menu is about to be shown, and not in idle time.

See: **Overview events**

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxUpdateUIEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxUpdateUIEventType` to subscribe to events of this type.

## Data Types

```
wxUpdateUIEvent() = wx:wx_object()
wxUpdateUI() =
    #wxUpdateUI{type = wxUpdateUIEvent:wxUpdateUIEventType()}
wxUpdateUIEventType() = update_ui
```

## Exports

```
canUpdate(Window) -> boolean()
```

Types:

```
Window = wxWindow:wxWindow()
```

Returns true if it is appropriate to update (send UI update events to) this window.

This function looks at the mode used (see `setMode/1`), the wxWS_EX_PROCESS_UI_UPDATES flag in `window`, the time update events were last sent in idle time, and the update interval, to determine whether events should be sent to this window now. By default this will always return true because the update mode is initially wxUPDATE_UI_PROCESS_ALL and the interval is set to 0; so update events will be sent as often as possible. You can reduce the frequency that events are sent by changing the mode and/or setting an update interval.

See: `resetUpdateTime/0`, `setUpdateInterval/1`, `setMode/1`

```
check(This, Check) -> ok
```

Types:

```
   This = wxUpdateUIEvent()
   Check = boolean()
```

Check or uncheck the UI element.

```
enable(This, Enable) -> ok
```

Types:

```
   This = wxUpdateUIEvent()
   Enable = boolean()
```

Enable or disable the UI element.

```
show(This, Show) -> ok
```

Types:

```
   This = wxUpdateUIEvent()
   Show = boolean()
```

Show or hide the UI element.

```
getChecked(This) -> boolean()
```

Types:

```
   This = wxUpdateUIEvent()
```

Returns true if the UI element should be checked.

```
getEnabled(This) -> boolean()
```

Types:

```
   This = wxUpdateUIEvent()
```

Returns true if the UI element should be enabled.

```
getShown(This) -> boolean()
```

Types:

```
   This = wxUpdateUIEvent()
```

Returns true if the UI element should be shown.

```
getSetChecked(This) -> boolean()
```

Types:

```
This = wxUpdateUIEvent()
```

Returns true if the application has called `check/2`.

For wxWidgets internal use only.

`getSetEnabled(This) -> boolean()`

Types:

```
This = wxUpdateUIEvent()
```

Returns true if the application has called `enable/2`.

For wxWidgets internal use only.

`getSetShown(This) -> boolean()`

Types:

```
This = wxUpdateUIEvent()
```

Returns true if the application has called `show/2`.

For wxWidgets internal use only.

`getSetText(This) -> boolean()`

Types:

```
This = wxUpdateUIEvent()
```

Returns true if the application has called `setText/2`.

For wxWidgets internal use only.

`getText(This) -> unicode:charlist()`

Types:

```
This = wxUpdateUIEvent()
```

Returns the text that should be set for the UI element.

`getMode() -> wx:wx_enum()`

Static function returning a value specifying how wxWidgets will send update events: to all windows, or only to those which specify that they will process the events.

See: `setMode/1`

`getUpdateInterval() -> integer()`

Returns the current interval between updates in milliseconds.

The value -1 disables updates, 0 updates as frequently as possible.

See: `setUpdateInterval/1`

`resetUpdateTime() -> ok`

Used internally to reset the last-updated time to the current time.

It is assumed that update events are normally sent in idle time, so this is called at the end of idle processing.

See: `canUpdate/1`, `setUpdateInterval/1`, `setMode/1`

```
setMode(Mode) -> ok
```

Types:

```
Mode = wx:wx_enum()
```

Specify how wxWidgets will send update events: to all windows, or only to those which specify that they will process the events.

```
setText(This, Text) -> ok
```

Types:

```
This = wxUpdateUIEvent()
Text = unicode:chardata()
```

Sets the text for this UI element.

```
setUpdateInterval(UpdateInterval) -> ok
```

Types:

```
UpdateInterval = integer()
```

Sets the interval between updates in milliseconds.

Set to -1 to disable updates, or to 0 to update as frequently as possible. The default is 0.

Use this to reduce the overhead of UI update events if your application has a lot of windows. If you set the value to -1 or greater than 0, you may also need to call `wxWindow:updateWindowUI/2` at appropriate points in your application, such as when a dialog is about to be shown.

# wxWebView

Erlang module

This control may be used to render web (HTML / CSS / javascript) documents. It is designed to allow the creation of multiple backends for each port, although currently just one is available. It differs from `wxHtmlWindow` in that each backend is actually a full rendering engine, Trident on MSW and Webkit on macOS and GTK. This allows the correct viewing of complex pages with javascript and css.

Backend Descriptions

Par: The IE backend uses Microsoft's Trident rendering engine, specifically the version used by the locally installed copy of Internet Explorer. As such it is only available for the MSW port. By default recent versions of the **WebBrowser** control, which this backend uses, emulate Internet Explorer 7. This can be changed with a registry setting by wxWebView::MSWSetEmulationLevel() see **this** article for more information. This backend has full support for custom schemes and virtual file systems.

Par: The Edge (Chromium) backend uses Microsoft's **Edge WebView2**. It is available for Windows 7 and newer. The following features are currently unsupported with this backend: virtual filesystems, custom urls, find.

This backend is not enabled by default, to build it follow these steps:

Par: Under GTK the WebKit backend uses **WebKitGTK+**. The current minimum version required is 1.3.1 which ships by default with Ubuntu Natty and Debian Wheezy and has the package name libwebkitgtk-dev. Custom schemes and virtual files systems are supported under this backend, however embedded resources such as images and stylesheets are currently loaded using the data:// scheme.

Par: Under GTK3 the WebKit2 version of **WebKitGTK+** is used. In Ubuntu the required package name is libwebkit2gtk-4.0-dev and under Fedora it is webkitgtk4-devel. All wxWEBVIEW_WEBKIT features are supported except for clearing and enabling / disabling the history.

Par: The macOS WebKit backend uses Apple's **WebView** class. This backend has full support for custom schemes and virtual file systems.

Asynchronous Notifications

Many of the methods in `wxWebView` are asynchronous, i.e. they return immediately and perform their work in the background. This includes functions such as `loadURL/2` and `reload/2`. To receive notification of the progress and completion of these functions you need to handle the events that are provided. Specifically `wxEVT_WEBVIEW_LOADED` notifies when the page or a sub-frame has finished loading and `wxEVT_WEBVIEW_ERROR` notifies that an error has occurred.

Virtual File Systems and Custom Schemes

`wxWebView` supports the registering of custom scheme handlers, for example `file` or `http`. To do this create a new class which inherits from `wxWebViewHandler` (not implemented in wx), where wxWebHandler::GetFile() returns a pointer to a `wxFSFile` (not implemented in wx) which represents the given url. You can then register your handler with `RegisterHandler()` (not implemented in wx) it will be called for all pages and resources.

`wxWebViewFSHandler` (not implemented in wx) is provided to access the virtual file system encapsulated by `wxFileSystem` (not implemented in wx). The `wxMemoryFSHandler` (not implemented in wx) documentation gives an example of how this may be used.

`wxWebViewArchiveHandler` (not implemented in wx) is provided to allow the navigation of pages inside a zip archive. It supports paths of the form: `scheme:///C`:/example/docs.zip;protocol=zip/main.htm

Since: 2.9.3

See: `wxWebViewHandler` (not implemented in wx), `wxWebViewEvent`

This class is derived (and can use functions) from: `wxControl` `wxWindow` `wxEvtHandler`

wxWidgets docs: **wxWebView**

## Events

Event types emitted from this class: `webview_navigating`, `webview_navigated`, `webview_loaded`, `webview_error`, `webview_newwindow`, `webview_title_changed`

## Data Types

```
wxWebView() = wx:wx_object()
```

## Exports

```
new(Parent, Id) -> wxWebView()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
```

```
new(Parent, Id, Options :: [Option]) -> wxWebView()
```

Types:

```
   Parent = wxWindow:wxWindow()
   Id = integer()
   Option =
       {url, unicode:chardata()} |
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {backend, unicode:chardata()} |
       {style, integer()}
```

Factory function to create a new `wxWebView` using a `wxWebViewFactory` (not implemented in wx).

Return: The created `wxWebView`, or `NULL` if the requested backend is not available

Since: 2.9.5

```
getCurrentTitle(This) -> unicode:charlist()
```

Types:

```
   This = wxWebView()
```

Get the title of the current web page, or its URL/path if title is not available.

```
getCurrentURL(This) -> unicode:charlist()
```

Types:

```
   This = wxWebView()
```

Get the URL of the currently displayed document.

```
getPageSource(This) -> unicode:charlist()
```

Types:

```
This = wxWebView()
```

Get the HTML source code of the currently displayed document.

Return: The HTML source code, or an empty string if no page is currently shown.

```
getPageText(This) -> unicode:charlist()
```

Types:

```
This = wxWebView()
```

Get the text of the current page.

```
isBusy(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns whether the web control is currently busy (e.g. loading a page).

```
isEditable(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns whether the web control is currently editable.

```
loadURL(This, Url) -> ok
```

Types:

```
This = wxWebView()
Url = unicode:chardata()
```

Load a web page from a URL.

Note: Web engines generally report errors asynchronously, so if you wish to know whether loading the URL was successful, register to receive navigation error events.

```
print(This) -> ok
```

Types:

```
This = wxWebView()
```

Opens a print dialog so that the user may print the currently displayed page.

```
reload(This) -> ok
```

Types:

```
This = wxWebView()
```

```
reload(This, Options :: [Option]) -> ok
```

Types:

```
This = wxWebView()
Option = {flags, wx:wx_enum()}
```

Reload the currently displayed URL.

Note: The flags are ignored by the edge backend.

```
runScript(This, Javascript) -> Result
```

Types:

```
Result = {Res :: boolean(), Output :: unicode:charlist()}
This = wxWebView()
Javascript = unicode:chardata()
```

Runs the given JavaScript code.

JavaScript code is executed inside the browser control and has full access to DOM and other browser-provided functionality. For example, this code will replace the current page contents with the provided string.

If `output` is non-null, it is filled with the result of executing this code on success, e.g. a JavaScript value such as a string, a number (integer or floating point), a boolean or JSON representation for non-primitive types such as arrays and objects. For example:

This function has a few platform-specific limitations:

Also notice that under MSW converting JavaScript objects to JSON is not supported in the default emulation mode. `wxWebView` implements its own object-to-JSON conversion as a fallback for this case, however it is not as full-featured, well-tested or performing as the implementation of this functionality in the browser control itself, so it is recommended to use MSWSetEmulationLevel() to change emulation level to a more modern one in which JSON conversion is done by the control itself.

Return: true if there is a result, false if there is an error.

```
setEditable(This) -> ok
```

Types:

```
This = wxWebView()
```

```
setEditable(This, Options :: [Option]) -> ok
```

Types:

```
This = wxWebView()
Option = {enable, boolean()}
```

Set the editable property of the web control.

Enabling allows the user to edit the page even if the `contenteditable` attribute is not set. The exact capabilities vary with the backend being used.

```
setPage(This, Html, BaseUrl) -> ok
```

Types:

```
This = wxWebView()
Html = BaseUrl = unicode:chardata()
```

Set the displayed page source to the contents of the given string.

Note: When using `wxWEBVIEW_BACKEND_IE` you must wait for the current page to finish loading before calling `setPage/3`. The baseURL parameter is not used in this backend and the edge backend.

```
stop(This) -> ok
```

Types:

```
This = wxWebView()
```

Stop the current page loading process, if any.

May trigger an error event of type `wxWEBVIEW_NAV_ERR_USER_CANCELLED`. TODO: make `wxWEBVIEW_NAV_ERR_USER_CANCELLED` errors uniform across ports.

```
canCopy(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns true if the current selection can be copied.

Note: This always returns `true` on the macOS WebKit backend.

```
canCut(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns true if the current selection can be cut.

Note: This always returns `true` on the macOS WebKit backend.

```
canPaste(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns true if data can be pasted.

Note: This always returns `true` on the macOS WebKit backend.

```
copy(This) -> ok
```

Types:

```
This = wxWebView()
```

Copies the current selection.

```
cut(This) -> ok
```

Types:

```
This = wxWebView()
```

Cuts the current selection.

```
paste(This) -> ok
```

Types:

```
This = wxWebView()
```

Pastes the current data.

```
enableContextMenu(This) -> ok
```

Types:

```
This = wxWebView()
```

```
enableContextMenu(This, Options :: [Option]) -> ok
```

Types:

```
This = wxWebView()
Option = {enable, boolean()}
```

Enable or disable the right click context menu.

By default the standard context menu is enabled, this method can be used to disable it or re-enable it later.

Since: 2.9.5

```
isContextMenuEnabled(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns true if a context menu will be shown on right click.

Since: 2.9.5

```
canGoBack(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns true if it is possible to navigate backward in the history of visited pages.

```
canGoForward(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns true if it is possible to navigate forward in the history of visited pages.

```
clearHistory(This) -> ok
```

Types:

```
This = wxWebView()
```

Clear the history, this will also remove the visible page.

Note: This is not implemented on the WebKit2GTK+ backend.

```
enableHistory(This) -> ok
```

Types:

```
This = wxWebView()
```

```
enableHistory(This, Options :: [Option]) -> ok
```

Types:

```
This = wxWebView()
Option = {enable, boolean()}
```

Enable or disable the history.

This will also clear the history.

Note: This is not implemented on the WebKit2GTK+ backend.

```
goBack(This) -> ok
```

Types:

```
This = wxWebView()
```

Navigate back in the history of visited pages.

Only valid if `canGoBack/1` returns true.

```
goForward(This) -> ok
```

Types:

```
This = wxWebView()
```

Navigate forward in the history of visited pages.

Only valid if `canGoForward/1` returns true.

```
clearSelection(This) -> ok
```

Types:

```
This = wxWebView()
```

Clears the current selection.

```
deleteSelection(This) -> ok
```

Types:

```
This = wxWebView()
```

Deletes the current selection.

Note that for `wxWEBVIEW_BACKEND_WEBKIT` the selection must be editable, either through SetEditable or the correct HTML attribute.

```
getSelectedSource(This) -> unicode:charlist()
```

Types:

```
This = wxWebView()
```

Returns the currently selected source, if any.

```
getSelectedText(This) -> unicode:charlist()
```

Types:

```
This = wxWebView()
```

Returns the currently selected text, if any.

```
hasSelection(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns true if there is a current selection.

```
selectAll(This) -> ok
```

Types:

```
This = wxWebView()
```

Selects the entire page.

```
canRedo(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns true if there is an action to redo.

```
canUndo(This) -> boolean()
```

Types:

```
This = wxWebView()
```

Returns true if there is an action to undo.

```
redo(This) -> ok
```

Types:

```
This = wxWebView()
```

Redos the last action.

```
undo(This) -> ok
```

Types:

```
This = wxWebView()
```

Undos the last action.

```
find(This, Text) -> integer()
```

Types:

```
This = wxWebView()
Text = unicode:chardata()
```

```
find(This, Text, Options :: [Option]) -> integer()
```

Types:

```
This = wxWebView()
Text = unicode:chardata()
Option = {flags, wx:wx_enum()}
```

Finds a phrase on the current page and if found, the control will scroll the phrase into view and select it.

Return: If search phrase was not found in combination with the flags then `wxNOT_FOUND` is returned. If called for the first time with search phrase then the total number of results will be returned. Then for every time its called with the same search phrase it will return the number of the current match.

Note: This function will restart the search if the flags `wxWEBVIEW_FIND_ENTIRE_WORD` or `wxWEBVIEW_FIND_MATCH_CASE` are changed, since this will require a new search. To reset the search, for example resetting the highlights call the function with an empty search phrase. This always returns `wxNOT_FOUND` on the macOS WebKit backend.

Since: 2.9.5

```
canSetZoomType(This, Type) -> boolean()
```

Types:

```
This = wxWebView()
Type = wx:wx_enum()
```

Retrieve whether the current HTML engine supports a zoom type.

Return: Whether this type of zoom is supported by this HTML engine (and thus can be set through `setZoomType/2`).

```
getZoom(This) -> wx:wx_enum()
```

Types:

```
This = wxWebView()
```

Get the zoom level of the page.

See `getZoomFactor/1` to get more precise zoom scale value other than as provided by `wxWebViewZoom`.

Return: The current level of zoom.

```
getZoomType(This) -> wx:wx_enum()
```

Types:

```
This = wxWebView()
```

Get how the zoom factor is currently interpreted.

Return: How the zoom factor is currently interpreted by the HTML engine.

```
setZoom(This, Zoom) -> ok
```

Types:

```
This = wxWebView()
Zoom = wx:wx_enum()
```

Set the zoom level of the page.

See `setZoomFactor/2` for more precise scaling other than the measured steps provided by `wxWebViewZoom`.

```
setZoomType(This, ZoomType) -> ok
```

Types:

```
This = wxWebView()
ZoomType = wx:wx_enum()
```

Set how to interpret the zoom factor.

Note: invoke `canSetZoomType/2` first, some HTML renderers may not support all zoom types.

```
getZoomFactor(This) -> number()
```

Types:

```
This = wxWebView()
```

Get the zoom factor of the page.

Return: The current factor of zoom.

Since: 3.1.4

```
setZoomFactor(This, Zoom) -> ok
```

Types:

```
This = wxWebView()
Zoom = number()
```

Set the zoom factor of the page.

Note: zoom scale in IE will be converted into `wxWebViewZoom` levels for `wxWebViewZoomType` of `wxWEBVIEW_ZOOM_TYPE_TEXT`.

Since: 3.1.4

```
isBackendAvailable(Backend) -> boolean()
```

Types:

```
Backend = unicode:chardata()
```

Allows to check if a specific backend is currently available.

Since: 3.1.4

# wxWebViewEvent

Erlang module

A navigation event holds information about events associated with `wxWebView` objects.

Since: 2.9.3

See: `wxWebView`

This class is derived (and can use functions) from: `wxNotifyEvent` `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxWebViewEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxWebViewEventType` to subscribe to events of this type.

## Data Types

```
wxWebViewEvent() = wx:wx_object()
wxWebView() =
    #wxWebView{type = wxWebViewEvent:wxWebViewEventType(),
               string = unicode:chardata(),
               int = integer(),
               target = unicode:chardata(),
               url = unicode:chardata()}
wxWebViewEventType() =
    webview_navigating | webview_navigated | webview_loaded |
    webview_error | webview_newwindow | webview_title_changed
```

## Exports

```
getString(This) -> unicode:charlist()
```

Types:

```
   This = wxWebViewEvent()
```

Returns item string for a listbox or choice selection event.

If one or several items have been deselected, returns the index of the first deselected item. If some items have been selected and others deselected at the same time, it will return the index of the first selected item.

```
getInt(This) -> integer()
```

Types:

```
   This = wxWebViewEvent()
```

Returns the integer identifier corresponding to a listbox, choice or radiobox selection (only if the event was a selection, not a deselection), or a boolean value representing the value of a checkbox.

For a menu item, this method returns -1 if the item is not checkable or a boolean value (true or false) for checkable items indicating the new state of the item.

```
getTarget(This) -> unicode:charlist()
```

Types:

```
This = wxWebViewEvent()
```

Get the name of the target frame which the url of this event has been or will be loaded into.

This may return an empty string if the frame is not available.

```
getURL(This) -> unicode:charlist()
```

Types:

```
This = wxWebViewEvent()
```

Get the URL being visited.

# wxWindow

Erlang module

`wxWindow` is the base class for all windows and represents any visible object on screen. All controls, top level windows and so on are windows. Sizers and device contexts are not, however, as they don't appear on screen themselves.

Please note that all children of the window will be deleted automatically by the destructor before the window itself is deleted which means that you don't have to worry about deleting them manually. Please see the window deletion overview for more information.

Also note that in this, and many others, wxWidgets classes some `GetXXX()` methods may be overloaded (as, for example, `getSize/1` or `getClientSize/1`). In this case, the overloads are non-virtual because having multiple virtual functions with the same name results in a virtual function name hiding at the derived class level (in English, this means that the derived class has to override all overloaded variants if it overrides any of them). To allow overriding them in the derived class, wxWidgets uses a unique protected virtual `DoGetXXX()` method and all `GetXXX()` ones are forwarded to it, so overriding the former changes the behaviour of the latter.

Styles

This class supports the following styles:

Extra Styles

This class supports the following extra styles:

See: **Overview events**, **Overview windowsizing**

This class is derived (and can use functions) from: `wxEvtHandler`

wxWidgets docs: **wxWindow**

## Events

Event types emitted from this class: `activate`, `child_focus`, `context_menu`, `help`, `drop_files`, `erase_background`, `set_focus`, `kill_focus`, `idle`, `joy_button_down`, `joy_button_up`, `joy_move`, `joy_zmove`, `key_down`, `key_up`, `char`, `char_hook`, `mouse_capture_lost`, `mouse_capture_changed`, `left_down`, `left_up`, `middle_down`, `middle_up`, `right_down`, `right_up`, `motion`, `enter_window`, `leave_window`, `left_dclick`, `middle_dclick`, `right_dclick`, `mousewheel`, `aux1_down`, `aux1_up`, `aux1_dclick`, `aux2_down`, `aux2_up`, `aux2_dclick`, `paint`, `scrollwin_top`, `scrollwin_bottom`, `scrollwin_lineup`, `scrollwin_linedown`, `scrollwin_pageup`, `scrollwin_pagedown`, `scrollwin_thumbtrack`, `scrollwin_thumbrelease`, `set_cursor`, `size`, `sys_colour_changed`

## Data Types

```
wxWindow() = wx:wx_object()
```

## Exports

```
new() -> wxWindow()
```

Default constructor.

```
new(Parent, Id) -> wxWindow()
```

Types:

```
   Parent = wxWindow()
   Id = integer()

new(Parent, Id, Options :: [Option]) -> wxWindow()
```

Types:

```
   Parent = wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Constructs a window, which can be a child of a frame, dialog or any other non-control window.

```
destroy(This :: wxWindow()) -> ok
```

Destructor.

Deletes all sub-windows, then deletes itself. Instead of using the `delete` operator explicitly, you should normally use `'Destroy'/1` so that wxWidgets can delete a window only when it is safe to do so, in idle time.

See: Window Deletion Overview, `'Destroy'/1`, `wxCloseEvent`

```
create(This, Parent, Id) -> boolean()
```

Types:

```
   This = Parent = wxWindow()
   Id = integer()

create(This, Parent, Id, Options :: [Option]) -> boolean()
```

Types:

```
   This = Parent = wxWindow()
   Id = integer()
   Option =
       {pos, {X :: integer(), Y :: integer()}} |
       {size, {W :: integer(), H :: integer()}} |
       {style, integer()}
```

Construct the actual window object after creating the C++ object.

The non-default constructor of `wxWindow` class does two things: it initializes the C++ object and it also creates the window object in the underlying graphical toolkit. The `create/4` method can be used to perform the second part later, while the default constructor can be used to perform the first part only.

Please note that the underlying window must be created exactly once, i.e. if you use the default constructor, which doesn't do this, you `must` call `create/4` before using the window and if you use the non-default constructor, you can `not` call `create/4`, as the underlying window is already created.

Note that it is possible and, in fact, useful, to call some methods on the object between creating the C++ object itself and calling `create/4` on it, e.g. a common pattern to avoid showing the contents of a window before it is fully initialized is:

Also note that it is possible to create an object of a derived type and then call `create/4` on it: This is notably used by overview_xrc.

The parameters of this method have exactly the same meaning as the non-default constructor parameters, please refer to them for their description.

Return: true if window creation succeeded or false if it failed

```
cacheBestSize(This, Size) -> ok
```

Types:

```
This = wxWindow()
Size = {W :: integer(), H :: integer()}
```

Sets the cached best size value.

See: `getBestSize/1`

```
captureMouse(This) -> ok
```

Types:

```
This = wxWindow()
```

Directs all mouse input to this window.

Call `releaseMouse/1` to release the capture.

Note that wxWidgets maintains the stack of windows having captured the mouse and when the mouse is released the capture returns to the window which had had captured it previously and it is only really released if there were no previous window. In particular, this means that you must release the mouse as many times as you capture it, unless the window receives the `wxMouseCaptureLostEvent` event.

Any application which captures the mouse in the beginning of some operation must handle `wxMouseCaptureLostEvent` and cancel this operation when it receives the event. The event handler must not recapture mouse.

See: `releaseMouse/1`, `wxMouseCaptureLostEvent`

```
center(This) -> ok
```

Types:

```
This = wxWindow()
```

```
centre(This) -> ok
```

Types:

```
This = wxWindow()
```

```
center(This, Options :: [Option]) -> ok
```

Types:

```
This = wxWindow()
Option = {dir, integer()}
```

See: `centre/2`.

```
centre(This, Options :: [Option]) -> ok
```

Types:

```
This = wxWindow()
Option = {dir, integer()}
```

Centres the window.

Remark: If the window is a top level one (i.e. doesn't have a parent), it will be centred relative to the screen anyhow.

See: `center/2`

```
centerOnParent(This) -> ok
```

Types:

```
This = wxWindow()
```

```
centreOnParent(This) -> ok
```

Types:

```
This = wxWindow()
```

```
centerOnParent(This, Options :: [Option]) -> ok
```

Types:

```
This = wxWindow()
Option = {dir, integer()}
```

See: `centreOnParent/2`.

```
centreOnParent(This, Options :: [Option]) -> ok
```

Types:

```
This = wxWindow()
Option = {dir, integer()}
```

Centres the window on its parent.

This is a more readable synonym for `centre/2`.

Remark: This methods provides for a way to centre top level windows over their parents instead of the entire screen. If there is no parent or if the window is not a top level window, then behaviour is the same as `centre/2`.

See: `wxTopLevelWindow:centreOnScreen/2`

```
clearBackground(This) -> ok
```

Types:

```
This = wxWindow()
```

Clears the window by filling it with the current background colour.

Does not cause an erase background event to be generated.

Notice that this uses `wxClientDC` to draw on the window and the results of doing it while also drawing on `wxPaintDC` for this window are undefined. Hence this method shouldn't be used from EVT_PAINT handlers, just use `wxDC:clear/1` on the `wxPaintDC` you already use there instead.

```
clientToScreen(This, Pt) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxWindow()
Pt = {X :: integer(), Y :: integer()}
```

Converts to screen coordinates from coordinates relative to this window.

```
clientToScreen(This, X, Y) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxWindow()
X = Y = integer()
```

Converts to screen coordinates from coordinates relative to this window.

```
close(This) -> boolean()
```

Types:

```
This = wxWindow()
```

```
close(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxWindow()
Option = {force, boolean()}
```

This function simply generates a `wxCloseEvent` whose handler usually tries to close the window.

It doesn't close the window itself, however.

Return: true if the event was handled and not vetoed, false otherwise.

Remark: Close calls the close handler for the window, providing an opportunity for the window to choose whether to destroy the window. Usually it is only used with the top level windows (`wxFrame` and `wxDialog` classes) as the others are not supposed to have any special OnClose() logic. The close handler should check whether the window is being deleted forcibly, using `wxCloseEvent:canVeto/1`, in which case it should destroy the window using `'Destroy'/1`. Note that calling Close does not guarantee that the window will be destroyed; but it provides a way to simulate a manual close of a window, which may or may not be implemented by destroying the window. The default implementation of wxDialog::OnCloseWindow does not necessarily delete the dialog, since it will simply simulate an wxID_CANCEL event which is handled by the appropriate button event handler and may do anything at all. To guarantee that the window will be destroyed, call `'Destroy'/1` instead

See: Window Deletion Overview, `'Destroy'/1`, `wxCloseEvent`

```
convertDialogToPixels(This, Sz) ->
                         {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
Sz = {W :: integer(), H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
convertPixelsToDialog(This, Sz) ->
                         {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
Sz = {W :: integer(), H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

`'Destroy'(This) -> boolean()`

Types:

```
This = wxWindow()
```

Destroys the window safely.

Use this function instead of the delete operator, since different window classes can be destroyed differently. Frames and dialogs are not destroyed immediately when this function is called - they are added to a list of windows to be deleted on idle time, when all the window's events have been processed. This prevents problems with events being sent to non-existent windows.

Return: true if the window has either been successfully deleted, or it has been added to the list of windows pending real deletion.

`destroyChildren(This) -> boolean()`

Types:

```
This = wxWindow()
```

Destroys all children of a window.

Called automatically by the destructor.

`disable(This) -> boolean()`

Types:

```
This = wxWindow()
```

Disables the window.

Same as `enable/2` Enable(false).

Return: Returns true if the window has been disabled, false if it had been already disabled before the call to this function.

`dragAcceptFiles(This, Accept) -> ok`

Types:

```
This = wxWindow()
Accept = boolean()
```

Enables or disables eligibility for drop file events (OnDropFiles).

Remark: Windows only until version 2.8.9, available on all platforms since 2.8.10. Cannot be used together with `setDropTarget/2` on non-Windows platforms.

See: `setDropTarget/2`

`enable(This) -> boolean()`

Types:

```
This = wxWindow()
```

```
enable(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxWindow()
Option = {enable, boolean()}
```

Enable or disable the window for user input.

Note that when a parent window is disabled, all of its children are disabled as well and they are re-enabled again when the parent is.

A window can be created initially disabled by calling this method on it `before` calling `create/4` to create the actual underlying window, e.g.

Return: Returns true if the window has been enabled or disabled, false if nothing was done, i.e. if the window had already been in the specified state.

See: `isEnabled/1`, `disable/1`, `wxRadioBox:enable/3`

```
findFocus() -> wxWindow()
```

Finds the window or control which currently has the keyboard focus.

Remark: Note that this is a static function, so it can be called without needing a `wxWindow` pointer.

See: `setFocus/1`, `HasFocus()` (not implemented in wx)

```
findWindow(This, Id) -> wxWindow()
findWindow(This, Name) -> wxWindow()
```

Types:

```
This = wxWindow()
Name = unicode:chardata()
```

Find a child of this window, by name.

May return `this` if it matches itself.

Notice that only real children, not top level windows using this window as parent, are searched by this function.

```
findWindowById(Id) -> wxWindow()
```

Types:

```
Id = integer()
```

```
findWindowById(Id, Options :: [Option]) -> wxWindow()
```

Types:

```
Id = integer()
Option = {parent, wxWindow()}
```

Find the first window with the given `id`.

If `parent` is NULL, the search will start from all top-level frames and dialog boxes; if non-NULL, the search will be limited to the given window hierarchy. The search is recursive in both cases.

See: `findWindow/2`

Return: Window with the given `id` or NULL if not found.

```
findWindowByName(Name) -> wxWindow()
```

Types:

```
Name = unicode:chardata()
```

```
findWindowByName(Name, Options :: [Option]) -> wxWindow()
```

Types:

```
Name = unicode:chardata()
Option = {parent, wxWindow()}
```

Find a window by its name (as given in a window constructor or `create/4` function call).

If `parent` is NULL, the search will start from all top-level frames and dialog boxes; if non-NULL, the search will be limited to the given window hierarchy.

The search is recursive in both cases and, unlike `findWindow/2`, recurses into top level child windows too.

If no window with such name is found, `findWindowByLabel/2` is called, i.e. the name is interpreted as (internal) name first but if this fails, it's internal as (user-visible) label. As this behaviour may be confusing, it is usually better to use either the `findWindow/2` overload taking the name or `findWindowByLabel/2` directly.

Return: Window with the given `name` or NULL if not found.

```
findWindowByLabel(Label) -> wxWindow()
```

Types:

```
Label = unicode:chardata()
```

```
findWindowByLabel(Label, Options :: [Option]) -> wxWindow()
```

Types:

```
Label = unicode:chardata()
Option = {parent, wxWindow()}
```

Find a window by its label.

Depending on the type of window, the label may be a window title or panel item label. If `parent` is NULL, the search will start from all top-level frames and dialog boxes; if non-NULL, the search will be limited to the given window hierarchy.

The search is recursive in both cases and, unlike with `findWindow/2`, recurses into top level child windows too.

See: `findWindow/2`

Return: Window with the given `label` or NULL if not found.

```
fit(This) -> ok
```

Types:

```
This = wxWindow()
```

Sizes the window to fit its best size.

Using this function is equivalent to setting window size to the return value of `getBestSize/1`.

Note that, unlike `setSizerAndFit/3`, this function only changes the current window size and doesn't change its minimal size.

See: **Overview windowsizing**

```
fitInside(This) -> ok
```

Types:

```
This = wxWindow()
```

Similar to `fit/1`, but sizes the interior (virtual) size of a window.

Mainly useful with scrolled windows to reset scrollbars after sizing changes that do not trigger a size event, and/or scrolled windows without an interior sizer. This function similarly won't do anything if there are no subwindows.

```
freeze(This) -> ok
```

Types:

```
This = wxWindow()
```

Freezes the window or, in other words, prevents any updates from taking place on screen, the window is not redrawn at all.

`thaw/1` must be called to re-enable window redrawing. Calls to these two functions may be nested but to ensure that the window is properly repainted again, you must thaw it exactly as many times as you froze it.

If the window has any children, they are recursively frozen too.

This method is useful for visual appearance optimization (for example, it is a good idea to use it before doing many large text insertions in a row into a `wxTextCtrl` under wxGTK) but is not implemented on all platforms nor for all controls so it is mostly just a hint to wxWidgets and not a mandatory directive.

See: `wxWindowUpdateLocker` (not implemented in wx), `thaw/1`, `isFrozen/1`

```
getAcceleratorTable(This) ->
                       wxAcceleratorTable:wxAcceleratorTable()
```

Types:

```
This = wxWindow()
```

Gets the accelerator table for this window.

See `wxAcceleratorTable`.

```
getBackgroundColour(This) -> wx:wx_colour4()
```

Types:

```
This = wxWindow()
```

Returns the background colour of the window.

See: `setBackgroundColour/2`, `setForegroundColour/2`, `getForegroundColour/1`

```
getBackgroundStyle(This) -> wx:wx_enum()
```

Types:

```
This = wxWindow()
```

Returns the background style of the window.

See: `setBackgroundColour/2`, `getForegroundColour/1`, `setBackgroundStyle/2`, `setTransparent/2`

```
getBestSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
```

This functions returns the best acceptable minimal size for the window.

For example, for a static control, it will be the minimal size such that the control label is not truncated. For windows containing subwindows (typically `wxPanel`), the size returned by this function will be the same as the size the window would have had after calling `fit/1`.

Override virtual `DoGetBestSize()` (not implemented in wx) or, better, because it's usually more convenient, `DoGetBestClientSize()` (not implemented in wx) when writing your own custom window class to change the value returned by this public non-virtual method.

Notice that the best size respects the minimal and maximal size explicitly set for the window, if any. So even if some window believes that it needs 200 pixels horizontally, calling `setMaxSize/2` with a width of 100 would ensure that `getBestSize/1` returns the width of at most 100 pixels.

See: `cacheBestSize/2`, **Overview windowsizing**

```
getCaret(This) -> wxCaret:wxCaret()
```

Types:

```
This = wxWindow()
```

Returns the caret() associated with the window.

```
getCapture() -> wxWindow()
```

Returns the currently captured window.

See: `hasCapture/1`, `captureMouse/1`, `releaseMouse/1`, `wxMouseCaptureLostEvent`, `wxMouseCaptureChangedEvent`

```
getCharHeight(This) -> integer()
```

Types:

```
This = wxWindow()
```

Returns the character height for this window.

```
getCharWidth(This) -> integer()
```

Types:

```
This = wxWindow()
```

Returns the average character width for this window.

```
getChildren(This) -> [wxWindow()]
```

Types:

```
This = wxWindow()
```

Returns a const reference to the list of the window's children.

`wxWindowList` is a type-safe wxList-like class whose elements are of type `wxWindow*`.

```
getClientSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
getContainingSizer(This) -> wxSizer:wxSizer()
```

Types:

```
This = wxWindow()
```

Returns the sizer of which this window is a member, if any, otherwise NULL.

```
getCursor(This) -> wxCursor:wxCursor()
```

Types:

```
This = wxWindow()
```

Return the cursor associated with this window.

See: `setCursor/2`

```
getDropTarget(This) -> wx:wx_object()
```

Types:

```
This = wxWindow()
```

Returns the associated drop target, which may be NULL.

See: `setDropTarget/2`, **Overview dnd**

```
getDPIScaleFactor(This) -> number()
```

Types:

```
This = wxWindow()
```

Returns the ratio of the DPI used by this window to the standard DPI.

The returned value is 1 for standard DPI screens or 2 for "200% scaling" and, unlike for `getContentScaleFactor/1`, is the same under all platforms.

This factor should be used to increase the size of icons and similar windows whose best size is not based on text metrics when using DPI scaling.

E.g. the program may load a 32px bitmap if the content scale factor is 1.0 or 64px version of the same bitmap if it is 2.0 or bigger.

Notice that this method should `not` be used for window sizes expressed in pixels, as they are already scaled by this factor by the underlying toolkit under some platforms. Use `fromDIP/2` for anything window-related instead.

Since: 3.1.4

```
getExtraStyle(This) -> integer()
```

Types:

```
This = wxWindow()
```

Returns the extra style bits for the window.

```
getFont(This) -> wxFont:wxFont()
```

Types:

```
This = wxWindow()
```

Returns the font for this window.

See: `setFont/2`

`getForegroundColour(This) -> wx:wx_colour4()`

Types:

```
This = wxWindow()
```

Returns the foreground colour of the window.

Remark: The meaning of foreground colour varies according to the window class; it may be the text colour or other colour, or it may not be used at all.

See: `setForegroundColour/2`, `setBackgroundColour/2`, `getBackgroundColour/1`

`getGrandParent(This) -> wxWindow()`

Types:

```
This = wxWindow()
```

Returns the grandparent of a window, or NULL if there isn't one.

`getHandle(This) -> integer()`

Types:

```
This = wxWindow()
```

Returns the platform-specific handle of the physical window.

Cast it to an appropriate handle, such as `HWND` for Windows, `Widget` for Motif or `GtkWidget` for GTK.

`getHelpText(This) -> unicode:charlist()`

Types:

```
This = wxWindow()
```

Gets the help text to be used as context-sensitive help for this window.

Note that the text is actually stored by the current `wxHelpProvider` (not implemented in wx) implementation, and not in the window object itself.

See: `setHelpText/2`, `GetHelpTextAtPoint()` (not implemented in wx), `wxHelpProvider` (not implemented in wx)

`getId(This) -> integer()`

Types:

```
This = wxWindow()
```

Returns the identifier of the window.

Remark: Each window has an integer identifier. If the application has not provided one (or the default wxID_ANY) a unique identifier with a negative value will be generated.

See: `setId/2`, **Overview windowids**

`getLabel(This) -> unicode:charlist()`

Types:

```
This = wxWindow()
```

Generic way of getting a label from any window, for identification purposes.

Remark: The interpretation of this function differs from class to class. For frames and dialogs, the value returned is the title. For buttons or static text controls, it is the button text. This function can be useful for meta-programs (such as testing tools or special-needs access programs) which need to identify windows by name.

```
getMaxSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
```

Returns the maximum size of the window.

This is an indication to the sizer layout mechanism that this is the maximum possible size as well as the upper bound on window's size settable using `setSize/6`.

See: `GetMaxClientSize()` (not implemented in wx), **Overview windowsizing**

```
getMinSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
```

Returns the minimum size of the window, an indication to the sizer layout mechanism that this is the minimum required size.

This method normally just returns the value set by `setMinSize/2`, but it can be overridden to do the calculation on demand.

See: `GetMinClientSize()` (not implemented in wx), **Overview windowsizing**

```
getName(This) -> unicode:charlist()
```

Types:

```
This = wxWindow()
```

Returns the window's name.

Remark: This name is not guaranteed to be unique; it is up to the programmer to supply an appropriate name in the window constructor or via `setName/2`.

See: `setName/2`

```
getParent(This) -> wxWindow()
```

Types:

```
This = wxWindow()
```

Returns the parent of the window, or NULL if there is no parent.

```
getPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxWindow()
```

This gets the position of the window in pixels, relative to the parent window for the child windows or relative to the display origin for the top level windows.

See: `getScreenPosition/1`

```
getRect(This) ->
           {X :: integer(),
            Y :: integer(),
            W :: integer(),
            H :: integer()}
```

Types:

```
This = wxWindow()
```

Returns the position and size of the window as a {X,Y,W,H} object.

See: `getScreenRect/1`

```
getScreenPosition(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxWindow()
```

Returns the window position in screen coordinates, whether the window is a child window or a top level one.

See: `getPosition/1`

```
getScreenRect(This) ->
                 {X :: integer(),
                  Y :: integer(),
                  W :: integer(),
                  H :: integer()}
```

Types:

```
This = wxWindow()
```

Returns the position and size of the window on the screen as a {X,Y,W,H} object.

See: `getRect/1`

```
getScrollPos(This, Orientation) -> integer()
```

Types:

```
This = wxWindow()
Orientation = integer()
```

Returns the built-in scrollbar position.

See: `setScrollbar/6`

```
getScrollRange(This, Orientation) -> integer()
```

Types:

```
This = wxWindow()
Orientation = integer()
```

Returns the built-in scrollbar range.

See: `setScrollbar/6`

```
getScrollThumb(This, Orientation) -> integer()
```

Types:

```
This = wxWindow()
Orientation = integer()
```

Returns the built-in scrollbar thumb size.

See: `setScrollbar/6`

```
getSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
```

See the GetSize(int*,int*) overload for more info.

```
getSizer(This) -> wxSizer:wxSizer()
```

Types:

```
This = wxWindow()
```

Returns the sizer associated with the window by a previous call to `setSizer/3`, or NULL.

```
getTextExtent(This, String) -> Result
```

Types:

```
Result =
    {W :: integer(),
     H :: integer(),
     Descent :: integer(),
     ExternalLeading :: integer()}
This = wxWindow()
String = unicode:chardata()
```

```
getTextExtent(This, String, Options :: [Option]) -> Result
```

Types:

```
Result =
    {W :: integer(),
     H :: integer(),
     Descent :: integer(),
     ExternalLeading :: integer()}
This = wxWindow()
String = unicode:chardata()
Option = {theFont, wxFont:wxFont()}
```

Gets the dimensions of the string as it would be drawn on the window with the currently selected font.

The text extent is returned in the `w` and `h` pointers.

```
getThemeEnabled(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Returns true if the window uses the system theme for drawing its background.

See: `setThemeEnabled/2`

```
getToolTip(This) -> wxToolTip:wxToolTip()
```

Types:

```
This = wxWindow()
```

Get the associated tooltip or NULL if none.

```
getUpdateRegion(This) -> wxRegion:wxRegion()
```

Types:

```
This = wxWindow()
```

Gets the dimensions of the string as it would be drawn on the window with the currently selected font.

Returns the region specifying which parts of the window have been damaged. Should only be called within an `wxPaintEvent` handler.

See: `wxRegion`, `wxRegionIterator` (not implemented in wx)

```
getVirtualSize(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
```

This gets the virtual size of the window in pixels.

By default it returns the client size of the window, but after a call to `setVirtualSize/3` it will return the size set with that method.

See: **Overview windowsizing**

```
getWindowStyleFlag(This) -> integer()
```

Types:

```
This = wxWindow()
```

Gets the window style that was passed to the constructor or `create/4` method.

`GetWindowStyle()` (not implemented in wx) is another name for the same function.

```
getWindowVariant(This) -> wx:wx_enum()
```

Types:

```
This = wxWindow()
```

Returns the value previously passed to `setWindowVariant/2`.

```
hasCapture(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Returns true if this window has the current mouse capture.

See: `captureMouse/1`, `releaseMouse/1`, `wxMouseCaptureLostEvent`, `wxMouseCaptureChangedEvent`

```
hasScrollbar(This, Orient) -> boolean()
```

Types:

```
This = wxWindow()
Orient = integer()
```

Returns true if this window currently has a scroll bar for this orientation.

This method may return false even when `CanScroll()` (not implemented in wx) for the same orientation returns true, but if `CanScroll()` (not implemented in wx) returns false, i.e. scrolling in this direction is not enabled at all, `hasScrollbar/2` always returns false as well.

### hasTransparentBackground(This) -> boolean()

Types:

```
This = wxWindow()
```

Returns true if this window background is transparent (as, for example, for `wxStaticText`) and should show the parent window background.

This method is mostly used internally by the library itself and you normally shouldn't have to call it. You may, however, have to override it in your wxWindow-derived class to ensure that background is painted correctly.

### hide(This) -> boolean()

Types:

```
This = wxWindow()
```

Equivalent to calling `show/2`(false).

### inheritAttributes(This) -> ok

Types:

```
This = wxWindow()
```

This function is (or should be, in case of custom controls) called during window creation to intelligently set up the window visual attributes, that is the font and the foreground and background colours.

By "intelligently" the following is meant: by default, all windows use their own `GetClassDefaultAttributes()` (not implemented in wx) default attributes. However if some of the parents attributes are explicitly (that is, using `setFont/2` and not `setOwnFont/2`) changed and if the corresponding attribute hadn't been explicitly set for this window itself, then this window takes the same value as used by the parent. In addition, if the window overrides `shouldInheritColours/1` to return false, the colours will not be changed no matter what and only the font might.

This rather complicated logic is necessary in order to accommodate the different usage scenarios. The most common one is when all default attributes are used and in this case, nothing should be inherited as in modern GUIs different controls use different fonts (and colours) than their siblings so they can't inherit the same value from the parent. However it was also deemed desirable to allow to simply change the attributes of all children at once by just changing the font or colour of their common parent, hence in this case we do inherit the parents attributes.

### initDialog(This) -> ok

Types:

```
This = wxWindow()
```

Sends an `wxEVT_INIT_DIALOG` event, whose handler usually transfers data to the dialog via validators.

### invalidateBestSize(This) -> ok

Types:

```
This = wxWindow()
```

Resets the cached best size value so it will be recalculated the next time it is needed.

See: `cacheBestSize/2`

```
isFrozen(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Returns true if the window is currently frozen by a call to `freeze/1`.

See: `freeze/1`, `thaw/1`

```
isEnabled(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Returns true if the window is enabled, i.e. if it accepts user input, false otherwise.

Notice that this method can return false even if this window itself hadn't been explicitly disabled when one of its parent windows is disabled. To get the intrinsic status of this window, use `IsThisEnabled()` (not implemented in wx)

See: `enable/2`

```
isExposed(This, Pt) -> boolean()
isExposed(This, Rect) -> boolean()
```

Types:

```
This = wxWindow()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
isExposed(This, X, Y) -> boolean()
```

Types:

```
This = wxWindow()
X = Y = integer()
```

Returns true if the given point or rectangle area has been exposed since the last repaint.

Call this in an paint event handler to optimize redrawing by only redrawing those areas, which have been exposed.

```
isExposed(This, X, Y, W, H) -> boolean()
```

Types:

```
This = wxWindow()
X = Y = W = H = integer()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

`isRetained(This) -> boolean()`

Types:

```
This = wxWindow()
```

Returns true if the window is retained, false otherwise.

Remark: Retained windows are only available on X platforms.

`isShown(This) -> boolean()`

Types:

```
This = wxWindow()
```

Returns true if the window is shown, false if it has been hidden.

See: `isShownOnScreen/1`

`isTopLevel(This) -> boolean()`

Types:

```
This = wxWindow()
```

Returns true if the given window is a top-level one.

Currently all frames and dialogs are considered to be top-level windows (even if they have a parent window).

`isShownOnScreen(This) -> boolean()`

Types:

```
This = wxWindow()
```

Returns true if the window is physically visible on the screen, i.e. it is shown and all its parents up to the toplevel window are shown as well.

See: `isShown/1`

`layout(This) -> boolean()`

Types:

```
This = wxWindow()
```

Lays out the children of this window using the associated sizer.

If a sizer hadn't been associated with this window (see `setSizer/3`), this function doesn't do anything, unless this is a top level window (see `layout/1`).

Note that this method is called automatically when the window size changes if it has the associated sizer (or if `setAutoLayout/2` with true argument had been explicitly called), ensuring that it is always laid out correctly.

See: **Overview windowsizing**

Return: Always returns true, the return value is not useful.

```
lineDown(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Same as `scrollLines/2` (1).

```
lineUp(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Same as `scrollLines/2` (-1).

```
lower(This) -> ok
```

Types:

```
This = wxWindow()
```

Lowers the window to the bottom of the window hierarchy (Z-order).

Remark: This function only works for wxTopLevelWindow-derived classes.

See: `raise/1`

```
move(This, Pt) -> ok
```

Types:

```
This = wxWindow()
Pt = {X :: integer(), Y :: integer()}
```

```
move(This, X, Y) -> ok
move(This, Pt, Y :: [Option]) -> ok
```

Types:

```
This = wxWindow()
Pt = {X :: integer(), Y :: integer()}
Option = {flags, integer()}
```

Moves the window to the given position.

Remark: Implementations of `setSize/6` can also implicitly implement the `move/4` function, which is defined in the base `wxWindow` class as the call:

See: `setSize/6`

```
move(This, X, Y, Options :: [Option]) -> ok
```

Types:

```
This = wxWindow()
X = Y = integer()
Option = {flags, integer()}
```

Moves the window to the given position.

Remark: Implementations of SetSize can also implicitly implement the `move/4` function, which is defined in the base `wxWindow` class as the call:

See: `setSize/6`

```
moveAfterInTabOrder(This, Win) -> ok
```

Types:

```
This = Win = wxWindow()
```

Moves this window in the tab navigation order after the specified `win`.

This means that when the user presses `TAB` key on that other window, the focus switches to this window.

Default tab order is the same as creation order, this function and `moveBeforeInTabOrder/2` allow to change it after creating all the windows.

```
moveBeforeInTabOrder(This, Win) -> ok
```

Types:

```
This = Win = wxWindow()
```

Same as `moveAfterInTabOrder/2` except that it inserts this window just before `win` instead of putting it right after it.

```
navigate(This) -> boolean()
```

Types:

```
This = wxWindow()
```

```
navigate(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxWindow()
Option = {flags, integer()}
```

Performs a keyboard navigation action starting from this window.

This method is equivalent to calling `NavigateIn()` (not implemented in wx) method on the parent window.

Return: Returns true if the focus was moved to another window or false if nothing changed.

Remark: You may wish to call this from a text control custom keypress handler to do the default navigation behaviour for the tab key, since the standard default behaviour for a multiline text control with the wxTE_PROCESS_TAB style is to insert a tab and not navigate to the next control. See also `wxNavigationKeyEvent` and HandleAsNavigationKey.

```
pageDown(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Same as `scrollPages/2` (1).

```
pageUp(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Same as `scrollPages/2` (-1).

```
popupMenu(This, Menu) -> boolean()
```

Types:

```
This = wxWindow()
Menu = wxMenu:wxMenu()
```

```
popupMenu(This, Menu, Options :: [Option]) -> boolean()
```

Types:

```
This = wxWindow()
Menu = wxMenu:wxMenu()
Option = {pos, {X :: integer(), Y :: integer()}}
```

Pops up the given menu at the specified coordinates, relative to this window, and returns control when the user has dismissed the menu.

If a menu item is selected, the corresponding menu event is generated and will be processed as usual. If coordinates are not specified, the current mouse cursor position is used.

`menu` is the menu to pop up.

The position where the menu will appear can be specified either as a {X,Y} `pos` or by two integers (`x` and `y`).

Note that this function switches focus to this window before showing the menu.

Remark: Just before the menu is popped up, `wxMenu::UpdateUI` (not implemented in wx) is called to ensure that the menu items are in the correct state. The menu does not get deleted by the window. It is recommended to not explicitly specify coordinates when calling PopupMenu in response to mouse click, because some of the ports (namely, wxGTK) can do a better job of positioning the menu in that case.

See: `wxMenu`

```
popupMenu(This, Menu, X, Y) -> boolean()
```

Types:

```
This = wxWindow()
Menu = wxMenu:wxMenu()
X = Y = integer()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
raise(This) -> ok
```

Types:

```
This = wxWindow()
```

Raises the window to the top of the window hierarchy (Z-order).

Notice that this function only requests the window manager to raise this window to the top of Z-order. Depending on its configuration, the window manager may raise the window, not do it at all or indicate that a window requested to be raised in some other way, e.g. by flashing its icon if it is minimized.

Remark: This function only works for wxTopLevelWindow-derived classes.

See: `lower/1`

```
refresh(This) -> ok
```

Types:

```
   This = wxWindow()

refresh(This, Options :: [Option]) -> ok
```

Types:

```
   This = wxWindow()
   Option =
       {eraseBackground, boolean()} |
       {rect,
        {X :: integer(),
         Y :: integer(),
         W :: integer(),
         H :: integer()}}
```

Causes this window, and all of its children recursively (except under wxGTK1 where this is not implemented), to be repainted.

Note that repainting doesn't happen immediately but only during the next event loop iteration, if you need to update the window immediately you should use `update/1` instead.

See: `refreshRect/3`

```
refreshRect(This, Rect) -> ok
```

Types:

```
   This = wxWindow()
   Rect =
       {X :: integer(),
        Y :: integer(),
        W :: integer(),
        H :: integer()}

refreshRect(This, Rect, Options :: [Option]) -> ok
```

Types:

```
   This = wxWindow()
   Rect =
       {X :: integer(),
        Y :: integer(),
        W :: integer(),
        H :: integer()}
   Option = {eraseBackground, boolean()}
```

Redraws the contents of the given rectangle: only the area inside it will be repainted.

This is the same as `refresh/2` but has a nicer syntax as it can be called with a temporary {X,Y,W,H} object as argument like this `RefreshRect(wxRect(x, y, w, h))`.

```
releaseMouse(This) -> ok
```

Types:

```
   This = wxWindow()
```

Releases mouse input captured with `captureMouse/1`.

See: `captureMouse/1`, `hasCapture/1`, `releaseMouse/1`, `wxMouseCaptureLostEvent`, `wxMouseCaptureChangedEvent`

```
removeChild(This, Child) -> ok
```

Types:

```
This = Child = wxWindow()
```

Removes a child window.

This is called automatically by window deletion functions so should not be required by the application programmer. Notice that this function is mostly internal to wxWidgets and shouldn't be called by the user code.

```
reparent(This, NewParent) -> boolean()
```

Types:

```
This = NewParent = wxWindow()
```

Reparents the window, i.e. the window will be removed from its current parent window (e.g.

a non-standard toolbar in a `wxFrame`) and then re-inserted into another.

Notice that currently you need to explicitly call `wxBookCtrlBase:removePage/2` before reparenting a notebook page.

```
screenToClient(This) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxWindow()
```

Converts from screen to client window coordinates.

```
screenToClient(This, Pt) -> {X :: integer(), Y :: integer()}
```

Types:

```
This = wxWindow()
Pt = {X :: integer(), Y :: integer()}
```

Converts from screen to client window coordinates.

```
scrollLines(This, Lines) -> boolean()
```

Types:

```
This = wxWindow()
Lines = integer()
```

Scrolls the window by the given number of lines down (if `lines` is positive) or up.

Return: Returns true if the window was scrolled, false if it was already on top/bottom and nothing was done.

Remark: This function is currently only implemented under MSW and `wxTextCtrl` under wxGTK (it also works for `wxScrolled` (not implemented in wx) classes under all platforms).

See: `scrollPages/2`

```
scrollPages(This, Pages) -> boolean()
```

Types:

```
This = wxWindow()
Pages = integer()
```

Scrolls the window by the given number of pages down (if `pages` is positive) or up.

Return: Returns true if the window was scrolled, false if it was already on top/bottom and nothing was done.

Remark: This function is currently only implemented under MSW and wxGTK.

See: scrollLines/2

```
scrollWindow(This, Dx, Dy) -> ok
```

Types:

```
This = wxWindow()
Dx = Dy = integer()
```

```
scrollWindow(This, Dx, Dy, Options :: [Option]) -> ok
```

Types:

```
This = wxWindow()
Dx = Dy = integer()
Option =
    {rect,
     {X :: integer(),
      Y :: integer(),
      W :: integer(),
      H :: integer()}}
```

Physically scrolls the pixels in the window and move child windows accordingly.

Remark: Note that you can often use wxScrolled (not implemented in wx) instead of using this function directly.

```
setAcceleratorTable(This, Accel) -> ok
```

Types:

```
This = wxWindow()
Accel = wxAcceleratorTable:wxAcceleratorTable()
```

Sets the accelerator table for this window.

See wxAcceleratorTable.

```
setAutoLayout(This, AutoLayout) -> ok
```

Types:

```
This = wxWindow()
AutoLayout = boolean()
```

Determines whether the layout/1 function will be called automatically when the window is resized.

This method is called implicitly by setSizer/3 but if you use SetConstraints() (not implemented in wx) you should call it manually or otherwise the window layout won't be correctly updated when its size changes.

See: setSizer/3, SetConstraints() (not implemented in wx)

```
setBackgroundColour(This, Colour) -> boolean()
```

Types:

```
This = wxWindow()
Colour = wx:wx_colour()
```

Sets the background colour of the window.

Notice that as with setForegroundColour/2, setting the background colour of a native control may not affect the entire control and could be not supported at all depending on the control and platform.

Please see inheritAttributes/1 for explanation of the difference between this method and setOwnBackgroundColour/2.

Remark: The background colour is usually painted by the default wxEraseEvent event handler function under Windows and automatically under GTK. Note that setting the background colour does not cause an immediate refresh, so you may wish to call clearBackground/1 or refresh/2 after calling this function. Using this function will disable attempts to use themes for this window, if the system supports them. Use with care since usually the themes represent the appearance chosen by the user to be used for all applications on the system.

Return: true if the colour was really changed, false if it was already set to this colour and nothing was done.

See: getBackgroundColour/1, setForegroundColour/2, getForegroundColour/1, clearBackground/1, refresh/2, wxEraseEvent, wxSystemSettings

```
setBackgroundStyle(This, Style) -> boolean()
```

Types:

```
This = wxWindow()
Style = wx:wx_enum()
```

Sets the background style of the window.

The default background style is wxBG_STYLE_ERASE which indicates that the window background may be erased in EVT_ERASE_BACKGROUND handler. This is a safe, compatibility default; however you may want to change it to wxBG_STYLE_SYSTEM if you don't define any erase background event handlers at all, to avoid unnecessary generation of erase background events and always let system erase the background. And you should change the background style to wxBG_STYLE_PAINT if you define an EVT_PAINT handler which completely overwrites the window background as in this case erasing it previously, either in EVT_ERASE_BACKGROUND handler or in the system default handler, would result in flicker as the background pixels will be repainted twice every time the window is redrawn. Do ensure that the background is entirely erased by your EVT_PAINT handler in this case however as otherwise garbage may be left on screen.

Notice that in previous versions of wxWidgets a common way to work around the above mentioned flickering problem was to define an empty EVT_ERASE_BACKGROUND handler. Setting background style to wxBG_STYLE_PAINT is a simpler and more efficient solution to the same problem.

Under wxGTK and wxOSX, you can use ?wxBG_STYLE_TRANSPARENT to obtain full transparency of the window background. Note that wxGTK supports this only since GTK 2.12 with a compositing manager enabled, call IsTransparentBackgroundSupported() (not implemented in wx) to check whether this is the case.

Also, in order for SetBackgroundStyle(wxBG_STYLE_TRANSPARENT) to work, it must be called before create/4. If you're using your own wxWindow-derived class you should write your code in the following way:

See: setBackgroundColour/2, getForegroundColour/1, setTransparent/2, IsTransparentBackgroundSupported() (not implemented in wx)

```
setCaret(This, Caret) -> ok
```

Types:

```
This = wxWindow()
Caret = wxCaret:wxCaret()
```

Sets the caret() associated with the window.

```
setClientSize(This, Size) -> ok
setClientSize(This, Rect) -> ok
```

Types:

```
This = wxWindow()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setClientSize(This, Width, Height) -> ok
```

Types:

```
This = wxWindow()
Width = Height = integer()
```

This sets the size of the window client area in pixels.

Using this function to size a window tends to be more device-independent than `setSize/6`, since the application need not worry about what dimensions the border or title bar have when trying to fit the window around panel items, for example.

See: **Overview windowsizing**

```
setContainingSizer(This, Sizer) -> ok
```

Types:

```
This = wxWindow()
Sizer = wxSizer:wxSizer()
```

Used by `wxSizer` internally to notify the window about being managed by the given sizer.

This method should not be called from outside the library, unless you're implementing a custom sizer class - and in the latter case you must call this method with the pointer to the sizer itself whenever a window is added to it and with NULL argument when the window is removed from it.

```
setCursor(This, Cursor) -> boolean()
```

Types:

```
This = wxWindow()
Cursor = wxCursor:wxCursor()
```

Sets the window's cursor.

Notice that the window cursor also sets it for the children of the window implicitly.

The `cursor` may be `wxNullCursor` in which case the window cursor will be reset back to default.

See: `wx_misc:setCursor/1`, `wxCursor`

```
setMaxSize(This, Size) -> ok
```

Types:

```
This = wxWindow()
Size = {W :: integer(), H :: integer()}
```

Sets the maximum size of the window, to indicate to the sizer layout mechanism that this is the maximum possible size.

See: `SetMaxClientSize()` (not implemented in wx), **Overview windowsizing**

```
setMinSize(This, Size) -> ok
```

Types:

```
This = wxWindow()
Size = {W :: integer(), H :: integer()}
```

Sets the minimum size of the window, to indicate to the sizer layout mechanism that this is the minimum required size.

You may need to call this if you change the window size after construction and before adding to its parent sizer.

Notice that calling this method doesn't prevent the program from making the window explicitly smaller than the specified size by calling `setSize/6`, it just ensures that it won't become smaller than this size during the automatic layout.

See: `SetMinClientSize()` (not implemented in wx), **Overview windowsizing**

```
setOwnBackgroundColour(This, Colour) -> ok
```

Types:

```
This = wxWindow()
Colour = wx:wx_colour()
```

Sets the background colour of the window but prevents it from being inherited by the children of this window.

See: `setBackgroundColour/2`, `inheritAttributes/1`

```
setOwnFont(This, Font) -> ok
```

Types:

```
This = wxWindow()
Font = wxFont:wxFont()
```

Sets the font of the window but prevents it from being inherited by the children of this window.

See: `setFont/2`, `inheritAttributes/1`

```
setOwnForegroundColour(This, Colour) -> ok
```

Types:

```
This = wxWindow()
Colour = wx:wx_colour()
```

Sets the foreground colour of the window but prevents it from being inherited by the children of this window.

See: `setForegroundColour/2`, `inheritAttributes/1`

```
setDropTarget(This, Target) -> ok
```

Types:

```
This = wxWindow()
Target = wx:wx_object()
```

Associates a drop target with this window.

If the window already has a drop target, it is deleted.

See: `getDropTarget/1`, **Overview dnd**

```
setExtraStyle(This, ExStyle) -> ok
```

Types:

```
This = wxWindow()
ExStyle = integer()
```

Sets the extra style bits for the window.

The currently defined extra style bits are reported in the class description.

```
setFocus(This) -> ok
```

Types:

```
This = wxWindow()
```

This sets the window to receive keyboard input.

See: `HasFocus()` (not implemented in wx), `wxFocusEvent`, `setFocus/1`, `wxPanel:setFocusIgnoringChildren/1`

```
setFocusFromKbd(This) -> ok
```

Types:

```
This = wxWindow()
```

This function is called by wxWidgets keyboard navigation code when the user gives the focus to this window from keyboard (e.g. using TAB key).

By default this method simply calls `setFocus/1` but can be overridden to do something in addition to this in the derived classes.

```
setFont(This, Font) -> boolean()
```

Types:

```
This = wxWindow()
Font = wxFont:wxFont()
```

Sets the font for this window.

This function should not be called for the parent window if you don't want its font to be inherited by its children, use `setOwnFont/2` instead in this case and see `inheritAttributes/1` for more explanations.

Please notice that the given font is not automatically used for `wxPaintDC` objects associated with this window, you need to call `wxDC:setFont/2` too. However this font is used by any standard controls for drawing their text as well as by `getTextExtent/3`.

Return: true if the font was really changed, false if it was already set to this font and nothing was done.

See: `getFont/1`, `inheritAttributes/1`

```
setForegroundColour(This, Colour) -> boolean()
```

Types:

```
This = wxWindow()
Colour = wx:wx_colour()
```

Sets the foreground colour of the window.

The meaning of foreground colour varies according to the window class; it may be the text colour or other colour, or it may not be used at all. Additionally, not all native controls support changing their foreground colour so this method may change their colour only partially or even not at all.

Please see `inheritAttributes/1` for explanation of the difference between this method and `setOwnForegroundColour/2`.

Return: true if the colour was really changed, false if it was already set to this colour and nothing was done.

See: `getForegroundColour/1`, `setBackgroundColour/2`, `getBackgroundColour/1`, `shouldInheritColours/1`

```
setHelpText(This, HelpText) -> ok
```

Types:

```
This = wxWindow()
HelpText = unicode:chardata()
```

Sets the help text to be used as context-sensitive help for this window.

Note that the text is actually stored by the current `wxHelpProvider` (not implemented in wx) implementation, and not in the window object itself.

See: `getHelpText/1`, `wxHelpProvider::AddHelp()` (not implemented in wx)

```
setId(This, Winid) -> ok
```

Types:

```
This = wxWindow()
Winid = integer()
```

Sets the identifier of the window.

Remark: Each window has an integer identifier. If the application has not provided one, an identifier will be generated. Normally, the identifier should be provided on creation and should not be modified subsequently.

See: `getId/1`, **Overview windowids**

```
setLabel(This, Label) -> ok
```

Types:

```
This = wxWindow()
Label = unicode:chardata()
```

Sets the window's label.

See: `getLabel/1`

```
setName(This, Name) -> ok
```

Types:

```
This = wxWindow()
Name = unicode:chardata()
```

Sets the window's name.

See: `getName/1`

```
setPalette(This, Pal) -> ok
```

Types:

```
This = wxWindow()
Pal = wxPalette:wxPalette()
```

Deprecated: use `wxDC:setPalette/2` instead.

```
setScrollbar(This, Orientation, Position, ThumbSize, Range) -> ok
```

Types:

```
This = wxWindow()
Orientation = Position = ThumbSize = Range = integer()
```

```
setScrollbar(This, Orientation, Position, ThumbSize, Range,
             Options :: [Option]) ->
                ok
```

Types:

```
This = wxWindow()
Orientation = Position = ThumbSize = Range = integer()
Option = {refresh, boolean()}
```

Sets the scrollbar properties of a built-in scrollbar.

Remark: Let's say you wish to display 50 lines of text, using the same font. The window is sized so that you can only see 16 lines at a time. You would use: Note that with the window at this size, the thumb position can never go above 50 minus 16, or 34. You can determine how many lines are currently visible by dividing the current view size by the character height in pixels. When defining your own scrollbar behaviour, you will always need to recalculate the scrollbar settings when the window size changes. You could therefore put your scrollbar calculations and SetScrollbar call into a function named AdjustScrollbars, which can be called initially and also from your `wxSizeEvent` handler function.

See: **Overview scrolling**, `wxScrollBar`, `wxScrolled` (not implemented in wx), `wxScrollWinEvent`

```
setScrollPos(This, Orientation, Pos) -> ok
```

Types:

```
This = wxWindow()
Orientation = Pos = integer()
```

```
setScrollPos(This, Orientation, Pos, Options :: [Option]) -> ok
```

Types:

```
This = wxWindow()
Orientation = Pos = integer()
Option = {refresh, boolean()}
```

Sets the position of one of the built-in scrollbars.

Remark: This function does not directly affect the contents of the window: it is up to the application to take note of scrollbar attributes and redraw contents accordingly.

See: `setScrollbar/6`, `getScrollPos/2`, `getScrollThumb/2`, `wxScrollBar`, `wxScrolled` (not implemented in wx)

```
setSize(This, Rect) -> ok
setSize(This, Size) -> ok
```

Types:

```
This = wxWindow()
Size = {W :: integer(), H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setSize(This, Width, Height) -> ok
setSize(This, Rect, Height :: [Option]) -> ok
```

Types:

```
This = wxWindow()
Rect =
    {X :: integer(),
     Y :: integer(),
     W :: integer(),
     H :: integer()}
Option = {sizeFlags, integer()}
```

Sets the size of the window in pixels.

The size is specified using a {X,Y,W,H}, {Width,Height} or by a couple of `int` objects.

Remark: This form must be used with non-default width and height values.

See: `move/4`, **Overview windowsizing**

```
setSize(This, X, Y, Width, Height) -> ok
```

Types:

```
This = wxWindow()
X = Y = Width = Height = integer()
```

```
setSize(This, X, Y, Width, Height, Options :: [Option]) -> ok
```

Types:

```
This = wxWindow()
X = Y = Width = Height = integer()
Option = {sizeFlags, integer()}
```

Sets the size of the window in pixels.

Remark: This overload sets the position and optionally size, of the window. Parameters may be wxDefaultCoord to indicate either that a default should be supplied by wxWidgets, or that the current value of the dimension should be used.

See: `move/4`, **Overview windowsizing**

```
setSizeHints(This, MinSize) -> ok
```

Types:

```
   This = wxWindow()
   MinSize = {W :: integer(), H :: integer()}

setSizeHints(This, MinW, MinH) -> ok
setSizeHints(This, MinSize, MinH :: [Option]) -> ok
```

Types:

```
   This = wxWindow()
   MinSize = {W :: integer(), H :: integer()}
   Option =
       {maxSize, {W :: integer(), H :: integer()}} |
       {incSize, {W :: integer(), H :: integer()}}
```

Use of this function for windows which are not toplevel windows (such as `wxDialog` or `wxFrame`) is discouraged.

Please use `setMinSize/2` and `setMaxSize/2` instead.

See: `setSizeHints/4`, **Overview windowsizing**

```
setSizeHints(This, MinW, MinH, Options :: [Option]) -> ok
```

Types:

```
   This = wxWindow()
   MinW = MinH = integer()
   Option =
       {maxW, integer()} |
       {maxH, integer()} |
       {incW, integer()} |
       {incH, integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setSizer(This, Sizer) -> ok
```

Types:

```
   This = wxWindow()
   Sizer = wxSizer:wxSizer()

setSizer(This, Sizer, Options :: [Option]) -> ok
```

Types:

```
   This = wxWindow()
   Sizer = wxSizer:wxSizer()
   Option = {deleteOld, boolean()}
```

Sets the window to have the given layout sizer.

The window will then own the object, and will take care of its deletion. If an existing layout constraints object is already owned by the window, it will be deleted if the `deleteOld` parameter is true.

Note that this function will also call `setAutoLayout/2` implicitly with true parameter if the `sizer` is non-NULL and false otherwise so that the sizer will be effectively used to layout the window children whenever it is resized.

Remark: SetSizer enables and disables Layout automatically.

```
setSizerAndFit(This, Sizer) -> ok
```

Types:

```
This = wxWindow()
Sizer = wxSizer:wxSizer()
```

```
setSizerAndFit(This, Sizer, Options :: [Option]) -> ok
```

Types:

```
This = wxWindow()
Sizer = wxSizer:wxSizer()
Option = {deleteOld, boolean()}
```

Associate the sizer with the window and set the window size and minimal size accordingly.

This method calls `setSizer/3` and then `wxSizer:setSizeHints/2` which sets the initial window size to the size needed to accommodate all sizer elements and sets the minimal size to the same size, this preventing the user from resizing this window to be less than this minimal size (if it's a top-level window which can be directly resized by the user).

```
setThemeEnabled(This, Enable) -> ok
```

Types:

```
This = wxWindow()
Enable = boolean()
```

This function tells a window if it should use the system's "theme" code to draw the windows' background instead of its own background drawing code.

This does not always have any effect since the underlying platform obviously needs to support the notion of themes in user defined windows. One such platform is GTK+ where windows can have (very colourful) backgrounds defined by a user's selected theme.

Dialogs, notebook pages and the status bar have this flag set to true by default so that the default look and feel is simulated best.

See: `getThemeEnabled/1`

```
setToolTip(This, TipString) -> ok
setToolTip(This, Tip) -> ok
```

Types:

```
This = wxWindow()
Tip = wxToolTip:wxToolTip()
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setVirtualSize(This, Size) -> ok
```

Types:

```
This = wxWindow()
Size = {W :: integer(), H :: integer()}
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

```
setVirtualSize(This, Width, Height) -> ok
```

Types:

```
This = wxWindow()
Width = Height = integer()
```

Sets the virtual size of the window in pixels.

See: **Overview windowsizing**

```
setWindowStyle(This, Style) -> ok
```

Types:

```
This = wxWindow()
Style = integer()
```

See `setWindowStyleFlag/2` for more info.

```
setWindowStyleFlag(This, Style) -> ok
```

Types:

```
This = wxWindow()
Style = integer()
```

Sets the style of the window.

Please note that some styles cannot be changed after the window creation and that `refresh/2` might need to be called after changing the others for the change to take place immediately.

See Window styles for more information about flags.

See: `getWindowStyleFlag/1`

```
setWindowVariant(This, Variant) -> ok
```

Types:

```
This = wxWindow()
Variant = wx:wx_enum()
```

Chooses a different variant of the window display to use.

Window variants currently just differ in size, as can be seen from ?wxWindowVariant documentation. Under all platforms but macOS, this function does nothing more than change the font used by the window. However under macOS it is implemented natively and selects the appropriate variant of the native widget, which has better appearance than just scaled down or up version of the normal variant, so it should be preferred to directly tweaking the font size.

By default the controls naturally use the normal variant.

```
shouldInheritColours(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Return true from here to allow the colours of this window to be changed by `inheritAttributes/1`.

Returning false forbids inheriting them from the parent window.

The base class version returns false, but this method is overridden in `wxControl` where it returns true.

```
show(This) -> boolean()
```

Types:

```
This = wxWindow()
```

```
show(This, Options :: [Option]) -> boolean()
```

Types:

```
This = wxWindow()
Option = {show, boolean()}
```

Shows or hides the window.

You may need to call `raise/1` for a top level window if you want to bring it to top, although this is not needed if `show/2` is called immediately after the frame creation.

Notice that the default state of newly created top level windows is hidden (to allow you to create their contents without flicker) unlike for all the other, not derived from `wxTopLevelWindow`, windows that are by default created in the shown state.

Return: true if the window has been shown or hidden or false if nothing was done because it already was in the requested state.

See: `isShown/1`, `hide/1`, `wxRadioBox:show/3`, `wxShowEvent`

```
thaw(This) -> ok
```

Types:

```
This = wxWindow()
```

Re-enables window updating after a previous call to `freeze/1`.

To really thaw the control, it must be called exactly the same number of times as `freeze/1`.

If the window has any children, they are recursively thawed too.

See: `wxWindowUpdateLocker` (not implemented in wx), `freeze/1`, `isFrozen/1`

```
transferDataFromWindow(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Transfers values from child controls to data areas specified by their validators.

Returns false if a transfer failed.

Notice that this also calls `transferDataFromWindow/1` for all children recursively.

See: `transferDataToWindow/1`, `wxValidator` (not implemented in wx), `validate/1`

```
transferDataToWindow(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Transfers values to child controls from data areas specified by their validators.

Notice that this also calls `transferDataToWindow/1` for all children recursively.

Return: Returns false if a transfer failed.

See: `transferDataFromWindow/1`, `wxValidator` (not implemented in wx), `validate/1`

```
update(This) -> ok
```

Types:

```
This = wxWindow()
```

Calling this method immediately repaints the invalidated area of the window and all of its children recursively (this normally only happens when the flow of control returns to the event loop).

Notice that this function doesn't invalidate any area of the window so nothing happens if nothing has been invalidated (i.e. marked as requiring a redraw). Use `refresh/2` first if you want to immediately redraw the window unconditionally.

```
updateWindowUI(This) -> ok
```

Types:

```
This = wxWindow()
```

```
updateWindowUI(This, Options :: [Option]) -> ok
```

Types:

```
This = wxWindow()
Option = {flags, integer()}
```

This function sends one or more `wxUpdateUIEvent` to the window.

The particular implementation depends on the window; for example a `wxToolBar` will send an update UI event for each toolbar button, and a `wxFrame` will send an update UI event for each menubar menu item.

You can call this function from your application to ensure that your UI is up-to-date at this point (as far as your `wxUpdateUIEvent` handlers are concerned). This may be necessary if you have called `wxUpdateUIEvent:setMode/1` or `wxUpdateUIEvent:setUpdateInterval/1` to limit the overhead that wxWidgets incurs by sending update UI events in idle time. `flags` should be a bitlist of one or more of the ? wxUpdateUI enumeration.

If you are calling this function from an OnInternalIdle or OnIdle function, make sure you pass the wxUPDATE_UI_FROMIDLE flag, since this tells the window to only update the UI elements that need to be updated in idle time. Some windows update their elements only when necessary, for example when a menu is about to be shown. The following is an example of how to call UpdateWindowUI from an idle function.

See: `wxUpdateUIEvent`, `DoUpdateWindowUI()` (not implemented in wx), `OnInternalIdle()` (not implemented in wx)

```
validate(This) -> boolean()
```

Types:

```
This = wxWindow()
```

Validates the current values of the child controls using their validators.

Notice that this also calls `validate/1` for all children recursively.

Return: Returns false if any of the validations failed.

See: `transferDataFromWindow/1`, `transferDataToWindow/1`, `wxValidator` (not implemented in wx)

```
warpPointer(This, X, Y) -> ok
```

Types:

```
This = wxWindow()
X = Y = integer()
```

Moves the pointer to the given position on the window.

Note: Apple Human Interface Guidelines forbid moving the mouse cursor programmatically so you should avoid using this function in Mac applications (and probably avoid using it under the other platforms without good reason as well).

### setTransparent(This, Alpha) -> boolean()

Types:

```
This = wxWindow()
Alpha = integer()
```

Set the transparency of the window.

If the system supports transparent windows, returns true, otherwise returns false and the window remains fully opaque. See also `canSetTransparent/1`.

The parameter `alpha` is in the range 0..255 where 0 corresponds to a fully transparent window and 255 to the fully opaque one. The constants `wxIMAGE_ALPHA_TRANSPARENT` and `wxIMAGE_ALPHA_OPAQUE` can be used.

### canSetTransparent(This) -> boolean()

Types:

```
This = wxWindow()
```

Returns true if the system supports transparent windows and calling `setTransparent/2` may succeed.

If this function returns false, transparent windows are definitely not supported by the current system.

### isDoubleBuffered(This) -> boolean()

Types:

```
This = wxWindow()
```

Returns true if the window contents is double-buffered by the system, i.e. if any drawing done on the window is really done on a temporary backing surface and transferred to the screen all at once later.

See: `wxBufferedDC`

### setDoubleBuffered(This, On) -> ok

Types:

```
This = wxWindow()
On = boolean()
```

Turn on or off double buffering of the window if the system supports it.

### getContentScaleFactor(This) -> number()

Types:

```
This = wxWindow()
```

Returns the factor mapping logical pixels of this window to physical pixels.

This function can be used to portably determine the number of physical pixels in a window of the given size, by multiplying the window size by the value returned from it. I.e. it returns the factor converting window coordinates to "content view" coordinates, where the view can be just a simple window displaying a `wxBitmap` or `wxGLCanvas` or any other kind of window rendering arbitrary "content" on screen.

For the platforms not doing any pixel mapping, i.e. where logical and physical pixels are one and the same, this function always returns 1.0 and so using it is, in principle, unnecessary and could be avoided by using preprocessor check for `wxHAVE_DPI_INDEPENDENT_PIXELS` not being defined, however using this function unconditionally under all platforms is usually simpler and so preferable.

Note: Current behaviour of this function is compatible with wxWidgets 3.0, but different from its behaviour in versions 3.1.0 to 3.1.3, where it returned the same value as `getDPIScaleFactor/1`. Please use the other function if you need to use a scaling factor greater than 1.0 even for the platforms without `wxHAVE_DPI_INDEPENDENT_PIXELS`, such as wxMSW.

Since: 2.9.5

```
getDPI(This) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
```

Return the DPI of the display used by this window.

The returned value can be different for different windows on systems with support for per-monitor DPI values, such as Microsoft Windows 10.

If the DPI is not available, returns {Width,Height} object.

See: `wxDisplay:getPPI/1`, `wxDPIChangedEvent` (not implemented in wx)

Since: 3.1.3

```
fromDIP(D, W) -> integer()
fromDIP(Sz, W) -> {W :: integer(), H :: integer()}
fromDIP(This, D) -> integer()
fromDIP(This, Sz) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
Sz = {W :: integer(), H :: integer()}
```

Convert DPI-independent pixel values to the value in pixels appropriate for the current toolkit.

A DPI-independent pixel is just a pixel at the standard 96 DPI resolution. To keep the same physical size at higher resolution, the physical pixel value must be scaled by `getDPIScaleFactor/1` but this scaling may be already done by the underlying toolkit (GTK+, Cocoa, ...) automatically. This method performs the conversion only if it is not already done by the lower level toolkit and so by using it with pixel values you can guarantee that the physical size of the corresponding elements will remain the same in all resolutions under all platforms. For example, instead of creating a bitmap of the hard coded size of 32 pixels you should use to avoid using tiny bitmaps on high DPI screens.

Notice that this function is only needed when using hard coded pixel values. It is not necessary if the sizes are already based on the DPI-independent units such as dialog units or if you are relying on the controls automatic best size determination and using sizers to lay out them.

Also note that if either component of `sz` has the special value of -1, it is returned unchanged independently of the current DPI, to preserve the special value of -1 in wxWidgets API (it is often used to mean "unspecified").

Since: 3.1.0

```
toDIP(D, W) -> integer()
toDIP(Sz, W) -> {W :: integer(), H :: integer()}
toDIP(This, D) -> integer()
toDIP(This, Sz) -> {W :: integer(), H :: integer()}
```

Types:

```
This = wxWindow()
Sz = {W :: integer(), H :: integer()}
```

Convert pixel values of the current toolkit to DPI-independent pixel values.

A DPI-independent pixel is just a pixel at the standard 96 DPI resolution. To keep the same physical size at higher resolution, the physical pixel value must be scaled by `getDPIScaleFactor/1` but this scaling may be already done by the underlying toolkit (GTK+, Cocoa, ...) automatically. This method performs the conversion only if it is not already done by the lower level toolkit, For example, you may want to use this to store window sizes and positions so that they can be re-used regardless of the display DPI:

Also note that if either component of `sz` has the special value of -1, it is returned unchanged independently of the current DPI, to preserve the special value of -1 in wxWidgets API (it is often used to mean "unspecified").

Since: 3.1.0

# wxWindowCreateEvent

Erlang module

This event is sent just after the actual window associated with a `wxWindow` object has been created.

Since it is derived from `wxCommandEvent`, the event propagates up the window hierarchy.

See: **Overview events**, `wxWindowDestroyEvent`

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxWindowCreateEvent**

## Events

Use `wxEvtHandler:connect/3` with `wxWindowCreateEventType` to subscribe to events of this type.

## Data Types

```
wxWindowCreateEvent() = wx:wx_object()
wxWindowCreate() =
    #wxWindowCreate{type =
                        wxWindowCreateEvent:wxWindowCreateEventType()}
wxWindowCreateEventType() = create
```

# wxWindowDC

Erlang module

A `wxWindowDC` must be constructed if an application wishes to paint on the whole area of a window (client and decorations). This should normally be constructed as a temporary stack object; don't store a `wxWindowDC` object.

To draw on a window from inside an EVT_PAINT() handler, construct a `wxPaintDC` object instead.

To draw on the client area of a window from outside an EVT_PAINT() handler, construct a `wxClientDC` object.

A `wxWindowDC` object is initialized to use the same font and colours as the window it is associated with.

See: `wxDC`, `wxMemoryDC`, `wxPaintDC`, `wxClientDC`, `wxScreenDC`

This class is derived (and can use functions) from: `wxDC`

wxWidgets docs: **wxWindowDC**

## Data Types

```
wxWindowDC() = wx:wx_object()
```

## Exports

```
new(Window) -> wxWindowDC()
```

Types:

```
Window = wxWindow:wxWindow()
```

Constructor.

Pass a pointer to the window on which you wish to paint.

```
destroy(This :: wxWindowDC()) -> ok
```

Destroys the object.

# wxWindowDestroyEvent

Erlang module

This event is sent as early as possible during the window destruction process.

For the top level windows, as early as possible means that this is done by `wxFrame` or `wxDialog` destructor, i.e. after the destructor of the derived class was executed and so any methods specific to the derived class can't be called any more from this event handler. If you need to do this, you must call `wxWindow::SendDestroyEvent()` (not implemented in wx) from your derived class destructor.

For the child windows, this event is generated just before deleting the window from `wxWindow:'Destroy'/1` (which is also called when the parent window is deleted) or from the window destructor if operator `delete` was used directly (which is not recommended for this very reason).

It is usually pointless to handle this event in the window itself but it ca be very useful to receive notifications about the window destruction in the parent window or in any other object interested in this window.

See: **Overview events**, `wxWindowCreateEvent`

This class is derived (and can use functions) from: `wxCommandEvent` `wxEvent`

wxWidgets docs: **wxWindowDestroyEvent**

## Data Types

```
wxWindowDestroyEvent() = wx:wx_object()
wxWindowDestroy() =
    #wxWindowDestroy{type =
                         wxWindowDestroyEvent:wxWindowDestroyEventType()}
wxWindowDestroyEventType() = destroy
```

# wxXmlResource

Erlang module

This is the main class for interacting with the XML-based resource system.

The class holds XML resources from one or more .xml files, binary files or zip archive files.

Note that this is a singleton class and you'll never allocate/deallocate it. Just use the static `get/0` getter.

See: **Overview xrc**, **Overview xrcformat**

wxWidgets docs: **wxXmlResource**

## Data Types

```
wxXmlResource() = wx:wx_object()
```

## Exports

```
new() -> wxXmlResource()
```

```
new(Options :: [Option]) -> wxXmlResource()
```

Types:

```
   Option = {flags, integer()} | {domain, unicode:chardata()}
```

Constructor.

```
new(Filemask, Options :: [Option]) -> wxXmlResource()
```

Types:

```
   Filemask = unicode:chardata()
   Option = {flags, integer()} | {domain, unicode:chardata()}
```

Constructor.

```
destroy(This :: wxXmlResource()) -> ok
```

Destructor.

```
attachUnknownControl(This, Name, Control) -> boolean()
```

Types:

```
   This = wxXmlResource()
   Name = unicode:chardata()
   Control = wxWindow:wxWindow()
```

```
attachUnknownControl(This, Name, Control, Options :: [Option]) ->
                        boolean()
```

Types:

```
This = wxXmlResource()
Name = unicode:chardata()
Control = wxWindow:wxWindow()
Option = {parent, wxWindow:wxWindow()}
```

Attaches an unknown control to the given panel/window/dialog.

Unknown controls are used in conjunction with <object class="unknown">.

```
clearHandlers(This) -> ok
```

Types:

```
This = wxXmlResource()
```

Removes all handlers and deletes them (this means that any handlers added using `AddHandler()` (not implemented in wx) must be allocated on the heap).

```
compareVersion(This, Major, Minor, Release, Revision) -> integer()
```

Types:

```
This = wxXmlResource()
Major = Minor = Release = Revision = integer()
```

Compares the XRC version to the argument.

Returns -1 if the XRC version is less than the argument, +1 if greater, and 0 if they are equal.

```
get() -> wxXmlResource()
```

Gets the global resources object or creates one if none exists.

```
getFlags(This) -> integer()
```

Types:

```
This = wxXmlResource()
```

Returns flags, which may be a bitlist of ?wxXmlResourceFlags enumeration values.

```
getVersion(This) -> integer()
```

Types:

```
This = wxXmlResource()
```

Returns version information (a.b.c.d = d + 256*c + 2562*b + 2563*a).

```
getXRCID(Str_id) -> integer()
```

Types:

```
Str_id = unicode:chardata()
```

```
getXRCID(Str_id, Options :: [Option]) -> integer()
```

Types:

```
Str_id = unicode:chardata()
Option = {value_if_not_found, integer()}
```

Returns a numeric ID that is equivalent to the string ID used in an XML resource.

If an unknown `str_id` is requested (i.e. other than wxID_XXX or integer), a new record is created which associates the given string with a number.

If `value_if_not_found` is `wxID_NONE`, the number is obtained via `wx_misc:newId/0`. Otherwise `value_if_not_found` is used.

Macro `XRCID(name)` is provided for convenient use in event tables.

Note: IDs returned by XRCID() cannot be used with the `EVT_*_RANGE` macros, because the order in which they are assigned to symbolic `name` values is not guaranteed.

```
initAllHandlers(This) -> ok
```

Types:

```
This = wxXmlResource()
```

Initializes handlers for all supported controls/windows.

This will make the executable quite big because it forces linking against most of the wxWidgets library.

```
load(This, Filemask) -> boolean()
```

Types:

```
This = wxXmlResource()
Filemask = unicode:chardata()
```

Loads resources from XML files that match given filemask.

Example:

Note: If wxUSE_FILESYS is enabled, this method understands `wxFileSystem` (not implemented in wx) URLs (see `wxFileSystem::FindFirst()` (not implemented in wx)).

Note: If you are sure that the argument is name of single XRC file (rather than an URL or a wildcard), use `LoadFile()` (not implemented in wx) instead.

See: `LoadFile()` (not implemented in wx), `LoadAllFiles()` (not implemented in wx)

```
loadBitmap(This, Name) -> wxBitmap:wxBitmap()
```

Types:

```
This = wxXmlResource()
Name = unicode:chardata()
```

Loads a bitmap resource from a file.

```
loadDialog(This, Parent, Name) -> wxDialog:wxDialog()
```

Types:

```
This = wxXmlResource()
Parent = wxWindow:wxWindow()
Name = unicode:chardata()
```

Loads a dialog.

`parent` points to parent window (if any).

```
loadDialog(This, Dlg, Parent, Name) -> boolean()
```

Types:

```
This = wxXmlResource()
Dlg = wxDialog:wxDialog()
Parent = wxWindow:wxWindow()
Name = unicode:chardata()
```

Loads a dialog.

`parent` points to parent window (if any).

This form is used to finish creation of an already existing instance (the main reason for this is that you may want to use derived class with a new event table). Example:

```
loadFrame(This, Parent, Name) -> wxFrame:wxFrame()
```

Types:

```
This = wxXmlResource()
Parent = wxWindow:wxWindow()
Name = unicode:chardata()
```

Loads a frame from the resource.

`parent` points to parent window (if any).

```
loadFrame(This, Frame, Parent, Name) -> boolean()
```

Types:

```
This = wxXmlResource()
Frame = wxFrame:wxFrame()
Parent = wxWindow:wxWindow()
Name = unicode:chardata()
```

Loads the contents of a frame onto an existing `wxFrame`.

This form is used to finish creation of an already existing instance (the main reason for this is that you may want to use derived class with a new event table).

```
loadIcon(This, Name) -> wxIcon:wxIcon()
```

Types:

```
This = wxXmlResource()
Name = unicode:chardata()
```

Loads an icon resource from a file.

```
loadMenu(This, Name) -> wxMenu:wxMenu()
```

Types:

```
This = wxXmlResource()
Name = unicode:chardata()
```

Loads menu from resource.

Returns NULL on failure.

```
loadMenuBar(This, Name) -> wxMenuBar:wxMenuBar()
```

Types:

```
   This = wxXmlResource()
   Name = unicode:chardata()
```

```
loadMenuBar(This, Parent, Name) -> wxMenuBar:wxMenuBar()
```

Types:

```
   This = wxXmlResource()
   Parent = wxWindow:wxWindow()
   Name = unicode:chardata()
```

Loads a menubar from resource.

Returns NULL on failure.

```
loadPanel(This, Parent, Name) -> wxPanel:wxPanel()
```

Types:

```
   This = wxXmlResource()
   Parent = wxWindow:wxWindow()
   Name = unicode:chardata()
```

Loads a panel.

`parent` points to the parent window.

```
loadPanel(This, Panel, Parent, Name) -> boolean()
```

Types:

```
   This = wxXmlResource()
   Panel = wxPanel:wxPanel()
   Parent = wxWindow:wxWindow()
   Name = unicode:chardata()
```

Loads a panel.

`parent` points to the parent window. This form is used to finish creation of an already existing instance.

```
loadToolBar(This, Parent, Name) -> wxToolBar:wxToolBar()
```

Types:

```
   This = wxXmlResource()
   Parent = wxWindow:wxWindow()
   Name = unicode:chardata()
```

Loads a toolbar.

```
set(Res) -> wxXmlResource()
```

Types:

```
   Res = wxXmlResource()
```

Sets the global resources object and returns a pointer to the previous one (may be NULL).

```
setFlags(This, Flags) -> ok
```

Types:

```
This = wxXmlResource()
Flags = integer()
```

Sets flags (bitlist of ?wxXmlResourceFlags enumeration values).

```
unload(This, Filename) -> boolean()
```

Types:

```
This = wxXmlResource()
Filename = unicode:chardata()
```

This function unloads a resource previously loaded by `load/2`.

Returns true if the resource was successfully unloaded and false if it hasn't been found in the list of loaded resources.

```
xrcctrl(Window, Name, Type) -> wx:wx_object()
```

Types:

```
Window = wxWindow:wxWindow()
Name = string()
Type = atom()
```

Looks up a control.

Get a control with `Name` in a window created with XML resources. You can use it to set/get values from controls. The object is type casted to `Type`. Example:

# wx_misc

Erlang module

Miscellaneous functions.

## Exports

`displaySize() -> {Width :: integer(), Height :: integer()}`

Returns the display size in pixels.

Note: Use of this function is not recommended in the new code as it only works for the primary display. Use `wxDisplay:getGeometry/1` to retrieve the size of the appropriate display instead.

Either of output pointers can be NULL if the caller is not interested in the corresponding value.

See: `wxGetDisplaySize()` (not implemented in wx), `wxDisplay`

`setCursor(Cursor) -> ok`

Types:

```
Cursor = wxCursor:wxCursor()
```

Globally sets the cursor; only has an effect on Windows, Mac and GTK+.

You should call this function with wxNullCursor to restore the system cursor.

See: `wxCursor`, `wxWindow:setCursor/2`

`getKeyState(Key) -> boolean()`

Types:

```
Key = wx:wx_enum()
```

For normal keys, returns true if the specified key is currently down.

For togglable keys (Caps Lock, Num Lock and Scroll Lock), returns true if the key is toggled such that its LED indicator is lit. There is currently no way to test whether togglable keys are up or down.

Even though there are virtual key codes defined for mouse buttons, they cannot be used with this function currently.

In wxGTK, this function can be only used with modifier keys (`WXK_ALT`, `WXK_CONTROL` and `WXK_SHIFT`) when not using X11 backend currently.

`getMousePosition() -> {X :: integer(), Y :: integer()}`

Returns the mouse position in screen coordinates.

`getMouseState() -> wx:wx_wxMouseState()`

Returns the current state of the mouse.

Returns a `wx_wxMouseState()` instance that contains the current position of the mouse pointer in screen coordinates, as well as boolean values indicating the up/down status of the mouse buttons and the modifier keys.

`setDetectableAutoRepeat(Flag) -> boolean()`

Types:

```
Flag = boolean()
```

Don't synthesize KeyUp events holding down a key and producing KeyDown events with autorepeat.

On by default and always on in wxMSW.

```
bell() -> ok
```

Ring the system bell.

Note: This function is categorized as a GUI one and so is not thread-safe.

```
findMenuItemId(Frame, MenuString, ItemString) -> integer()
```

Types:

```
Frame = wxFrame:wxFrame()
MenuString = ItemString = unicode:chardata()
```

Find a menu item identifier associated with the given frame's menu bar.

```
findWindowAtPoint(Pt) -> wxWindow:wxWindow()
```

Types:

```
Pt = {X :: integer(), Y :: integer()}
```

Find the deepest window at the given mouse position in screen coordinates, returning the window if found, or NULL if not.

This function takes child windows at the given position into account even if they are disabled. The hidden children are however skipped by it.

```
beginBusyCursor() -> ok
```

```
beginBusyCursor(Options :: [Option]) -> ok
```

Types:

```
Option = {cursor, wxCursor:wxCursor()}
```

Changes the cursor to the given cursor for all windows in the application.

Use `endBusyCursor/0` to revert the cursor back to its previous state. These two calls can be nested, and a counter ensures that only the outer calls take effect.

See: `isBusy/0`, `wxBusyCursor` (not implemented in wx)

```
endBusyCursor() -> ok
```

Changes the cursor back to the original cursor, for all windows in the application.

Use with `beginBusyCursor/1`.

See: `isBusy/0`, `wxBusyCursor` (not implemented in wx)

```
isBusy() -> boolean()
```

Returns true if between two `beginBusyCursor/1` and `endBusyCursor/0` calls.

See: `wxBusyCursor` (not implemented in wx)

```
shutdown() -> boolean()

shutdown(Options :: [Option]) -> boolean()
```

Types:

```
Option = {flags, integer()}
```

This function shuts down or reboots the computer depending on the value of the `flags`.

Note: Note that performing the shutdown requires the corresponding access rights (superuser under Unix, SE_SHUTDOWN privilege under Windows) and that this function is only implemented under Unix and MSW.

Return: true on success, false if an error occurred.

```
shell() -> boolean()

shell(Options :: [Option]) -> boolean()
```

Types:

```
Option = {command, unicode:chardata()}
```

Executes a command in an interactive shell window.

If no command is specified, then just the shell is spawned.

See: `wxExecute()` (not implemented in wx), **Examples**

```
launchDefaultBrowser(Url) -> boolean()
```

Types:

```
Url = unicode:chardata()

launchDefaultBrowser(Url, Options :: [Option]) -> boolean()
```

Types:

```
Url = unicode:chardata()
Option = {flags, integer()}
```

Opens the `url` in user's default browser.

If the `flags` parameter contains `wxBROWSER_NEW_WINDOW` flag, a new window is opened for the URL (currently this is only supported under Windows).

And unless the `flags` parameter contains `wxBROWSER_NOBUSYCURSOR` flag, a busy cursor is shown while the browser is being launched (using `wxBusyCursor` (not implemented in wx)).

The parameter `url` is interpreted as follows:

Returns true if the application was successfully launched.

Note: For some configurations of the running user, the application which is launched to open the given URL may be URL-dependent (e.g. a browser may be used for local URLs while another one may be used for remote URLs).

See: `wxLaunchDefaultApplication()` (not implemented in wx), `wxExecute()` (not implemented in wx)

```
getEmailAddress() -> unicode:charlist()
```

Copies the user's email address into the supplied buffer, by concatenating the values returned by `wxGetFullHostName()` (not implemented in wx) and `getUserId/0`.

Return: true if successful, false otherwise.

`getUserId() -> unicode:charlist()`

This function returns the "user id" also known as "login name" under Unix (i.e.

something like "jsmith"). It uniquely identifies the current user (on this system). Under Windows or NT, this function first looks in the environment variables USER and LOGNAME; if neither of these is found, the entry `UserId` in the `wxWidgets` section of the WIN.INI file is tried.

Return: The login name if successful or an empty string otherwise.

See: `wxGetUserName()` (not implemented in wx)

`getHomeDir() -> unicode:charlist()`

Return the (current) user's home directory.

See: `wxGetUserHome()` (not implemented in wx), `wxStandardPaths` (not implemented in wx)

`newId() -> integer()`

Deprecated: Ids generated by it can conflict with the Ids defined by the user code, use `wxID_ANY` to assign ids which are guaranteed to not conflict with the user-defined ids for the controls and menu items you create instead of using this function.

Generates an integer identifier unique to this run of the program.

`registerId(Id) -> ok`

Types:

```
Id = integer()
```

Ensures that Ids subsequently generated by `newId/0` do not clash with the given `id`.

`getCurrentId() -> integer()`

Returns the current id.

`getOsDescription() -> unicode:charlist()`

Returns the string containing the description of the current platform in a user-readable form.

For example, this function may return strings like "Windows 10 (build 10240), 64-bit edition" or "Linux 4.1.4 i386".

See: `wxGetOsVersion()` (not implemented in wx)

`isPlatformLittleEndian() -> boolean()`

Returns true if the current platform is little endian (instead of big endian).

The check is performed at run-time.

`isPlatform64Bit() -> boolean()`

Returns true if the operating system the program is running under is 64 bit.

The check is performed at run-time and may differ from the value available at compile-time (at compile-time you can just check if `sizeof(void*) == 8`) since the program could be running in emulation mode or in a mixed 32/64 bit system (bi-architecture operating system).

Note: This function is not 100% reliable on some systems given the fact that there isn't always a standard way to do a reliable check on the OS architecture.

# gl

Erlang module

Standard OpenGL API

This documents the functions as a brief version of the complete **OpenGL reference pages.**

## Data Types

```
clamp() = float()
offset() = integer() >= 0
i() = integer()
f() = float()
enum() = integer() >= 0
matrix() = m12() | m16()
m12() =
    {f(), f(), f(), f(), f(), f(), f(), f(), f(), f(), f(), f()}
m16() =
    {f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f()}
mem() = binary() | tuple()
```

## Exports

`accum(Op :: enum(), Value :: f()) -> ok`

The accumulation buffer is an extended-range color buffer. Images are not rendered into it. Rather, images rendered into one of the color buffers are added to the contents of the accumulation buffer after rendering. Effects such as antialiasing (of points, lines, and polygons), motion blur, and depth of field can be created by accumulating images generated with different transformation matrices.

**External documentation.**

```
activeShaderProgram(Pipeline :: i(), Program :: i()) -> ok
```

`gl:activeShaderProgram/2` sets the linked program named by `Program` to be the active program for the program pipeline object `Pipeline`. The active program in the active program pipeline object is the target of calls to `gl:uniform()` when no program has been made current through a call to `gl:useProgram/1`.

**External documentation.**

```
activeTexture(Texture :: enum()) -> ok
```

`gl:activeTexture/1` selects which texture unit subsequent texture state calls will affect. The number of texture units an implementation supports is implementation dependent, but must be at least 80.

**External documentation.**

```
alphaFunc(Func :: enum(), Ref :: clamp()) -> ok
```

The alpha test discards fragments depending on the outcome of a comparison between an incoming fragment's alpha value and a constant reference value. `gl:alphaFunc/2` specifies the reference value and the comparison function. The comparison is performed only if alpha testing is enabled. By default, it is not enabled. (See `gl:enable/1` and `gl:disable/1` of `?GL_ALPHA_TEST`.)

**External documentation.**

```
areTexturesResident(Textures :: [i()]) ->
                       {0 | 1, Residences :: [0 | 1]}
```

GL establishes a ``working set'' of textures that are resident in texture memory. These textures can be bound to a texture target much more efficiently than textures that are not resident.

**External documentation.**

```
arrayElement(I :: i()) -> ok
```

`gl:arrayElement/1` commands are used within `gl:'begin'/1`/`gl:'end'/0` pairs to specify vertex and attribute data for point, line, and polygon primitives. If `?GL_VERTEX_ARRAY` is enabled when `gl:arrayElement/1` is called, a single vertex is drawn, using vertex and attribute data taken from location `I` of the enabled arrays. If `?GL_VERTEX_ARRAY` is not enabled, no drawing occurs but the attributes corresponding to the enabled arrays are modified.

**External documentation.**

```
attachShader(Program :: i(), Shader :: i()) -> ok
```

In order to create a complete shader program, there must be a way to specify the list of things that will be linked together. Program objects provide this mechanism. Shaders that are to be linked together in a program object must first be attached to that program object. `gl:attachShader/2` attaches the shader object specified by `Shader` to the program object specified by `Program`. This indicates that `Shader` will be included in link operations that will be performed on `Program`.

**External documentation.**

```
'begin'(Mode :: enum()) -> ok
'end'() -> ok
```

`gl:'begin'/1` and `gl:'end'/0` delimit the vertices that define a primitive or a group of like primitives. `gl:'begin'/1` accepts a single argument that specifies in which of ten ways the vertices are interpreted. Taking n as an integer count starting at one, and N as the total number of vertices specified, the interpretations are as follows:

**External documentation.**

```
beginConditionalRender(Id :: i(), Mode :: enum()) -> ok
endConditionalRender() -> ok
```

Conditional rendering is started using `gl:beginConditionalRender/2` and ended using `gl:endConditionalRender/0`. During conditional rendering, all vertex array commands, as well as `gl:clear/1` and `gl:clearBuffer()` have no effect if the (`?GL_SAMPLES_PASSED`) result of the query object `Id` is zero, or if the (`?GL_ANY_SAMPLES_PASSED`) result is `?GL_FALSE`. The results of commands setting the current vertex state, such as `gl:vertexAttrib()` are undefined. If the (`?GL_SAMPLES_PASSED`) result is non-zero or if the (`?GL_ANY_SAMPLES_PASSED`) result is `?GL_TRUE`, such commands are not discarded. The `Id` parameter to `gl:beginConditionalRender/2` must be the name of a query object previously returned from a call to `gl:genQueries/1`. `Mode` specifies how the results of the query object are to be interpreted. If `Mode` is `?GL_QUERY_WAIT`, the GL waits for the results of the query to be available and then uses the results to determine if subsequent rendering commands are discarded. If `Mode` is `?GL_QUERY_NO_WAIT`, the GL may choose to unconditionally execute the subsequent rendering commands without waiting for the query to complete.

**External documentation.**

```
beginQuery(Target :: enum(), Id :: i()) -> ok
endQuery(Target :: enum()) -> ok
```

`gl:beginQuery/2` and `gl:endQuery/1` delimit the boundaries of a query object. `Query` must be a name previously returned from a call to `gl:genQueries/1`. If a query object with name `Id` does not yet exist it is created with the type determined by `Target`. `Target` must be one of `?GL_SAMPLES_PASSED`, `?GL_ANY_SAMPLES_PASSED`, `?GL_PRIMITIVES_GENERATED`, `?GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`, or `?GL_TIME_ELAPSED`. The behavior of the query object depends on its type and is as follows.

**External documentation.**

```
beginQueryIndexed(Target :: enum(), Index :: i(), Id :: i()) -> ok
endQueryIndexed(Target :: enum(), Index :: i()) -> ok
```

`gl:beginQueryIndexed/3` and `gl:endQueryIndexed/2` delimit the boundaries of a query object. `Query` must be a name previously returned from a call to `gl:genQueries/1`. If a query object with name `Id` does not yet exist it is created with the type determined by `Target`. `Target` must be one of `?GL_SAMPLES_PASSED`, `?GL_ANY_SAMPLES_PASSED`, `?GL_PRIMITIVES_GENERATED`, `?GL_TRANSFORM_FEEDBACK_PRIMITIVES_WRITTEN`, or `?GL_TIME_ELAPSED`. The behavior of the query object depends on its type and is as follows.

**External documentation.**

```
beginTransformFeedback(PrimitiveMode :: enum()) -> ok
endTransformFeedback() -> ok
```

Transform feedback mode captures the values of varying variables written by the vertex shader (or, if active, the geometry shader). Transform feedback is said to be active after a call to `gl:beginTransformFeedback/1` until a subsequent call to `gl:endTransformFeedback/0`. Transform feedback commands must be paired.

**External documentation.**

```
bindAttribLocation(Program :: i(), Index :: i(), Name :: string()) ->
```

```
                    ok
```

`gl:bindAttribLocation/3` is used to associate a user-defined attribute variable in the program object specified by `Program` with a generic vertex attribute index. The name of the user-defined attribute variable is passed as a null terminated string in `Name`. The generic vertex attribute index to be bound to this variable is specified by `Index`. When `Program` is made part of current state, values provided via the generic vertex attribute `Index` will modify the value of the user-defined attribute variable specified by `Name`.

**External documentation.**

```
bindBuffer(Target :: enum(), Buffer :: i()) -> ok
```

`gl:bindBuffer/2` binds a buffer object to the specified buffer binding point. Calling `gl:bindBuffer/2` with `Target` set to one of the accepted symbolic constants and `Buffer` set to the name of a buffer object binds that buffer object name to the target. If no buffer object with name `Buffer` exists, one is created with that name. When a buffer object is bound to a target, the previous binding for that target is automatically broken.

**External documentation.**

```
bindBufferBase(Target :: enum(), Index :: i(), Buffer :: i()) ->
                  ok
```

`gl:bindBufferBase/3` binds the buffer object `Buffer` to the binding point at index `Index` of the array of targets specified by `Target`. Each `Target` represents an indexed array of buffer binding points, as well as a single general binding point that can be used by other buffer manipulation functions such as `gl:bindBuffer/2` or `glMapBuffer`. In addition to binding `Buffer` to the indexed buffer binding target, `gl:bindBufferBase/3` also binds `Buffer` to the generic buffer binding point specified by `Target`.

**External documentation.**

```
bindBufferRange(Target :: enum(),
                Index :: i(),
                Buffer :: i(),
                Offset :: i(),
                Size :: i()) ->
                   ok
```

`gl:bindBufferRange/5` binds a range the buffer object `Buffer` represented by `Offset` and `Size` to the binding point at index `Index` of the array of targets specified by `Target`. Each `Target` represents an indexed array of buffer binding points, as well as a single general binding point that can be used by other buffer manipulation functions such as `gl:bindBuffer/2` or `glMapBuffer`. In addition to binding a range of `Buffer` to the indexed buffer binding target, `gl:bindBufferRange/5` also binds the range to the generic buffer binding point specified by `Target`.

**External documentation.**

```
bindBuffersBase(Target :: enum(), First :: i(), Buffers :: [i()]) ->
                   ok
```

`gl:bindBuffersBase/3` binds a set of `Count` buffer objects whose names are given in the array `Buffers` to the `Count` consecutive binding points starting from index `First` of the array of targets specified by `Target`. If `Buffers` is `?NULL` then `gl:bindBuffersBase/3` unbinds any buffers that are currently bound to the referenced binding points. Assuming no errors are generated, it is equivalent to the following pseudo-code, which calls `gl:bindBufferBase/3`, with the exception that the non-indexed `Target` is not changed by `gl:bindBuffersBase/3`:

**External documentation.**

```
bindBuffersRange(Target :: enum(),
                 First :: i(),
                 Buffers :: [i()],
                 Offsets :: [i()],
                 Sizes :: [i()]) ->
                    ok
```

`gl:bindBuffersRange/5` binds a set of `Count` ranges from buffer objects whose names are given in the array `Buffers` to the `Count` consecutive binding points starting from index `First` of the array of targets specified by `Target`. `Offsets` specifies the address of an array containing `Count` starting offsets within the buffers, and `Sizes` specifies the address of an array of `Count` sizes of the ranges. If `Buffers` is `?NULL` then `Offsets` and `Sizes` are ignored and `gl:bindBuffersRange/5` unbinds any buffers that are currently bound to the referenced binding points. Assuming no errors are generated, it is equivalent to the following pseudo-code, which calls `gl:bindBufferRange/5`, with the exception that the non-indexed `Target` is not changed by `gl:bindBuffersRange/5`:

**External documentation.**

```
bindFragDataLocation(Program :: i(),
                     Color :: i(),
                     Name :: string()) ->
                        ok
```

`gl:bindFragDataLocation/3` explicitly specifies the binding of the user-defined varying out variable `Name` to fragment shader color number `ColorNumber` for program `Program`. If `Name` was bound previously, its assigned binding is replaced with `ColorNumber`. `Name` must be a null-terminated string. `ColorNumber` must be less than `?GL_MAX_DRAW_BUFFERS`.

**External documentation.**

```
bindFragDataLocationIndexed(Program :: i(),
                            ColorNumber :: i(),
                            Index :: i(),
                            Name :: string()) ->
                               ok
```

`gl:bindFragDataLocationIndexed/4` specifies that the varying out variable `Name` in `Program` should be bound to fragment color `ColorNumber` when the program is next linked. `Index` may be zero or one to specify that the color be used as either the first or second color input to the blend equation, respectively.

**External documentation.**

```
bindFramebuffer(Target :: enum(), Framebuffer :: i()) -> ok
```

`gl:bindFramebuffer/2` binds the framebuffer object with name `Framebuffer` to the framebuffer target specified by `Target`. `Target` must be either `?GL_DRAW_FRAMEBUFFER`, `?GL_READ_FRAMEBUFFER` or `?GL_FRAMEBUFFER`. If a framebuffer object is bound to `?GL_DRAW_FRAMEBUFFER` or `?GL_READ_FRAMEBUFFER`, it becomes the target for rendering or readback operations, respectively, until it is deleted or another framebuffer is bound to the corresponding bind point. Calling `gl:bindFramebuffer/2` with `Target` set to `?GL_FRAMEBUFFER` binds `Framebuffer` to both the read and draw framebuffer targets. `Framebuffer` is the name of a framebuffer object previously returned from a call to `gl:genFramebuffers/1`, or zero to break the existing binding of a framebuffer object to `Target`.

**External documentation.**

```
bindImageTexture(Unit, Texture, Level, Layered, Layer, Access,
```

```
                Format) ->
                   ok
```

Types:

```
Unit = Texture = Level = i()
Layered = 0 | 1
Layer = i()
Access = Format = enum()
```

`gl:bindImageTexture/7` binds a single level of a texture to an image unit for the purpose of reading and writing it from shaders. `Unit` specifies the zero-based index of the image unit to which to bind the texture level. `Texture` specifies the name of an existing texture object to bind to the image unit. If `Texture` is zero, then any existing binding to the image unit is broken. `Level` specifies the level of the texture to bind to the image unit.

**External documentation.**

### bindImageTextures(First :: i(), Textures :: [i()]) -> ok

`gl:bindImageTextures/2` binds images from an array of existing texture objects to a specified number of consecutive image units. `Count` specifies the number of texture objects whose names are stored in the array `Textures`. That number of texture names are read from the array and bound to the `Count` consecutive texture units starting from `First`. If the name zero appears in the `Textures` array, any existing binding to the image unit is reset. Any non-zero entry in `Textures` must be the name of an existing texture object. When a non-zero entry in `Textures` is present, the image at level zero is bound, the binding is considered layered, with the first layer set to zero, and the image is bound for read-write access. The image unit format parameter is taken from the internal format of the image at level zero of the texture object. For cube map textures, the internal format of the positive X image of level zero is used. If `Textures` is `?NULL` then it is as if an appropriately sized array containing only zeros had been specified.

**External documentation.**

### bindProgramPipeline(Pipeline :: i()) -> ok

`gl:bindProgramPipeline/1` binds a program pipeline object to the current context. `Pipeline` must be a name previously returned from a call to `gl:genProgramPipelines/1`. If no program pipeline exists with name `Pipeline` then a new pipeline object is created with that name and initialized to the default state vector.

**External documentation.**

### bindRenderbuffer(Target :: enum(), Renderbuffer :: i()) -> ok

`gl:bindRenderbuffer/2` binds the renderbuffer object with name `Renderbuffer` to the renderbuffer target specified by `Target`. `Target` must be `?GL_RENDERBUFFER`. `Renderbuffer` is the name of a renderbuffer object previously returned from a call to `gl:genRenderbuffers/1`, or zero to break the existing binding of a renderbuffer object to `Target`.

**External documentation.**

### bindSampler(Unit :: i(), Sampler :: i()) -> ok

`gl:bindSampler/2` binds `Sampler` to the texture unit at index `Unit`. `Sampler` must be zero or the name of a sampler object previously returned from a call to `gl:genSamplers/1`. `Unit` must be less than the value of `?GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS`.

**External documentation.**

```
bindSamplers(First :: i(), Samplers :: [i()]) -> ok
```

`gl:bindSamplers/2` binds samplers from an array of existing sampler objects to a specified number of consecutive sampler units. `Count` specifies the number of sampler objects whose names are stored in the array `Samplers`. That number of sampler names is read from the array and bound to the `Count` consecutive sampler units starting from `First`.

**External documentation.**

```
bindTexture(Target :: enum(), Texture :: i()) -> ok
```

`gl:bindTexture/2` lets you create or use a named texture. Calling `gl:bindTexture/2` with `Target` set to `?GL_TEXTURE_1D`, `?GL_TEXTURE_2D`, `?GL_TEXTURE_3D`, `?GL_TEXTURE_1D_ARRAY`, `?GL_TEXTURE_2D_ARRAY`, `?GL_TEXTURE_RECTANGLE`, `?GL_TEXTURE_CUBE_MAP`, `?GL_TEXTURE_CUBE_MAP_ARRAY`, `?GL_TEXTURE_BUFFER`, `?GL_TEXTURE_2D_MULTISAMPLE` or `?GL_TEXTURE_2D_MULTISAMPLE_ARRAY` and `Texture` set to the name of the new texture binds the texture name to the target. When a texture is bound to a target, the previous binding for that target is automatically broken.

**External documentation.**

```
bindTextureUnit(Unit :: i(), Texture :: i()) -> ok
```

`gl:bindTextureUnit/2` binds an existing texture object to the texture unit numbered `Unit`.

**External documentation.**

```
bindTextures(First :: i(), Textures :: [i()]) -> ok
```

`gl:bindTextures/2` binds an array of existing texture objects to a specified number of consecutive texture units. `Count` specifies the number of texture objects whose names are stored in the array `Textures`. That number of texture names are read from the array and bound to the `Count` consecutive texture units starting from `First`. The target, or type of texture is deduced from the texture object and each texture is bound to the corresponding target of the texture unit. If the name zero appears in the `Textures` array, any existing binding to any target of the texture unit is reset and the default texture for that target is bound in its place. Any non-zero entry in `Textures` must be the name of an existing texture object. If `Textures` is `?NULL` then it is as if an appropriately sized array containing only zeros had been specified.

**External documentation.**

```
bindTransformFeedback(Target :: enum(), Id :: i()) -> ok
```

`gl:bindTransformFeedback/2` binds the transform feedback object with name `Id` to the current GL state. `Id` must be a name previously returned from a call to `gl:genTransformFeedbacks/1`. If `Id` has not previously been bound, a new transform feedback object with name `Id` and initialized with the default transform state vector is created.

**External documentation.**

```
bindVertexArray(Array :: i()) -> ok
```

`gl:bindVertexArray/1` binds the vertex array object with name `Array`. `Array` is the name of a vertex array object previously returned from a call to `gl:genVertexArrays/1`, or zero to break the existing vertex array object binding.

**External documentation.**

```
bindVertexBuffer(Bindingindex :: i(),
                 Buffer :: i(),
                 Offset :: i(),
                 Stride :: i()) ->
                    ok
vertexArrayVertexBuffer(Vaobj :: i(),
                        Bindingindex :: i(),
                        Buffer :: i(),
                        Offset :: i(),
                        Stride :: i()) ->
                           ok
```

`gl:bindVertexBuffer/4` and `gl:vertexArrayVertexBuffer/5` bind the buffer named `Buffer` to the vertex buffer binding point whose index is given by `Bindingindex`. `gl:bindVertexBuffer/4` modifies the binding of the currently bound vertex array object, whereas `gl:vertexArrayVertexBuffer/5` allows the caller to specify ID of the vertex array object with an argument named `Vaobj`, for which the binding should be modified. `Offset` and `Stride` specify the offset of the first element within the buffer and the distance between elements within the buffer, respectively, and are both measured in basic machine units. `Bindingindex` must be less than the value of `?GL_MAX_VERTEX_ATTRIB_BINDINGS`. `Offset` and `Stride` must be greater than or equal to zero. If `Buffer` is zero, then any buffer currently bound to the specified binding point is unbound.

**External documentation.**

```
bindVertexBuffers(First :: i(),
                  Buffers :: [i()],
                  Offsets :: [i()],
                  Strides :: [i()]) ->
                     ok
vertexArrayVertexBuffers(Vaobj :: i(),
                         First :: i(),
                         Buffers :: [i()],
                         Offsets :: [i()],
                         Strides :: [i()]) ->
                            ok
```

`gl:bindVertexBuffers/4` and `gl:vertexArrayVertexBuffers/5` bind storage from an array of existing buffer objects to a specified number of consecutive vertex buffer binding points units in a vertex array object. For `gl:bindVertexBuffers/4`, the vertex array object is the currently bound vertex array object. For `gl:vertexArrayVertexBuffers/5`, `Vaobj` is the name of the vertex array object.

**External documentation.**

```
bitmap(Width, Height, Xorig, Yorig, Xmove, Ymove, Bitmap) -> ok
```

Types:

```
Width = Height = i()
Xorig = Yorig = Xmove = Ymove = f()
Bitmap = offset() | mem()
```

A bitmap is a binary image. When drawn, the bitmap is positioned relative to the current raster position, and frame buffer pixels corresponding to 1's in the bitmap are written using the current raster color or index. Frame buffer pixels corresponding to 0's in the bitmap are not modified.

**External documentation.**

```
blendColor(Red :: clamp(),
           Green :: clamp(),
           Blue :: clamp(),
           Alpha :: clamp()) ->
              ok
```

The ?GL_BLEND_COLOR may be used to calculate the source and destination blending factors. The color components are clamped to the range [0 1] before being stored. See gl:blendFunc/2 for a complete description of the blending operations. Initially the ?GL_BLEND_COLOR is set to (0, 0, 0, 0).

**External documentation.**

```
blendEquation(Mode :: enum()) -> ok
blendEquationi(Buf :: i(), Mode :: enum()) -> ok
```

The blend equations determine how a new pixel (the "source" color) is combined with a pixel already in the framebuffer (the "destination" color). This function sets both the RGB blend equation and the alpha blend equation to a single equation. gl:blendEquationi/2 specifies the blend equation for a single draw buffer whereas gl:blendEquation/1 sets the blend equation for all draw buffers.

**External documentation.**

```
blendEquationSeparate(ModeRGB :: enum(), ModeAlpha :: enum()) ->
                         ok
blendEquationSeparatei(Buf :: i(),
                       ModeRGB :: enum(),
                       ModeAlpha :: enum()) ->
                          ok
```

The blend equations determines how a new pixel (the "source" color) is combined with a pixel already in the framebuffer (the "destination" color). These functions specify one blend equation for the RGB-color components and one blend equation for the alpha component. gl:blendEquationSeparatei/3 specifies the blend equations for a single draw buffer whereas gl:blendEquationSeparate/2 sets the blend equations for all draw buffers.

**External documentation.**

```
blendFunc(Sfactor :: enum(), Dfactor :: enum()) -> ok
blendFunci(Buf :: i(), Src :: enum(), Dst :: enum()) -> ok
```

Pixels can be drawn using a function that blends the incoming (source) RGBA values with the RGBA values that are already in the frame buffer (the destination values). Blending is initially disabled. Use gl:enable/1 and gl:disable/1 with argument ?GL_BLEND to enable and disable blending.

**External documentation.**

```
blendFuncSeparate(SfactorRGB, DfactorRGB, SfactorAlpha,
                  DfactorAlpha) ->
                     ok
blendFuncSeparatei(Buf :: i(),
                   SrcRGB :: enum(),
                   DstRGB :: enum(),
                   SrcAlpha :: enum(),
                   DstAlpha :: enum()) ->
```

```
                      ok
```

Pixels can be drawn using a function that blends the incoming (source) RGBA values with the RGBA values that are already in the frame buffer (the destination values). Blending is initially disabled. Use `gl:enable/1` and `gl:disable/1` with argument `?GL_BLEND` to enable and disable blending.

**External documentation.**

```
blitFramebuffer(SrcX0, SrcY0, SrcX1, SrcY1, DstX0, DstY0, DstX1,
                DstY1, Mask, Filter) ->
                   ok
```

Types:

```
   SrcX0 = SrcY0 = SrcX1 = SrcY1 = DstX0 = DstY0 = DstX1 = DstY1 = Mask = i()
   Filter = enum()
```

`gl:blitFramebuffer/10` and `glBlitNamedFramebuffer` transfer a rectangle of pixel values from one region of a read framebuffer to another region of a draw framebuffer.

**External documentation.**

```
bufferData(Target :: enum(),
           Size :: i(),
           Data :: offset() | mem(),
           Usage :: enum()) ->
              ok
```

`gl:bufferData/4` and `glNamedBufferData` create a new data store for a buffer object. In case of `gl:bufferData/4`, the buffer object currently bound to `Target` is used. For `glNamedBufferData`, a buffer object associated with ID specified by the caller in `Buffer` will be used instead.

**External documentation.**

```
bufferStorage(Target :: enum(),
              Size :: i(),
              Data :: offset() | mem(),
              Flags :: i()) ->
                 ok
```

`gl:bufferStorage/4` and `glNamedBufferStorage` create a new immutable data store. For `gl:bufferStorage/4`, the buffer object currently bound to `Target` will be initialized. For `glNamedBufferStorage`, `Buffer` is the name of the buffer object that will be configured. The size of the data store is specified by `Size`. If an initial data is available, its address may be supplied in `Data`. Otherwise, to create an uninitialized data store, `Data` should be `?NULL`.

**External documentation.**

```
bufferSubData(Target :: enum(),
              Offset :: i(),
              Size :: i(),
              Data :: offset() | mem()) ->
```

```
                ok
```

`gl:bufferSubData/4` and `glNamedBufferSubData` redefine some or all of the data store for the specified buffer object. Data starting at byte offset `Offset` and extending for `Size` bytes is copied to the data store from the memory pointed to by `Data`. `Offset` and `Size` must define a range lying entirely within the buffer object's data store.

**External documentation.**

```
callList(List :: i()) -> ok
```

`gl:callList/1` causes the named display list to be executed. The commands saved in the display list are executed in order, just as if they were called without using a display list. If `List` has not been defined as a display list, `gl:callList/1` is ignored.

**External documentation.**

```
callLists(Lists :: [i()]) -> ok
```

`gl:callLists/1` causes each display list in the list of names passed as `Lists` to be executed. As a result, the commands saved in each display list are executed in order, just as if they were called without using a display list. Names of display lists that have not been defined are ignored.

**External documentation.**

```
checkFramebufferStatus(Target :: enum()) -> enum()
```

`gl:checkFramebufferStatus/1` and `glCheckNamedFramebufferStatus` return the completeness status of a framebuffer object when treated as a read or draw framebuffer, depending on the value of `Target`.

**External documentation.**

```
clampColor(Target :: enum(), Clamp :: enum()) -> ok
```

`gl:clampColor/2` controls color clamping that is performed during `gl:readPixels/7`. `Target` must be `?GL_CLAMP_READ_COLOR`. If `Clamp` is `?GL_TRUE`, read color clamping is enabled; if `Clamp` is `?GL_FALSE`, read color clamping is disabled. If `Clamp` is `?GL_FIXED_ONLY`, read color clamping is enabled only if the selected read buffer has fixed point components and disabled otherwise.

**External documentation.**

```
clear(Mask :: i()) -> ok
```

`gl:clear/1` sets the bitplane area of the window to values previously selected by `gl:clearColor/4`, `gl:clearDepth/1`, and `gl:clearStencil/1`. Multiple color buffers can be cleared simultaneously by selecting more than one buffer at a time using `gl:drawBuffer/1`.

**External documentation.**

```
clearAccum(Red :: f(), Green :: f(), Blue :: f(), Alpha :: f()) ->
              ok
```

`gl:clearAccum/4` specifies the red, green, blue, and alpha values used by `gl:clear/1` to clear the accumulation buffer.

**External documentation.**

```
clearBufferData(Target, Internalformat, Format, Type, Data) -> ok
clearBufferSubData(Target, Internalformat, Offset, Size, Format,
```

```
                  Type, Data) ->
                     ok
clearBufferfi(Buffer :: enum(),
              Drawbuffer :: i(),
              Depth :: f(),
              Stencil :: i()) ->
                 ok
clearBufferfv(Buffer :: enum(),
              Drawbuffer :: i(),
              Value :: tuple()) ->
                 ok
clearBufferiv(Buffer :: enum(),
              Drawbuffer :: i(),
              Value :: tuple()) ->
                 ok
clearBufferuiv(Buffer :: enum(),
               Drawbuffer :: i(),
               Value :: tuple()) ->
                  ok
```

These commands clear a specified buffer of a framebuffer to specified value(s). For `gl:clearBuffer*()`, the framebuffer is the currently bound draw framebuffer object. For `glClearNamedFramebuffer*`, `Framebuffer` is zero, indicating the default draw framebuffer, or the name of a framebuffer object.

**External documentation.**

```
clearColor(Red :: clamp(),
           Green :: clamp(),
           Blue :: clamp(),
           Alpha :: clamp()) ->
              ok
```

`gl:clearColor/4` specifies the red, green, blue, and alpha values used by `gl:clear/1` to clear the color buffers. Values specified by `gl:clearColor/4` are clamped to the range [0 1].

**External documentation.**

```
clearDepth(Depth :: clamp()) -> ok
clearDepthf(D :: f()) -> ok
```

`gl:clearDepth/1` specifies the depth value used by `gl:clear/1` to clear the depth buffer. Values specified by `gl:clearDepth/1` are clamped to the range [0 1].

**External documentation.**

```
clearIndex(C :: f()) -> ok
```

`gl:clearIndex/1` specifies the index used by `gl:clear/1` to clear the color index buffers. `C` is not clamped. Rather, `C` is converted to a fixed-point value with unspecified precision to the right of the binary point. The integer part of this value is then masked with 2 m-1, where m is the number of bits in a color index stored in the frame buffer.

**External documentation.**

```
clearStencil(S :: i()) -> ok
```

`gl:clearStencil/1` specifies the index used by `gl:clear/1` to clear the stencil buffer. `S` is masked with 2 m-1, where m is the number of bits in the stencil buffer.

**External documentation.**

```
clearTexImage(Texture :: i(),
              Level :: i(),
              Format :: enum(),
              Type :: enum(),
              Data :: offset() | mem()) ->
                 ok
```

`gl:clearTexImage/5` fills all an image contained in a texture with an application supplied value. `Texture` must be the name of an existing texture. Further, `Texture` may not be the name of a buffer texture, nor may its internal format be compressed.

**External documentation.**

```
clearTexSubImage(Texture, Level, Xoffset, Yoffset, Zoffset, Width,
                 Height, Depth, Format, Type, Data) ->
                    ok
```

Types:

```
   Texture = Level = Xoffset = Yoffset = Zoffset = Width = Height = Depth =
   i()
   Format = Type = enum()
   Data = offset() | mem()
```

`gl:clearTexSubImage/11` fills all or part of an image contained in a texture with an application supplied value. `Texture` must be the name of an existing texture. Further, `Texture` may not be the name of a buffer texture, nor may its internal format be compressed.

**External documentation.**

```
clientActiveTexture(Texture :: enum()) -> ok
```

`gl:clientActiveTexture/1` selects the vertex array client state parameters to be modified by `gl:texCoordPointer/4`, and enabled or disabled with `gl:enableClientState/1` or `gl:disableClientState/1`, respectively, when called with a parameter of `?GL_TEXTURE_COORD_ARRAY`.

**External documentation.**

```
clientWaitSync(Sync :: i(), Flags :: i(), Timeout :: i()) ->
                  enum()
```

`gl:clientWaitSync/3` causes the client to block and wait for the sync object specified by `Sync` to become signaled. If `Sync` is signaled when `gl:clientWaitSync/3` is called, `gl:clientWaitSync/3` returns immediately, otherwise it will block and wait for up to `Timeout` nanoseconds for `Sync` to become signaled.

**External documentation.**

```
clipControl(Origin :: enum(), Depth :: enum()) -> ok
```

`gl:clipControl/2` controls the clipping volume behavior and the clip coordinate to window coordinate transformation behavior.

**External documentation.**

```
clipPlane(Plane :: enum(), Equation :: {f(), f(), f(), f()}) -> ok
```

Geometry is always clipped against the boundaries of a six-plane frustum in `x`, `y`, and `z`. `gl:clipPlane/2` allows the specification of additional planes, not necessarily perpendicular to the `x`, `y`, or `z` axis, against which all geometry is clipped. To determine the maximum number of additional clipping planes, call `gl:getIntegerv/1` with argument `?GL_MAX_CLIP_PLANES`. All implementations support at least six such clipping planes. Because the resulting clipping region is the intersection of the defined half-spaces, it is always convex.

**External documentation.**

```
color3b(Red :: i(), Green :: i(), Blue :: i()) -> ok
color3bv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) -> ok
color3d(Red :: f(), Green :: f(), Blue :: f()) -> ok
color3dv(X1 :: {Red :: f(), Green :: f(), Blue :: f()}) -> ok
color3f(Red :: f(), Green :: f(), Blue :: f()) -> ok
color3fv(X1 :: {Red :: f(), Green :: f(), Blue :: f()}) -> ok
color3i(Red :: i(), Green :: i(), Blue :: i()) -> ok
color3iv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) -> ok
color3s(Red :: i(), Green :: i(), Blue :: i()) -> ok
color3sv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) -> ok
color3ub(Red :: i(), Green :: i(), Blue :: i()) -> ok
color3ubv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) -> ok
color3ui(Red :: i(), Green :: i(), Blue :: i()) -> ok
color3uiv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) -> ok
color3us(Red :: i(), Green :: i(), Blue :: i()) -> ok
color3usv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) -> ok
color4b(Red :: i(), Green :: i(), Blue :: i(), Alpha :: i()) -> ok
color4bv(X1 ::
             {Red :: i(), Green :: i(), Blue :: i(), Alpha :: i()}) ->
            ok
color4d(Red :: f(), Green :: f(), Blue :: f(), Alpha :: f()) -> ok
color4dv(X1 ::
             {Red :: f(), Green :: f(), Blue :: f(), Alpha :: f()}) ->
            ok
color4f(Red :: f(), Green :: f(), Blue :: f(), Alpha :: f()) -> ok
color4fv(X1 ::
             {Red :: f(), Green :: f(), Blue :: f(), Alpha :: f()}) ->
            ok
color4i(Red :: i(), Green :: i(), Blue :: i(), Alpha :: i()) -> ok
color4iv(X1 ::
             {Red :: i(), Green :: i(), Blue :: i(), Alpha :: i()}) ->
```

```
            ok
color4s(Red :: i(), Green :: i(), Blue :: i(), Alpha :: i()) -> ok
color4sv(X1 ::
             {Red :: i(), Green :: i(), Blue :: i(), Alpha :: i()}) ->
            ok
color4ub(Red :: i(), Green :: i(), Blue :: i(), Alpha :: i()) ->
            ok
color4ubv(X1 ::
              {Red :: i(),
               Green :: i(),
               Blue :: i(),
               Alpha :: i()}) ->
             ok
color4ui(Red :: i(), Green :: i(), Blue :: i(), Alpha :: i()) ->
            ok
color4uiv(X1 ::
              {Red :: i(),
               Green :: i(),
               Blue :: i(),
               Alpha :: i()}) ->
             ok
color4us(Red :: i(), Green :: i(), Blue :: i(), Alpha :: i()) ->
            ok
color4usv(X1 ::
              {Red :: i(),
               Green :: i(),
               Blue :: i(),
               Alpha :: i()}) ->
             ok
```

The GL stores both a current single-valued color index and a current four-valued RGBA color. `gl:color()` sets a new four-valued RGBA color. `gl:color()` has two major variants: `gl:color3()` and `gl:color4()`. `gl:color3()` variants specify new red, green, and blue values explicitly and set the current alpha value to 1.0 (full intensity) implicitly. `gl:color4()` variants specify all four color components explicitly.

**External documentation.**

```
colorMask(Red :: 0 | 1,
          Green :: 0 | 1,
          Blue :: 0 | 1,
          Alpha :: 0 | 1) ->
             ok
colorMaski(Index :: i(),
           R :: 0 | 1,
           G :: 0 | 1,
           B :: 0 | 1,
           A :: 0 | 1) ->
              ok
```

`gl:colorMask/4` and `gl:colorMaski/5` specify whether the individual color components in the frame buffer can or cannot be written. `gl:colorMaski/5` sets the mask for a specific draw buffer, whereas `gl:colorMask/4`

sets the mask for all draw buffers. If Red is ?GL_FALSE, for example, no change is made to the red component of any pixel in any of the color buffers, regardless of the drawing operation attempted.

**External documentation.**

```
colorMaterial(Face :: enum(), Mode :: enum()) -> ok
```

gl:colorMaterial/2 specifies which material parameters track the current color. When ?GL_COLOR_MATERIAL is enabled, the material parameter or parameters specified by Mode, of the material or materials specified by Face, track the current color at all times.

**External documentation.**

```
colorPointer(Size :: i(),
             Type :: enum(),
             Stride :: i(),
             Ptr :: offset() | mem()) ->
                ok
```

gl:colorPointer/4 specifies the location and data format of an array of color components to use when rendering. Size specifies the number of components per color, and must be 3 or 4. Type specifies the data type of each color component, and Stride specifies the byte stride from one color to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see gl:interleavedArrays/3.)

**External documentation.**

```
colorSubTable(Target, Start, Count, Format, Type, Data) -> ok
```

Types:

```
   Target = enum()
   Start = Count = i()
   Format = Type = enum()
   Data = offset() | mem()
```

gl:colorSubTable/6 is used to respecify a contiguous portion of a color table previously defined using gl:colorTable/6. The pixels referenced by Data replace the portion of the existing table from indices Start to start+count-1, inclusive. This region may not include any entries outside the range of the color table as it was originally specified. It is not an error to specify a subtexture with width of 0, but such a specification has no effect.

**External documentation.**

```
colorTable(Target, Internalformat, Width, Format, Type, Table) ->
              ok
```

Types:

```
   Target = Internalformat = enum()
   Width = i()
   Format = Type = enum()
   Table = offset() | mem()
```

gl:colorTable/6 may be used in two ways: to test the actual size and color resolution of a lookup table given a particular set of parameters, or to load the contents of a color lookup table. Use the targets ?GL_PROXY_* for the first case and the other targets for the second case.

**External documentation.**

```
colorTableParameterfv(Target :: enum(),
                      Pname :: enum(),
                      Params :: {f(), f(), f(), f()}) ->
                         ok
colorTableParameteriv(Target :: enum(),
                      Pname :: enum(),
                      Params :: {i(), i(), i(), i()}) ->
                         ok
```

`gl:colorTableParameter()` is used to specify the scale factors and bias terms applied to color components when they are loaded into a color table. `Target` indicates which color table the scale and bias terms apply to; it must be set to `?GL_COLOR_TABLE`, `?GL_POST_CONVOLUTION_COLOR_TABLE`, or `?GL_POST_COLOR_MATRIX_COLOR_TABLE`.

**External documentation.**

```
compileShader(Shader :: i()) -> ok
```

`gl:compileShader/1` compiles the source code strings that have been stored in the shader object specified by `Shader`.

**External documentation.**

```
compressedTexImage1D(Target, Level, Internalformat, Width, Border,
                     ImageSize, Data) ->
                        ok
```

Types:

```
Target = enum()
Level = i()
Internalformat = enum()
Width = Border = ImageSize = i()
Data = offset() | mem()
```

Texturing allows elements of an image array to be read by shaders.

**External documentation.**

```
compressedTexImage2D(Target, Level, Internalformat, Width, Height,
                     Border, ImageSize, Data) ->
                        ok
```

Types:

```
Target = enum()
Level = i()
Internalformat = enum()
Width = Height = Border = ImageSize = i()
Data = offset() | mem()
```

Texturing allows elements of an image array to be read by shaders.

**External documentation.**

```
compressedTexImage3D(Target, Level, Internalformat, Width, Height,
```

```
                    Depth, Border, ImageSize, Data) ->
                       ok
```

Types:

```
   Target = enum()
   Level = i()
   Internalformat = enum()
   Width = Height = Depth = Border = ImageSize = i()
   Data = offset() | mem()
```

Texturing allows elements of an image array to be read by shaders.

**External documentation.**

```
compressedTexSubImage1D(Target, Level, Xoffset, Width, Format,
                        ImageSize, Data) ->
                           ok
compressedTextureSubImage1D(Texture, Level, Xoffset, Width,
                            Format, ImageSize, Data) ->
                               ok
```

Types:

```
   Texture = Level = Xoffset = Width = i()
   Format = enum()
   ImageSize = i()
   Data = offset() | mem()
```

Texturing allows elements of an image array to be read by shaders.

**External documentation.**

```
compressedTexSubImage2D(Target, Level, Xoffset, Yoffset, Width,
                        Height, Format, ImageSize, Data) ->
                           ok
compressedTextureSubImage2D(Texture, Level, Xoffset, Yoffset,
                            Width, Height, Format, ImageSize,
                            Data) ->
                               ok
```

Types:

```
   Texture = Level = Xoffset = Yoffset = Width = Height = i()
   Format = enum()
   ImageSize = i()
   Data = offset() | mem()
```

Texturing allows elements of an image array to be read by shaders.

**External documentation.**

```
compressedTexSubImage3D(Target, Level, Xoffset, Yoffset, Zoffset,
                        Width, Height, Depth, Format, ImageSize,
                        Data) ->
```

```
                            ok
compressedTextureSubImage3D(Texture, Level, Xoffset, Yoffset,
                            Zoffset, Width, Height, Depth, Format,
                            ImageSize, Data) ->
                               ok
```

Types:

```
   Texture = Level = Xoffset = Yoffset = Zoffset = Width = Height = Depth =
   i()
   Format = enum()
   ImageSize = i()
   Data = offset() | mem()
```

Texturing allows elements of an image array to be read by shaders.

**External documentation.**

```
convolutionFilter1D(Target, Internalformat, Width, Format, Type,
                    Image) ->
                       ok
```

Types:

```
   Target = Internalformat = enum()
   Width = i()
   Format = Type = enum()
   Image = offset() | mem()
```

`gl:convolutionFilter1D/6` builds a one-dimensional convolution filter kernel from an array of pixels.

**External documentation.**

```
convolutionFilter2D(Target, Internalformat, Width, Height, Format,
                    Type, Image) ->
                       ok
```

Types:

```
   Target = Internalformat = enum()
   Width = Height = i()
   Format = Type = enum()
   Image = offset() | mem()
```

`gl:convolutionFilter2D/7` builds a two-dimensional convolution filter kernel from an array of pixels.

**External documentation.**

```
convolutionParameterf(Target :: enum(),
                      Pname :: enum(),
                      Params :: tuple()) ->
                         ok
convolutionParameterfv(Target :: enum(),
                       Pname :: enum(),
                       Params :: tuple()) ->
```

```
                          ok
convolutionParameteri(Target :: enum(),
                      Pname :: enum(),
                      Params :: tuple()) ->
                         ok
convolutionParameteriv(Target :: enum(),
                       Pname :: enum(),
                       Params :: tuple()) ->
                          ok
```

`gl:convolutionParameter()` sets the value of a convolution parameter.

**External documentation.**

```
copyBufferSubData(ReadTarget, WriteTarget, ReadOffset,
                  WriteOffset, Size) ->
                     ok
```

Types:

```
   ReadTarget = WriteTarget = enum()
   ReadOffset = WriteOffset = Size = i()
```

`gl:copyBufferSubData/5` and `glCopyNamedBufferSubData` copy part of the data store attached to a source buffer object to the data store attached to a destination buffer object. The number of basic machine units indicated by `Size` is copied from the source at offset `ReadOffset` to the destination at `WriteOffset`. `ReadOffset`, `WriteOffset` and `Size` are in terms of basic machine units.

**External documentation.**

```
copyColorSubTable(Target :: enum(),
                  Start :: i(),
                  X :: i(),
                  Y :: i(),
                  Width :: i()) ->
                     ok
```

`gl:copyColorSubTable/5` is used to respecify a contiguous portion of a color table previously defined using `gl:colorTable/6`. The pixels copied from the framebuffer replace the portion of the existing table from indices `Start` to start+x-1, inclusive. This region may not include any entries outside the range of the color table, as was originally specified. It is not an error to specify a subtexture with width of 0, but such a specification has no effect.

**External documentation.**

```
copyColorTable(Target :: enum(),
               Internalformat :: enum(),
               X :: i(),
               Y :: i(),
               Width :: i()) ->
                  ok
```

`gl:copyColorTable/5` loads a color table with pixels from the current `?GL_READ_BUFFER` (rather than from main memory, as is the case for `gl:colorTable/6`).

**External documentation.**

```
copyConvolutionFilter1D(Target :: enum(),
                        Internalformat :: enum(),
                        X :: i(),
                        Y :: i(),
                        Width :: i()) ->
                           ok
```

`gl:copyConvolutionFilter1D/5` defines a one-dimensional convolution filter kernel with pixels from the current `?GL_READ_BUFFER` (rather than from main memory, as is the case for `gl:convolutionFilter1D/6`).

**External documentation.**

```
copyConvolutionFilter2D(Target :: enum(),
                        Internalformat :: enum(),
                        X :: i(),
                        Y :: i(),
                        Width :: i(),
                        Height :: i()) ->
                           ok
```

`gl:copyConvolutionFilter2D/6` defines a two-dimensional convolution filter kernel with pixels from the current `?GL_READ_BUFFER` (rather than from main memory, as is the case for `gl:convolutionFilter2D/7`).

**External documentation.**

```
copyImageSubData(SrcName, SrcTarget, SrcLevel, SrcX, SrcY, SrcZ,
                 DstName, DstTarget, DstLevel, DstX, DstY, DstZ,
                 SrcWidth, SrcHeight, SrcDepth) ->
                    ok
```

Types:

```
SrcName = i()
SrcTarget = enum()
SrcLevel = SrcX = SrcY = SrcZ = DstName = i()
DstTarget = enum()
DstLevel = DstX = DstY = DstZ = SrcWidth = SrcHeight = SrcDepth = i()
```

`gl:copyImageSubData/15` may be used to copy data from one image (i.e. texture or renderbuffer) to another. `gl:copyImageSubData/15` does not perform general-purpose conversions such as scaling, resizing, blending, color-space, or format conversions. It should be considered to operate in a manner similar to a CPU memcpy. CopyImageSubData can copy between images with different internal formats, provided the formats are compatible.

**External documentation.**

```
copyPixels(X :: i(),
           Y :: i(),
           Width :: i(),
           Height :: i(),
           Type :: enum()) ->
              ok
```

`gl:copyPixels/5` copies a screen-aligned rectangle of pixels from the specified frame buffer location to a region relative to the current raster position. Its operation is well defined only if the entire pixel source region is within the

exposed portion of the window. Results of copies from outside the window, or from regions of the window that are not exposed, are hardware dependent and undefined.

**External documentation.**

```
copyTexImage1D(Target, Level, Internalformat, X, Y, Width, Border) ->
                  ok
```

Types:

```
   Target = enum()
   Level = i()
   Internalformat = enum()
   X = Y = Width = Border = i()
```

`gl:copyTexImage1D/7` defines a one-dimensional texture image with pixels from the current `?GL_READ_BUFFER`.

**External documentation.**

```
copyTexImage2D(Target, Level, Internalformat, X, Y, Width, Height,
               Border) ->
                  ok
```

Types:

```
   Target = enum()
   Level = i()
   Internalformat = enum()
   X = Y = Width = Height = Border = i()
```

`gl:copyTexImage2D/8` defines a two-dimensional texture image, or cube-map texture image with pixels from the current `?GL_READ_BUFFER`.

**External documentation.**

```
copyTexSubImage1D(Target :: enum(),
                  Level :: i(),
                  Xoffset :: i(),
                  X :: i(),
                  Y :: i(),
                  Width :: i()) ->
                     ok
```

`gl:copyTexSubImage1D/6` and `glCopyTextureSubImage1D` replace a portion of a one-dimensional texture image with pixels from the current `?GL_READ_BUFFER` (rather than from main memory, as is the case for `gl:texSubImage1D/7`). For `gl:copyTexSubImage1D/6`, the texture object that is bound to `Target` will be used for the process. For `glCopyTextureSubImage1D`, `Texture` tells which texture object should be used for the purpose of the call.

**External documentation.**

```
copyTexSubImage2D(Target, Level, Xoffset, Yoffset, X, Y, Width,
                  Height) ->
```

```
                    ok
```

Types:

```
   Target = enum()
   Level = Xoffset = Yoffset = X = Y = Width = Height = i()
```

`gl:copyTexSubImage2D/8` and `glCopyTextureSubImage2D` replace a rectangular portion of a two-dimensional texture image, cube-map texture image, rectangular image, or a linear portion of a number of slices of a one-dimensional array texture with pixels from the current `?GL_READ_BUFFER` (rather than from main memory, as is the case for `gl:texSubImage2D/9`).

**External documentation.**

```
copyTexSubImage3D(Target, Level, Xoffset, Yoffset, Zoffset, X, Y,
                  Width, Height) ->
                     ok
```

Types:

```
   Target = enum()
   Level = Xoffset = Yoffset = Zoffset = X = Y = Width = Height = i()
```

`gl:copyTexSubImage3D/9` and `glCopyTextureSubImage3D` functions replace a rectangular portion of a three-dimensional or two-dimensional array texture image with pixels from the current `?GL_READ_BUFFER` (rather than from main memory, as is the case for `gl:texSubImage3D/11`).

**External documentation.**

```
createBuffers(N :: i()) -> [i()]
```

`gl:createBuffers/1` returns `N` previously unused buffer names in `Buffers`, each representing a new buffer object initialized as if it had been bound to an unspecified target.

**External documentation.**

```
createFramebuffers(N :: i()) -> [i()]
```

`gl:createFramebuffers/1` returns `N` previously unused framebuffer names in `Framebuffers`, each representing a new framebuffer object initialized to the default state.

**External documentation.**

```
createProgram() -> i()
```

`gl:createProgram/0` creates an empty program object and returns a non-zero value by which it can be referenced. A program object is an object to which shader objects can be attached. This provides a mechanism to specify the shader objects that will be linked to create a program. It also provides a means for checking the compatibility of the shaders that will be used to create a program (for instance, checking the compatibility between a vertex shader and a fragment shader). When no longer needed as part of a program object, shader objects can be detached.

**External documentation.**

```
createProgramPipelines(N :: i()) -> [i()]
```

`gl:createProgramPipelines/1` returns `N` previously unused program pipeline names in `Pipelines`, each representing a new program pipeline object initialized to the default state.

**External documentation.**

```
createQueries(Target :: enum(), N :: i()) -> [i()]
```

`gl:createQueries/2` returns `N` previously unused query object names in `Ids`, each representing a new query object with the specified `Target`.

**External documentation.**

```
createRenderbuffers(N :: i()) -> [i()]
```

`gl:createRenderbuffers/1` returns `N` previously unused renderbuffer object names in `Renderbuffers`, each representing a new renderbuffer object initialized to the default state.

**External documentation.**

```
createSamplers(N :: i()) -> [i()]
```

`gl:createSamplers/1` returns `N` previously unused sampler names in `Samplers`, each representing a new sampler object initialized to the default state.

**External documentation.**

```
createShader(Type :: enum()) -> i()
```

`gl:createShader/1` creates an empty shader object and returns a non-zero value by which it can be referenced. A shader object is used to maintain the source code strings that define a shader. `ShaderType` indicates the type of shader to be created. Five types of shader are supported. A shader of type `?GL_COMPUTE_SHADER` is a shader that is intended to run on the programmable compute processor. A shader of type `?GL_VERTEX_SHADER` is a shader that is intended to run on the programmable vertex processor. A shader of type `?GL_TESS_CONTROL_SHADER` is a shader that is intended to run on the programmable tessellation processor in the control stage. A shader of type `?GL_TESS_EVALUATION_SHADER` is a shader that is intended to run on the programmable tessellation processor in the evaluation stage. A shader of type `?GL_GEOMETRY_SHADER` is a shader that is intended to run on the programmable geometry processor. A shader of type `?GL_FRAGMENT_SHADER` is a shader that is intended to run on the programmable fragment processor.

**External documentation.**

```
createShaderProgramv(Type :: enum(),
                     Strings :: [unicode:chardata()]) ->
                        i()
```

`gl:createShaderProgram()` creates a program object containing compiled and linked shaders for a single stage specified by `Type`. `Strings` refers to an array of `Count` strings from which to create the shader executables.

**External documentation.**

```
createTextures(Target :: enum(), N :: i()) -> [i()]
```

`gl:createTextures/2` returns `N` previously unused texture names in `Textures`, each representing a new texture object of the dimensionality and type specified by `Target` and initialized to the default values for that texture type.

**External documentation.**

```
createTransformFeedbacks(N :: i()) -> [i()]
```

`gl:createTransformFeedbacks/1` returns `N` previously unused transform feedback object names in `Ids`, each representing a new transform feedback object initialized to the default state.

**External documentation.**

```
createVertexArrays(N :: i()) -> [i()]
```

`gl:createVertexArrays/1` returns `N` previously unused vertex array object names in `Arrays`, each representing a new vertex array object initialized to the default state.

**External documentation.**

```
cullFace(Mode :: enum()) -> ok
```

`gl:cullFace/1` specifies whether front- or back-facing facets are culled (as specified by `mode`) when facet culling is enabled. Facet culling is initially disabled. To enable and disable facet culling, call the `gl:enable/1` and `gl:disable/1` commands with the argument `?GL_CULL_FACE`. Facets include triangles, quadrilaterals, polygons, and rectangles.

**External documentation.**

```
debugMessageControl(Source :: enum(),
                    Type :: enum(),
                    Severity :: enum(),
                    Ids :: [i()],
                    Enabled :: 0 | 1) ->
                       ok
```

`gl:debugMessageControl/5` controls the reporting of debug messages generated by a debug context. The parameters `Source`, `Type` and `Severity` form a filter to select messages from the pool of potential messages generated by the GL.

**External documentation.**

```
debugMessageInsert(Source :: enum(),
                   Type :: enum(),
                   Id :: i(),
                   Severity :: enum(),
                   Buf :: string()) ->
                      ok
```

`gl:debugMessageInsert/5` inserts a user-supplied message into the debug output queue. `Source` specifies the source that will be used to classify the message and must be `?GL_DEBUG_SOURCE_APPLICATION` or `?GL_DEBUG_SOURCE_THIRD_PARTY`. All other sources are reserved for use by the GL implementation. `Type` indicates the type of the message to be inserted and may be one of `?GL_DEBUG_TYPE_ERROR`, `?GL_DEBUG_TYPE_DEPRECATED_BEHAVIOR`, `?GL_DEBUG_TYPE_UNDEFINED_BEHAVIOR`, `?GL_DEBUG_TYPE_PORTABILITY`, `?GL_DEBUG_TYPE_PERFORMANCE`, `?GL_DEBUG_TYPE_MARKER`, `?GL_DEBUG_TYPE_PUSH_GROUP`, `?GL_DEBUG_TYPE_POP_GROUP`, or `?GL_DEBUG_TYPE_OTHER`. `Severity` indicates the severity of the message and may be `?GL_DEBUG_SEVERITY_LOW`, `?GL_DEBUG_SEVERITY_MEDIUM`, `?GL_DEBUG_SEVERITY_HIGH` or `?GL_DEBUG_SEVERITY_NOTIFICATION`. `Id` is available for application defined use and may be any value. This value will be recorded and used to identify the message.

**External documentation.**

```
deleteBuffers(Buffers :: [i()]) -> ok
```

`gl:deleteBuffers/1` deletes `N` buffer objects named by the elements of the array `Buffers`. After a buffer object is deleted, it has no contents, and its name is free for reuse (for example by `gl:genBuffers/1`). If a buffer object that is currently bound is deleted, the binding reverts to 0 (the absence of any buffer object).

**External documentation.**

```
deleteFramebuffers(Framebuffers :: [i()]) -> ok
```

`gl:deleteFramebuffers/1` deletes the `N` framebuffer objects whose names are stored in the array addressed by `Framebuffers`. The name zero is reserved by the GL and is silently ignored, should it occur in `Framebuffers`, as are other unused names. Once a framebuffer object is deleted, its name is again unused and it has no attachments. If a framebuffer that is currently bound to one or more of the targets `?GL_DRAW_FRAMEBUFFER` or `?GL_READ_FRAMEBUFFER` is deleted, it is as though `gl:bindFramebuffer/2` had been executed with the corresponding `Target` and `Framebuffer` zero.

**External documentation.**

```
deleteLists(List :: i(), Range :: i()) -> ok
```

`gl:deleteLists/2` causes a contiguous group of display lists to be deleted. `List` is the name of the first display list to be deleted, and `Range` is the number of display lists to delete. All display lists d with list<= d<= list +range-1 are deleted.

**External documentation.**

```
deleteProgram(Program :: i()) -> ok
```

`gl:deleteProgram/1` frees the memory and invalidates the name associated with the program object specified by `Program.` This command effectively undoes the effects of a call to `gl:createProgram/0`.

**External documentation.**

```
deleteProgramPipelines(Pipelines :: [i()]) -> ok
```

`gl:deleteProgramPipelines/1` deletes the `N` program pipeline objects whose names are stored in the array `Pipelines`. Unused names in `Pipelines` are ignored, as is the name zero. After a program pipeline object is deleted, its name is again unused and it has no contents. If program pipeline object that is currently bound is deleted, the binding for that object reverts to zero and no program pipeline object becomes current.

**External documentation.**

```
deleteQueries(Ids :: [i()]) -> ok
```

`gl:deleteQueries/1` deletes `N` query objects named by the elements of the array `Ids`. After a query object is deleted, it has no contents, and its name is free for reuse (for example by `gl:genQueries/1`).

**External documentation.**

```
deleteRenderbuffers(Renderbuffers :: [i()]) -> ok
```

`gl:deleteRenderbuffers/1` deletes the `N` renderbuffer objects whose names are stored in the array addressed by `Renderbuffers`. The name zero is reserved by the GL and is silently ignored, should it occur in `Renderbuffers`, as are other unused names. Once a renderbuffer object is deleted, its name is again unused and it has no contents. If a renderbuffer that is currently bound to the target `?GL_RENDERBUFFER` is deleted, it is as though `gl:bindRenderbuffer/2` had been executed with a `Target` of `?GL_RENDERBUFFER` and a `Name` of zero.

**External documentation.**

```
deleteSamplers(Samplers :: [i()]) -> ok
```

gl:deleteSamplers/1 deletes N sampler objects named by the elements of the array Samplers. After a sampler object is deleted, its name is again unused. If a sampler object that is currently bound to a sampler unit is deleted, it is as though gl:bindSampler/2 is called with unit set to the unit the sampler is bound to and sampler zero. Unused names in samplers are silently ignored, as is the reserved name zero.

**External documentation.**

```
deleteShader(Shader :: i()) -> ok
```

gl:deleteShader/1 frees the memory and invalidates the name associated with the shader object specified by Shader. This command effectively undoes the effects of a call to gl:createShader/1.

**External documentation.**

```
deleteSync(Sync :: i()) -> ok
```

gl:deleteSync/1 deletes the sync object specified by Sync. If the fence command corresponding to the specified sync object has completed, or if no gl:waitSync/3 or gl:clientWaitSync/3 commands are blocking on Sync, the object is deleted immediately. Otherwise, Sync is flagged for deletion and will be deleted when it is no longer associated with any fence command and is no longer blocking any gl:waitSync/3 or gl:clientWaitSync/3 command. In either case, after gl:deleteSync/1 returns, the name Sync is invalid and can no longer be used to refer to the sync object.

**External documentation.**

```
deleteTextures(Textures :: [i()]) -> ok
```

gl:deleteTextures/1 deletes N textures named by the elements of the array Textures. After a texture is deleted, it has no contents or dimensionality, and its name is free for reuse (for example by gl:genTextures/1). If a texture that is currently bound is deleted, the binding reverts to 0 (the default texture).

**External documentation.**

```
deleteTransformFeedbacks(Ids :: [i()]) -> ok
```

gl:deleteTransformFeedbacks/1 deletes the N transform feedback objects whose names are stored in the array Ids. Unused names in Ids are ignored, as is the name zero. After a transform feedback object is deleted, its name is again unused and it has no contents. If an active transform feedback object is deleted, its name immediately becomes unused, but the underlying object is not deleted until it is no longer active.

**External documentation.**

```
deleteVertexArrays(Arrays :: [i()]) -> ok
```

gl:deleteVertexArrays/1 deletes N vertex array objects whose names are stored in the array addressed by Arrays. Once a vertex array object is deleted it has no contents and its name is again unused. If a vertex array object that is currently bound is deleted, the binding for that object reverts to zero and the default vertex array becomes current. Unused names in Arrays are silently ignored, as is the value zero.

**External documentation.**

```
depthFunc(Func :: enum()) -> ok
```

gl:depthFunc/1 specifies the function used to compare each incoming pixel depth value with the depth value present in the depth buffer. The comparison is performed only if depth testing is enabled. (See gl:enable/1 and gl:disable/1 of ?GL_DEPTH_TEST.)

**External documentation.**

```
depthMask(Flag :: 0 | 1) -> ok
```

`gl:depthMask/1` specifies whether the depth buffer is enabled for writing. If `Flag` is `?GL_FALSE`, depth buffer writing is disabled. Otherwise, it is enabled. Initially, depth buffer writing is enabled.

**External documentation.**

```
depthRange(Near_val :: clamp(), Far_val :: clamp()) -> ok
depthRangef(N :: f(), F :: f()) -> ok
```

After clipping and division by `w`, depth coordinates range from -1 to 1, corresponding to the near and far clipping planes. `gl:depthRange/2` specifies a linear mapping of the normalized depth coordinates in this range to window depth coordinates. Regardless of the actual depth buffer implementation, window coordinate depth values are treated as though they range from 0 through 1 (like color components). Thus, the values accepted by `gl:depthRange/2` are both clamped to this range before they are accepted.

**External documentation.**

```
depthRangeArrayv(First :: i(), V :: [{f(), f()}]) -> ok
```

After clipping and division by `w`, depth coordinates range from -1 to 1, corresponding to the near and far clipping planes. Each viewport has an independent depth range specified as a linear mapping of the normalized depth coordinates in this range to window depth coordinates. Regardless of the actual depth buffer implementation, window coordinate depth values are treated as though they range from 0 through 1 (like color components). `gl:depthRangeArray()` specifies a linear mapping of the normalized depth coordinates in this range to window depth coordinates for each viewport in the range [`First`, `First` + `Count`). Thus, the values accepted by `gl:depthRangeArray()` are both clamped to this range before they are accepted.

**External documentation.**

```
depthRangeIndexed(Index :: i(), N :: f(), F :: f()) -> ok
```

After clipping and division by `w`, depth coordinates range from -1 to 1, corresponding to the near and far clipping planes. Each viewport has an independent depth range specified as a linear mapping of the normalized depth coordinates in this range to window depth coordinates. Regardless of the actual depth buffer implementation, window coordinate depth values are treated as though they range from 0 through 1 (like color components). `gl:depthRangeIndexed/3` specifies a linear mapping of the normalized depth coordinates in this range to window depth coordinates for a specified viewport. Thus, the values accepted by `gl:depthRangeIndexed/3` are both clamped to this range before they are accepted.

**External documentation.**

```
detachShader(Program :: i(), Shader :: i()) -> ok
```

`gl:detachShader/2` detaches the shader object specified by `Shader` from the program object specified by `Program`. This command can be used to undo the effect of the command `gl:attachShader/2`.

**External documentation.**

```
dispatchCompute(Num_groups_x :: i(),
                Num_groups_y :: i(),
                Num_groups_z :: i()) ->
```

```
                    ok
```

`gl:dispatchCompute/3` launches one or more compute work groups. Each work group is processed by the active program object for the compute shader stage. While the individual shader invocations within a work group are executed as a unit, work groups are executed completely independently and in unspecified order. `Num_groups_x`, `Num_groups_y` and `Num_groups_z` specify the number of local work groups that will be dispatched in the X, Y and Z dimensions, respectively.

**External documentation.**

```
dispatchComputeIndirect(Indirect :: i()) -> ok
```

`gl:dispatchComputeIndirect/1` launches one or more compute work groups using parameters stored in the buffer object currently bound to the `?GL_DISPATCH_INDIRECT_BUFFER` target. Each work group is processed by the active program object for the compute shader stage. While the individual shader invocations within a work group are executed as a unit, work groups are executed completely independently and in unspecified order. `Indirect` contains the offset into the data store of the buffer object bound to the `?GL_DISPATCH_INDIRECT_BUFFER` target at which the parameters are stored.

**External documentation.**

```
drawArrays(Mode :: enum(), First :: i(), Count :: i()) -> ok
```

`gl:drawArrays/3` specifies multiple geometric primitives with very few subroutine calls. Instead of calling a GL procedure to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertices, normals, and colors and use them to construct a sequence of primitives with a single call to `gl:drawArrays/3`.

**External documentation.**

```
drawArraysIndirect(Mode :: enum(), Indirect :: offset() | mem()) ->
                      ok
```

`gl:drawArraysIndirect/2` specifies multiple geometric primitives with very few subroutine calls. `gl:drawArraysIndirect/2` behaves similarly to `gl:drawArraysInstancedBaseInstance/5`, execept that the parameters to `gl:drawArraysInstancedBaseInstance/5` are stored in memory at the address given by `Indirect`.

**External documentation.**

```
drawArraysInstanced(Mode :: enum(),
                    First :: i(),
                    Count :: i(),
                    Instancecount :: i()) ->
                       ok
```

`gl:drawArraysInstanced/4` behaves identically to `gl:drawArrays/3` except that `Instancecount` instances of the range of elements are executed and the value of the internal counter `InstanceID` advances for each iteration. `InstanceID` is an internal 32-bit integer counter that may be read by a vertex shader as `?gl_InstanceID`.

**External documentation.**

```
drawArraysInstancedBaseInstance(Mode :: enum(),
                                First :: i(),
                                Count :: i(),
                                Instancecount :: i(),
```

```
                                Baseinstance :: i()) ->
                                   ok
```

`gl:drawArraysInstancedBaseInstance/5` behaves identically to `gl:drawArrays/3` except that `Instancecount` instances of the range of elements are executed and the value of the internal counter `InstanceID` advances for each iteration. `InstanceID` is an internal 32-bit integer counter that may be read by a vertex shader as `?gl_InstanceID`.

**External documentation.**

```
drawBuffer(Mode :: enum()) -> ok
```

When colors are written to the frame buffer, they are written into the color buffers specified by `gl:drawBuffer/1`. One of the following values can be used for default framebuffer:

**External documentation.**

```
drawBuffers(Bufs :: [enum()]) -> ok
```

`gl:drawBuffers/1` and `glNamedFramebufferDrawBuffers` define an array of buffers into which outputs from the fragment shader data will be written. If a fragment shader writes a value to one or more user defined output variables, then the value of each variable will be written into the buffer specified at a location within `Bufs` corresponding to the location assigned to that user defined output. The draw buffer used for user defined outputs assigned to locations greater than or equal to `N` is implicitly set to `?GL_NONE` and any data written to such an output is discarded.

**External documentation.**

```
drawElements(Mode :: enum(),
             Count :: i(),
             Type :: enum(),
             Indices :: offset() | mem()) ->
                ok
```

`gl:drawElements/4` specifies multiple geometric primitives with very few subroutine calls. Instead of calling a GL function to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertices, normals, and so on, and use them to construct a sequence of primitives with a single call to `gl:drawElements/4`.

**External documentation.**

```
drawElementsBaseVertex(Mode, Count, Type, Indices, Basevertex) ->
                          ok
```

Types:

```
Mode = enum()
Count = i()
Type = enum()
Indices = offset() | mem()
Basevertex = i()
```

`gl:drawElementsBaseVertex/5` behaves identically to `gl:drawElements/4` except that the `i`th element transferred by the corresponding draw call will be taken from element `Indices`[i] + `Basevertex` of each enabled array. If the resulting value is larger than the maximum value representable by `Type`, it is as if the calculation were upconverted to 32-bit unsigned integers (with wrapping on overflow conditions). The operation is undefined if the sum would be negative.

**External documentation.**

```
drawElementsIndirect(Mode :: enum(),
                     Type :: enum(),
                     Indirect :: offset() | mem()) ->
                        ok
```

`gl:drawElementsIndirect/3` specifies multiple indexed geometric primitives with very few subroutine calls. `gl:drawElementsIndirect/3` behaves similarly to `gl:drawElementsInstancedBaseVertexBaseInstance/7`, execpt that the parameters to `gl:drawElementsInstancedBaseVertexBaseInstance/7` are stored in memory at the address given by `Indirect`.

**External documentation.**

```
drawElementsInstanced(Mode, Count, Type, Indices, Instancecount) ->
                         ok
```

Types:

```
Mode = enum()
Count = i()
Type = enum()
Indices = offset() | mem()
Instancecount = i()
```

`gl:drawElementsInstanced/5` behaves identically to `gl:drawElements/4` except that `Instancecount` instances of the set of elements are executed and the value of the internal counter `InstanceID` advances for each iteration. `InstanceID` is an internal 32-bit integer counter that may be read by a vertex shader as `?gl_InstanceID`.

**External documentation.**

```
drawElementsInstancedBaseInstance(Mode, Count, Type, Indices,
                                  Instancecount, Baseinstance) ->
                                     ok
```

Types:

```
Mode = enum()
Count = i()
Type = enum()
Indices = offset() | mem()
Instancecount = Baseinstance = i()
```

`gl:drawElementsInstancedBaseInstance/6` behaves identically to `gl:drawElements/4` except that `Instancecount` instances of the set of elements are executed and the value of the internal counter `InstanceID` advances for each iteration. `InstanceID` is an internal 32-bit integer counter that may be read by a vertex shader as `?gl_InstanceID`.

**External documentation.**

```
drawElementsInstancedBaseVertex(Mode, Count, Type, Indices,
                                Instancecount, Basevertex) ->
                                   ok
```

Types:

```
   Mode = enum()
   Count = i()
   Type = enum()
   Indices = offset() | mem()
   Instancecount = Basevertex = i()
```

`gl:drawElementsInstancedBaseVertex/6` behaves identically to `gl:drawElementsInstanced/5` except that the `i`th element transferred by the corresponding draw call will be taken from element `Indices`[i] + `Basevertex` of each enabled array. If the resulting value is larger than the maximum value representable by `Type`, it is as if the calculation were upconverted to 32-bit unsigned integers (with wrapping on overflow conditions). The operation is undefined if the sum would be negative.

**External documentation.**

```
drawElementsInstancedBaseVertexBaseInstance(Mode, Count, Type,
                                            Indices,
                                            Instancecount,
                                            Basevertex,
                                            Baseinstance) ->
                                               ok
```

Types:

```
   Mode = enum()
   Count = i()
   Type = enum()
   Indices = offset() | mem()
   Instancecount = Basevertex = Baseinstance = i()
```

`gl:drawElementsInstancedBaseVertexBaseInstance/7` behaves identically to `gl:drawElementsInstanced/5` except that the `i`th element transferred by the corresponding draw call will be taken from element `Indices`[i] + `Basevertex` of each enabled array. If the resulting value is larger than the maximum value representable by `Type`, it is as if the calculation were upconverted to 32-bit unsigned integers (with wrapping on overflow conditions). The operation is undefined if the sum would be negative.

**External documentation.**

```
drawPixels(Width :: i(),
           Height :: i(),
           Format :: enum(),
           Type :: enum(),
           Pixels :: offset() | mem()) ->
              ok
```

`gl:drawPixels/5` reads pixel data from memory and writes it into the frame buffer relative to the current raster position, provided that the raster position is valid. Use `gl:rasterPos()` or `gl:windowPos()` to set the current raster position; use `gl:get()` with argument `?GL_CURRENT_RASTER_POSITION_VALID` to determine if the specified raster position is valid, and `gl:get()` with argument `?GL_CURRENT_RASTER_POSITION` to query the raster position.

**External documentation.**

```
drawRangeElements(Mode, Start, End, Count, Type, Indices) -> ok
```

Types:

```
Mode = enum()
Start = End = Count = i()
Type = enum()
Indices = offset() | mem()
```

`gl:drawRangeElements/6` is a restricted form of `gl:drawElements/4`. `Mode`, and `Count` match the corresponding arguments to `gl:drawElements/4`, with the additional constraint that all values in the arrays `Count` must lie between `Start` and `End`, inclusive.

**External documentation.**

```
drawRangeElementsBaseVertex(Mode, Start, End, Count, Type,
                            Indices, Basevertex) ->
                               ok
```

Types:

```
Mode = enum()
Start = End = Count = i()
Type = enum()
Indices = offset() | mem()
Basevertex = i()
```

`gl:drawRangeElementsBaseVertex/7` is a restricted form of `gl:drawElementsBaseVertex/5`. `Mode`, `Count` and `Basevertex` match the corresponding arguments to `gl:drawElementsBaseVertex/5`, with the additional constraint that all values in the array `Indices` must lie between `Start` and `End`, inclusive, prior to adding `Basevertex`. Index values lying outside the range [`Start`, `End`] are treated in the same way as `gl:drawElementsBaseVertex/5`. The `i`th element transferred by the corresponding draw call will be taken from element `Indices`[i] + `Basevertex` of each enabled array. If the resulting value is larger than the maximum value representable by `Type`, it is as if the calculation were upconverted to 32-bit unsigned integers (with wrapping on overflow conditions). The operation is undefined if the sum would be negative.

**External documentation.**

```
drawTransformFeedback(Mode :: enum(), Id :: i()) -> ok
```

`gl:drawTransformFeedback/2` draws primitives of a type specified by `Mode` using a count retrieved from the transform feedback specified by `Id`. Calling `gl:drawTransformFeedback/2` is equivalent to calling `gl:drawArrays/3` with `Mode` as specified, `First` set to zero, and `Count` set to the number of vertices captured on vertex stream zero the last time transform feedback was active on the transform feedback object named by `Id`.

**External documentation.**

```
drawTransformFeedbackInstanced(Mode :: enum(),
                               Id :: i(),
                               Instancecount :: i()) ->
                                  ok
```

`gl:drawTransformFeedbackInstanced/3` draws multiple copies of a range of primitives of a type specified by `Mode` using a count retrieved from the transform feedback stream specified by `Stream` of the transform feedback object specified by `Id`. Calling `gl:drawTransformFeedbackInstanced/3` is equivalent to calling `gl:drawArraysInstanced/4` with `Mode` and `Instancecount` as specified, `First` set to zero, and `Count` set to the number of vertices captured on vertex stream zero the last time transform feedback was active on the transform feedback object named by `Id`.

**External documentation.**

```
drawTransformFeedbackStream(Mode :: enum(),
                            Id :: i(),
                            Stream :: i()) ->
                               ok
```

`gl:drawTransformFeedbackStream/3` draws primitives of a type specified by `Mode` using a count retrieved from the transform feedback stream specified by `Stream` of the transform feedback object specified by `Id`. Calling `gl:drawTransformFeedbackStream/3` is equivalent to calling `gl:drawArrays/3` with `Mode` as specified, `First` set to zero, and `Count` set to the number of vertices captured on vertex stream `Stream` the last time transform feedback was active on the transform feedback object named by `Id`.

**External documentation.**

```
drawTransformFeedbackStreamInstanced(Mode :: enum(),
                                     Id :: i(),
                                     Stream :: i(),
                                     Instancecount :: i()) ->
                                        ok
```

`gl:drawTransformFeedbackStreamInstanced/4` draws multiple copies of a range of primitives of a type specified by `Mode` using a count retrieved from the transform feedback stream specified by `Stream` of the transform feedback object specified by `Id`. Calling `gl:drawTransformFeedbackStreamInstanced/4` is equivalent to calling `gl:drawArraysInstanced/4` with `Mode` and `Instancecount` as specified, `First` set to zero, and `Count` set to the number of vertices captured on vertex stream `Stream` the last time transform feedback was active on the transform feedback object named by `Id`.

**External documentation.**

```
edgeFlag(Flag :: 0 | 1) -> ok
edgeFlagv(X1 :: {Flag :: 0 | 1}) -> ok
```

Each vertex of a polygon, separate triangle, or separate quadrilateral specified between a `gl:'begin'/1`/`gl:'end'/0` pair is marked as the start of either a boundary or nonboundary edge. If the current edge flag is true when the vertex is specified, the vertex is marked as the start of a boundary edge. Otherwise, the vertex is marked as the start of a nonboundary edge. `gl:edgeFlag/1` sets the edge flag bit to `?GL_TRUE` if `Flag` is `?GL_TRUE` and to `?GL_FALSE` otherwise.

**External documentation.**

```
edgeFlagPointer(Stride :: i(), Ptr :: offset() | mem()) -> ok
```

`gl:edgeFlagPointer/2` specifies the location and data format of an array of boolean edge flags to use when rendering. `Stride` specifies the byte stride from one edge flag to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays.

**External documentation.**

```
disable(Cap :: enum()) -> ok
disablei(Target :: enum(), Index :: i()) -> ok
enable(Cap :: enum()) -> ok
enablei(Target :: enum(), Index :: i()) -> ok
```

`gl:enable/1` and `gl:disable/1` enable and disable various capabilities. Use `gl:isEnabled/1` or `gl:get()` to determine the current setting of any capability. The initial value for each capability with the

exception of ?GL_DITHER and ?GL_MULTISAMPLE is ?GL_FALSE. The initial value for ?GL_DITHER and ?GL_MULTISAMPLE is ?GL_TRUE.

**External documentation.**

```
disableClientState(Cap :: enum()) -> ok
enableClientState(Cap :: enum()) -> ok
```

gl:enableClientState/1 and gl:disableClientState/1 enable or disable individual client-side capabilities. By default, all client-side capabilities are disabled. Both gl:enableClientState/1 and gl:disableClientState/1 take a single argument, Cap, which can assume one of the following values:

**External documentation.**

```
disableVertexArrayAttrib(Vaobj :: i(), Index :: i()) -> ok
disableVertexAttribArray(Index :: i()) -> ok
enableVertexArrayAttrib(Vaobj :: i(), Index :: i()) -> ok
enableVertexAttribArray(Index :: i()) -> ok
```

gl:enableVertexAttribArray/1 and gl:enableVertexArrayAttrib/2 enable the generic vertex attribute array specified by Index. gl:enableVertexAttribArray/1 uses currently bound vertex array object for the operation, whereas gl:enableVertexArrayAttrib/2 updates state of the vertex array object with ID Vaobj.

**External documentation.**

```
evalCoord1d(U :: f()) -> ok
evalCoord1dv(X1 :: {U :: f()}) -> ok
evalCoord1f(U :: f()) -> ok
evalCoord1fv(X1 :: {U :: f()}) -> ok
evalCoord2d(U :: f(), V :: f()) -> ok
evalCoord2dv(X1 :: {U :: f(), V :: f()}) -> ok
evalCoord2f(U :: f(), V :: f()) -> ok
evalCoord2fv(X1 :: {U :: f(), V :: f()}) -> ok
```

gl:evalCoord1() evaluates enabled one-dimensional maps at argument U. gl:evalCoord2() does the same for two-dimensional maps using two domain values, U and V. To define a map, call glMap1 and glMap2; to enable and disable it, call gl:enable/1 and gl:disable/1.

**External documentation.**

```
evalMesh1(Mode :: enum(), I1 :: i(), I2 :: i()) -> ok
evalMesh2(Mode :: enum(),
          I1 :: i(),
          I2 :: i(),
          J1 :: i(),
          J2 :: i()) ->
             ok
```

gl:mapGrid() and gl:evalMesh() are used in tandem to efficiently generate and evaluate a series of evenly-spaced map domain values. gl:evalMesh() steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by glMap1 and glMap2. Mode determines whether the resulting vertices are connected as points, lines, or filled polygons.

**External documentation.**

```
evalPoint1(I :: i()) -> ok
evalPoint2(I :: i(), J :: i()) -> ok
```

`gl:mapGrid()` and `gl:evalMesh()` are used in tandem to efficiently generate and evaluate a series of evenly spaced map domain values. `gl:evalPoint()` can be used to evaluate a single grid point in the same gridspace that is traversed by `gl:evalMesh()`. Calling `gl:evalPoint1/1` is equivalent to calling glEvalCoord1( i.ð u +u 1 ); where ð u=(u 2-u 1)/n

**External documentation.**

```
feedbackBuffer(Size :: i(), Type :: enum(), Buffer :: mem()) -> ok
```

The `gl:feedbackBuffer/3` function controls feedback. Feedback, like selection, is a GL mode. The mode is selected by calling `gl:renderMode/1` with `?GL_FEEDBACK`. When the GL is in feedback mode, no pixels are produced by rasterization. Instead, information about primitives that would have been rasterized is fed back to the application using the GL.

**External documentation.**

```
fenceSync(Condition :: enum(), Flags :: i()) -> i()
```

`gl:fenceSync/2` creates a new fence sync object, inserts a fence command into the GL command stream and associates it with that sync object, and returns a non-zero name corresponding to the sync object.

**External documentation.**

```
finish() -> ok
```

`gl:finish/0` does not return until the effects of all previously called GL commands are complete. Such effects include all changes to GL state, all changes to connection state, and all changes to the frame buffer contents.

**External documentation.**

```
flush() -> ok
```

Different GL implementations buffer commands in several different locations, including network buffers and the graphics accelerator itself. `gl:flush/0` empties all of these buffers, causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine. Though this execution may not be completed in any particular time period, it does complete in finite time.

**External documentation.**

```
flushMappedBufferRange(Target :: enum(),
                       Offset :: i(),
                       Length :: i()) ->
                          ok
flushMappedNamedBufferRange(Buffer :: i(),
                            Offset :: i(),
                            Length :: i()) ->
                               ok
```

`gl:flushMappedBufferRange/3` indicates that modifications have been made to a range of a mapped buffer object. The buffer object must previously have been mapped with the `?GL_MAP_FLUSH_EXPLICIT_BIT` flag.

**External documentation.**

```
fogf(Pname :: enum(), Param :: f()) -> ok
fogfv(Pname :: enum(), Params :: tuple()) -> ok
fogi(Pname :: enum(), Param :: i()) -> ok
fogiv(Pname :: enum(), Params :: tuple()) -> ok
```

Fog is initially disabled. While enabled, fog affects rasterized geometry, bitmaps, and pixel blocks, but not buffer clear operations. To enable and disable fog, call `gl:enable/1` and `gl:disable/1` with argument `?GL_FOG`.

**External documentation.**

```
fogCoordd(Coord :: f()) -> ok
fogCoorddv(X1 :: {Coord :: f()}) -> ok
fogCoordf(Coord :: f()) -> ok
fogCoordfv(X1 :: {Coord :: f()}) -> ok
```

`gl:fogCoord()` specifies the fog coordinate that is associated with each vertex and the current raster position. The value specified is interpolated and used in computing the fog color (see `gl:fog()`).

**External documentation.**

```
fogCoordPointer(Type :: enum(),
                Stride :: i(),
                Pointer :: offset() | mem()) ->
                   ok
```

`gl:fogCoordPointer/3` specifies the location and data format of an array of fog coordinates to use when rendering. `Type` specifies the data type of each fog coordinate, and `Stride` specifies the byte stride from one fog coordinate to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays.

**External documentation.**

```
framebufferParameteri(Target :: enum(),
                      Pname :: enum(),
                      Param :: i()) ->
                         ok
```

`gl:framebufferParameteri/3` and `glNamedFramebufferParameteri` modify the value of the parameter named `Pname` in the specified framebuffer object. There are no modifiable parameters of the default draw and read framebuffer, so they are not valid targets of these commands.

**External documentation.**

```
framebufferRenderbuffer(Target, Attachment, Renderbuffertarget,
                        Renderbuffer) ->
                           ok
```

Types:

```
Target = Attachment = Renderbuffertarget = enum()
Renderbuffer = i()
```

`gl:framebufferRenderbuffer/4` and `glNamedFramebufferRenderbuffer` attaches a renderbuffer as one of the logical buffers of the specified framebuffer object. Renderbuffers cannot be attached to the default draw and read framebuffer, so they are not valid targets of these commands.

**External documentation.**

```
framebufferTexture(Target :: enum(),
                   Attachment :: enum(),
                   Texture :: i(),
                   Level :: i()) ->
                      ok
framebufferTexture1D(Target :: enum(),
                     Attachment :: enum(),
                     Textarget :: enum(),
                     Texture :: i(),
                     Level :: i()) ->
                        ok
framebufferTexture2D(Target :: enum(),
                     Attachment :: enum(),
                     Textarget :: enum(),
                     Texture :: i(),
                     Level :: i()) ->
                        ok
framebufferTexture3D(Target, Attachment, Textarget, Texture,
                     Level, Zoffset) ->
                        ok
framebufferTextureFaceARB(Target :: enum(),
                          Attachment :: enum(),
                          Texture :: i(),
                          Level :: i(),
                          Face :: enum()) ->
                             ok
framebufferTextureLayer(Target :: enum(),
                        Attachment :: enum(),
                        Texture :: i(),
                        Level :: i(),
                        Layer :: i()) ->
                           ok
```

These commands attach a selected mipmap level or image of a texture object as one of the logical buffers of the specified framebuffer object. Textures cannot be attached to the default draw and read framebuffer, so they are not valid targets of these commands.

**External documentation.**

```
frontFace(Mode :: enum()) -> ok
```

In a scene composed entirely of opaque closed surfaces, back-facing polygons are never visible. Eliminating these invisible polygons has the obvious benefit of speeding up the rendering of the image. To enable and disable elimination of back-facing polygons, call `gl:enable/1` and `gl:disable/1` with argument `?GL_CULL_FACE`.

**External documentation.**

```
frustum(Left :: f(),
        Right :: f(),
        Bottom :: f(),
        Top :: f(),
        Near_val :: f(),
        Far_val :: f()) ->
```

```
ok
```

`gl:frustum/6` describes a perspective matrix that produces a perspective projection. The current matrix (see `gl:matrixMode/1`) is multiplied by this matrix and the result replaces the current matrix, as if `gl:multMatrix()` were called with the following matrix as its argument:

**External documentation.**

```
genBuffers(N :: i()) -> [i()]
```

`gl:genBuffers/1` returns `N` buffer object names in `Buffers`. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `gl:genBuffers/1`.

**External documentation.**

```
genFramebuffers(N :: i()) -> [i()]
```

`gl:genFramebuffers/1` returns `N` framebuffer object names in `Ids`. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `gl:genFramebuffers/1`.

**External documentation.**

```
genLists(Range :: i()) -> i()
```

`gl:genLists/1` has one argument, `Range`. It returns an integer `n` such that `Range` contiguous empty display lists, named n, n+1, ..., n+range-1, are created. If `Range` is 0, if there is no group of `Range` contiguous names available, or if any error is generated, no display lists are generated, and 0 is returned.

**External documentation.**

```
genProgramPipelines(N :: i()) -> [i()]
```

`gl:genProgramPipelines/1` returns `N` previously unused program pipeline object names in `Pipelines`. These names are marked as used, for the purposes of `gl:genProgramPipelines/1` only, but they acquire program pipeline state only when they are first bound.

**External documentation.**

```
genQueries(N :: i()) -> [i()]
```

`gl:genQueries/1` returns `N` query object names in `Ids`. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `gl:genQueries/1`.

**External documentation.**

```
genRenderbuffers(N :: i()) -> [i()]
```

`gl:genRenderbuffers/1` returns `N` renderbuffer object names in `Renderbuffers`. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `gl:genRenderbuffers/1`.

**External documentation.**

```
genSamplers(Count :: i()) -> [i()]
```

`gl:genSamplers/1` returns `N` sampler object names in `Samplers`. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `gl:genSamplers/1`.

**External documentation.**

```
genTextures(N :: i()) -> [i()]
```

`gl:genTextures/1` returns `N` texture names in `Textures`. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `gl:genTextures/1`.

**External documentation.**

```
genTransformFeedbacks(N :: i()) -> [i()]
```

`gl:genTransformFeedbacks/1` returns `N` previously unused transform feedback object names in `Ids`. These names are marked as used, for the purposes of `gl:genTransformFeedbacks/1` only, but they acquire transform feedback state only when they are first bound.

**External documentation.**

```
genVertexArrays(N :: i()) -> [i()]
```

`gl:genVertexArrays/1` returns `N` vertex array object names in `Arrays`. There is no guarantee that the names form a contiguous set of integers; however, it is guaranteed that none of the returned names was in use immediately before the call to `gl:genVertexArrays/1`.

**External documentation.**

```
generateMipmap(Target :: enum()) -> ok
generateTextureMipmap(Texture :: i()) -> ok
```

`gl:generateMipmap/1` and `gl:generateTextureMipmap/1` generates mipmaps for the specified texture object. For `gl:generateMipmap/1`, the texture object that is bound to `Target`. For `gl:generateTextureMipmap/1`, `Texture` is the name of the texture object.

**External documentation.**

```
getBooleani_v(Target :: enum(), Index :: i()) -> [0 | 1]
getBooleanv(Pname :: enum()) -> [0 | 1]
getDoublei_v(Target :: enum(), Index :: i()) -> [f()]
getDoublev(Pname :: enum()) -> [f()]
getFloati_v(Target :: enum(), Index :: i()) -> [f()]
getFloatv(Pname :: enum()) -> [f()]
getInteger64i_v(Target :: enum(), Index :: i()) -> [i()]
getInteger64v(Pname :: enum()) -> [i()]
getIntegeri_v(Target :: enum(), Index :: i()) -> [i()]
getIntegerv(Pname :: enum()) -> [i()]
```

These commands return values for simple state variables in GL. `Pname` is a symbolic constant indicating the state variable to be returned, and `Data` is a pointer to an array of the indicated type in which to place the returned data.

**External documentation.**

```
getActiveAttrib(Program :: i(), Index :: i(), BufSize :: i()) ->
                   {Size :: i(), Type :: enum(), Name :: string()}
```

gl:getActiveAttrib/3 returns information about an active attribute variable in the program object specified by Program. The number of active attributes can be obtained by calling gl:getProgram() with the value ?GL_ACTIVE_ATTRIBUTES. A value of 0 for Index selects the first active attribute variable. Permissible values for Index range from zero to the number of active attribute variables minus one.

**External documentation.**

```
getActiveSubroutineName(Program :: i(),
                        Shadertype :: enum(),
                        Index :: i(),
                        Bufsize :: i()) ->
                           string()
```

gl:getActiveSubroutineName/4 queries the name of an active shader subroutine uniform from the program object given in Program. Index specifies the index of the shader subroutine uniform within the shader stage given by Stage, and must between zero and the value of ?GL_ACTIVE_SUBROUTINES minus one for the shader stage.

**External documentation.**

```
getActiveSubroutineUniformName(Program :: i(),
                               Shadertype :: enum(),
                               Index :: i(),
                               Bufsize :: i()) ->
                                  string()
```

gl:getActiveSubroutineUniformName/4 retrieves the name of an active shader subroutine uniform. Program contains the name of the program containing the uniform. Shadertype specifies the stage for which the uniform location, given by Index, is valid. Index must be between zero and the value of ?GL_ACTIVE_SUBROUTINE_UNIFORMS minus one for the shader stage.

**External documentation.**

```
getActiveUniform(Program :: i(), Index :: i(), BufSize :: i()) ->
                    {Size :: i(),
                     Type :: enum(),
                     Name :: string()}
```

gl:getActiveUniform/3 returns information about an active uniform variable in the program object specified by Program. The number of active uniform variables can be obtained by calling gl:getProgram() with the value ?GL_ACTIVE_UNIFORMS. A value of 0 for Index selects the first active uniform variable. Permissible values for Index range from zero to the number of active uniform variables minus one.

**External documentation.**

```
getActiveUniformBlockiv(Program :: i(),
                        UniformBlockIndex :: i(),
                        Pname :: enum(),
                        Params :: mem()) ->
                           ok
```

gl:getActiveUniformBlockiv/4 retrieves information about an active uniform block within Program.

**External documentation.**

```
getActiveUniformBlockName(Program :: i(),
                          UniformBlockIndex :: i(),
                          BufSize :: i()) ->
                             string()
```

`gl:getActiveUniformBlockName/3` retrieves the name of the active uniform block at `UniformBlockIndex` within `Program`.

**External documentation.**

```
getActiveUniformName(Program :: i(),
                     UniformIndex :: i(),
                     BufSize :: i()) ->
                        string()
```

`gl:getActiveUniformName/3` returns the name of the active uniform at `UniformIndex` within `Program`. If `UniformName` is not NULL, up to `BufSize` characters (including a nul-terminator) will be written into the array whose address is specified by `UniformName`. If `Length` is not NULL, the number of characters that were (or would have been) written into `UniformName` (not including the nul-terminator) will be placed in the variable whose address is specified in `Length`. If `Length` is NULL, no length is returned. The length of the longest uniform name in `Program` is given by the value of `?GL_ACTIVE_UNIFORM_MAX_LENGTH`, which can be queried with `gl:getProgram()`.

**External documentation.**

```
getActiveUniformsiv(Program :: i(),
                    UniformIndices :: [i()],
                    Pname :: enum()) ->
                       [i()]
```

`gl:getActiveUniformsiv/3` queries the value of the parameter named `Pname` for each of the uniforms within `Program` whose indices are specified in the array of `UniformCount` unsigned integers `UniformIndices`. Upon success, the value of the parameter for each uniform is written into the corresponding entry in the array whose address is given in `Params`. If an error is generated, nothing is written into `Params`.

**External documentation.**

```
getAttachedShaders(Program :: i(), MaxCount :: i()) -> [i()]
```

`gl:getAttachedShaders/2` returns the names of the shader objects attached to `Program`. The names of shader objects that are attached to `Program` will be returned in `Shaders`. The actual number of shader names written into `Shaders` is returned in `Count`. If no shader objects are attached to `Program`, `Count` is set to 0. The maximum number of shader names that may be returned in `Shaders` is specified by `MaxCount`.

**External documentation.**

```
getAttribLocation(Program :: i(), Name :: string()) -> i()
```

`gl:getAttribLocation/2` queries the previously linked program object specified by `Program` for the attribute variable specified by `Name` and returns the index of the generic vertex attribute that is bound to that attribute variable. If `Name` is a matrix attribute variable, the index of the first column of the matrix is returned. If the named attribute variable is not an active attribute in the specified program object or if `Name` starts with the reserved prefix "gl_", a value of -1 is returned.

**External documentation.**

```
getBufferParameteri64v(Target :: enum(), Pname :: enum()) -> [i()]
getBufferParameterivARB(Target :: enum(), Pname :: enum()) ->
                           [i()]
```

These functions return in `Data` a selected parameter of the specified buffer object.

**External documentation.**

```
getBufferParameteriv(Target :: enum(), Pname :: enum()) -> i()
```

`gl:getBufferParameteriv/2` returns in `Data` a selected parameter of the buffer object specified by `Target`.

**External documentation.**

```
getBufferSubData(Target :: enum(),
                 Offset :: i(),
                 Size :: i(),
                 Data :: mem()) ->
                    ok
```

`gl:getBufferSubData/4` and `glGetNamedBufferSubData` return some or all of the data contents of the data store of the specified buffer object. Data starting at byte offset `Offset` and extending for `Size` bytes is copied from the buffer object's data store to the memory pointed to by `Data`. An error is thrown if the buffer object is currently mapped, or if `Offset` and `Size` together define a range beyond the bounds of the buffer object's data store.

**External documentation.**

```
getClipPlane(Plane :: enum()) -> {f(), f(), f(), f()}
```

`gl:getClipPlane/1` returns in `Equation` the four coefficients of the plane equation for `Plane`.

**External documentation.**

```
getColorTable(Target :: enum(),
              Format :: enum(),
              Type :: enum(),
              Table :: mem()) ->
                 ok
```

`gl:getColorTable/4` returns in `Table` the contents of the color table specified by `Target`. No pixel transfer operations are performed, but pixel storage modes that are applicable to `gl:readPixels/7` are performed.

**External documentation.**

```
getColorTableParameterfv(Target :: enum(), Pname :: enum()) ->
                            {f(), f(), f(), f()}
getColorTableParameteriv(Target :: enum(), Pname :: enum()) ->
                            {i(), i(), i(), i()}
```

Returns parameters specific to color table `Target`.

**External documentation.**

```
getCompressedTexImage(Target :: enum(), Lod :: i(), Img :: mem()) ->
                         ok
```

`gl:getCompressedTexImage/3` and `glGetnCompressedTexImage` return the compressed texture image associated with `Target` and `Lod` into `Pixels`. `glGetCompressedTextureImage` serves the same purpose,

but instead of taking a texture target, it takes the ID of the texture object. `Pixels` should be an array of `BufSize` bytes for `glGetnCompresedTexImage` and `glGetCompressedTextureImage` functions, and of `?GL_TEXTURE_COMPRESSED_IMAGE_SIZE` bytes in case of `gl:getCompressedTexImage/3`. If the actual data takes less space than `BufSize`, the remaining bytes will not be touched. `Target` specifies the texture target, to which the texture the data the function should extract the data from is bound to. `Lod` specifies the level-of-detail number of the desired image.

**External documentation.**

```
getConvolutionFilter(Target :: enum(),
                     Format :: enum(),
                     Type :: enum(),
                     Image :: mem()) ->
                        ok
```

`gl:getConvolutionFilter/4` returns the current 1D or 2D convolution filter kernel as an image. The one- or two-dimensional image is placed in `Image` according to the specifications in `Format` and `Type`. No pixel transfer operations are performed on this image, but the relevant pixel storage modes are applied.

**External documentation.**

```
getConvolutionParameterfv(Target :: enum(), Pname :: enum()) ->
                             {f(), f(), f(), f()}
getConvolutionParameteriv(Target :: enum(), Pname :: enum()) ->
                             {i(), i(), i(), i()}
```

`gl:getConvolutionParameter()` retrieves convolution parameters. `Target` determines which convolution filter is queried. `Pname` determines which parameter is returned:

**External documentation.**

```
getDebugMessageLog(Count :: i(), BufSize :: i()) ->
                       {i(),
                        Sources :: [enum()],
                        Types :: [enum()],
                        Ids :: [i()],
                        Severities :: [enum()],
                        MessageLog :: [string()]}
```

`gl:getDebugMessageLog/2` retrieves messages from the debug message log. A maximum of `Count` messages are retrieved from the log. If `Sources` is not NULL then the source of each message is written into up to `Count` elements of the array. If `Types` is not NULL then the type of each message is written into up to `Count` elements of the array. If `Id` is not NULL then the identifier of each message is written into up to `Count` elements of the array. If `Severities` is not NULL then the severity of each message is written into up to `Count` elements of the array. If `Lengths` is not NULL then the length of each message is written into up to `Count` elements of the array.

**External documentation.**

```
getError() -> enum()
```

`gl:getError/0` returns the value of the error flag. Each detectable error is assigned a numeric code and symbolic name. When an error occurs, the error flag is set to the appropriate error code value. No other errors are recorded until `gl:getError/0` is called, the error code is returned, and the flag is reset to `?GL_NO_ERROR`. If a call to `gl:getError/0` returns `?GL_NO_ERROR`, there has been no detectable error since the last call to `gl:getError/0`, or since the GL was initialized.

**External documentation.**

```
getFragDataIndex(Program :: i(), Name :: string()) -> i()
```

`gl:getFragDataIndex/2` returns the index of the fragment color to which the variable `Name` was bound when the program object `Program` was last linked. If `Name` is not a varying out variable of `Program`, or if an error occurs, -1 will be returned.

**External documentation.**

```
getFragDataLocation(Program :: i(), Name :: string()) -> i()
```

`gl:getFragDataLocation/2` retrieves the assigned color number binding for the user-defined varying out variable `Name` for program `Program`. `Program` must have previously been linked. `Name` must be a null-terminated string. If `Name` is not the name of an active user-defined varying out fragment shader variable within `Program`, -1 will be returned.

**External documentation.**

```
getFramebufferAttachmentParameteriv(Target :: enum(),
                                    Attachment :: enum(),
                                    Pname :: enum()) ->
                                       i()
```

`gl:getFramebufferAttachmentParameteriv/3` and `glGetNamedFramebufferAttachmentParameteriv` return parameters of attachments of a specified framebuffer object.

**External documentation.**

```
getFramebufferParameteriv(Target :: enum(), Pname :: enum()) ->
                             i()
```

`gl:getFramebufferParameteriv/2` and `glGetNamedFramebufferParameteriv` query parameters of a specified framebuffer object.

**External documentation.**

```
getGraphicsResetStatus() -> enum()
```

Certain events can result in a reset of the GL context. Such a reset causes all context state to be lost and requires the application to recreate all objects in the affected context.

**External documentation.**

```
getHistogram(Target :: enum(),
             Reset :: 0 | 1,
             Format :: enum(),
             Type :: enum(),
             Values :: mem()) ->
                ok
```

`gl:getHistogram/5` returns the current histogram table as a one-dimensional image with the same width as the histogram. No pixel transfer operations are performed on this image, but pixel storage modes that are applicable to 1D images are honored.

**External documentation.**

```
getHistogramParameterfv(Target :: enum(), Pname :: enum()) ->
                           {f()}
getHistogramParameteriv(Target :: enum(), Pname :: enum()) ->
                           {i()}
```

`gl:getHistogramParameter()` is used to query parameter values for the current histogram or for a proxy. The histogram state information may be queried by calling `gl:getHistogramParameter()` with a `Target` of `?GL_HISTOGRAM` (to obtain information for the current histogram table) or `?GL_PROXY_HISTOGRAM` (to obtain information from the most recent proxy request) and one of the following values for the `Pname` argument:

**External documentation.**

```
getInternalformati64v(Target :: enum(),
                      Internalformat :: enum(),
                      Pname :: enum(),
                      BufSize :: i()) ->
                         [i()]
getInternalformativ(Target :: enum(),
                    Internalformat :: enum(),
                    Pname :: enum(),
                    BufSize :: i()) ->
                       [i()]
```

No documentation available.

```
getLightfv(Light :: enum(), Pname :: enum()) ->
              {f(), f(), f(), f()}
getLightiv(Light :: enum(), Pname :: enum()) ->
              {i(), i(), i(), i()}
```

`gl:getLight()` returns in `Params` the value or values of a light source parameter. `Light` names the light and is a symbolic name of the form `?GL_LIGHT` i where i ranges from 0 to the value of `?GL_MAX_LIGHTS` - 1. `?GL_MAX_LIGHTS` is an implementation dependent constant that is greater than or equal to eight. `Pname` specifies one of ten light source parameters, again by symbolic name.

**External documentation.**

```
getMapdv(Target :: enum(), Query :: enum(), V :: mem()) -> ok
getMapfv(Target :: enum(), Query :: enum(), V :: mem()) -> ok
getMapiv(Target :: enum(), Query :: enum(), V :: mem()) -> ok
```

`glMap1` and `glMap2` define evaluators. `gl:getMap()` returns evaluator parameters. `Target` chooses a map, `Query` selects a specific parameter, and `V` points to storage where the values will be returned.

**External documentation.**

```
getMaterialfv(Face :: enum(), Pname :: enum()) ->
                 {f(), f(), f(), f()}
getMaterialiv(Face :: enum(), Pname :: enum()) ->
                 {i(), i(), i(), i()}
```

`gl:getMaterial()` returns in `Params` the value or values of parameter `Pname` of material `Face`. Six parameters are defined:

**External documentation.**

```
getMinmax(Target :: enum(),
          Reset :: 0 | 1,
          Format :: enum(),
          Types :: enum(),
          Values :: mem()) ->
             ok
```

`gl:getMinmax/5` returns the accumulated minimum and maximum pixel values (computed on a per-component basis) in a one-dimensional image of width 2. The first set of return values are the minima, and the second set of return values are the maxima. The format of the return values is determined by `Format`, and their type is determined by `Types`.

**External documentation.**

```
getMinmaxParameterfv(Target :: enum(), Pname :: enum()) -> {f()}
getMinmaxParameteriv(Target :: enum(), Pname :: enum()) -> {i()}
```

`gl:getMinmaxParameter()` retrieves parameters for the current minmax table by setting `Pname` to one of the following values:

**External documentation.**

```
getMultisamplefv(Pname :: enum(), Index :: i()) -> {f(), f()}
```

`gl:getMultisamplefv/2` queries the location of a given sample. `Pname` specifies the sample parameter to retrieve and must be `?GL_SAMPLE_POSITION`. `Index` corresponds to the sample for which the location should be returned. The sample location is returned as two floating-point values in `Val[0]` and `Val[1]`, each between 0 and 1, corresponding to the X and Y locations respectively in the GL pixel space of that sample. (0.5, 0.5) this corresponds to the pixel center. `Index` must be between zero and the value of `?GL_SAMPLES` minus one.

**External documentation.**

```
getPixelMapfv(Map :: enum(), Values :: mem()) -> ok
getPixelMapuiv(Map :: enum(), Values :: mem()) -> ok
getPixelMapusv(Map :: enum(), Values :: mem()) -> ok
```

See the `gl:pixelMap()` reference page for a description of the acceptable values for the `Map` parameter. `gl:getPixelMap()` returns in `Data` the contents of the pixel map specified in `Map`. Pixel maps are used during the execution of `gl:readPixels/7`, `gl:drawPixels/5`, `gl:copyPixels/5`, `gl:texImage1D/8`, `gl:texImage2D/9`, `gl:texImage3D/10`, `gl:texSubImage1D/7`, `gl:texSubImage2D/9`, `gl:texSubImage3D/11`, `gl:copyTexImage1D/7`, `gl:copyTexImage2D/8`, `gl:copyTexSubImage1D/6`, `gl:copyTexSubImage2D/8`, and `gl:copyTexSubImage3D/9`. to map color indices, stencil indices, color components, and depth components to other values.

**External documentation.**

```
getPolygonStipple() -> binary()
```

`gl:getPolygonStipple/0` returns to `Pattern` a 32×32 polygon stipple pattern. The pattern is packed into memory as if `gl:readPixels/7` with both `height` and `width` of 32, `type` of `?GL_BITMAP`, and `format` of `?GL_COLOR_INDEX` were called, and the stipple pattern were stored in an internal 32×32 color index buffer. Unlike `gl:readPixels/7`, however, pixel transfer operations (shift, offset, pixel map) are not applied to the returned stipple image.

**External documentation.**

```
getProgramiv(Program :: i(), Pname :: enum()) -> i()
```

`gl:getProgram()` returns in `Params` the value of a parameter for a specific program object. The following parameters are defined:

**External documentation.**

```
getProgramBinary(Program :: i(), BufSize :: i()) ->
                    {BinaryFormat :: enum(), Binary :: binary()}
```

`gl:getProgramBinary/2` returns a binary representation of the compiled and linked executable for `Program` into the array of bytes whose address is specified in `Binary`. The maximum number of bytes that may be written into `Binary` is specified by `BufSize`. If the program binary is greater in size than `BufSize` bytes, then an error is generated, otherwise the actual number of bytes written into `Binary` is returned in the variable whose address is given by `Length`. If `Length` is `?NULL`, then no length is returned.

**External documentation.**

```
getProgramInfoLog(Program :: i(), BufSize :: i()) -> string()
```

`gl:getProgramInfoLog/2` returns the information log for the specified program object. The information log for a program object is modified when the program object is linked or validated. The string that is returned will be null terminated.

**External documentation.**

```
getProgramInterfaceiv(Program :: i(),
                      ProgramInterface :: enum(),
                      Pname :: enum()) ->
                         i()
```

`gl:getProgramInterfaceiv/3` queries the property of the interface identifed by `ProgramInterface` in `Program`, the property name of which is given by `Pname`.

**External documentation.**

```
getProgramPipelineiv(Pipeline :: i(), Pname :: enum()) -> i()
```

`gl:getProgramPipelineiv/2` retrieves the value of a property of the program pipeline object `Pipeline`. `Pname` specifies the name of the parameter whose value to retrieve. The value of the parameter is written to the variable whose address is given by `Params`.

**External documentation.**

```
getProgramPipelineInfoLog(Pipeline :: i(), BufSize :: i()) ->
                             string()
```

`gl:getProgramPipelineInfoLog/2` retrieves the info log for the program pipeline object `Pipeline`. The info log, including its null terminator, is written into the array of characters whose address is given by `InfoLog`. The maximum number of characters that may be written into `InfoLog` is given by `BufSize`, and the actual number of characters written into `InfoLog` is returned in the integer whose address is given by `Length`. If `Length` is `?NULL`, no length is returned.

**External documentation.**

```
getProgramResourceIndex(Program :: i(),
                        ProgramInterface :: enum(),
```

```
                        Name :: string()) ->
                           i()
```

gl:getProgramResourceIndex/3 returns the unsigned integer index assigned to a resource named Name in the interface type ProgramInterface of program object Program.

**External documentation.**

```
getProgramResourceLocation(Program :: i(),
                           ProgramInterface :: enum(),
                           Name :: string()) ->
                              i()
```

gl:getProgramResourceLocation/3 returns the location assigned to the variable named Name in interface ProgramInterface of program object Program. Program must be the name of a program that has been linked successfully. ProgramInterface must be one of ?GL_UNIFORM, ?GL_PROGRAM_INPUT, ?GL_PROGRAM_OUTPUT, ?GL_VERTEX_SUBROUTINE_UNIFORM, ?GL_TESS_CONTROL_SUBROUTINE_UNIFORM, ?GL_TESS_EVALUATION_SUBROUTINE_UNIFORM, ?GL_GEOMETRY_SUBROUTINE_UNIFORM, ?GL_FRAGMENT_SUBROUTINE_UNIFORM, ?GL_COMPUTE_SUBROUTINE_UNIFORM, or ?GL_TRANSFORM_FEEDBACK_BUFFER.

**External documentation.**

```
getProgramResourceLocationIndex(Program :: i(),
                                ProgramInterface :: enum(),
                                Name :: string()) ->
                                   i()
```

gl:getProgramResourceLocationIndex/3 returns the fragment color index assigned to the variable named Name in interface ProgramInterface of program object Program. Program must be the name of a program that has been linked successfully. ProgramInterface must be ?GL_PROGRAM_OUTPUT.

**External documentation.**

```
getProgramResourceName(Program :: i(),
                       ProgramInterface :: enum(),
                       Index :: i(),
                       BufSize :: i()) ->
                          string()
```

gl:getProgramResourceName/4 retrieves the name string assigned to the single active resource with an index of Index in the interface ProgramInterface of program object Program. Index must be less than the number of entries in the active resource list for ProgramInterface.

**External documentation.**

```
getProgramStageiv(Program :: i(),
                  Shadertype :: enum(),
                  Pname :: enum()) ->
                     i()
```

gl:getProgramStage() queries a parameter of a shader stage attached to a program object. Program contains the name of the program to which the shader is attached. Shadertype specifies the stage from which to query the parameter. Pname specifies which parameter should be queried. The value or values of the parameter to be queried is returned in the variable whose address is given in Values.

**External documentation.**

```
getQueryIndexediv(Target :: enum(), Index :: i(), Pname :: enum()) ->
                     i()
```

`gl:getQueryIndexediv/3` returns in `Params` a selected parameter of the indexed query object target specified by `Target` and `Index`. `Index` specifies the index of the query object target and must be between zero and a target-specific maxiumum.

**External documentation.**

```
getQueryBufferObjecti64v(Id :: i(),
                         Buffer :: i(),
                         Pname :: enum(),
                         Offset :: i()) ->
                            ok
getQueryBufferObjectiv(Id :: i(),
                       Buffer :: i(),
                       Pname :: enum(),
                       Offset :: i()) ->
                          ok
getQueryBufferObjectui64v(Id :: i(),
                          Buffer :: i(),
                          Pname :: enum(),
                          Offset :: i()) ->
                             ok
getQueryBufferObjectuiv(Id :: i(),
                        Buffer :: i(),
                        Pname :: enum(),
                        Offset :: i()) ->
                           ok
getQueryObjecti64v(Id :: i(), Pname :: enum()) -> i()
getQueryObjectiv(Id :: i(), Pname :: enum()) -> i()
getQueryObjectui64v(Id :: i(), Pname :: enum()) -> i()
getQueryObjectuiv(Id :: i(), Pname :: enum()) -> i()
```

These commands return a selected parameter of the query object specified by `Id`. `gl:getQueryObject()` returns in `Params` a selected parameter of the query object specified by `Id`. `gl:getQueryBufferObject()` returns in `Buffer` a selected parameter of the query object specified by `Id`, by writing it to `Buffer`'s data store at the byte offset specified by `Offset`.

**External documentation.**

```
getQueryiv(Target :: enum(), Pname :: enum()) -> i()
```

`gl:getQueryiv/2` returns in `Params` a selected parameter of the query object target specified by `Target`.

**External documentation.**

```
getRenderbufferParameteriv(Target :: enum(), Pname :: enum()) ->
                              i()
```

`gl:getRenderbufferParameteriv/2` and `glGetNamedRenderbufferParameteriv` query parameters of a specified renderbuffer object.

**External documentation.**

```
getSamplerParameterIiv(Sampler :: i(), Pname :: enum()) -> [i()]
getSamplerParameterIuiv(Sampler :: i(), Pname :: enum()) -> [i()]
getSamplerParameterfv(Sampler :: i(), Pname :: enum()) -> [f()]
getSamplerParameteriv(Sampler :: i(), Pname :: enum()) -> [i()]
```

`gl:getSamplerParameter()` returns in `Params` the value or values of the sampler parameter specified as `Pname`. `Sampler` defines the target sampler, and must be the name of an existing sampler object, returned from a previous call to `gl:genSamplers/1`. `Pname` accepts the same symbols as `gl:samplerParameter()`, with the same interpretations:

**External documentation.**

```
getShaderiv(Shader :: i(), Pname :: enum()) -> i()
```

`gl:getShader()` returns in `Params` the value of a parameter for a specific shader object. The following parameters are defined:

**External documentation.**

```
getShaderInfoLog(Shader :: i(), BufSize :: i()) -> string()
```

`gl:getShaderInfoLog/2` returns the information log for the specified shader object. The information log for a shader object is modified when the shader is compiled. The string that is returned will be null terminated.

**External documentation.**

```
getShaderPrecisionFormat(Shadertype :: enum(),
                         Precisiontype :: enum()) ->
                            {Range :: {i(), i()},
                             Precision :: i()}
```

`gl:getShaderPrecisionFormat/2` retrieves the numeric range and precision for the implementation's representation of quantities in different numeric formats in specified shader type. `ShaderType` specifies the type of shader for which the numeric precision and range is to be retrieved and must be one of `?GL_VERTEX_SHADER` or `?GL_FRAGMENT_SHADER`. `PrecisionType` specifies the numeric format to query and must be one of `?GL_LOW_FLOAT`, `?GL_MEDIUM_FLOAT?GL_HIGH_FLOAT`, `?GL_LOW_INT`, `?GL_MEDIUM_INT`, or `?GL_HIGH_INT`.

**External documentation.**

```
getShaderSource(Shader :: i(), BufSize :: i()) -> string()
```

`gl:getShaderSource/2` returns the concatenation of the source code strings from the shader object specified by `Shader`. The source code strings for a shader object are the result of a previous call to `gl:shaderSource/2`. The string returned by the function will be null terminated.

**External documentation.**

```
getString(Name :: enum()) -> string()
getStringi(Name :: enum(), Index :: i()) -> string()
```

`gl:getString/1` returns a pointer to a static string describing some aspect of the current GL connection. `Name` can be one of the following:

**External documentation.**

```
getSubroutineIndex(Program :: i(),
```

```
                  Shadertype :: enum(),
                  Name :: string()) ->
                     i()
```

`gl:getSubroutineIndex/3` returns the index of a subroutine uniform within a shader stage attached to a program object. `Program` contains the name of the program to which the shader is attached. `Shadertype` specifies the stage from which to query shader subroutine index. `Name` contains the null-terminated name of the subroutine uniform whose name to query.

**External documentation.**

```
getSubroutineUniformLocation(Program :: i(),
                             Shadertype :: enum(),
                             Name :: string()) ->
                                i()
```

`gl:getSubroutineUniformLocation/3` returns the location of the subroutine uniform variable `Name` in the shader stage of type `Shadertype` attached to `Program`, with behavior otherwise identical to `gl:getUniformLocation/2`.

**External documentation.**

```
getSynciv(Sync :: i(), Pname :: enum(), BufSize :: i()) -> [i()]
```

`gl:getSynciv/3` retrieves properties of a sync object. `Sync` specifies the name of the sync object whose properties to retrieve.

**External documentation.**

```
getTexEnvfv(Target :: enum(), Pname :: enum()) ->
               {f(), f(), f(), f()}
getTexEnviv(Target :: enum(), Pname :: enum()) ->
               {i(), i(), i(), i()}
```

`gl:getTexEnv()` returns in `Params` selected values of a texture environment that was specified with `gl:texEnv()`. `Target` specifies a texture environment.

**External documentation.**

```
getTexGendv(Coord :: enum(), Pname :: enum()) ->
               {f(), f(), f(), f()}
getTexGenfv(Coord :: enum(), Pname :: enum()) ->
               {f(), f(), f(), f()}
getTexGeniv(Coord :: enum(), Pname :: enum()) ->
               {i(), i(), i(), i()}
```

`gl:getTexGen()` returns in `Params` selected parameters of a texture coordinate generation function that was specified using `gl:texGen()`. `Coord` names one of the (`s`, `t`, `r`, `q`) texture coordinates, using the symbolic constant `?GL_S`, `?GL_T`, `?GL_R`, or `?GL_Q`.

**External documentation.**

```
getTexImage(Target :: enum(),
            Level :: i(),
            Format :: enum(),
            Type :: enum(),
```

```
            Pixels :: mem()) ->
               ok
```

gl:getTexImage/5, glGetnTexImage and glGetTextureImage functions return a texture image into Pixels. For gl:getTexImage/5 and glGetnTexImage, Target specifies whether the desired texture image is one specified by gl:texImage1D/8 (?GL_TEXTURE_1D), gl:texImage2D/9 (?GL_TEXTURE_1D_ARRAY, ?GL_TEXTURE_RECTANGLE, ?GL_TEXTURE_2D or any of ?GL_TEXTURE_CUBE_MAP_*), or gl:texImage3D/10 (?GL_TEXTURE_2D_ARRAY, ?GL_TEXTURE_3D, ?GL_TEXTURE_CUBE_MAP_ARRAY). For glGetTextureImage, Texture specifies the texture object name. In addition to types of textures accepted by gl:getTexImage/5 and glGetnTexImage, the function also accepts cube map texture objects (with effective target ?GL_TEXTURE_CUBE_MAP). Level specifies the level-of-detail number of the desired image. Format and Type specify the format and type of the desired image array. See the reference page for gl:texImage1D/8 for a description of the acceptable values for the Format and Type parameters, respectively. For glGetnTexImage and glGetTextureImage functions, bufSize tells the size of the buffer to receive the retrieved pixel data. glGetnTexImage and glGetTextureImage do not write more than BufSize bytes into Pixels.

**External documentation.**

```
getTexLevelParameterfv(Target :: enum(),
                       Level :: i(),
                       Pname :: enum()) ->
                          {f()}
getTexLevelParameteriv(Target :: enum(),
                       Level :: i(),
                       Pname :: enum()) ->
                          {i()}
```

gl:getTexLevelParameterfv/3, gl:getTexLevelParameteriv/3, glGetTextureLevelParameterfv and glGetTextureLevelParameteriv return in Params texture parameter values for a specific level-of-detail value, specified as Level. For the first two functions, Target defines the target texture, either ?GL_TEXTURE_1D, ?GL_TEXTURE_2D, ?GL_TEXTURE_3D, ?GL_PROXY_TEXTURE_1D, ?GL_PROXY_TEXTURE_2D, ?GL_PROXY_TEXTURE_3D, ?GL_TEXTURE_CUBE_MAP_POSITIVE_X, ?GL_TEXTURE_CUBE_MAP_NEGATIVE_X, ?GL_TEXTURE_CUBE_MAP_POSITIVE_Y, ?GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, ?GL_TEXTURE_CUBE_MAP_POSITIVE_Z, ?GL_TEXTURE_CUBE_MAP_NEGATIVE_Z, or ?GL_PROXY_TEXTURE_CUBE_MAP. The remaining two take a Texture argument which specifies the name of the texture object.

**External documentation.**

```
getTexParameterIiv(Target :: enum(), Pname :: enum()) ->
                      {i(), i(), i(), i()}
getTexParameterIuiv(Target :: enum(), Pname :: enum()) ->
                       {i(), i(), i(), i()}
getTexParameterfv(Target :: enum(), Pname :: enum()) ->
                     {f(), f(), f(), f()}
getTexParameteriv(Target :: enum(), Pname :: enum()) ->
                    {i(), i(), i(), i()}
```

gl:getTexParameter() and glGetTextureParameter return in Params the value or values of the texture parameter specified as Pname. Target defines the target texture. ?GL_TEXTURE_1D, ?GL_TEXTURE_2D, ?GL_TEXTURE_3D, ?GL_TEXTURE_1D_ARRAY, ?GL_TEXTURE_2D_ARRAY, ?GL_TEXTURE_RECTANGLE, ?GL_TEXTURE_CUBE_MAP, ?GL_TEXTURE_CUBE_MAP_ARRAY, ?GL_TEXTURE_2D_MULTISAMPLE, or ?

GL_TEXTURE_2D_MULTISAMPLE_ARRAY specify one-, two-, or three-dimensional, one-dimensional array, two-dimensional array, rectangle, cube-mapped or cube-mapped array, two-dimensional multisample, or two-dimensional multisample array texturing, respectively. Pname accepts the same symbols as gl:texParameter(), with the same interpretations:

**External documentation.**

```
getTransformFeedbackVarying(Program :: i(),
                            Index :: i(),
                            BufSize :: i()) ->
                               {Size :: i(),
                                Type :: enum(),
                                Name :: string()}
```

Information about the set of varying variables in a linked program that will be captured during transform feedback may be retrieved by calling gl:getTransformFeedbackVarying/3. gl:getTransformFeedbackVarying/3 provides information about the varying variable selected by Index. An Index of 0 selects the first varying variable specified in the Varyings array passed to gl:transformFeedbackVaryings/3, and an Index of the value of ? GL_TRANSFORM_FEEDBACK_VARYINGS minus one selects the last such variable.

**External documentation.**

```
getUniformdv(Program :: i(), Location :: i()) -> matrix()
getUniformfv(Program :: i(), Location :: i()) -> matrix()
getUniformiv(Program :: i(), Location :: i()) ->
                {i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i(),
                 i()}
getUniformuiv(Program :: i(), Location :: i()) ->
                 {i(),
                  i(),
                  i(),
                  i(),
                  i(),
                  i(),
                  i(),
                  i(),
                  i(),
                  i(),
                  i(),
```

```
                  i(),
                  i(),
                  i(),
                  i(),
                  i(),
                  i()}
```

`gl:getUniform()` and `glGetnUniform` return in `Params` the value(s) of the specified uniform variable. The type of the uniform variable specified by `Location` determines the number of values returned. If the uniform variable is defined in the shader as a boolean, int, or float, a single value will be returned. If it is defined as a vec2, ivec2, or bvec2, two values will be returned. If it is defined as a vec3, ivec3, or bvec3, three values will be returned, and so on. To query values stored in uniform variables declared as arrays, call `gl:getUniform()` for each element of the array. To query values stored in uniform variables declared as structures, call `gl:getUniform()` for each field in the structure. The values for uniform variables declared as a matrix will be returned in column major order.

**External documentation.**

```
getUniformBlockIndex(Program :: i(), UniformBlockName :: string()) ->
                        i()
```

`gl:getUniformBlockIndex/2` retrieves the index of a uniform block within `Program`.

**External documentation.**

```
getUniformIndices(Program :: i(),
                  UniformNames :: [unicode:chardata()]) ->
                     [i()]
```

`gl:getUniformIndices/2` retrieves the indices of a number of uniforms within `Program`.

**External documentation.**

```
getUniformLocation(Program :: i(), Name :: string()) -> i()
```

`glGetUniformLocation` returns an integer that represents the location of a specific uniform variable within a program object. `Name` must be a null terminated string that contains no white space. `Name` must be an active uniform variable name in `Program` that is not a structure, an array of structures, or a subcomponent of a vector or a matrix. This function returns -1 if `Name` does not correspond to an active uniform variable in `Program`, if `Name` starts with the reserved prefix "gl_", or if `Name` is associated with an atomic counter or a named uniform block.

**External documentation.**

```
getUniformSubroutineuiv(Shadertype :: enum(), Location :: i()) ->
                           {i(),
                            i(),
                            i(),
                            i(),
                            i(),
                            i(),
                            i(),
                            i(),
                            i(),
                            i(),
                            i(),
                            i(),
                            i(),
                            i(),
```

```
                          i(),
                          i(),
                          i()}
```

`gl:getUniformSubroutine()` retrieves the value of the subroutine uniform at location `Location` for shader stage `Shadertype` of the current program. `Location` must be less than the value of `?GL_ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS` for the shader currently in use at shader stage `Shadertype`. The value of the subroutine uniform is returned in `Values`.

**External documentation.**

```
getVertexAttribIiv(Index :: i(), Pname :: enum()) ->
                      {i(), i(), i(), i()}
getVertexAttribIuiv(Index :: i(), Pname :: enum()) ->
                       {i(), i(), i(), i()}
getVertexAttribLdv(Index :: i(), Pname :: enum()) ->
                      {f(), f(), f(), f()}
getVertexAttribdv(Index :: i(), Pname :: enum()) ->
                     {f(), f(), f(), f()}
getVertexAttribfv(Index :: i(), Pname :: enum()) ->
                     {f(), f(), f(), f()}
getVertexAttribiv(Index :: i(), Pname :: enum()) ->
                     {i(), i(), i(), i()}
```

`gl:getVertexAttrib()` returns in `Params` the value of a generic vertex attribute parameter. The generic vertex attribute to be queried is specified by `Index`, and the parameter to be queried is specified by `Pname`.

**External documentation.**

```
hint(Target :: enum(), Mode :: enum()) -> ok
```

Certain aspects of GL behavior, when there is room for interpretation, can be controlled with hints. A hint is specified with two arguments. `Target` is a symbolic constant indicating the behavior to be controlled, and `Mode` is another symbolic constant indicating the desired behavior. The initial value for each `Target` is `?GL_DONT_CARE`. `Mode` can be one of the following:

**External documentation.**

```
histogram(Target :: enum(),
          Width :: i(),
          Internalformat :: enum(),
          Sink :: 0 | 1) ->
             ok
```

When `?GL_HISTOGRAM` is enabled, RGBA color components are converted to histogram table indices by clamping to the range [0,1], multiplying by the width of the histogram table, and rounding to the nearest integer. The table entries selected by the RGBA indices are then incremented. (If the internal format of the histogram table includes luminance, then the index derived from the R color component determines the luminance table entry to be incremented.) If a histogram table entry is incremented beyond its maximum value, then its value becomes undefined. (This is not an error.)

**External documentation.**

```
indexd(C :: f()) -> ok
indexdv(X1 :: {C :: f()}) -> ok
indexf(C :: f()) -> ok
indexfv(X1 :: {C :: f()}) -> ok
indexi(C :: i()) -> ok
indexiv(X1 :: {C :: i()}) -> ok
indexs(C :: i()) -> ok
indexsv(X1 :: {C :: i()}) -> ok
indexub(C :: i()) -> ok
indexubv(X1 :: {C :: i()}) -> ok
```

`gl:index()` updates the current (single-valued) color index. It takes one argument, the new value for the current color index.

**External documentation.**

```
indexMask(Mask :: i()) -> ok
```

`gl:indexMask/1` controls the writing of individual bits in the color index buffers. The least significant n bits of `Mask`, where n is the number of bits in a color index buffer, specify a mask. Where a 1 (one) appears in the mask, it's possible to write to the corresponding bit in the color index buffer (or buffers). Where a 0 (zero) appears, the corresponding bit is write-protected.

**External documentation.**

```
indexPointer(Type :: enum(),
             Stride :: i(),
             Ptr :: offset() | mem()) ->
                ok
```

`gl:indexPointer/3` specifies the location and data format of an array of color indexes to use when rendering. `Type` specifies the data type of each color index and `Stride` specifies the byte stride from one color index to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays.

**External documentation.**

```
initNames() -> ok
```

The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers. `gl:initNames/0` causes the name stack to be initialized to its default empty state.

**External documentation.**

```
interleavedArrays(Format :: enum(),
                  Stride :: i(),
                  Pointer :: offset() | mem()) ->
                     ok
```

`gl:interleavedArrays/3` lets you specify and enable individual color, normal, texture and vertex arrays whose elements are part of a larger aggregate array element. For some implementations, this is more efficient than specifying the arrays separately.

**External documentation.**

```
invalidateBufferData(Buffer :: i()) -> ok
```

`gl:invalidateBufferData/1` invalidates all of the content of the data store of a buffer object. After invalidation, the content of the buffer's data store becomes undefined.

**External documentation.**

```
invalidateBufferSubData(Buffer :: i(),
                        Offset :: i(),
                        Length :: i()) ->
                           ok
```

`gl:invalidateBufferSubData/3` invalidates all or part of the content of the data store of a buffer object. After invalidation, the content of the specified range of the buffer's data store becomes undefined. The start of the range is given by `Offset` and its size is given by `Length`, both measured in basic machine units.

**External documentation.**

```
invalidateFramebuffer(Target :: enum(), Attachments :: [enum()]) ->
                         ok
```

`gl:invalidateFramebuffer/2` and `glInvalidateNamedFramebufferData` invalidate the entire contents of a specified set of attachments of a framebuffer.

**External documentation.**

```
invalidateSubFramebuffer(Target :: enum(),
                         Attachments :: [enum()],
                         X :: i(),
                         Y :: i(),
                         Width :: i(),
                         Height :: i()) ->
                            ok
```

`gl:invalidateSubFramebuffer/6` and `glInvalidateNamedFramebufferSubData` invalidate the contents of a specified region of a specified set of attachments of a framebuffer.

**External documentation.**

```
invalidateTexImage(Texture :: i(), Level :: i()) -> ok
```

`gl:invalidateTexSubImage/8` invalidates all of a texture image. `Texture` and `Level` indicated which texture image is being invalidated. After this command, data in the texture image has undefined values.

**External documentation.**

```
invalidateTexSubImage(Texture, Level, Xoffset, Yoffset, Zoffset,
                      Width, Height, Depth) ->
                         ok
```

Types:

```
Texture = Level = Xoffset = Yoffset = Zoffset = Width = Height = Depth =
i()
```

`gl:invalidateTexSubImage/8` invalidates all or part of a texture image. `Texture` and `Level` indicated which texture image is being invalidated. After this command, data in that subregion have undefined values. `Xoffset`,

Yoffset, Zoffset, Width, Height, and Depth are interpreted as they are in gl:texSubImage3D/11. For texture targets that don't have certain dimensions, this command treats those dimensions as having a size of 1. For example, to invalidate a portion of a two- dimensional texture, the application would use Zoffset equal to zero and Depth equal to one. Cube map textures are treated as an array of six slices in the z-dimension, where a value of Zoffset is interpreted as specifying face ?GL_TEXTURE_CUBE_MAP_POSITIVE_X + Zoffset.

**External documentation.**

```
isBuffer(Buffer :: i()) -> 0 | 1
```

gl:isBuffer/1 returns ?GL_TRUE if Buffer is currently the name of a buffer object. If Buffer is zero, or is a non-zero value that is not currently the name of a buffer object, or if an error occurs, gl:isBuffer/1 returns ?GL_FALSE.

**External documentation.**

```
isEnabled(Cap :: enum()) -> 0 | 1
isEnabledi(Target :: enum(), Index :: i()) -> 0 | 1
```

gl:isEnabled/1 returns ?GL_TRUE if Cap is an enabled capability and returns ?GL_FALSE otherwise. Boolean states that are indexed may be tested with gl:isEnabledi/2. For gl:isEnabledi/2, Index specifies the index of the capability to test. Index must be between zero and the count of indexed capabilities for Cap. Initially all capabilities except ?GL_DITHER are disabled; ?GL_DITHER is initially enabled.

**External documentation.**

```
isFramebuffer(Framebuffer :: i()) -> 0 | 1
```

gl:isFramebuffer/1 returns ?GL_TRUE if Framebuffer is currently the name of a framebuffer object. If Framebuffer is zero, or if ?framebuffer is not the name of a framebuffer object, or if an error occurs, gl:isFramebuffer/1 returns ?GL_FALSE. If Framebuffer is a name returned by gl:genFramebuffers/1, by that has not yet been bound through a call to gl:bindFramebuffer/2, then the name is not a framebuffer object and gl:isFramebuffer/1 returns ?GL_FALSE.

**External documentation.**

```
isList(List :: i()) -> 0 | 1
```

gl:isList/1 returns ?GL_TRUE if List is the name of a display list and returns ?GL_FALSE if it is not, or if an error occurs.

**External documentation.**

```
isProgram(Program :: i()) -> 0 | 1
```

gl:isProgram/1 returns ?GL_TRUE if Program is the name of a program object previously created with gl:createProgram/0 and not yet deleted with gl:deleteProgram/1. If Program is zero or a non-zero value that is not the name of a program object, or if an error occurs, gl:isProgram/1 returns ?GL_FALSE.

**External documentation.**

```
isProgramPipeline(Pipeline :: i()) -> 0 | 1
```

gl:isProgramPipeline/1 returns ?GL_TRUE if Pipeline is currently the name of a program pipeline object. If Pipeline is zero, or if ?pipeline is not the name of a program pipeline object, or if an error occurs, gl:isProgramPipeline/1 returns ?GL_FALSE. If Pipeline is a name returned by gl:genProgramPipelines/1, but that has not yet been bound through

a call to gl:bindProgramPipeline/1, then the name is not a program pipeline object and gl:isProgramPipeline/1 returns ?GL_FALSE.

**External documentation.**

### isQuery(Id :: i()) -> 0 | 1

gl:isQuery/1 returns ?GL_TRUE if Id is currently the name of a query object. If Id is zero, or is a non-zero value that is not currently the name of a query object, or if an error occurs, gl:isQuery/1 returns ?GL_FALSE.

**External documentation.**

### isRenderbuffer(Renderbuffer :: i()) -> 0 | 1

gl:isRenderbuffer/1 returns ?GL_TRUE if Renderbuffer is currently the name of a renderbuffer object. If Renderbuffer is zero, or if Renderbuffer is not the name of a renderbuffer object, or if an error occurs, gl:isRenderbuffer/1 returns ?GL_FALSE. If Renderbuffer is a name returned by gl:genRenderbuffers/1, by that has not yet been bound through a call to gl:bindRenderbuffer/2 or gl:framebufferRenderbuffer/4, then the name is not a renderbuffer object and gl:isRenderbuffer/1 returns ?GL_FALSE.

**External documentation.**

### isSampler(Sampler :: i()) -> 0 | 1

gl:isSampler/1 returns ?GL_TRUE if Id is currently the name of a sampler object. If Id is zero, or is a non-zero value that is not currently the name of a sampler object, or if an error occurs, gl:isSampler/1 returns ?GL_FALSE.

**External documentation.**

### isShader(Shader :: i()) -> 0 | 1

gl:isShader/1 returns ?GL_TRUE if Shader is the name of a shader object previously created with gl:createShader/1 and not yet deleted with gl:deleteShader/1. If Shader is zero or a non-zero value that is not the name of a shader object, or if an error occurs, glIsShader returns ?GL_FALSE.

**External documentation.**

### isSync(Sync :: i()) -> 0 | 1

gl:isSync/1 returns ?GL_TRUE if Sync is currently the name of a sync object. If Sync is not the name of a sync object, or if an error occurs, gl:isSync/1 returns ?GL_FALSE. Note that zero is not the name of a sync object.

**External documentation.**

### isTexture(Texture :: i()) -> 0 | 1

gl:isTexture/1 returns ?GL_TRUE if Texture is currently the name of a texture. If Texture is zero, or is a non-zero value that is not currently the name of a texture, or if an error occurs, gl:isTexture/1 returns ?GL_FALSE.

**External documentation.**

### isTransformFeedback(Id :: i()) -> 0 | 1

gl:isTransformFeedback/1 returns ?GL_TRUE if Id is currently the name of a transform feedback object. If Id is zero, or if ?id is not the name of a transform feedback object,

or if an error occurs, `gl:isTransformFeedback/1` returns `?GL_FALSE`. If `Id` is a name returned by `gl:genTransformFeedbacks/1`, but that has not yet been bound through a call to `gl:bindTransformFeedback/2`, then the name is not a transform feedback object and `gl:isTransformFeedback/1` returns `?GL_FALSE`.

**External documentation.**

```
isVertexArray(Array :: i()) -> 0 | 1
```

`gl:isVertexArray/1` returns `?GL_TRUE` if `Array` is currently the name of a vertex array object. If `Array` is zero, or if `Array` is not the name of a vertex array object, or if an error occurs, `gl:isVertexArray/1` returns `?GL_FALSE`. If `Array` is a name returned by `gl:genVertexArrays/1`, by that has not yet been bound through a call to `gl:bindVertexArray/1`, then the name is not a vertex array object and `gl:isVertexArray/1` returns `?GL_FALSE`.

**External documentation.**

```
lightf(Light :: enum(), Pname :: enum(), Param :: f()) -> ok
lightfv(Light :: enum(), Pname :: enum(), Params :: tuple()) -> ok
lighti(Light :: enum(), Pname :: enum(), Param :: i()) -> ok
lightiv(Light :: enum(), Pname :: enum(), Params :: tuple()) -> ok
```

`gl:light()` sets the values of individual light source parameters. `Light` names the light and is a symbolic name of the form `?GL_LIGHT` i, where i ranges from 0 to the value of `?GL_MAX_LIGHTS` - 1. `Pname` specifies one of ten light source parameters, again by symbolic name. `Params` is either a single value or a pointer to an array that contains the new values.

**External documentation.**

```
lightModelf(Pname :: enum(), Param :: f()) -> ok
lightModelfv(Pname :: enum(), Params :: tuple()) -> ok
lightModeli(Pname :: enum(), Param :: i()) -> ok
lightModeliv(Pname :: enum(), Params :: tuple()) -> ok
```

`gl:lightModel()` sets the lighting model parameter. `Pname` names a parameter and `Params` gives the new value. There are three lighting model parameters:

**External documentation.**

```
lineStipple(Factor :: i(), Pattern :: i()) -> ok
```

Line stippling masks out certain fragments produced by rasterization; those fragments will not be drawn. The masking is achieved by using three parameters: the 16-bit line stipple pattern `Pattern`, the repeat count `Factor`, and an integer stipple counter s.

**External documentation.**

```
lineWidth(Width :: f()) -> ok
```

`gl:lineWidth/1` specifies the rasterized width of both aliased and antialiased lines. Using a line width other than 1 has different effects, depending on whether line antialiasing is enabled. To enable and disable line antialiasing, call `gl:enable/1` and `gl:disable/1` with argument `?GL_LINE_SMOOTH`. Line antialiasing is initially disabled.

**External documentation.**

```
linkProgram(Program :: i()) -> ok
```

`gl:linkProgram/1` links the program object specified by `Program`. If any shader objects of type `?GL_VERTEX_SHADER` are attached to `Program`, they will be used to create an executable that will run on the programmable vertex processor. If any shader objects of type `?GL_GEOMETRY_SHADER` are attached to `Program`, they will be used to create an executable that will run on the programmable geometry processor. If any shader objects of type `?GL_FRAGMENT_SHADER` are attached to `Program`, they will be used to create an executable that will run on the programmable fragment processor.

**External documentation.**

```
listBase(Base :: i()) -> ok
```

`gl:callLists/1` specifies an array of offsets. Display-list names are generated by adding `Base` to each offset. Names that reference valid display lists are executed; the others are ignored.

**External documentation.**

```
loadIdentity() -> ok
```

`gl:loadIdentity/0` replaces the current matrix with the identity matrix. It is semantically equivalent to calling `gl:loadMatrix()` with the identity matrix

**External documentation.**

```
loadMatrixd(M :: matrix()) -> ok
loadMatrixf(M :: matrix()) -> ok
```

`gl:loadMatrix()` replaces the current matrix with the one whose elements are specified by `M`. The current matrix is the projection matrix, modelview matrix, or texture matrix, depending on the current matrix mode (see `gl:matrixMode/1`).

**External documentation.**

```
loadName(Name :: i()) -> ok
```

The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers and is initially empty.

**External documentation.**

```
loadTransposeMatrixd(M :: matrix()) -> ok
loadTransposeMatrixf(M :: matrix()) -> ok
```

`gl:loadTransposeMatrix()` replaces the current matrix with the one whose elements are specified by `M`. The current matrix is the projection matrix, modelview matrix, or texture matrix, depending on the current matrix mode (see `gl:matrixMode/1`).

**External documentation.**

```
logicOp(Opcode :: enum()) -> ok
```

`gl:logicOp/1` specifies a logical operation that, when enabled, is applied between the incoming RGBA color and the RGBA color at the corresponding location in the frame buffer. To enable or disable the logical operation, call `gl:enable/1` and `gl:disable/1` using the symbolic constant `?GL_COLOR_LOGIC_OP`. The initial value is disabled.

**External documentation.**

```
map1d(Target :: enum(),
      U1 :: f(),
      U2 :: f(),
      Stride :: i(),
      Order :: i(),
      Points :: binary()) ->
         ok
map1f(Target :: enum(),
      U1 :: f(),
      U2 :: f(),
      Stride :: i(),
      Order :: i(),
      Points :: binary()) ->
         ok
```

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent to further stages of GL processing just as if they had been presented using `gl:vertex()`, `gl:normal()`, `gl:texCoord()`, and `gl:color()` commands, except that the generated values do not update the current normal, texture coordinates, or color.

**External documentation.**

```
map2d(Target, U1, U2, Ustride, Uorder, V1, V2, Vstride, Vorder,
      Points) ->
         ok
map2f(Target, U1, U2, Ustride, Uorder, V1, V2, Vstride, Vorder,
      Points) ->
         ok
```

Types:

```
   Target = enum()
   U1 = U2 = f()
   Ustride = Uorder = i()
   V1 = V2 = f()
   Vstride = Vorder = i()
   Points = binary()
```

Evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates, and colors. The values produced by an evaluator are sent on to further stages of GL processing just as if they had been presented using `gl:vertex()`, `gl:normal()`, `gl:texCoord()`, and `gl:color()` commands, except that the generated values do not update the current normal, texture coordinates, or color.

**External documentation.**

```
mapGrid1d(Un :: i(), U1 :: f(), U2 :: f()) -> ok
mapGrid1f(Un :: i(), U1 :: f(), U2 :: f()) -> ok
mapGrid2d(Un :: i(),
          U1 :: f(),
          U2 :: f(),
          Vn :: i(),
          V1 :: f(),
          V2 :: f()) ->
```

```
             ok
mapGrid2f(Un :: i(),
          U1 :: f(),
          U2 :: f(),
          Vn :: i(),
          V1 :: f(),
          V2 :: f()) ->
             ok
```

`gl:mapGrid()` and `gl:evalMesh()` are used together to efficiently generate and evaluate a series of evenly-spaced map domain values. `gl:evalMesh()` steps through the integer domain of a one- or two-dimensional grid, whose range is the domain of the evaluation maps specified by `glMap1` and `glMap2`.

**External documentation.**

```
materialf(Face :: enum(), Pname :: enum(), Param :: f()) -> ok
materialfv(Face :: enum(), Pname :: enum(), Params :: tuple()) ->
              ok
materiali(Face :: enum(), Pname :: enum(), Param :: i()) -> ok
materialiv(Face :: enum(), Pname :: enum(), Params :: tuple()) ->
              ok
```

`gl:material()` assigns values to material parameters. There are two matched sets of material parameters. One, the `front-facing` set, is used to shade points, lines, bitmaps, and all polygons (when two-sided lighting is disabled), or just front-facing polygons (when two-sided lighting is enabled). The other set, `back-facing`, is used to shade back-facing polygons only when two-sided lighting is enabled. Refer to the `gl:lightModel()` reference page for details concerning one- and two-sided lighting calculations.

**External documentation.**

```
matrixMode(Mode :: enum()) -> ok
```

`gl:matrixMode/1` sets the current matrix mode. `Mode` can assume one of four values:

**External documentation.**

```
memoryBarrier(Barriers :: i()) -> ok
memoryBarrierByRegion(Barriers :: i()) -> ok
```

`gl:memoryBarrier/1` defines a barrier ordering the memory transactions issued prior to the command relative to those issued after the barrier. For the purposes of this ordering, memory transactions performed by shaders are considered to be issued by the rendering command that triggered the execution of the shader. `Barriers` is a bitfield indicating the set of operations that are synchronized with shader stores; the bits used in `Barriers` are as follows:

**External documentation.**

```
minSampleShading(Value :: f()) -> ok
```

`gl:minSampleShading/1` specifies the rate at which samples are shaded within a covered pixel. Sample-rate shading is enabled by calling `gl:enable/1` with the parameter `?GL_SAMPLE_SHADING`. If `?GL_MULTISAMPLE` or `?GL_SAMPLE_SHADING` is disabled, sample shading has no effect. Otherwise, an implementation must provide at least as many unique color values for each covered fragment as specified by `Value` times `Samples` where `Samples` is the value of `?GL_SAMPLES` for the current framebuffer. At least 1 sample for each covered fragment is generated.

**External documentation.**

```
minmax(Target :: enum(), Internalformat :: enum(), Sink :: 0 | 1) ->
          ok
```

When `?GL_MINMAX` is enabled, the RGBA components of incoming pixels are compared to the minimum and maximum values for each component, which are stored in the two-element minmax table. (The first element stores the minima, and the second element stores the maxima.) If a pixel component is greater than the corresponding component in the maximum element, then the maximum element is updated with the pixel component value. If a pixel component is less than the corresponding component in the minimum element, then the minimum element is updated with the pixel component value. (In both cases, if the internal format of the minmax table includes luminance, then the R color component of incoming pixels is used for comparison.) The contents of the minmax table may be retrieved at a later time by calling `gl:getMinmax/5`. The minmax operation is enabled or disabled by calling `gl:enable/1` or `gl:disable/1`, respectively, with an argument of `?GL_MINMAX`.

**External documentation.**

```
multMatrixd(M :: matrix()) -> ok
multMatrixf(M :: matrix()) -> ok
```

`gl:multMatrix()` multiplies the current matrix with the one specified using `M`, and replaces the current matrix with the product.

**External documentation.**

```
multTransposeMatrixd(M :: matrix()) -> ok
multTransposeMatrixf(M :: matrix()) -> ok
```

`gl:multTransposeMatrix()` multiplies the current matrix with the one specified using `M`, and replaces the current matrix with the product.

**External documentation.**

```
multiDrawArrays(Mode :: enum(),
                First :: [integer()] | mem(),
                Count :: [integer()] | mem()) ->
                   ok
```

`gl:multiDrawArrays/3` specifies multiple sets of geometric primitives with very few subroutine calls. Instead of calling a GL procedure to pass each individual vertex, normal, texture coordinate, edge flag, or color, you can prespecify separate arrays of vertices, normals, and colors and use them to construct a sequence of primitives with a single call to `gl:multiDrawArrays/3`.

**External documentation.**

```
multiDrawArraysIndirect(Mode :: enum(),
                        Indirect :: offset() | mem(),
                        Drawcount :: i(),
                        Stride :: i()) ->
                           ok
```

`gl:multiDrawArraysIndirect/4` specifies multiple geometric primitives with very few subroutine calls. `gl:multiDrawArraysIndirect/4` behaves similarly to a multitude of calls to `gl:drawArraysInstancedBaseInstance/5`, execept that the parameters to each call to `gl:drawArraysInstancedBaseInstance/5` are stored in an array in memory at the address given by `Indirect`, separated by the stride, in basic machine units, specified by `Stride`. If `Stride` is zero, then the array is assumed to be tightly packed in memory.

**External documentation.**

```
multiDrawArraysIndirectCount(Mode, Indirect, Drawcount,
                             Maxdrawcount, Stride) ->
                                ok
```

Types:

```
Mode = enum()
Indirect = offset() | mem()
Drawcount = Maxdrawcount = Stride = i()
```

No documentation available.

```
multiTexCoord1d(Target :: enum(), S :: f()) -> ok
multiTexCoord1dv(Target :: enum(), X2 :: {S :: f()}) -> ok
multiTexCoord1f(Target :: enum(), S :: f()) -> ok
multiTexCoord1fv(Target :: enum(), X2 :: {S :: f()}) -> ok
multiTexCoord1i(Target :: enum(), S :: i()) -> ok
multiTexCoord1iv(Target :: enum(), X2 :: {S :: i()}) -> ok
multiTexCoord1s(Target :: enum(), S :: i()) -> ok
multiTexCoord1sv(Target :: enum(), X2 :: {S :: i()}) -> ok
multiTexCoord2d(Target :: enum(), S :: f(), T :: f()) -> ok
multiTexCoord2dv(Target :: enum(), X2 :: {S :: f(), T :: f()}) ->
                    ok
multiTexCoord2f(Target :: enum(), S :: f(), T :: f()) -> ok
multiTexCoord2fv(Target :: enum(), X2 :: {S :: f(), T :: f()}) ->
                    ok
multiTexCoord2i(Target :: enum(), S :: i(), T :: i()) -> ok
multiTexCoord2iv(Target :: enum(), X2 :: {S :: i(), T :: i()}) ->
                    ok
multiTexCoord2s(Target :: enum(), S :: i(), T :: i()) -> ok
multiTexCoord2sv(Target :: enum(), X2 :: {S :: i(), T :: i()}) ->
                    ok
multiTexCoord3d(Target :: enum(), S :: f(), T :: f(), R :: f()) ->
                   ok
multiTexCoord3dv(Target :: enum(),
                 X2 :: {S :: f(), T :: f(), R :: f()}) ->
                    ok
multiTexCoord3f(Target :: enum(), S :: f(), T :: f(), R :: f()) ->
                   ok
multiTexCoord3fv(Target :: enum(),
                 X2 :: {S :: f(), T :: f(), R :: f()}) ->
                    ok
multiTexCoord3i(Target :: enum(), S :: i(), T :: i(), R :: i()) ->
                   ok
multiTexCoord3iv(Target :: enum(),
                 X2 :: {S :: i(), T :: i(), R :: i()}) ->
```

```
                    ok
multiTexCoord3s(Target :: enum(), S :: i(), T :: i(), R :: i()) ->
                   ok
multiTexCoord3sv(Target :: enum(),
                 X2 :: {S :: i(), T :: i(), R :: i()}) ->
                    ok
multiTexCoord4d(Target :: enum(),
                S :: f(),
                T :: f(),
                R :: f(),
                Q :: f()) ->
                   ok
multiTexCoord4dv(Target :: enum(),
                 X2 :: {S :: f(), T :: f(), R :: f(), Q :: f()}) ->
                    ok
multiTexCoord4f(Target :: enum(),
                S :: f(),
                T :: f(),
                R :: f(),
                Q :: f()) ->
                   ok
multiTexCoord4fv(Target :: enum(),
                 X2 :: {S :: f(), T :: f(), R :: f(), Q :: f()}) ->
                    ok
multiTexCoord4i(Target :: enum(),
                S :: i(),
                T :: i(),
                R :: i(),
                Q :: i()) ->
                   ok
multiTexCoord4iv(Target :: enum(),
                 X2 :: {S :: i(), T :: i(), R :: i(), Q :: i()}) ->
                    ok
multiTexCoord4s(Target :: enum(),
                S :: i(),
                T :: i(),
                R :: i(),
                Q :: i()) ->
                   ok
multiTexCoord4sv(Target :: enum(),
                 X2 :: {S :: i(), T :: i(), R :: i(), Q :: i()}) ->
                    ok
```

`gl:multiTexCoord()` specifies texture coordinates in one, two, three, or four dimensions. `gl:multiTexCoord1()` sets the current texture coordinates to (s 0 0 1); a call to `gl:multiTexCoord2()` sets them to (s t 0 1). Similarly, `gl:multiTexCoord3()` specifies the texture coordinates as (s t r 1), and `gl:multiTexCoord4()` defines all four components explicitly as (s t r q).

**External documentation.**

```
endList() -> ok
newList(List :: i(), Mode :: enum()) -> ok
```

Display lists are groups of GL commands that have been stored for subsequent execution. Display lists are created with `gl:newList/2`. All subsequent commands are placed in the display list, in the order issued, until `gl:endList/0` is called.

**External documentation.**

```
normal3b(Nx :: i(), Ny :: i(), Nz :: i()) -> ok
normal3bv(X1 :: {Nx :: i(), Ny :: i(), Nz :: i()}) -> ok
normal3d(Nx :: f(), Ny :: f(), Nz :: f()) -> ok
normal3dv(X1 :: {Nx :: f(), Ny :: f(), Nz :: f()}) -> ok
normal3f(Nx :: f(), Ny :: f(), Nz :: f()) -> ok
normal3fv(X1 :: {Nx :: f(), Ny :: f(), Nz :: f()}) -> ok
normal3i(Nx :: i(), Ny :: i(), Nz :: i()) -> ok
normal3iv(X1 :: {Nx :: i(), Ny :: i(), Nz :: i()}) -> ok
normal3s(Nx :: i(), Ny :: i(), Nz :: i()) -> ok
normal3sv(X1 :: {Nx :: i(), Ny :: i(), Nz :: i()}) -> ok
```

The current normal is set to the given coordinates whenever `gl:normal()` is issued. Byte, short, or integer arguments are converted to floating-point format with a linear mapping that maps the most positive representable integer value to 1.0 and the most negative representable integer value to -1.0.

**External documentation.**

```
normalPointer(Type :: enum(),
              Stride :: i(),
              Ptr :: offset() | mem()) ->
                 ok
```

`gl:normalPointer/3` specifies the location and data format of an array of normals to use when rendering. `Type` specifies the data type of each normal coordinate, and `Stride` specifies the byte stride from one normal to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see `gl:interleavedArrays/3`.)

**External documentation.**

```
objectPtrLabel(Ptr :: offset() | mem(),
               Length :: i(),
               Label :: string()) ->
                  ok
```

`gl:objectPtrLabel/3` labels the sync object identified by `Ptr`.

**External documentation.**

```
ortho(Left :: f(),
      Right :: f(),
      Bottom :: f(),
      Top :: f(),
      Near_val :: f(),
      Far_val :: f()) ->
```

```
          ok
```

`gl:ortho/6` describes a transformation that produces a parallel projection. The current matrix (see `gl:matrixMode/1`) is multiplied by this matrix and the result replaces the current matrix, as if `gl:multMatrix()` were called with the following matrix as its argument:

**External documentation.**

```
passThrough(Token :: f()) -> ok
```

**External documentation.**

```
patchParameterfv(Pname :: enum(), Values :: [f()]) -> ok
patchParameteri(Pname :: enum(), Value :: i()) -> ok
```

`gl:patchParameter()` specifies the parameters that will be used for patch primitives. `Pname` specifies the parameter to modify and must be either `?GL_PATCH_VERTICES`, `?GL_PATCH_DEFAULT_OUTER_LEVEL` or `?GL_PATCH_DEFAULT_INNER_LEVEL`. For `gl:patchParameteri/2`, `Value` specifies the new value for the parameter specified by `Pname`. For `gl:patchParameterfv/2`, `Values` specifies the address of an array containing the new values for the parameter specified by `Pname`.

**External documentation.**

```
pauseTransformFeedback() -> ok
```

`gl:pauseTransformFeedback/0` pauses transform feedback operations on the currently active transform feedback object. When transform feedback operations are paused, transform feedback is still considered active and changing most transform feedback state related to the object results in an error. However, a new transform feedback object may be bound while transform feedback is paused.

**External documentation.**

```
pixelMapfv(Map :: enum(), Mapsize :: i(), Values :: binary()) ->
              ok
pixelMapuiv(Map :: enum(), Mapsize :: i(), Values :: binary()) ->
               ok
pixelMapusv(Map :: enum(), Mapsize :: i(), Values :: binary()) ->
               ok
```

`gl:pixelMap()` sets up translation tables, or `maps`, used by `gl:copyPixels/5`, `gl:copyTexImage1D/7`, `gl:copyTexImage2D/8`, `gl:copyTexSubImage1D/6`, `gl:copyTexSubImage2D/8`, `gl:copyTexSubImage3D/9`, `gl:drawPixels/5`, `gl:readPixels/7`, `gl:texImage1D/8`, `gl:texImage2D/9`, `gl:texImage3D/10`, `gl:texSubImage1D/7`, `gl:texSubImage2D/9`, and `gl:texSubImage3D/11`. Additionally, if the ARB_imaging subset is supported, the routines `gl:colorTable/6`, `gl:colorSubTable/6`, `gl:convolutionFilter1D/6`, `gl:convolutionFilter2D/7`, `gl:histogram/4`, `gl:minmax/3`, and `gl:separableFilter2D/8`. Use of these maps is described completely in the `gl:pixelTransfer()` reference page, and partly in the reference pages for the pixel and texture image commands. Only the specification of the maps is described in this reference page.

**External documentation.**

```
pixelStoref(Pname :: enum(), Param :: f()) -> ok
pixelStorei(Pname :: enum(), Param :: i()) -> ok
```

`gl:pixelStore()` sets pixel storage modes that affect the operation of subsequent `gl:readPixels/7` as well as the unpacking of texture patterns (see `gl:texImage1D/8`, `gl:texImage2D/9`, `gl:texImage3D/10`, `gl:texSubImage1D/7`, `gl:texSubImage2D/9`, `gl:texSubImage3D/11`), `gl:compressedTexImage1D/7`, `gl:compressedTexImage2D/8`, `gl:compressedTexImage3D/9`, `gl:compressedTexSubImage1D/7`, `gl:compressedTexSubImage2D/9` or `gl:compressedTexSubImage1D/7`.

**External documentation.**

```
pixelTransferf(Pname :: enum(), Param :: f()) -> ok
pixelTransferi(Pname :: enum(), Param :: i()) -> ok
```

`gl:pixelTransfer()` sets pixel transfer modes that affect the operation of subsequent `gl:copyPixels/5`, `gl:copyTexImage1D/7`, `gl:copyTexImage2D/8`, `gl:copyTexSubImage1D/6`, `gl:copyTexSubImage2D/8`, `gl:copyTexSubImage3D/9`, `gl:drawPixels/5`, `gl:readPixels/7`, `gl:texImage1D/8`, `gl:texImage2D/9`, `gl:texImage3D/10`, `gl:texSubImage1D/7`, `gl:texSubImage2D/9`, and `gl:texSubImage3D/11` commands. Additionally, if the ARB_imaging subset is supported, the routines `gl:colorTable/6`, `gl:colorSubTable/6`, `gl:convolutionFilter1D/6`, `gl:convolutionFilter2D/7`, `gl:histogram/4`, `gl:minmax/3`, and `gl:separableFilter2D/8` are also affected. The algorithms that are specified by pixel transfer modes operate on pixels after they are read from the frame buffer (`gl:copyPixels/5gl:copyTexImage1D/7`, `gl:copyTexImage2D/8`, `gl:copyTexSubImage1D/6`, `gl:copyTexSubImage2D/8`, `gl:copyTexSubImage3D/9`, and `gl:readPixels/7`), or unpacked from client memory (`gl:drawPixels/5`, `gl:texImage1D/8`, `gl:texImage2D/9`, `gl:texImage3D/10`, `gl:texSubImage1D/7`, `gl:texSubImage2D/9`, and `gl:texSubImage3D/11`). Pixel transfer operations happen in the same order, and in the same manner, regardless of the command that resulted in the pixel operation. Pixel storage modes (see `gl:pixelStore()`) control the unpacking of pixels being read from client memory and the packing of pixels being written back into client memory.

**External documentation.**

```
pixelZoom(Xfactor :: f(), Yfactor :: f()) -> ok
```

`gl:pixelZoom/2` specifies values for the x and y zoom factors. During the execution of `gl:drawPixels/5` or `gl:copyPixels/5`, if ( xr, yr) is the current raster position, and a given element is in the mth row and nth column of the pixel rectangle, then pixels whose centers are in the rectangle with corners at

**External documentation.**

```
pointParameterf(Pname :: enum(), Param :: f()) -> ok
pointParameterfv(Pname :: enum(), Params :: tuple()) -> ok
pointParameteri(Pname :: enum(), Param :: i()) -> ok
pointParameteriv(Pname :: enum(), Params :: tuple()) -> ok
```

The following values are accepted for `Pname`:

**External documentation.**

```
pointSize(Size :: f()) -> ok
```

`gl:pointSize/1` specifies the rasterized diameter of points. If point size mode is disabled (see `gl:enable/1` with parameter `?GL_PROGRAM_POINT_SIZE`), this value will be used to rasterize points. Otherwise, the value written to the shading language built-in variable gl_PointSize will be used.

**External documentation.**

```
polygonMode(Face :: enum(), Mode :: enum()) -> ok
```

`gl:polygonMode/2` controls the interpretation of polygons for rasterization. `Face` describes which polygons `Mode` applies to: both front and back-facing polygons (`?GL_FRONT_AND_BACK`). The polygon mode affects only the final rasterization of polygons. In particular, a polygon's vertices are lit and the polygon is clipped and possibly culled before these modes are applied.

**External documentation.**

```
polygonOffset(Factor :: f(), Units :: f()) -> ok
```

When `?GL_POLYGON_OFFSET_FILL`, `?GL_POLYGON_OFFSET_LINE`, or `?GL_POLYGON_OFFSET_POINT` is enabled, each fragment's `depth` value will be offset after it is interpolated from the `depth` values of the appropriate vertices. The value of the offset is factor×DZ+r×units, where DZ is a measurement of the change in depth relative to the screen area of the polygon, and r is the smallest value that is guaranteed to produce a resolvable offset for a given implementation. The offset is added before the depth test is performed and before the value is written into the depth buffer.

**External documentation.**

```
polygonOffsetClamp(Factor :: f(), Units :: f(), Clamp :: f()) ->
                      ok
```

No documentation available.

```
polygonStipple(Mask :: binary()) -> ok
```

Polygon stippling, like line stippling (see `gl:lineStipple/2`), masks out certain fragments produced by rasterization, creating a pattern. Stippling is independent of polygon antialiasing.

**External documentation.**

```
primitiveRestartIndex(Index :: i()) -> ok
```

`gl:primitiveRestartIndex/1` specifies a vertex array element that is treated specially when primitive restarting is enabled. This is known as the primitive restart index.

**External documentation.**

```
prioritizeTextures(Textures :: [i()], Priorities :: [clamp()]) ->
                      ok
```

`gl:prioritizeTextures/2` assigns the `N` texture priorities given in `Priorities` to the `N` textures named in `Textures`.

**External documentation.**

```
programBinary(Program :: i(),
              BinaryFormat :: enum(),
              Binary :: binary()) ->
                 ok
```

`gl:programBinary/3` loads a program object with a program binary previously returned from `gl:getProgramBinary/2`. `BinaryFormat` and `Binary` must be those returned by a previous call to `gl:getProgramBinary/2`, and `Length` must be the length returned by `gl:getProgramBinary/2`, or by

gl:getProgram() when called with Pname set to ?GL_PROGRAM_BINARY_LENGTH. If these conditions are not met, loading the program binary will fail and Program's ?GL_LINK_STATUS will be set to ?GL_FALSE.

**External documentation.**

```
programParameteri(Program :: i(), Pname :: enum(), Value :: i()) ->
                     ok
```

gl:programParameter() specifies a new value for the parameter nameed by Pname for the program object Program.

**External documentation.**

```
programUniform1d(Program :: i(), Location :: i(), V0 :: f()) -> ok
programUniform1dv(Program :: i(), Location :: i(), Value :: [f()]) ->
                     ok
programUniform1f(Program :: i(), Location :: i(), V0 :: f()) -> ok
programUniform1fv(Program :: i(), Location :: i(), Value :: [f()]) ->
                     ok
programUniform1i(Program :: i(), Location :: i(), V0 :: i()) -> ok
programUniform1iv(Program :: i(), Location :: i(), Value :: [i()]) ->
                     ok
programUniform1ui(Program :: i(), Location :: i(), V0 :: i()) ->
                     ok
programUniform1uiv(Program :: i(),
                   Location :: i(),
                   Value :: [i()]) ->
                      ok
programUniform2d(Program :: i(),
                 Location :: i(),
                 V0 :: f(),
                 V1 :: f()) ->
                    ok
programUniform2dv(Program :: i(),
                  Location :: i(),
                  Value :: [{f(), f()}]) ->
                     ok
programUniform2f(Program :: i(),
                 Location :: i(),
                 V0 :: f(),
                 V1 :: f()) ->
                    ok
programUniform2fv(Program :: i(),
                  Location :: i(),
                  Value :: [{f(), f()}]) ->
                     ok
programUniform2i(Program :: i(),
                 Location :: i(),
                 V0 :: i(),
                 V1 :: i()) ->
```

```
                    ok
programUniform2iv(Program :: i(),
                  Location :: i(),
                  Value :: [{i(), i()}]) ->
                     ok
programUniform2ui(Program :: i(),
                  Location :: i(),
                  V0 :: i(),
                  V1 :: i()) ->
                     ok
programUniform2uiv(Program :: i(),
                   Location :: i(),
                   Value :: [{i(), i()}]) ->
                      ok
programUniform3d(Program :: i(),
                 Location :: i(),
                 V0 :: f(),
                 V1 :: f(),
                 V2 :: f()) ->
                    ok
programUniform3dv(Program :: i(),
                  Location :: i(),
                  Value :: [{f(), f(), f()}]) ->
                     ok
programUniform3f(Program :: i(),
                 Location :: i(),
                 V0 :: f(),
                 V1 :: f(),
                 V2 :: f()) ->
                    ok
programUniform3fv(Program :: i(),
                  Location :: i(),
                  Value :: [{f(), f(), f()}]) ->
                     ok
programUniform3i(Program :: i(),
                 Location :: i(),
                 V0 :: i(),
                 V1 :: i(),
                 V2 :: i()) ->
                    ok
programUniform3iv(Program :: i(),
                  Location :: i(),
                  Value :: [{i(), i(), i()}]) ->
                     ok
programUniform3ui(Program :: i(),
                  Location :: i(),
                  V0 :: i(),
                  V1 :: i(),
                  V2 :: i()) ->
```

```
                     ok
programUniform3uiv(Program :: i(),
                   Location :: i(),
                   Value :: [{i(), i(), i()}]) ->
                      ok
programUniform4d(Program :: i(),
                 Location :: i(),
                 V0 :: f(),
                 V1 :: f(),
                 V2 :: f(),
                 V3 :: f()) ->
                    ok
programUniform4dv(Program :: i(),
                  Location :: i(),
                  Value :: [{f(), f(), f(), f()}]) ->
                     ok
programUniform4f(Program :: i(),
                 Location :: i(),
                 V0 :: f(),
                 V1 :: f(),
                 V2 :: f(),
                 V3 :: f()) ->
                    ok
programUniform4fv(Program :: i(),
                  Location :: i(),
                  Value :: [{f(), f(), f(), f()}]) ->
                     ok
programUniform4i(Program :: i(),
                 Location :: i(),
                 V0 :: i(),
                 V1 :: i(),
                 V2 :: i(),
                 V3 :: i()) ->
                    ok
programUniform4iv(Program :: i(),
                  Location :: i(),
                  Value :: [{i(), i(), i(), i()}]) ->
                     ok
programUniform4ui(Program :: i(),
                  Location :: i(),
                  V0 :: i(),
                  V1 :: i(),
                  V2 :: i(),
                  V3 :: i()) ->
                     ok
programUniform4uiv(Program :: i(),
                   Location :: i(),
                   Value :: [{i(), i(), i(), i()}]) ->
                      ok
programUniformMatrix2dv(Program :: i(),
```

```
                        Location :: i(),
                        Transpose :: 0 | 1,
                        Value :: [{f(), f(), f(), f()}]) ->
                           ok
programUniformMatrix2fv(Program :: i(),
                        Location :: i(),
                        Transpose :: 0 | 1,
                        Value :: [{f(), f(), f(), f()}]) ->
                           ok
programUniformMatrix2x3dv(Program :: i(),
                          Location :: i(),
                          Transpose :: 0 | 1,
                          Value ::
                              [{f(), f(), f(), f(), f(), f()}]) ->
                             ok
programUniformMatrix2x3fv(Program :: i(),
                          Location :: i(),
                          Transpose :: 0 | 1,
                          Value ::
                              [{f(), f(), f(), f(), f(), f()}]) ->
                             ok
programUniformMatrix2x4dv(Program, Location, Transpose, Value) ->
                             ok
programUniformMatrix2x4fv(Program, Location, Transpose, Value) ->
                             ok
programUniformMatrix3dv(Program, Location, Transpose, Value) -> ok
programUniformMatrix3fv(Program, Location, Transpose, Value) -> ok
programUniformMatrix3x2dv(Program :: i(),
                          Location :: i(),
                          Transpose :: 0 | 1,
                          Value ::
                              [{f(), f(), f(), f(), f(), f()}]) ->
                             ok
programUniformMatrix3x2fv(Program :: i(),
                          Location :: i(),
                          Transpose :: 0 | 1,
                          Value ::
                              [{f(), f(), f(), f(), f(), f()}]) ->
                             ok
programUniformMatrix3x4dv(Program, Location, Transpose, Value) ->
                             ok
programUniformMatrix3x4fv(Program, Location, Transpose, Value) ->
                             ok
programUniformMatrix4dv(Program, Location, Transpose, Value) -> ok
programUniformMatrix4fv(Program, Location, Transpose, Value) -> ok
programUniformMatrix4x2dv(Program, Location, Transpose, Value) ->
                             ok
programUniformMatrix4x2fv(Program, Location, Transpose, Value) ->
```

```
                              ok
programUniformMatrix4x3dv(Program, Location, Transpose, Value) ->
                              ok
programUniformMatrix4x3fv(Program, Location, Transpose, Value) ->
                              ok
```

Types:

```
Program = Location = i()
Transpose = 0 | 1
Value =
    [{f(), f(), f(), f(), f(), f(), f(), f(), f(), f(), f(), f()}]
```

`gl:programUniform()` modifies the value of a uniform variable or a uniform variable array. The location of the uniform variable to be modified is specified by `Location`, which should be a value returned by `gl:getUniformLocation/2`. `gl:programUniform()` operates on the program object specified by `Program`.

**External documentation.**

```
provokingVertex(Mode :: enum()) -> ok
```

`Flatshading` a vertex shader varying output means to assign all vetices of the primitive the same value for that output. The vertex from which these values is derived is known as the `provoking vertex` and `gl:provokingVertex/1` specifies which vertex is to be used as the source of data for flat shaded varyings.

**External documentation.**

```
popAttrib() -> ok
pushAttrib(Mask :: i()) -> ok
```

`gl:pushAttrib/1` takes one argument, a mask that indicates which groups of state variables to save on the attribute stack. Symbolic constants are used to set bits in the mask. `Mask` is typically constructed by specifying the bitwise-or of several of these constants together. The special mask `?GL_ALL_ATTRIB_BITS` can be used to save all stackable states.

**External documentation.**

```
popClientAttrib() -> ok
pushClientAttrib(Mask :: i()) -> ok
```

`gl:pushClientAttrib/1` takes one argument, a mask that indicates which groups of client-state variables to save on the client attribute stack. Symbolic constants are used to set bits in the mask. `Mask` is typically constructed by specifying the bitwise-or of several of these constants together. The special mask `?GL_CLIENT_ALL_ATTRIB_BITS` can be used to save all stackable client state.

**External documentation.**

```
popDebugGroup() -> ok
pushDebugGroup(Source :: enum(),
               Id :: i(),
               Length :: i(),
               Message :: string()) ->
```

```
ok
```

`gl:pushDebugGroup/4` pushes a debug group described by the string `Message` into the command stream. The value of `Id` specifies the ID of messages generated. The parameter `Length` contains the number of characters in `Message`. If `Length` is negative, it is implied that `Message` contains a null terminated string. The message has the specified `Source` and `Id`, the `Type?GL_DEBUG_TYPE_PUSH_GROUP`, and `Severity?GL_DEBUG_SEVERITY_NOTIFICATION`. The GL will put a new debug group on top of the debug group stack which inherits the control of the volume of debug output of the debug group previously residing on the top of the debug group stack. Because debug groups are strictly hierarchical, any additional control of the debug output volume will only apply within the active debug group and the debug groups pushed on top of the active debug group.

**External documentation.**

```
popMatrix() -> ok
pushMatrix() -> ok
```

There is a stack of matrices for each of the matrix modes. In `?GL_MODELVIEW` mode, the stack depth is at least 32. In the other modes, `?GL_COLOR`, `?GL_PROJECTION`, and `?GL_TEXTURE`, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode.

**External documentation.**

```
popName() -> ok
pushName(Name :: i()) -> ok
```

The name stack is used during selection mode to allow sets of rendering commands to be uniquely identified. It consists of an ordered set of unsigned integers and is initially empty.

**External documentation.**

```
queryCounter(Id :: i(), Target :: enum()) -> ok
```

`gl:queryCounter/2` causes the GL to record the current time into the query object named `Id`. `Target` must be `?GL_TIMESTAMP`. The time is recorded after all previous commands on the GL client and server state and the framebuffer have been fully realized. When the time is recorded, the query result for that object is marked available. `gl:queryCounter/2` timer queries can be used within a `gl:beginQuery/2` / `gl:endQuery/1` block where the target is `?GL_TIME_ELAPSED` and it does not affect the result of that query object.

**External documentation.**

```
rasterPos2d(X :: f(), Y :: f()) -> ok
rasterPos2dv(X1 :: {X :: f(), Y :: f()}) -> ok
rasterPos2f(X :: f(), Y :: f()) -> ok
rasterPos2fv(X1 :: {X :: f(), Y :: f()}) -> ok
rasterPos2i(X :: i(), Y :: i()) -> ok
rasterPos2iv(X1 :: {X :: i(), Y :: i()}) -> ok
rasterPos2s(X :: i(), Y :: i()) -> ok
rasterPos2sv(X1 :: {X :: i(), Y :: i()}) -> ok
rasterPos3d(X :: f(), Y :: f(), Z :: f()) -> ok
rasterPos3dv(X1 :: {X :: f(), Y :: f(), Z :: f()}) -> ok
rasterPos3f(X :: f(), Y :: f(), Z :: f()) -> ok
rasterPos3fv(X1 :: {X :: f(), Y :: f(), Z :: f()}) -> ok
rasterPos3i(X :: i(), Y :: i(), Z :: i()) -> ok
rasterPos3iv(X1 :: {X :: i(), Y :: i(), Z :: i()}) -> ok
rasterPos3s(X :: i(), Y :: i(), Z :: i()) -> ok
rasterPos3sv(X1 :: {X :: i(), Y :: i(), Z :: i()}) -> ok
rasterPos4d(X :: f(), Y :: f(), Z :: f(), W :: f()) -> ok
rasterPos4dv(X1 :: {X :: f(), Y :: f(), Z :: f(), W :: f()}) -> ok
rasterPos4f(X :: f(), Y :: f(), Z :: f(), W :: f()) -> ok
rasterPos4fv(X1 :: {X :: f(), Y :: f(), Z :: f(), W :: f()}) -> ok
rasterPos4i(X :: i(), Y :: i(), Z :: i(), W :: i()) -> ok
rasterPos4iv(X1 :: {X :: i(), Y :: i(), Z :: i(), W :: i()}) -> ok
rasterPos4s(X :: i(), Y :: i(), Z :: i(), W :: i()) -> ok
rasterPos4sv(X1 :: {X :: i(), Y :: i(), Z :: i(), W :: i()}) -> ok
```

The GL maintains a 3D position in window coordinates. This position, called the raster position, is used to position pixel and bitmap write operations. It is maintained with subpixel accuracy. See `gl:bitmap/7`, `gl:drawPixels/5`, and `gl:copyPixels/5`.

**External documentation.**

```
readBuffer(Mode :: enum()) -> ok
```

`gl:readBuffer/1` specifies a color buffer as the source for subsequent `gl:readPixels/7`, `gl:copyTexImage1D/7`, `gl:copyTexImage2D/8`, `gl:copyTexSubImage1D/6`, `gl:copyTexSubImage2D/8`, and `gl:copyTexSubImage3D/9` commands. `Mode` accepts one of twelve or more predefined values. In a fully configured system, `?GL_FRONT`, `?GL_LEFT`, and `?GL_FRONT_LEFT` all name the front left buffer, `?GL_FRONT_RIGHT` and `?GL_RIGHT` name the front right buffer, and `?GL_BACK_LEFT` and `?GL_BACK` name the back left buffer. Further more, the constants `?GL_COLOR_ATTACHMENTi` may be used to indicate the `i`th color attachment where `i` ranges from zero to the value of `?GL_MAX_COLOR_ATTACHMENTS` minus one.

**External documentation.**

```
readPixels(X, Y, Width, Height, Format, Type, Pixels) -> ok
```

Types:

```
X = Y = Width = Height = i()
Format = Type = enum()
Pixels = mem()
```

`gl:readPixels/7` and `glReadnPixels` return pixel data from the frame buffer, starting with the pixel whose lower left corner is at location (`X`, `Y`), into client memory starting at location `Data`. Several parameters control the processing of the pixel data before it is placed into client memory. These parameters are set with `gl:pixelStore()`. This reference page describes the effects on `gl:readPixels/7` and `glReadnPixels` of most, but not all of the parameters specified by these three commands.

**External documentation.**

```
rectd(X1 :: f(), Y1 :: f(), X2 :: f(), Y2 :: f()) -> ok
rectdv(V1 :: {f(), f()}, V2 :: {f(), f()}) -> ok
rectf(X1 :: f(), Y1 :: f(), X2 :: f(), Y2 :: f()) -> ok
rectfv(V1 :: {f(), f()}, V2 :: {f(), f()}) -> ok
recti(X1 :: i(), Y1 :: i(), X2 :: i(), Y2 :: i()) -> ok
rectiv(V1 :: {i(), i()}, V2 :: {i(), i()}) -> ok
rects(X1 :: i(), Y1 :: i(), X2 :: i(), Y2 :: i()) -> ok
rectsv(V1 :: {i(), i()}, V2 :: {i(), i()}) -> ok
```

`gl:rect()` supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of (x y) coordinates or as two pointers to arrays, each containing an (x y) pair. The resulting rectangle is defined in the z=0 plane.

**External documentation.**

```
releaseShaderCompiler() -> ok
```

`gl:releaseShaderCompiler/0` provides a hint to the implementation that it may free internal resources associated with its shader compiler. `gl:compileShader/1` may subsequently be called and the implementation may at that time reallocate resources previously freed by the call to `gl:releaseShaderCompiler/0`.

**External documentation.**

```
renderMode(Mode :: enum()) -> i()
```

`gl:renderMode/1` sets the rasterization mode. It takes one argument, `Mode`, which can assume one of three predefined values:

**External documentation.**

```
renderbufferStorage(Target :: enum(),
                    Internalformat :: enum(),
                    Width :: i(),
                    Height :: i()) ->
                       ok
```

`gl:renderbufferStorage/4` is equivalent to calling `gl:renderbufferStorageMultisample/5` with the `Samples` set to zero, and `glNamedRenderbufferStorage` is equivalent to calling `glNamedRenderbufferStorageMultisample` with the samples set to zero.

**External documentation.**

```
renderbufferStorageMultisample(Target :: enum(),
                               Samples :: i(),
                               Internalformat :: enum(),
                               Width :: i(),
                               Height :: i()) ->
                                  ok
```

`gl:renderbufferStorageMultisample/5` and `glNamedRenderbufferStorageMultisample` establish the data storage, format, dimensions and number of samples of a renderbuffer object's image.

**External documentation.**

```
resetHistogram(Target :: enum()) -> ok
```

`gl:resetHistogram/1` resets all the elements of the current histogram table to zero.

**External documentation.**

```
resetMinmax(Target :: enum()) -> ok
```

`gl:resetMinmax/1` resets the elements of the current minmax table to their initial values: the ``maximum'' element receives the minimum possible component values, and the ``minimum'' element receives the maximum possible component values.

**External documentation.**

```
resumeTransformFeedback() -> ok
```

`gl:resumeTransformFeedback/0` resumes transform feedback operations on the currently active transform feedback object. When transform feedback operations are paused, transform feedback is still considered active and changing most transform feedback state related to the object results in an error. However, a new transform feedback object may be bound while transform feedback is paused.

**External documentation.**

```
rotated(Angle :: f(), X :: f(), Y :: f(), Z :: f()) -> ok
rotatef(Angle :: f(), X :: f(), Y :: f(), Z :: f()) -> ok
```

`gl:rotate()` produces a rotation of `Angle` degrees around the vector (x y z). The current matrix (see `gl:matrixMode/1`) is multiplied by a rotation matrix with the product replacing the current matrix, as if `gl:multMatrix()` were called with the following matrix as its argument:

**External documentation.**

```
sampleCoverage(Value :: clamp(), Invert :: 0 | 1) -> ok
```

Multisampling samples a pixel multiple times at various implementation-dependent subpixel locations to generate antialiasing effects. Multisampling transparently antialiases points, lines, polygons, and images if it is enabled.

**External documentation.**

```
sampleMaski(MaskNumber :: i(), Mask :: i()) -> ok
```

`gl:sampleMaski/2` sets one 32-bit sub-word of the multi-word sample mask, `?GL_SAMPLE_MASK_VALUE`.

**External documentation.**

```
samplerParameterIiv(Sampler :: i(),
```

```
                    Pname :: enum(),
                    Param :: [i()]) ->
                       ok
samplerParameterIuiv(Sampler :: i(),
                     Pname :: enum(),
                     Param :: [i()]) ->
                        ok
samplerParameterf(Sampler :: i(), Pname :: enum(), Param :: f()) ->
                     ok
samplerParameterfv(Sampler :: i(),
                   Pname :: enum(),
                   Param :: [f()]) ->
                      ok
samplerParameteri(Sampler :: i(), Pname :: enum(), Param :: i()) ->
                     ok
samplerParameteriv(Sampler :: i(),
                   Pname :: enum(),
                   Param :: [i()]) ->
                      ok
```

`gl:samplerParameter()` assigns the value or values in `Params` to the sampler parameter specified as `Pname`. `Sampler` specifies the sampler object to be modified, and must be the name of a sampler object previously returned from a call to `gl:genSamplers/1`. The following symbols are accepted in `Pname`:

**External documentation.**

```
scaled(X :: f(), Y :: f(), Z :: f()) -> ok
scalef(X :: f(), Y :: f(), Z :: f()) -> ok
```

`gl:scale()` produces a nonuniform scaling along the `x`, `y`, and `z` axes. The three parameters indicate the desired scale factor along each of the three axes.

**External documentation.**

```
scissor(X :: i(), Y :: i(), Width :: i(), Height :: i()) -> ok
```

`gl:scissor/4` defines a rectangle, called the scissor box, in window coordinates. The first two arguments, `X` and `Y`, specify the lower left corner of the box. `Width` and `Height` specify the width and height of the box.

**External documentation.**

```
scissorArrayv(First :: i(), V :: [{i(), i(), i(), i()}]) -> ok
```

`gl:scissorArrayv/2` defines rectangles, called scissor boxes, in window coordinates for each viewport. `First` specifies the index of the first scissor box to modify and `Count` specifies the number of scissor boxes to modify. `First` must be less than the value of `?GL_MAX_VIEWPORTS`, and `First` + `Count` must be less than or equal to the value of `?GL_MAX_VIEWPORTS`. `V` specifies the address of an array containing integers specifying the lower left corner of the scissor boxes, and the width and height of the scissor boxes, in that order.

**External documentation.**

```
scissorIndexed(Index :: i(),
               Left :: i(),
               Bottom :: i(),
```

```
               Width :: i(),
               Height :: i()) ->
                  ok
scissorIndexedv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
```

`gl:scissorIndexed/5` defines the scissor box for a specified viewport. `Index` specifies the index of scissor box to modify. `Index` must be less than the value of `?GL_MAX_VIEWPORTS`. For `gl:scissorIndexed/5`, `Left`, `Bottom`, `Width` and `Height` specify the left, bottom, width and height of the scissor box, in pixels, respectively. For `gl:scissorIndexedv/2`, `V` specifies the address of an array containing integers specifying the lower left corner of the scissor box, and the width and height of the scissor box, in that order.

**External documentation.**

```
secondaryColor3b(Red :: i(), Green :: i(), Blue :: i()) -> ok
secondaryColor3bv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) ->
                     ok
secondaryColor3d(Red :: f(), Green :: f(), Blue :: f()) -> ok
secondaryColor3dv(X1 :: {Red :: f(), Green :: f(), Blue :: f()}) ->
                     ok
secondaryColor3f(Red :: f(), Green :: f(), Blue :: f()) -> ok
secondaryColor3fv(X1 :: {Red :: f(), Green :: f(), Blue :: f()}) ->
                     ok
secondaryColor3i(Red :: i(), Green :: i(), Blue :: i()) -> ok
secondaryColor3iv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) ->
                     ok
secondaryColor3s(Red :: i(), Green :: i(), Blue :: i()) -> ok
secondaryColor3sv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) ->
                     ok
secondaryColor3ub(Red :: i(), Green :: i(), Blue :: i()) -> ok
secondaryColor3ubv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) ->
                      ok
secondaryColor3ui(Red :: i(), Green :: i(), Blue :: i()) -> ok
secondaryColor3uiv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) ->
                      ok
secondaryColor3us(Red :: i(), Green :: i(), Blue :: i()) -> ok
secondaryColor3usv(X1 :: {Red :: i(), Green :: i(), Blue :: i()}) ->
                      ok
```

The GL stores both a primary four-valued RGBA color and a secondary four-valued RGBA color (where alpha is always set to 0.0) that is associated with every vertex.

**External documentation.**

```
secondaryColorPointer(Size :: i(),
                      Type :: enum(),
                      Stride :: i(),
                      Pointer :: offset() | mem()) ->
                         ok
```

`gl:secondaryColorPointer/4` specifies the location and data format of an array of color components to use when rendering. `Size` specifies the number of components per color, and must be 3. `Type` specifies the data type

of each color component, and `Stride` specifies the byte stride from one color to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays.

**External documentation.**

```
selectBuffer(Size :: i(), Buffer :: mem()) -> ok
```

`gl:selectBuffer/2` has two arguments: `Buffer` is a pointer to an array of unsigned integers, and `Size` indicates the size of the array. `Buffer` returns values from the name stack (see `gl:initNames/0`, `gl:loadName/1`, `gl:pushName/1`) when the rendering mode is `?GL_SELECT` (see `gl:renderMode/1`). `gl:selectBuffer/2` must be issued before selection mode is enabled, and it must not be issued while the rendering mode is `?GL_SELECT`.

**External documentation.**

```
separableFilter2D(Target, Internalformat, Width, Height, Format,
                  Type, Row, Column) ->
                     ok
```

Types:

```
   Target = Internalformat = enum()
   Width = Height = i()
   Format = Type = enum()
   Row = Column = offset() | mem()
```

`gl:separableFilter2D/8` builds a two-dimensional separable convolution filter kernel from two arrays of pixels.

**External documentation.**

```
shadeModel(Mode :: enum()) -> ok
```

GL primitives can have either flat or smooth shading. Smooth shading, the default, causes the computed colors of vertices to be interpolated as the primitive is rasterized, typically assigning different colors to each resulting pixel fragment. Flat shading selects the computed color of just one vertex and assigns it to all the pixel fragments generated by rasterizing a single primitive. In either case, the computed color of a vertex is the result of lighting if lighting is enabled, or it is the current color at the time the vertex was specified if lighting is disabled.

**External documentation.**

```
shaderBinary(Shaders :: [i()],
             Binaryformat :: enum(),
             Binary :: binary()) ->
                ok
```

`gl:shaderBinary/3` loads pre-compiled shader binary code into the `Count` shader objects whose handles are given in `Shaders`. `Binary` points to `Length` bytes of binary shader code stored in client memory. `BinaryFormat` specifies the format of the pre-compiled code.

**External documentation.**

```
shaderSource(Shader :: i(), String :: [unicode:chardata()]) -> ok
```

`gl:shaderSource/2` sets the source code in `Shader` to the source code in the array of strings specified by `String`. Any source code previously stored in the shader object is completely replaced. The number of strings in the array is specified by `Count`. If `Length` is `?NULL`, each string is assumed to be null terminated. If `Length` is

a value other than ?NULL, it points to an array containing a string length for each of the corresponding elements of String. Each element in the Length array may contain the length of the corresponding string (the null character is not counted as part of the string length) or a value less than 0 to indicate that the string is null terminated. The source code strings are not scanned or parsed at this time; they are simply copied into the specified shader object.

**External documentation.**

```
shaderStorageBlockBinding(Program :: i(),
                          StorageBlockIndex :: i(),
                          StorageBlockBinding :: i()) ->
                             ok
```

gl:shaderStorageBlockBinding/3, changes the active shader storage block with an assigned index of StorageBlockIndex in program object Program. StorageBlockIndex must be an active shader storage block index in Program. StorageBlockBinding must be less than the value of ?GL_MAX_SHADER_STORAGE_BUFFER_BINDINGS. If successful, gl:shaderStorageBlockBinding/3 specifies that Program will use the data store of the buffer object bound to the binding point StorageBlockBinding to read and write the values of the buffer variables in the shader storage block identified by StorageBlockIndex.

**External documentation.**

```
stencilFunc(Func :: enum(), Ref :: i(), Mask :: i()) -> ok
```

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. Stencil planes are first drawn into using GL drawing primitives, then geometry and images are rendered using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

**External documentation.**

```
stencilFuncSeparate(Face :: enum(),
                    Func :: enum(),
                    Ref :: i(),
                    Mask :: i()) ->
                       ok
```

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

**External documentation.**

```
stencilMask(Mask :: i()) -> ok
```

gl:stencilMask/1 controls the writing of individual bits in the stencil planes. The least significant n bits of Mask, where n is the number of bits in the stencil buffer, specify a mask. Where a 1 appears in the mask, it's possible to write to the corresponding bit in the stencil buffer. Where a 0 appears, the corresponding bit is write-protected. Initially, all bits are enabled for writing.

**External documentation.**

```
stencilMaskSeparate(Face :: enum(), Mask :: i()) -> ok
```

gl:stencilMaskSeparate/2 controls the writing of individual bits in the stencil planes. The least significant n bits of Mask, where n is the number of bits in the stencil buffer, specify a mask. Where a 1 appears in the mask,

it's possible to write to the corresponding bit in the stencil buffer. Where a 0 appears, the corresponding bit is write-protected. Initially, all bits are enabled for writing.

**External documentation.**

```
stencilOp(Fail :: enum(), Zfail :: enum(), Zpass :: enum()) -> ok
```

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

**External documentation.**

```
stencilOpSeparate(Face :: enum(),
                  Sfail :: enum(),
                  Dpfail :: enum(),
                  Dppass :: enum()) ->
                     ok
```

Stenciling, like depth-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering.

**External documentation.**

```
texBuffer(Target :: enum(),
          Internalformat :: enum(),
          Buffer :: i()) ->
             ok
textureBuffer(Texture :: i(),
              Internalformat :: enum(),
              Buffer :: i()) ->
                 ok
```

`gl:texBuffer/3` and `gl:textureBuffer/3` attaches the data store of a specified buffer object to a specified texture object, and specify the storage format for the texture image found in the buffer object. The texture object must be a buffer texture.

**External documentation.**

```
texBufferRange(Target :: enum(),
               Internalformat :: enum(),
               Buffer :: i(),
               Offset :: i(),
               Size :: i()) ->
                  ok
textureBufferRange(Texture :: i(),
                   Internalformat :: enum(),
                   Buffer :: i(),
                   Offset :: i(),
                   Size :: i()) ->
```

```
ok
```

`gl:texBufferRange/5` and `gl:textureBufferRange/5` attach a range of the data store of a specified buffer object to a specified texture object, and specify the storage format for the texture image found in the buffer object. The texture object must be a buffer texture.

**External documentation.**

```
texCoord1d(S :: f()) -> ok
texCoord1dv(X1 :: {S :: f()}) -> ok
texCoord1f(S :: f()) -> ok
texCoord1fv(X1 :: {S :: f()}) -> ok
texCoord1i(S :: i()) -> ok
texCoord1iv(X1 :: {S :: i()}) -> ok
texCoord1s(S :: i()) -> ok
texCoord1sv(X1 :: {S :: i()}) -> ok
texCoord2d(S :: f(), T :: f()) -> ok
texCoord2dv(X1 :: {S :: f(), T :: f()}) -> ok
texCoord2f(S :: f(), T :: f()) -> ok
texCoord2fv(X1 :: {S :: f(), T :: f()}) -> ok
texCoord2i(S :: i(), T :: i()) -> ok
texCoord2iv(X1 :: {S :: i(), T :: i()}) -> ok
texCoord2s(S :: i(), T :: i()) -> ok
texCoord2sv(X1 :: {S :: i(), T :: i()}) -> ok
texCoord3d(S :: f(), T :: f(), R :: f()) -> ok
texCoord3dv(X1 :: {S :: f(), T :: f(), R :: f()}) -> ok
texCoord3f(S :: f(), T :: f(), R :: f()) -> ok
texCoord3fv(X1 :: {S :: f(), T :: f(), R :: f()}) -> ok
texCoord3i(S :: i(), T :: i(), R :: i()) -> ok
texCoord3iv(X1 :: {S :: i(), T :: i(), R :: i()}) -> ok
texCoord3s(S :: i(), T :: i(), R :: i()) -> ok
texCoord3sv(X1 :: {S :: i(), T :: i(), R :: i()}) -> ok
texCoord4d(S :: f(), T :: f(), R :: f(), Q :: f()) -> ok
texCoord4dv(X1 :: {S :: f(), T :: f(), R :: f(), Q :: f()}) -> ok
texCoord4f(S :: f(), T :: f(), R :: f(), Q :: f()) -> ok
texCoord4fv(X1 :: {S :: f(), T :: f(), R :: f(), Q :: f()}) -> ok
texCoord4i(S :: i(), T :: i(), R :: i(), Q :: i()) -> ok
texCoord4iv(X1 :: {S :: i(), T :: i(), R :: i(), Q :: i()}) -> ok
texCoord4s(S :: i(), T :: i(), R :: i(), Q :: i()) -> ok
texCoord4sv(X1 :: {S :: i(), T :: i(), R :: i(), Q :: i()}) -> ok
```

`gl:texCoord()` specifies texture coordinates in one, two, three, or four dimensions. `gl:texCoord1()` sets the current texture coordinates to (s 0 0 1); a call to `gl:texCoord2()` sets them to (s t 0 1). Similarly, `gl:texCoord3()` specifies the texture coordinates as (s t r 1), and `gl:texCoord4()` defines all four components explicitly as (s t r q).

**External documentation.**

```
texCoordPointer(Size :: i(),
                Type :: enum(),
                Stride :: i(),
                Ptr :: offset() | mem()) ->
                   ok
```

`gl:texCoordPointer/4` specifies the location and data format of an array of texture coordinates to use when rendering. `Size` specifies the number of coordinates per texture coordinate set, and must be 1, 2, 3, or 4. `Type` specifies the data type of each texture coordinate, and `Stride` specifies the byte stride from one texture coordinate set to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see `gl:interleavedArrays/3`.)

**External documentation.**

```
texEnvf(Target :: enum(), Pname :: enum(), Param :: f()) -> ok
texEnvfv(Target :: enum(), Pname :: enum(), Params :: tuple()) ->
            ok
texEnvi(Target :: enum(), Pname :: enum(), Param :: i()) -> ok
texEnviv(Target :: enum(), Pname :: enum(), Params :: tuple()) ->
            ok
```

A texture environment specifies how texture values are interpreted when a fragment is textured. When `Target` is `?GL_TEXTURE_FILTER_CONTROL`, `Pname` must be `?GL_TEXTURE_LOD_BIAS`. When `Target` is `?GL_TEXTURE_ENV`, `Pname` can be `?GL_TEXTURE_ENV_MODE`, `?GL_TEXTURE_ENV_COLOR`, `?GL_COMBINE_RGB`, `?GL_COMBINE_ALPHA`, `?GL_RGB_SCALE`, `?GL_ALPHA_SCALE`, `?GL_SRC0_RGB`, `?GL_SRC1_RGB`, `?GL_SRC2_RGB`, `?GL_SRC0_ALPHA`, `?GL_SRC1_ALPHA`, or `?GL_SRC2_ALPHA`.

**External documentation.**

```
texGend(Coord :: enum(), Pname :: enum(), Param :: f()) -> ok
texGendv(Coord :: enum(), Pname :: enum(), Params :: tuple()) ->
            ok
texGenf(Coord :: enum(), Pname :: enum(), Param :: f()) -> ok
texGenfv(Coord :: enum(), Pname :: enum(), Params :: tuple()) ->
            ok
texGeni(Coord :: enum(), Pname :: enum(), Param :: i()) -> ok
texGeniv(Coord :: enum(), Pname :: enum(), Params :: tuple()) ->
            ok
```

`gl:texGen()` selects a texture-coordinate generation function or supplies coefficients for one of the functions. `Coord` names one of the (`s`, `t`, `r`, `q`) texture coordinates; it must be one of the symbols `?GL_S`, `?GL_T`, `?GL_R`, or `?GL_Q`. `Pname` must be one of three symbolic constants: `?GL_TEXTURE_GEN_MODE`, `?GL_OBJECT_PLANE`, or `?GL_EYE_PLANE`. If `Pname` is `?GL_TEXTURE_GEN_MODE`, then `Params` chooses a mode, one of `?GL_OBJECT_LINEAR`, `?GL_EYE_LINEAR`, `?GL_SPHERE_MAP`, `?GL_NORMAL_MAP`, or `?GL_REFLECTION_MAP`. If `Pname` is either `?GL_OBJECT_PLANE` or `?GL_EYE_PLANE`, `Params` contains coefficients for the corresponding texture generation function.

**External documentation.**

```
texImage1D(Target, Level, InternalFormat, Width, Border, Format,
           Type, Pixels) ->
              ok
```

Types:

```
Target = enum()
Level = InternalFormat = Width = Border = i()
Format = Type = enum()
Pixels = offset() | mem()
```

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable one-dimensional texturing, call `gl:enable/1` and `gl:disable/1` with argument `?GL_TEXTURE_1D`.

**External documentation.**

```
texImage2D(Target, Level, InternalFormat, Width, Height, Border,
           Format, Type, Pixels) ->
              ok
```

Types:

```
Target = enum()
Level = InternalFormat = Width = Height = Border = i()
Format = Type = enum()
Pixels = offset() | mem()
```

Texturing allows elements of an image array to be read by shaders.

**External documentation.**

```
texImage2DMultisample(Target, Samples, Internalformat, Width,
                      Height, Fixedsamplelocations) ->
                         ok
```

Types:

```
Target = enum()
Samples = i()
Internalformat = enum()
Width = Height = i()
Fixedsamplelocations = 0 | 1
```

`gl:texImage2DMultisample/6` establishes the data storage, format, dimensions and number of samples of a multisample texture's image.

**External documentation.**

```
texImage3D(Target, Level, InternalFormat, Width, Height, Depth,
           Border, Format, Type, Pixels) ->
              ok
```

Types:

```
Target = enum()
Level = InternalFormat = Width = Height = Depth = Border = i()
Format = Type = enum()
Pixels = offset() | mem()
```

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable and disable three-dimensional texturing, call `gl:enable/1` and `gl:disable/1` with argument `?GL_TEXTURE_3D`.

**External documentation.**

```
texImage3DMultisample(Target, Samples, Internalformat, Width,
                      Height, Depth, Fixedsamplelocations) ->
                         ok
```

Types:

```
Target = enum()
Samples = i()
Internalformat = enum()
Width = Height = Depth = i()
Fixedsamplelocations = 0 | 1
```

`gl:texImage3DMultisample/7` establishes the data storage, format, dimensions and number of samples of a multisample texture's image.

**External documentation.**

```
texParameterIiv(Target :: enum(),
                Pname :: enum(),
                Params :: tuple()) ->
                   ok
texParameterIuiv(Target :: enum(),
                 Pname :: enum(),
                 Params :: tuple()) ->
                    ok
texParameterf(Target :: enum(), Pname :: enum(), Param :: f()) ->
                 ok
texParameterfv(Target :: enum(),
               Pname :: enum(),
               Params :: tuple()) ->
                  ok
texParameteri(Target :: enum(), Pname :: enum(), Param :: i()) ->
                 ok
texParameteriv(Target :: enum(),
               Pname :: enum(),
               Params :: tuple()) ->
                  ok
```

`gl:texParameter()` and `gl:textureParameter()` assign the value or values in `Params` to the texture parameter specified as `Pname`. For `gl:texParameter()`, `Target` defines the target texture, either `?GL_TEXTURE_1D`, `?GL_TEXTURE_1D_ARRAY`, `?GL_TEXTURE_2D`, `?GL_TEXTURE_2D_ARRAY`,

?GL_TEXTURE_2D_MULTISAMPLE, ?GL_TEXTURE_2D_MULTISAMPLE_ARRAY, ?GL_TEXTURE_3D, ?GL_TEXTURE_CUBE_MAP, ?GL_TEXTURE_CUBE_MAP_ARRAY, or ?GL_TEXTURE_RECTANGLE. The following symbols are accepted in Pname:

**External documentation.**

```
texStorage1D(Target :: enum(),
             Levels :: i(),
             Internalformat :: enum(),
             Width :: i()) ->
                ok
```

gl:texStorage1D/4 and gl:textureStorage1D() specify the storage requirements for all levels of a one-dimensional texture simultaneously. Once a texture is specified with this command, the format and dimensions of all levels become immutable unless it is a proxy texture. The contents of the image may still be modified, however, its storage requirements may not change. Such a texture is referred to as an immutable-format texture.

**External documentation.**

```
texStorage2D(Target :: enum(),
             Levels :: i(),
             Internalformat :: enum(),
             Width :: i(),
             Height :: i()) ->
                ok
```

gl:texStorage2D/5 and gl:textureStorage2D() specify the storage requirements for all levels of a two-dimensional texture or one-dimensional texture array simultaneously. Once a texture is specified with this command, the format and dimensions of all levels become immutable unless it is a proxy texture. The contents of the image may still be modified, however, its storage requirements may not change. Such a texture is referred to as an immutable-format texture.

**External documentation.**

```
texStorage2DMultisample(Target, Samples, Internalformat, Width,
                        Height, Fixedsamplelocations) ->
                           ok
```

Types:

```
Target = enum()
Samples = i()
Internalformat = enum()
Width = Height = i()
Fixedsamplelocations = 0 | 1
```

gl:texStorage2DMultisample/6 and gl:textureStorage2DMultisample() specify the storage requirements for a two-dimensional multisample texture. Once a texture is specified with this command, its format and dimensions become immutable unless it is a proxy texture. The contents of the image may still be modified, however, its storage requirements may not change. Such a texture is referred to as an immutable-format texture.

**External documentation.**

```
texStorage3D(Target, Levels, Internalformat, Width, Height, Depth) ->
                ok
```

Types:

```
   Target = enum()
   Levels = i()
   Internalformat = enum()
   Width = Height = Depth = i()
```

`gl:texStorage3D/6` and `gl:textureStorage3D()` specify the storage requirements for all levels of a three-dimensional, two-dimensional array or cube-map array texture simultaneously. Once a texture is specified with this command, the format and dimensions of all levels become immutable unless it is a proxy texture. The contents of the image may still be modified, however, its storage requirements may not change. Such a texture is referred to as an `immutable-format` texture.

**External documentation.**

```
texStorage3DMultisample(Target, Samples, Internalformat, Width,
                        Height, Depth, Fixedsamplelocations) ->
                           ok
```

Types:

```
   Target = enum()
   Samples = i()
   Internalformat = enum()
   Width = Height = Depth = i()
   Fixedsamplelocations = 0 | 1
```

`gl:texStorage3DMultisample/7` and `gl:textureStorage3DMultisample()` specify the storage requirements for a two-dimensional multisample array texture. Once a texture is specified with this command, its format and dimensions become immutable unless it is a proxy texture. The contents of the image may still be modified, however, its storage requirements may not change. Such a texture is referred to as an `immutable-format` texture.

**External documentation.**

```
texSubImage1D(Target, Level, Xoffset, Width, Format, Type, Pixels) ->
                 ok
```

Types:

```
   Target = enum()
   Level = Xoffset = Width = i()
   Format = Type = enum()
   Pixels = offset() | mem()
```

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled. To enable or disable one-dimensional texturing, call `gl:enable/1` and `gl:disable/1` with argument `?GL_TEXTURE_1D`.

**External documentation.**

```
texSubImage2D(Target, Level, Xoffset, Yoffset, Width, Height,
              Format, Type, Pixels) ->
                 ok
```

Types:

```
Target = enum()
Level = Xoffset = Yoffset = Width = Height = i()
Format = Type = enum()
Pixels = offset() | mem()
```

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled.

**External documentation.**

```
texSubImage3D(Target, Level, Xoffset, Yoffset, Zoffset, Width,
              Height, Depth, Format, Type, Pixels) ->
                 ok
```

Types:

```
Target = enum()
Level = Xoffset = Yoffset = Zoffset = Width = Height = Depth = i()
Format = Type = enum()
Pixels = offset() | mem()
```

Texturing maps a portion of a specified texture image onto each graphical primitive for which texturing is enabled.

**External documentation.**

```
textureBarrier() -> ok
```

The values of rendered fragments are undefined when a shader stage fetches texels and the same texels are written via fragment shader outputs, even if the reads and writes are not in the same drawing command. To safely read the result of a written texel via a texel fetch in a subsequent drawing command, call `gl:textureBarrier/0` between the two drawing commands to guarantee that writes have completed and caches have been invalidated before subsequent drawing commands are executed.

**External documentation.**

```
textureView(Texture, Target, Origtexture, Internalformat,
            Minlevel, Numlevels, Minlayer, Numlayers) ->
               ok
```

Types:

```
Texture = i()
Target = enum()
Origtexture = i()
Internalformat = enum()
Minlevel = Numlevels = Minlayer = Numlayers = i()
```

`gl:textureView/8` initializes a texture object as an alias, or view of another texture object, sharing some or all of the parent texture's data store with the initialized texture. `Texture` specifies a name previously reserved by a successful call to `gl:genTextures/1` but that has not yet been bound or given a target. `Target` specifies the target for the newly initialized texture and must be compatible with the target of the parent texture, given in `Origtexture` as specified in the following table:

**External documentation.**

```
transformFeedbackBufferBase(Xfb :: i(),
                            Index :: i(),
                            Buffer :: i()) ->
                               ok
```

`gl:transformFeedbackBufferBase/3` binds the buffer object `Buffer` to the binding point at index `Index` of the transform feedback object `Xfb`.

**External documentation.**

```
transformFeedbackBufferRange(Xfb :: i(),
                             Index :: i(),
                             Buffer :: i(),
                             Offset :: i(),
                             Size :: i()) ->
                                ok
```

`gl:transformFeedbackBufferRange/5` binds a range of the buffer object `Buffer` represented by `Offset` and `Size` to the binding point at index `Index` of the transform feedback object `Xfb`.

**External documentation.**

```
transformFeedbackVaryings(Program :: i(),
                          Varyings :: [unicode:chardata()],
                          BufferMode :: enum()) ->
                             ok
```

The names of the vertex or geometry shader outputs to be recorded in transform feedback mode are specified using `gl:transformFeedbackVaryings/3`. When a geometry shader is active, transform feedback records the values of selected geometry shader output variables from the emitted vertices. Otherwise, the values of the selected vertex shader outputs are recorded.

**External documentation.**

```
translated(X :: f(), Y :: f(), Z :: f()) -> ok
translatef(X :: f(), Y :: f(), Z :: f()) -> ok
```

`gl:translate()` produces a translation by (x y z). The current matrix (see `gl:matrixMode/1`) is multiplied by this translation matrix, with the product replacing the current matrix, as if `gl:multMatrix()` were called with the following matrix for its argument:

**External documentation.**

```
uniform1d(Location :: i(), X :: f()) -> ok
uniform1dv(Location :: i(), Value :: [f()]) -> ok
uniform1f(Location :: i(), V0 :: f()) -> ok
uniform1fv(Location :: i(), Value :: [f()]) -> ok
uniform1i(Location :: i(), V0 :: i()) -> ok
uniform1iv(Location :: i(), Value :: [i()]) -> ok
uniform1ui(Location :: i(), V0 :: i()) -> ok
uniform1uiv(Location :: i(), Value :: [i()]) -> ok
uniform2d(Location :: i(), X :: f(), Y :: f()) -> ok
uniform2dv(Location :: i(), Value :: [{f(), f()}]) -> ok
uniform2f(Location :: i(), V0 :: f(), V1 :: f()) -> ok
uniform2fv(Location :: i(), Value :: [{f(), f()}]) -> ok
uniform2i(Location :: i(), V0 :: i(), V1 :: i()) -> ok
uniform2iv(Location :: i(), Value :: [{i(), i()}]) -> ok
uniform2ui(Location :: i(), V0 :: i(), V1 :: i()) -> ok
uniform2uiv(Location :: i(), Value :: [{i(), i()}]) -> ok
uniform3d(Location :: i(), X :: f(), Y :: f(), Z :: f()) -> ok
uniform3dv(Location :: i(), Value :: [{f(), f(), f()}]) -> ok
uniform3f(Location :: i(), V0 :: f(), V1 :: f(), V2 :: f()) -> ok
uniform3fv(Location :: i(), Value :: [{f(), f(), f()}]) -> ok
uniform3i(Location :: i(), V0 :: i(), V1 :: i(), V2 :: i()) -> ok
uniform3iv(Location :: i(), Value :: [{i(), i(), i()}]) -> ok
uniform3ui(Location :: i(), V0 :: i(), V1 :: i(), V2 :: i()) -> ok
uniform3uiv(Location :: i(), Value :: [{i(), i(), i()}]) -> ok
uniform4d(Location :: i(), X :: f(), Y :: f(), Z :: f(), W :: f()) ->
             ok
uniform4dv(Location :: i(), Value :: [{f(), f(), f(), f()}]) -> ok
uniform4f(Location :: i(),
          V0 :: f(),
          V1 :: f(),
          V2 :: f(),
          V3 :: f()) ->
             ok
uniform4fv(Location :: i(), Value :: [{f(), f(), f(), f()}]) -> ok
uniform4i(Location :: i(),
          V0 :: i(),
          V1 :: i(),
          V2 :: i(),
          V3 :: i()) ->
             ok
uniform4iv(Location :: i(), Value :: [{i(), i(), i(), i()}]) -> ok
uniform4ui(Location :: i(),
           V0 :: i(),
           V1 :: i(),
           V2 :: i(),
           V3 :: i()) ->
```

```
              ok
uniform4uiv(Location :: i(), Value :: [{i(), i(), i(), i()}]) ->
               ok
uniformMatrix2dv(Location :: i(),
                 Transpose :: 0 | 1,
                 Value :: [{f(), f(), f(), f()}]) ->
                    ok
uniformMatrix2fv(Location :: i(),
                 Transpose :: 0 | 1,
                 Value :: [{f(), f(), f(), f()}]) ->
                    ok
uniformMatrix2x3dv(Location :: i(),
                   Transpose :: 0 | 1,
                   Value :: [{f(), f(), f(), f(), f(), f()}]) ->
                      ok
uniformMatrix2x3fv(Location :: i(),
                   Transpose :: 0 | 1,
                   Value :: [{f(), f(), f(), f(), f(), f()}]) ->
                      ok
uniformMatrix2x4dv(Location :: i(),
                   Transpose :: 0 | 1,
                   Value ::
                       [{f(), f(), f(), f(), f(), f(), f(), f()}]) ->
                      ok
uniformMatrix2x4fv(Location :: i(),
                   Transpose :: 0 | 1,
                   Value ::
                       [{f(), f(), f(), f(), f(), f(), f(), f()}]) ->
                      ok
uniformMatrix3dv(Location :: i(),
                 Transpose :: 0 | 1,
                 Value ::
                     [{f(),
                       f(),
                       f(),
                       f(),
                       f(),
                       f(),
                       f(),
                       f(),
                       f()}]) ->
                    ok
uniformMatrix3fv(Location :: i(),
                 Transpose :: 0 | 1,
                 Value ::
                     [{f(),
                       f(),
                       f(),
                       f(),
                       f(),
```

```
                      f(),
                      f(),
                      f(),
                      f()}]) ->
                   ok
uniformMatrix3x2dv(Location :: i(),
                   Transpose :: 0 | 1,
                   Value :: [{f(), f(), f(), f(), f(), f()}]) ->
                      ok
uniformMatrix3x2fv(Location :: i(),
                   Transpose :: 0 | 1,
                   Value :: [{f(), f(), f(), f(), f(), f()}]) ->
                      ok
uniformMatrix3x4dv(Location, Transpose, Value) -> ok
uniformMatrix3x4fv(Location, Transpose, Value) -> ok
uniformMatrix4dv(Location, Transpose, Value) -> ok
uniformMatrix4fv(Location, Transpose, Value) -> ok
uniformMatrix4x2dv(Location :: i(),
                   Transpose :: 0 | 1,
                   Value ::
                       [{f(), f(), f(), f(), f(), f(), f(), f()}]) ->
                      ok
uniformMatrix4x2fv(Location :: i(),
                   Transpose :: 0 | 1,
                   Value ::
                       [{f(), f(), f(), f(), f(), f(), f(), f()}]) ->
                      ok
uniformMatrix4x3dv(Location, Transpose, Value) -> ok
uniformMatrix4x3fv(Location, Transpose, Value) -> ok
```

Types:

```
Location = i()
Transpose = 0 | 1
Value =
    [{f(), f(), f(), f(), f(), f(), f(), f(), f(), f(), f(), f()}]
```

`gl:uniform()` modifies the value of a uniform variable or a uniform variable array. The location of the uniform variable to be modified is specified by `Location`, which should be a value returned by `gl:getUniformLocation/2`. `gl:uniform()` operates on the program object that was made part of current state by calling `gl:useProgram/1`.

**External documentation.**

```
uniformBlockBinding(Program :: i(),
                    UniformBlockIndex :: i(),
                    UniformBlockBinding :: i()) ->
                       ok
```

Binding points for active uniform blocks are assigned using `gl:uniformBlockBinding/3`. Each of a program's active uniform blocks has a corresponding uniform buffer binding point. `Program` is the name of a program object for which the command `gl:linkProgram/1` has been issued in the past.

**External documentation.**

```
uniformSubroutinesuiv(Shadertype :: enum(), Indices :: [i()]) ->
                         ok
```

`gl:uniformSubroutines()` loads all active subroutine uniforms for shader stage `Shadertype` of the current program with subroutine indices from `Indices`, storing `Indices[i]` into the uniform at location `I`. `Count` must be equal to the value of `?GL_ACTIVE_SUBROUTINE_UNIFORM_LOCATIONS` for the program currently in use at shader stage `Shadertype`. Furthermore, all values in `Indices` must be less than the value of `?GL_ACTIVE_SUBROUTINES` for the shader stage.

**External documentation.**

```
useProgram(Program :: i()) -> ok
```

`gl:useProgram/1` installs the program object specified by `Program` as part of current rendering state. One or more executables are created in a program object by successfully attaching shader objects to it with `gl:attachShader/2`, successfully compiling the shader objects with `gl:compileShader/1`, and successfully linking the program object with `gl:linkProgram/1`.

**External documentation.**

```
useProgramStages(Pipeline :: i(), Stages :: i(), Program :: i()) ->
                    ok
```

`gl:useProgramStages/3` binds executables from a program object associated with a specified set of shader stages to the program pipeline object given by `Pipeline`. `Pipeline` specifies the program pipeline object to which to bind the executables. `Stages` contains a logical combination of bits indicating the shader stages to use within `Program` with the program pipeline object `Pipeline`. `Stages` must be a logical combination of `?GL_VERTEX_SHADER_BIT`, `?GL_TESS_CONTROL_SHADER_BIT`, `?GL_TESS_EVALUATION_SHADER_BIT`, `?GL_GEOMETRY_SHADER_BIT`, `?GL_FRAGMENT_SHADER_BIT` and `?GL_COMPUTE_SHADER_BIT`. Additionally, the special value `?GL_ALL_SHADER_BITS` may be specified to indicate that all executables contained in `Program` should be installed in `Pipeline`.

**External documentation.**

```
validateProgram(Program :: i()) -> ok
```

`gl:validateProgram/1` checks to see whether the executables contained in `Program` can execute given the current OpenGL state. The information generated by the validation process will be stored in `Program`'s information log. The validation information may consist of an empty string, or it may be a string containing information about how the current program object interacts with the rest of current OpenGL state. This provides a way for OpenGL implementers to convey more information about why the current program is inefficient, suboptimal, failing to execute, and so on.

**External documentation.**

```
validateProgramPipeline(Pipeline :: i()) -> ok
```

`gl:validateProgramPipeline/1` instructs the implementation to validate the shader executables contained in `Pipeline` against the current GL state. The implementation may use this as an opportunity to perform any internal shader modifications that may be required to ensure correct operation of the installed shaders given the current GL state.

**External documentation.**

```
vertex2d(X :: f(), Y :: f()) -> ok
vertex2dv(X1 :: {X :: f(), Y :: f()}) -> ok
vertex2f(X :: f(), Y :: f()) -> ok
vertex2fv(X1 :: {X :: f(), Y :: f()}) -> ok
vertex2i(X :: i(), Y :: i()) -> ok
vertex2iv(X1 :: {X :: i(), Y :: i()}) -> ok
vertex2s(X :: i(), Y :: i()) -> ok
vertex2sv(X1 :: {X :: i(), Y :: i()}) -> ok
vertex3d(X :: f(), Y :: f(), Z :: f()) -> ok
vertex3dv(X1 :: {X :: f(), Y :: f(), Z :: f()}) -> ok
vertex3f(X :: f(), Y :: f(), Z :: f()) -> ok
vertex3fv(X1 :: {X :: f(), Y :: f(), Z :: f()}) -> ok
vertex3i(X :: i(), Y :: i(), Z :: i()) -> ok
vertex3iv(X1 :: {X :: i(), Y :: i(), Z :: i()}) -> ok
vertex3s(X :: i(), Y :: i(), Z :: i()) -> ok
vertex3sv(X1 :: {X :: i(), Y :: i(), Z :: i()}) -> ok
vertex4d(X :: f(), Y :: f(), Z :: f(), W :: f()) -> ok
vertex4dv(X1 :: {X :: f(), Y :: f(), Z :: f(), W :: f()}) -> ok
vertex4f(X :: f(), Y :: f(), Z :: f(), W :: f()) -> ok
vertex4fv(X1 :: {X :: f(), Y :: f(), Z :: f(), W :: f()}) -> ok
vertex4i(X :: i(), Y :: i(), Z :: i(), W :: i()) -> ok
vertex4iv(X1 :: {X :: i(), Y :: i(), Z :: i(), W :: i()}) -> ok
vertex4s(X :: i(), Y :: i(), Z :: i(), W :: i()) -> ok
vertex4sv(X1 :: {X :: i(), Y :: i(), Z :: i(), W :: i()}) -> ok
```

`gl:vertex()` commands are used within `gl:'begin'/1`/`gl:'end'/0` pairs to specify point, line, and polygon vertices. The current color, normal, texture coordinates, and fog coordinate are associated with the vertex when `gl:vertex()` is called.

**External documentation.**

```
vertexArrayElementBuffer(Vaobj :: i(), Buffer :: i()) -> ok
```

`gl:vertexArrayElementBuffer/2` binds a buffer object with id `Buffer` to the element array buffer bind point of a vertex array object with id `Vaobj`. If `Buffer` is zero, any existing element array buffer binding to `Vaobj` is removed.

**External documentation.**

```
vertexAttrib1d(Index :: i(), X :: f()) -> ok
vertexAttrib1dv(Index :: i(), X2 :: {X :: f()}) -> ok
vertexAttrib1f(Index :: i(), X :: f()) -> ok
vertexAttrib1fv(Index :: i(), X2 :: {X :: f()}) -> ok
vertexAttrib1s(Index :: i(), X :: i()) -> ok
vertexAttrib1sv(Index :: i(), X2 :: {X :: i()}) -> ok
vertexAttrib2d(Index :: i(), X :: f(), Y :: f()) -> ok
vertexAttrib2dv(Index :: i(), X2 :: {X :: f(), Y :: f()}) -> ok
vertexAttrib2f(Index :: i(), X :: f(), Y :: f()) -> ok
vertexAttrib2fv(Index :: i(), X2 :: {X :: f(), Y :: f()}) -> ok
vertexAttrib2s(Index :: i(), X :: i(), Y :: i()) -> ok
vertexAttrib2sv(Index :: i(), X2 :: {X :: i(), Y :: i()}) -> ok
vertexAttrib3d(Index :: i(), X :: f(), Y :: f(), Z :: f()) -> ok
vertexAttrib3dv(Index :: i(),
                X2 :: {X :: f(), Y :: f(), Z :: f()}) ->
                   ok
vertexAttrib3f(Index :: i(), X :: f(), Y :: f(), Z :: f()) -> ok
vertexAttrib3fv(Index :: i(),
                X2 :: {X :: f(), Y :: f(), Z :: f()}) ->
                   ok
vertexAttrib3s(Index :: i(), X :: i(), Y :: i(), Z :: i()) -> ok
vertexAttrib3sv(Index :: i(),
                X2 :: {X :: i(), Y :: i(), Z :: i()}) ->
                   ok
vertexAttrib4Nbv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttrib4Niv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttrib4Nsv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttrib4Nub(Index :: i(),
                 X :: i(),
                 Y :: i(),
                 Z :: i(),
                 W :: i()) ->
                    ok
vertexAttrib4Nubv(Index :: i(),
                  X2 :: {X :: i(), Y :: i(), Z :: i(), W :: i()}) ->
                     ok
vertexAttrib4Nuiv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttrib4Nusv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttrib4bv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttrib4d(Index :: i(),
               X :: f(),
               Y :: f(),
               Z :: f(),
               W :: f()) ->
                  ok
vertexAttrib4dv(Index :: i(),
```

```
                X2 :: {X :: f(), Y :: f(), Z :: f(), W :: f()}) ->
                   ok
vertexAttrib4f(Index :: i(),
               X :: f(),
               Y :: f(),
               Z :: f(),
               W :: f()) ->
                  ok
vertexAttrib4fv(Index :: i(),
                X2 :: {X :: f(), Y :: f(), Z :: f(), W :: f()}) ->
                   ok
vertexAttrib4iv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttrib4s(Index :: i(),
               X :: i(),
               Y :: i(),
               Z :: i(),
               W :: i()) ->
                  ok
vertexAttrib4sv(Index :: i(),
                X2 :: {X :: i(), Y :: i(), Z :: i(), W :: i()}) ->
                   ok
vertexAttrib4ubv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttrib4uiv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttrib4usv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttribI1i(Index :: i(), X :: i()) -> ok
vertexAttribI1iv(Index :: i(), X2 :: {X :: i()}) -> ok
vertexAttribI1ui(Index :: i(), X :: i()) -> ok
vertexAttribI1uiv(Index :: i(), X2 :: {X :: i()}) -> ok
vertexAttribI2i(Index :: i(), X :: i(), Y :: i()) -> ok
vertexAttribI2iv(Index :: i(), X2 :: {X :: i(), Y :: i()}) -> ok
vertexAttribI2ui(Index :: i(), X :: i(), Y :: i()) -> ok
vertexAttribI2uiv(Index :: i(), X2 :: {X :: i(), Y :: i()}) -> ok
vertexAttribI3i(Index :: i(), X :: i(), Y :: i(), Z :: i()) -> ok
vertexAttribI3iv(Index :: i(),
                 X2 :: {X :: i(), Y :: i(), Z :: i()}) ->
                    ok
vertexAttribI3ui(Index :: i(), X :: i(), Y :: i(), Z :: i()) -> ok
vertexAttribI3uiv(Index :: i(),
                  X2 :: {X :: i(), Y :: i(), Z :: i()}) ->
                     ok
vertexAttribI4bv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttribI4i(Index :: i(),
                X :: i(),
                Y :: i(),
                Z :: i(),
                W :: i()) ->
```

```
                  ok
vertexAttribI4iv(Index :: i(),
                 X2 :: {X :: i(), Y :: i(), Z :: i(), W :: i()}) ->
                    ok
vertexAttribI4sv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttribI4ubv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttribI4ui(Index :: i(),
                 X :: i(),
                 Y :: i(),
                 Z :: i(),
                 W :: i()) ->
                    ok
vertexAttribI4uiv(Index :: i(),
                  X2 :: {X :: i(), Y :: i(), Z :: i(), W :: i()}) ->
                     ok
vertexAttribI4usv(Index :: i(), V :: {i(), i(), i(), i()}) -> ok
vertexAttribL1d(Index :: i(), X :: f()) -> ok
vertexAttribL1dv(Index :: i(), X2 :: {X :: f()}) -> ok
vertexAttribL2d(Index :: i(), X :: f(), Y :: f()) -> ok
vertexAttribL2dv(Index :: i(), X2 :: {X :: f(), Y :: f()}) -> ok
vertexAttribL3d(Index :: i(), X :: f(), Y :: f(), Z :: f()) -> ok
vertexAttribL3dv(Index :: i(),
                 X2 :: {X :: f(), Y :: f(), Z :: f()}) ->
                    ok
vertexAttribL4d(Index :: i(),
                X :: f(),
                Y :: f(),
                Z :: f(),
                W :: f()) ->
                   ok
vertexAttribL4dv(Index :: i(),
                 X2 :: {X :: f(), Y :: f(), Z :: f(), W :: f()}) ->
                    ok
```

The `gl:vertexAttrib()` family of entry points allows an application to pass generic vertex attributes in numbered locations.

**External documentation.**

```
vertexArrayAttribBinding(Vaobj :: i(),
                         Attribindex :: i(),
                         Bindingindex :: i()) ->
                            ok
vertexAttribBinding(Attribindex :: i(), Bindingindex :: i()) -> ok
```

`gl:vertexAttribBinding/2` and `gl:vertexArrayAttribBinding/3` establishes an association between the generic vertex attribute of a vertex array object whose index is given by `Attribindex`, and a vertex buffer binding whose index is given by `Bindingindex`. For `gl:vertexAttribBinding/2`, the vertex array object affected is that currently bound. For `gl:vertexArrayAttribBinding/3`, `Vaobj` is the name of the vertex array object.

**External documentation.**

```
vertexAttribDivisor(Index :: i(), Divisor :: i()) -> ok
```

`gl:vertexAttribDivisor/2` modifies the rate at which generic vertex attributes advance when rendering multiple instances of primitives in a single draw call. If `Divisor` is zero, the attribute at slot `Index` advances once per vertex. If `Divisor` is non-zero, the attribute advances once per `Divisor` instances of the set(s) of vertices being rendered. An attribute is referred to as instanced if its `?GL_VERTEX_ATTRIB_ARRAY_DIVISOR` value is non-zero.

**External documentation.**

```
vertexArrayAttribFormat(Vaobj, Attribindex, Size, Type,
                        Normalized, Relativeoffset) ->
                           ok
vertexArrayAttribIFormat(Vaobj :: i(),
                         Attribindex :: i(),
                         Size :: i(),
                         Type :: enum(),
                         Relativeoffset :: i()) ->
                            ok
vertexArrayAttribLFormat(Vaobj :: i(),
                         Attribindex :: i(),
                         Size :: i(),
                         Type :: enum(),
                         Relativeoffset :: i()) ->
                            ok
vertexAttribFormat(Attribindex :: i(),
                   Size :: i(),
                   Type :: enum(),
                   Normalized :: 0 | 1,
                   Relativeoffset :: i()) ->
                      ok
vertexAttribIFormat(Attribindex :: i(),
                    Size :: i(),
                    Type :: enum(),
                    Relativeoffset :: i()) ->
                       ok
vertexAttribIPointer(Index :: i(),
                     Size :: i(),
                     Type :: enum(),
                     Stride :: i(),
                     Pointer :: offset() | mem()) ->
                        ok
vertexAttribLFormat(Attribindex :: i(),
                    Size :: i(),
                    Type :: enum(),
                    Relativeoffset :: i()) ->
                       ok
vertexAttribLPointer(Index :: i(),
                     Size :: i(),
                     Type :: enum(),
```

```
                    Stride :: i(),
                    Pointer :: offset() | mem()) ->
                       ok
```

`gl:vertexAttribFormat/5`, `gl:vertexAttribIFormat/4` and `gl:vertexAttribLFormat/4`, as well as `gl:vertexArrayAttribFormat/6`, `gl:vertexArrayAttribIFormat/5` and `gl:vertexArrayAttribLFormat/5` specify the organization of data in vertex arrays. The first three calls operate on the bound vertex array object, whereas the last three ones modify the state of a vertex array object with ID `Vaobj`. `Attribindex` specifies the index of the generic vertex attribute array whose data layout is being described, and must be less than the value of `?GL_MAX_VERTEX_ATTRIBS`.

**External documentation.**

```
vertexAttribPointer(Index, Size, Type, Normalized, Stride,
                    Pointer) ->
                       ok
```

Types:

```
   Index = Size = i()
   Type = enum()
   Normalized = 0 | 1
   Stride = i()
   Pointer = offset() | mem()
```

`gl:vertexAttribPointer/6`, `gl:vertexAttribIPointer/5` and `gl:vertexAttribLPointer/5` specify the location and data format of the array of generic vertex attributes at index `Index` to use when rendering. `Size` specifies the number of components per attribute and must be 1, 2, 3, 4, or `?GL_BGRA`. `Type` specifies the data type of each component, and `Stride` specifies the byte stride from one attribute to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays.

**External documentation.**

```
vertexArrayBindingDivisor(Vaobj :: i(),
                          Bindingindex :: i(),
                          Divisor :: i()) ->
                             ok
vertexBindingDivisor(Bindingindex :: i(), Divisor :: i()) -> ok
```

`gl:vertexBindingDivisor/2` and `gl:vertexArrayBindingDivisor/3` modify the rate at which generic vertex attributes advance when rendering multiple instances of primitives in a single draw command. If `Divisor` is zero, the attributes using the buffer bound to `Bindingindex` advance once per vertex. If `Divisor` is non-zero, the attributes advance once per `Divisor` instances of the set(s) of vertices being rendered. An attribute is referred to as `instanced` if the corresponding `Divisor` value is non-zero.

**External documentation.**

```
vertexPointer(Size :: i(),
              Type :: enum(),
              Stride :: i(),
              Ptr :: offset() | mem()) ->
                 ok
```

`gl:vertexPointer/4` specifies the location and data format of an array of vertex coordinates to use when rendering. `Size` specifies the number of coordinates per vertex, and must be 2, 3, or 4. `Type` specifies the data type of each coordinate, and `Stride` specifies the byte stride from one vertex to the next, allowing vertices and attributes

to be packed into a single array or stored in separate arrays. (Single-array storage may be more efficient on some implementations; see `gl:interleavedArrays/3`.)

**External documentation.**

```
viewport(X :: i(), Y :: i(), Width :: i(), Height :: i()) -> ok
```

`gl:viewport/4` specifies the affine transformation of x and y from normalized device coordinates to window coordinates. Let (x nd y nd) be normalized device coordinates. Then the window coordinates (x w y w) are computed as follows:

**External documentation.**

```
viewportArrayv(First :: i(), V :: [{f(), f(), f(), f()}]) -> ok
```

`gl:viewportArrayv/2` specifies the parameters for multiple viewports simulataneously. `First` specifies the index of the first viewport to modify and `Count` specifies the number of viewports to modify. `First` must be less than the value of `?GL_MAX_VIEWPORTS`, and `First` + `Count` must be less than or equal to the value of `?GL_MAX_VIEWPORTS`. Viewports whose indices lie outside the range [`First`, `First` + `Count`) are not modified. `V` contains the address of an array of floating point values specifying the left ( x), bottom ( y), width ( w), and height ( h) of each viewport, in that order. x and y give the location of the viewport's lower left corner, and w and h give the width and height of the viewport, respectively. The viewport specifies the affine transformation of x and y from normalized device coordinates to window coordinates. Let (x nd y nd) be normalized device coordinates. Then the window coordinates (x w y w) are computed as follows:

**External documentation.**

```
viewportIndexedf(Index :: i(),
                 X :: f(),
                 Y :: f(),
                 W :: f(),
                 H :: f()) ->
                    ok
viewportIndexedfv(Index :: i(), V :: {f(), f(), f(), f()}) -> ok
```

`gl:viewportIndexedf/5` and `gl:viewportIndexedfv/2` specify the parameters for a single viewport. `Index` specifies the index of the viewport to modify. `Index` must be less than the value of `?GL_MAX_VIEWPORTS`. For `gl:viewportIndexedf/5`, `X`, `Y`, `W`, and `H` specify the left, bottom, width and height of the viewport in pixels, respectively. For `gl:viewportIndexedfv/2`, `V` contains the address of an array of floating point values specifying the left ( x), bottom ( y), width ( w), and height ( h) of each viewport, in that order. x and y give the location of the viewport's lower left corner, and w and h give the width and height of the viewport, respectively. The viewport specifies the affine transformation of x and y from normalized device coordinates to window coordinates. Let (x nd y nd) be normalized device coordinates. Then the window coordinates (x w y w) are computed as follows:

**External documentation.**

```
waitSync(Sync :: i(), Flags :: i(), Timeout :: i()) -> ok
```

`gl:waitSync/3` causes the GL server to block and wait until `Sync` becomes signaled. `Sync` is the name of an existing sync object upon which to wait. `Flags` and `Timeout` are currently not used and must be set to zero and the special value `?GL_TIMEOUT_IGNORED`, respectively

`Flags` and `Timeout` are placeholders for anticipated future extensions of sync object capabilities. They must have these reserved values in order that existing code calling `gl:waitSync/3` operate properly in the presence of such extensions.

**External documentation.**

```
windowPos2d(X :: f(), Y :: f()) -> ok
windowPos2dv(X1 :: {X :: f(), Y :: f()}) -> ok
windowPos2f(X :: f(), Y :: f()) -> ok
windowPos2fv(X1 :: {X :: f(), Y :: f()}) -> ok
windowPos2i(X :: i(), Y :: i()) -> ok
windowPos2iv(X1 :: {X :: i(), Y :: i()}) -> ok
windowPos2s(X :: i(), Y :: i()) -> ok
windowPos2sv(X1 :: {X :: i(), Y :: i()}) -> ok
windowPos3d(X :: f(), Y :: f(), Z :: f()) -> ok
windowPos3dv(X1 :: {X :: f(), Y :: f(), Z :: f()}) -> ok
windowPos3f(X :: f(), Y :: f(), Z :: f()) -> ok
windowPos3fv(X1 :: {X :: f(), Y :: f(), Z :: f()}) -> ok
windowPos3i(X :: i(), Y :: i(), Z :: i()) -> ok
windowPos3iv(X1 :: {X :: i(), Y :: i(), Z :: i()}) -> ok
windowPos3s(X :: i(), Y :: i(), Z :: i()) -> ok
windowPos3sv(X1 :: {X :: i(), Y :: i(), Z :: i()}) -> ok
```

The GL maintains a 3D position in window coordinates. This position, called the raster position, is used to position pixel and bitmap write operations. It is maintained with subpixel accuracy. See `gl:bitmap/7`, `gl:drawPixels/5`, and `gl:copyPixels/5`.

**External documentation.**

# glu

Erlang module

Standard OpenGL API

This documents the functions as a brief version of the complete **OpenGL reference pages.**

## Data Types

```
vertex() = {float(), float(), float()}
i() = integer()
f() = float()
enum() = integer() >= 0
matrix() = m12() | m16()
m12() =
    {f(), f(), f(), f(), f(), f(), f(), f(), f(), f(), f(), f()}
m16() =
    {f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f(),
     f()}
mem() = binary() | tuple()
```

## Exports

```
build1DMipmapLevels(Target, InternalFormat, Width, Format, Type,
                    Level, Base, Max, Data) ->
                       i()
```

Types:

```
   Target = enum()
   InternalFormat = Width = i()
   Format = Type = enum()
   Level = Base = Max = i()
   Data = binary()
```

`glu:build1DMipmapLevels/9` builds a subset of prefiltered one-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

**External documentation.**

```
build1DMipmaps(Target, InternalFormat, Width, Format, Type, Data) ->
                  i()
```

Types:

```
   Target = enum()
   InternalFormat = Width = i()
   Format = Type = enum()
   Data = binary()
```

`glu:build1DMipmaps/6` builds a series of prefiltered one-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

**External documentation.**

```
build2DMipmapLevels(Target, InternalFormat, Width, Height, Format,
                    Type, Level, Base, Max, Data) ->
                       i()
```

Types:

```
   Target = enum()
   InternalFormat = Width = Height = i()
   Format = Type = enum()
   Level = Base = Max = i()
   Data = binary()
```

`glu:build2DMipmapLevels/10` builds a subset of prefiltered two-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

**External documentation.**

```
build2DMipmaps(Target, InternalFormat, Width, Height, Format,
               Type, Data) ->
                  i()
```

Types:

```
   Target = enum()
   InternalFormat = Width = Height = i()
   Format = Type = enum()
   Data = binary()
```

`glu:build2DMipmaps/7` builds a series of prefiltered two-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture-mapped primitives.

**External documentation.**

```
build3DMipmapLevels(Target, InternalFormat, Width, Height, Depth,
                    Format, Type, Level, Base, Max, Data) ->
                       i()
```

Types:

```
   Target = enum()
   InternalFormat = Width = Height = Depth = i()
   Format = Type = enum()
   Level = Base = Max = i()
   Data = binary()
```

`glu:build3DMipmapLevels/11` builds a subset of prefiltered three-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture mapped primitives.

**External documentation.**

```
build3DMipmaps(Target, InternalFormat, Width, Height, Depth,
               Format, Type, Data) ->
                  i()
```

Types:

```
   Target = enum()
   InternalFormat = Width = Height = Depth = i()
   Format = Type = enum()
   Data = binary()
```

`glu:build3DMipmaps/8` builds a series of prefiltered three-dimensional texture maps of decreasing resolutions called a mipmap. This is used for the antialiasing of texture-mapped primitives.

**External documentation.**

```
checkExtension(ExtName :: string(), ExtString :: string()) ->
                  0 | 1
```

`glu:checkExtension/2` returns `?GLU_TRUE` if `ExtName` is supported otherwise `?GLU_FALSE` is returned.

**External documentation.**

```
cylinder(Quad :: i(),
         Base :: f(),
         Top :: f(),
         Height :: f(),
         Slices :: i(),
         Stacks :: i()) ->
            ok
```

`glu:cylinder/6` draws a cylinder oriented along the `z` axis. The base of the cylinder is placed at `z` = 0 and the top at z=height. Like a sphere, a cylinder is subdivided around the `z` axis into slices and along the `z` axis into stacks.

**External documentation.**

```
deleteQuadric(Quad :: i()) -> ok
```

`glu:deleteQuadric/1` destroys the quadrics object (created with `glu:newQuadric/0`) and frees any memory it uses. Once `glu:deleteQuadric/1` has been called, `Quad` cannot be used again.

**External documentation.**

```
disk(Quad :: i(),
     Inner :: f(),
     Outer :: f(),
     Slices :: i(),
     Loops :: i()) ->
        ok
```

`glu:disk/5` renders a disk on the `z` = 0 plane. The disk has a radius of `Outer` and contains a concentric circular hole with a radius of `Inner`. If `Inner` is 0, then no hole is generated. The disk is subdivided around the `z` axis into slices (like pizza slices) and also about the `z` axis into rings (as specified by `Slices` and `Loops`, respectively).

**External documentation.**

```
errorString(Error :: enum()) -> string()
```

`glu:errorString/1` produces an error string from a GL or GLU error code. The string is in ISO Latin 1 format. For example, `glu:errorString/1(?GLU_OUT_OF_MEMORY)` returns the string `out of memory`.

**External documentation.**

```
getString(Name :: enum()) -> string()
```

`glu:getString/1` returns a pointer to a static string describing the GLU version or the GLU extensions that are supported.

**External documentation.**

```
lookAt(EyeX, EyeY, EyeZ, CenterX, CenterY, CenterZ, UpX, UpY, UpZ) ->
          ok
```

Types:

```
   EyeX = EyeY = EyeZ = CenterX = CenterY = CenterZ = UpX = UpY = UpZ = f()
```

`glu:lookAt/9` creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an `UP` vector.

**External documentation.**

```
newQuadric() -> i()
```

`glu:newQuadric/0` creates and returns a pointer to a new quadrics object. This object must be referred to when calling quadrics rendering and control functions. A return value of 0 means that there is not enough memory to allocate the object.

**External documentation.**

```
ortho2D(Left :: f(), Right :: f(), Bottom :: f(), Top :: f()) ->
```

```
           ok
```

`glu:ortho2D/4` sets up a two-dimensional orthographic viewing region. This is equivalent to calling `gl:ortho/6` with near=-1 and far=1.

**External documentation.**

```
partialDisk(Quad, Inner, Outer, Slices, Loops, Start, Sweep) -> ok
```

Types:

```
   Quad = i()
   Inner = Outer = f()
   Slices = Loops = i()
   Start = Sweep = f()
```

`glu:partialDisk/7` renders a partial disk on the z=0 plane. A partial disk is similar to a full disk, except that only the subset of the disk from `Start` through `Start` + `Sweep` is included (where 0 degrees is along the +f2yf axis, 90 degrees along the +`x` axis, 180 degrees along the -`y` axis, and 270 degrees along the -`x` axis).

**External documentation.**

```
perspective(Fovy :: f(), Aspect :: f(), ZNear :: f(), ZFar :: f()) ->
              ok
```

`glu:perspective/4` specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in `glu:perspective/4` should match the aspect ratio of the associated viewport. For example, aspect=2.0 means the viewer's angle of view is twice as wide in `x` as it is in `y`. If the viewport is twice as wide as it is tall, it displays the image without distortion.

**External documentation.**

```
pickMatrix(X :: f(),
           Y :: f(),
           DelX :: f(),
           DelY :: f(),
           Viewport :: {i(), i(), i(), i()}) ->
              ok
```

`glu:pickMatrix/5` creates a projection matrix that can be used to restrict drawing to a small region of the viewport. This is typically useful to determine what objects are being drawn near the cursor. Use `glu:pickMatrix/5` to restrict drawing to a small region around the cursor. Then, enter selection mode (with `gl:renderMode/1`) and rerender the scene. All primitives that would have been drawn near the cursor are identified and stored in the selection buffer.

**External documentation.**

```
project(ObjX, ObjY, ObjZ, Model, Proj, View) ->
           {i(), WinX :: f(), WinY :: f(), WinZ :: f()}
```

Types:

```
ObjX = ObjY = ObjZ = f()
Model = Proj = matrix()
View = {i(), i(), i(), i()}
```

`glu:project/6` transforms the specified object coordinates into window coordinates using `Model`, `Proj`, and `View`. The result is stored in `WinX`, `WinY`, and `WinZ`. A return value of `?GLU_TRUE` indicates success, a return value of `?GLU_FALSE` indicates failure.

**External documentation.**

```
quadricDrawStyle(Quad :: i(), Draw :: enum()) -> ok
```

`glu:quadricDrawStyle/2` specifies the draw style for quadrics rendered with `Quad`. The legal values are as follows:

**External documentation.**

```
quadricNormals(Quad :: i(), Normal :: enum()) -> ok
```

`glu:quadricNormals/2` specifies what kind of normals are desired for quadrics rendered with `Quad`. The legal values are as follows:

**External documentation.**

```
quadricOrientation(Quad :: i(), Orientation :: enum()) -> ok
```

`glu:quadricOrientation/2` specifies what kind of orientation is desired for quadrics rendered with `Quad`. The `Orientation` values are as follows:

**External documentation.**

```
quadricTexture(Quad :: i(), Texture :: 0 | 1) -> ok
```

`glu:quadricTexture/2` specifies if texture coordinates should be generated for quadrics rendered with `Quad`. If the value of `Texture` is `?GLU_TRUE`, then texture coordinates are generated, and if `Texture` is `?GLU_FALSE`, they are not. The initial value is `?GLU_FALSE`.

**External documentation.**

```
scaleImage(Format, WIn, HIn, TypeIn, DataIn, WOut, HOut, TypeOut,
           DataOut) ->
              i()
```

Types:

```
Format = enum()
WIn = HIn = i()
TypeIn = enum()
DataIn = binary()
WOut = HOut = i()
TypeOut = enum()
DataOut = mem()
```

`glu:scaleImage/9` scales a pixel image using the appropriate pixel store modes to unpack data from the source image and pack data into the destination image.

**External documentation.**

```
sphere(Quad :: i(), Radius :: f(), Slices :: i(), Stacks :: i()) ->
          ok
```

`glu:sphere/4` draws a sphere of the given radius centered around the origin. The sphere is subdivided around the `z` axis into slices and along the `z` axis into stacks (similar to lines of longitude and latitude).

**External documentation.**

```
tesselate(Normal, Vs :: [Vs]) -> {Triangles, VertexPos}
```

Types:

```
Normal = Vs = vertex()
Triangles = [integer()]
VertexPos = binary()
```

Triangulates a polygon, the polygon is specified by a `Normal` and `Vs` a list of vertex positions.

The function returns a list of indices of the vertices and a binary (64bit native float) containing an array of vertex positions, it starts with the vertices in `Vs` and may contain newly created vertices in the end.

```
unProject(WinX, WinY, WinZ, Model, Proj, View) ->
             {i(), ObjX :: f(), ObjY :: f(), ObjZ :: f()}
unProject4(WinX, WinY, WinZ, ClipW, Model, Proj, View, NearVal,
           FarVal) ->
              {i(),
               ObjX :: f(),
               ObjY :: f(),
               ObjZ :: f(),
               ObjW :: f()}
```

Types:

```
WinX = WinY = WinZ = ClipW = f()
Model = Proj = matrix()
View = {i(), i(), i(), i()}
NearVal = FarVal = f()
```

`glu:unProject/6` maps the specified window coordinates into object coordinates using `Model`, `Proj`, and `View`. The result is stored in `ObjX`, `ObjY`, and `ObjZ`. A return value of `?GLU_TRUE` indicates success; a return value of `?GLU_FALSE` indicates failure.

**External documentation.**