MiniWin Embedded Window Manager
8 October 2018

1 Introduction

MiniWin is a generic open source window manager for small embedded systems with a touch screen. MiniWin is written in compliance with the C99 standard. The hardware interface is separated out into a small hardware abstraction layer making the rest of the system re-targetable to a wide variety of processors. MiniWin will run easily on ARM Cortex M3 or above, PIC32 or Renesas RX processors. It will not run on small 8 bit processors, for example PIC 18F.

2 Who is MiniWin For?

- Open source projects that need a quick-start user interface
- Small commercial products that do not have the budget to buy a window manager or the manpower to develop one
- Hardware constrained devices that have limited resources but also have limited requirements
- Student projects that can build on the supplied example code
- Developers who want a quick-start user interface without the need for extensive driver development effort

3 Features

- Written specifically for small embedded systems with a LCD display and a touch screen
- Apart from a small hardware abstraction layer, platform independent
- Supports multiple overlapped windows with Z ordering
- Incorporates a flexible graphics library
- Comes with a basic set of user interface controls in two sizes - a standard size for use with a touch screen and stylus, and large size for a touch screen with finger input.
- Includes a set of standard dialogs that need no further code written to use.
- No dynamic memory used - all data structures allocated at compile time
- Two fonts included - fixed width and proportional
- A clean easy to use API
- Comprehensive documentation and example code
- Runs in bare metal systems or within a single thread of a RTOS
- Example project showing FatFS file system integration for reading and writing to a USB pen drive
- Requires minimal memory - no off-screen buffering used
- Compiles without warning with GCC
- Doxygen documentation for every function and type
- In-built touch screen calibration capability the first time the window manager is started

4 User Interface Controls

- Button
- Check box
- List box with optional icons
- Radio buttons
- Menu bar
- Text label
- Progress bar
- Horizontal scroll bar
- Vertical scroll bar
- Integer number entry field
- Single line generic text entry field

These are the standard ones. Most controls (except number and text input) come in 2 sizes. Typically the smaller size is for use with a stylus operated touch screen and the larger size for a finger operated touch screen. You can easily add more control types.

5 MiniWin Windows

- Optional title bar, title and border
- Overlapped with unique Z order, fixed tiled or a combination
- Movable
- Resizable
- Maximise, minimise and close icons
- Minimisable to icon on desktop
- Customisable desktop colour or bitmap
- Can have a single system modal window
- Window-aware graphics library for drawing in a window

Of course, you don't have to use overlapped windows at all. If you want fixed dedicated areas of the display with each area being responsive to user input and having its own message and paint functions, that's possible too. Any window can be created with no border or title bar, and like this it's fixed on the screen.

6 Developer Utility Applications

MiniWin includes applications (with source code) to convert monochrome and 24 bit colour bitmaps in Microsoft Windows .bmp file format to C99 source code. These can then be included straight into your project and are flashed into your device's memory when you program your application into your board.

These utilities can be built for Windows or Linux from source. Project files for Atollic TrueStudio are included.

These applications are command line console/shell applications. Running either application with no arguments shows their usage.

7 Drivers and Examples

MiniWin comes with all source for the window manager, graphics libraries and user interface components. It also comes with 4 example projects - a simple getting started example, a fully comprehensive project showing usage of all MiniWin's features, an example that integrates the FatFS file system and a USB driver for a pen drive, and a non-overlapped tiled windows example.

MiniWin comes with three hal layer samples – a minimalist (but slow) embedded one for the STM32F407 that uses a generic IL9325 driver chip, another accelerated embedded one for the STM32F429DISC1 development board and a third one that allows the example applications to run within a Microsoft Windows window. Each of the 4 example projects can be built for all three of the hal drivers.

All sample projects are built with the free compiler/IDE Atollic TrueStudio, currently version 9.0.1.

8 Customisability

Because MiniWin is fully open source its look and feel is fully customisable. Don't like the look of a user interface component? Simple. Modify its paint function and draw it how you want it to look. More user interface controls and dialog windows can also be added by copying from the standard set provided.

9 Implementing the Hardware Abstraction Layer (hal)

Note: In this document HAL in capitals refers to the ST driver library used for STM32 devices; hal in lower case refers to the hardware interface part of MiniWin.

The hal sub-project (found under folder `hal` in the source code) collects all the drivers required by MiniWin in one location and implements as one low level layer. No other MiniWin code has any hardware dependency. To port the MiniWin code to your hardware you need to implement the drivers found here. The following sections describe what you need to do for each driver.

9.1 Delay Driver

Source file: `hal_delay.c, hal_delay.h`

These functions wait in a busy loop for a specified time. This routine is only used by other drivers under the hal sub-project. If your drivers do not need this capability you do not need to implement these functions, but some LCD drivers need delays during initialisation. Do not use these functions from within your user code as you will block the responsiveness of your user interface. Use the MiniWin tick timer and message handlers instead.

Function: `mw_hal_delay_init`
Any hardware initializations for the delay driver are here.

Function: `mw_hal_delay_ms`
Delay for a specified number of milliseconds.

Function: `mw_hal_delay_us`
Delay for a specified number of microseconds.

9.2 Init Driver

Source file: `hal_init.c, hal_init.h`

This is not a driver in itself but collects together the initialization routines of all the other drivers.

Function: `mw_hal_init`
Calls all other init routines in the hal sub-project

9.3 Timer Driver

Source file: `hal_timer.c, hal_timer.h`

This driver uses a hardware timer to drive the MiniWin tick counter at 20 Hz.

Function: `mw_hal_timer_init`
Any hardware initializations for the tick timer driver are here.
Function: varies, timer interrupt handler
This function, which the interrupt routine calls 20 times per second, increments the MiniWin tick counter on each call.

9.4 Non-Volatile Storage Driver

Source file: `hal_non_vol.c, hal_non_vol.h`

This driver stores settings data in non-volatile memory. This could be a section of internal flash memory, on chip EEPROM, a SD card or an external chip.

Function: `hal_non_vol_init`
Any hardware inititlisations for the non-vol driver are here.

Function: `hal_non_vol_load`
This function loads a specified length of data from non-volatile memory into a buffer owned by the caller.

Function: `hal_non_vol_save`
This function saves a specified length of data into non-volatile storage. It is up to the driver to decide where in non-volatile memory to save the data.

9.5 Touch Driver

Source file: `hal_touch.c, hal_touch.h`

This driver detects when the touch screen is pressed and can read the touch screen when requested.

Function: `mw_hal_touch_init`
Any hardware inititlisations for the touch driver are here.

Function: `mw_hal_touch_get_state`
This function returns whether the screen is currently being touched or not.

Function: mw_hal_touch_get_point
This function gets the current touch point. It should only be called directly after mw_hal_touch_get_state has been called (and reported that the screen is currently being touched) to ensure that there is a valid touch point to read. If there is no valid touch point to read then the function returns false.

This function must return a filtered stable touch point. If this requires multiple reads to be made and averaged (or otherwise filtered) then this must be performed within this driver function. The function returns the touch point in raw touch digitizer co-ordinates, which may not be the same as pixel coordinates. MiniWin incorporates a calibration routine which removes touch screen non-linearities and also transforms the touch point co-ordinates from raw digitizer co-ordinates (as returned by this function) to pixel co-ordinates.

9.6 LCD Driver

Source file: hal_lcd.c, hal_lcd.h

This driver implements basic write functions to the display device screen and defines the colour type and some colours. There is no screen read capability requirement for MiniWin.

Type: mw_hal_lcd_colour_t
This typedef represents the colour format used by MiniWin and as of version 2.0.0 is a 32 bit to hold a 24 bit colour representation.

Defines: MW_HAL_LCD_WIDTH and MW_HAL_LCD_HEIGHT
These are the size in pixels of the display device.

Defines: MW_HAL_LCD_BLACK, etc.
A variety of pre-named colours are defined, the minimum being colours for black and white (MW_HAL_LCD_BLACK and MW_HAL_LCD_WHITE).

Function: mw_hal_lcd_init
This function initializes the display device hardware.

Function: mw_hal_lcd_pixel
This function writes a single pixel with the specified colour to the display device. The call is ignored if the specified position is not within the screen maximum co-ordinates.

Function mw_hal_lcd_filled_rectangle_clip
This function fills a rectangle on the screen with a single colour. The co-ordinates of all points within the specified rectangle are clipped to the screen maximum co-ordinates and also to a caller specified rectangle width and height.

Function mw_hal_colour_bitmap_clip
This function fills a full colour rectangle on the screen with data from bitmap specified by the caller. The bitmap data must be in the format of 3 bytes per pixel. The image data must start at the top left and proceed across a full row before starting the next row. The co-ordinates of all points within the specified

rectangle are clipped to the screen maximum co-ordinates and also to a caller specified rectangle width and height.

Function `mw_hal_lcd_monochrome_bitmap_clip`
This function fills a monochrome rectangle on the screen with data from bitmap specified by the caller. The bitmap data must be in 8 pixels per byte format, left most pixel in the byte's msb. The image data must start at the top left and proceed across a full row before starting the next row. The co-ordinates of all points within the specified rectangle are clipped to the screen maximum co-ordinates and also to a caller specified rectangle width and height. The actual colours to use for the 2 possible colours are specified in the device colour format.

10 Paint Algorithm

MiniWin uses a painting algorithm that is optimized for use on a system without a shadow screen buffer, i.e. all writes to the display memory are shown immediately on the display. To avoid flickering each part of the display is written to only once for each repaint request.

The repaint algorithm used is the singly-combined sorted intersections algorithm. This algorithm has two parts.

When a repaint request is received for a rectangular area the first part the algorithm searches for all window edges that intersect this area. These edges are collected into two arrays, one of horizontal edges, the other of vertical edges. The edges of the rectangle being repainted are also added to the arrays. Then the contents of each array is sorted numerically.

For the second part of the algorithm there is a pair of nested loops that iterates through the array of horizontal edges and for each horizontal edge iterates through the array of vertical edges. For each horizontal/vertical edge intersection the highest Z order of the visible window (including root) at the point is found, along with the window that has this Z order. The paint algorithm is called for this window with a clip area with origin of this intersection point and a width and height of the difference between the current intersection in the edges arrays and the subsequent intersections.

The singly-combined part of the algorithm means that when a rectangle to repaint is found in the inner loop the window's paint function is not called immediately. Instead, the call to the repaint is suspended until the next iteration of the inner loop. If at the next iteration it is found that the subsequent rectangle has the same Z order as the previous then painting is delayed again. If the subsequent rectangle has a different Z order then only now is the previously identified rectangle (or combined rectangles) painted. In this way rectangles along the same row are combined, but rectangles on different rows are not - hence singly-combined.

Additional optimizations take place when deciding on how to repaint a rectangle. If a rectangle is completely within a window that has focus then nothing overlaps it and it is painted completely. If a rectangle is completely overlapped by higher Z order windows then its repaint request is ignored.

11 Z Orders

All windows have a Z order. This is the window's position in the stack of windows and controls which window is drawn on which. 0 is the lowest Z order and this value is reserved for the root window. Reasons for a Z ordering change are: a new window is created, a window is removed, a window is sent to the back, a window is sent to the front, a window is minimized, a window is restored, and a window is given focus. Whenever a Z ordering change occurs all the Z orders are rationalized, which means that they go from 1 to the number of currently used user windows with no gaps in the numbering. This is done automatically by MiniWin; no user interaction is required.

The window with the highest Z order is the window with focus. This window is drawn with a different colour title bar (blue by default) from unfocused windows (title bar grey by default). When a window is created or restored it is given the highest Z order and is given focus. When a window loses focus, because it is removed, minimized, or another window is brought to the front the next highest Z ordered window gains focus. A message is sent to the window message handler whenever a window gains or loses focus.

Controls are handled differently. They do not have a Z order compared to other controls in the same window. It is safest if controls are positioned so that they do no overlap as the painting order and hence which one appears on top of which is uncontrolled. If a control needs to appear temporarily on top of another control, for example simulating a cascading menu with a pop up list box, then it is up to the user to make sure that the underlying control is temporarily made invisible. This may seem as a limitation to using layered pop up controls, but it keeps MiniWin small and simple.

12 Multiple Instances of the Same Window

Most windows are used with only a single instance. In these cases any data the window needs to store its state can be within the window source file as single instance static data and the code within the message handler and paint function within the source file can access the locally stored data.

However, there may be cases where multiple instances of the same window are required to be displayed. In these cases no window data can be stored within the source file as static data as this would not allow different data for the different instances of the window. In these cases each window instance's data needs to be stored externally and a pointer to this data made available to the message handler and paint functions. Therefore each window has associated with it a void pointer which is used to store instance data where multiple instances of the same window are required. This pointer is set when the window is created in the `mw_add_window` function. If only a single instance of a window is required and window data is stored in the window's source file as static data then this pointer can be `NULL`.

13 Graphics Library

MiniWin comes with a standard graphics library supporting the normal lines, fonts and shapes as well as polygon and shape rotation. It includes simple patterned lines and texture fills.

All graphics library drawing routines are window aware - you do not have to worry about where your window is, if it is overlapped or partly off screen. You just draw to the client area regardless using the client area as your frame of reference. MiniWin performs all window offset translations and clipping required. You will never end up drawing outside of your window or scribbling somewhere you shouldn't.

The graphics library is for painting windows and controls. All painting should be done via this graphics library to ensure correct clipping of any graphics function coordinates that fall outside of a window.

The coordinate system used in all graphics commands in gl is that of the item being painted. If a window frame is being painted then the origin coordinate (0, 0) represents the top left corner of the window frame. If a window client area is being painted (0, 0) represents the top left corner of the client area, and for a control (0, 0) represents the top left corner of the control's client area.

Calling a gl function with coordinates beyond the boundaries of the item (window frame, client area, control) being painted (including negative coordinate values) will result in the pixels beyond boundary being automatically clipped. Pixels up to the boundary will be painted, so for example if a line is asked to be drawn where the origin is within the area being painted but the end point is outside the boundary the line will be drawn from the origin up to the boundary.

All window and control paint routines pass in a pointer to a `mw_gl_draw_info_t` parameter. This contains information on the frame of reference origin of the area item being painted and the clipping area extents. It is not necessary for the user to use any values in this data structure, simple pass it on to any function calls in gl.

The gl library uses the concept of a graphics context in its calls. Individual functions are not passed parameters describing the required colour, line style, fill pattern etc. These are set in a graphics context structure that belongs to gl using the gl API. Values in the graphics context are set first and then all subsequent calls to gl functions will use these values. Values that can be set in the graphics context are as follows:

- foreground colour
- background colour
- line pattern
- solid fill colour
- solid fill pattern
- background transparency
- border on or off

The contents of a graphics context are not preserved between subsequent calls to paint functions. It is necessary to set the values at every use. The gl library provides an accessor for a pointer to its internal gl structure so if setting the contents on every paint function call is not suitable the user can keep a local copy and use `memcpy()` to copy the local copy into gl's copy as an alternative.

14 Standard Controls Overview

There is a set of standard controls available in MiniWin. This set is smaller than that provided by more sophisticated window managers, but they can be combined to work collaboratively to produce a greater range - for example the list box control can be combined with the standalone vertical scrollbar to produce a scrolling list box. Some extra user code is required to do this.

Most of the controls provided by MiniWin come in two sizes – a small size for stylus operated touch screens and a large size for finger operated touch screens. The exceptions to this are the number and text entry controls which come only in the small size because of size limitations on the LCD display typically used for MiniWin projects. Progress bars are created with a user specified size and there is no small or large concept for this control type.

Each control has its own source and header files. For each control there are a minimum of 3 functions - the control message handler, the control paint function and a utility function to simplify the creation of the control. The controls' message handlers and paint functions must all have the same parameters and return type. These function signatures are declared in `miniwin.h`.

Each control has its own data structure storing fields common to all controls, for example visibility, size, currently used and enabled statuses. These data structures are stored in an array of all the controls. The size of this array is specified at compile time in the MiniWin configuration header file. In addition to this common data structure each control type has a control-specific data structure to store instance data unique to that control. For example, a button has a single label but a list box has an array of labels, icons and the array size. Each instance data structure is declared in the control's header file and the standard control common data structure contains a void pointer to reference it. The fields in the control-specific data structure are initialized when the control is created. Some fields are user modifiable and some are not; the non-user-modifiable fields contain control state and should not be changed by the user. Each control-specific data structure indicates which fields are user modifiable and which should not be changed in the structure's declaration it the control's header file.

It is possible in MiniWin to have multiple instances of the same control within the system, for example multiple buttons. Each instance requires its own entry in the array of control common structures and also requires its own instance of its control-specific data structure.

When controls are created there are 3 flags that can be specified to alter their appearance and behaviour. These flags and their effects are:

`MW_CONTROL_FLAGS_LARGE_SIZE`	Create the control as large size if supported
`MW_CONTROL_FLAG_IS_ENABLED`	Create the control so that it is enabled
`MW_CONTROL_FLAG_IS_VISIBLE`	Create the control so that it is visible

15 Enabling and Disabling Controls

All controls and control like features of a window frame (menu bar, vertical and horizontal scroll bars) can be enabled and disabled individually. When a control is disabled it accepts no input and is drawn greyed out. Controls and the menu bar can have multiple items that cause an input event, i.e. different menu items along the bar or different lines in a list box. As well as the global enable/disable feature these controls can also have each item enabled and disabled separately. This is done by setting bits in a 16 bit bit field. This means that the maximum number of items in a menu bar or list box is 16.

There are utilities functions in `miniwin_utilities.c` that allow a single bit in a 16 bit bit field to be set or read. There are also macros defined in `miniwin.h` for setting or clearing all bits.

If the global enable flag for this type of controls is false then all items are disabled. If the global flag is true then items that also have their bit field bits set are enabled, all others are disabled.

16 Standard Dialogs

MiniWin contains a set of standard dialogs. These are pre-coded windows that are instantiated with a single function call that can partly customise them. Dialogs are all shown as modal – that is the user must respond to them and dismiss them before continuing. Because of this only 1 dialog can be shown at any time and they are automatically given focus and sent to the top of all other windows when shown. They cannot be resized, moved or closed except by pressing one of the dismiss buttons. Data can be sent to any window from a dialog when it is closed. The following sections describe the dialogs that exist within MiniWin so far.

16.1 Single Button Message Box

This dialog shows a user configurable message in a window with a user configurable title and contains a single button with a user configurable label. When the dialog is dismissed by the button a message is sent to a specified window.

16.2 Double Button Message Box

This dialog shows a user configurable message in a window with a user configurable title and contains two buttons with user configurable labels. When the dialog is dismissed by one of the buttons a message is sent to a specified window containing which button was pressed in its data.

16.3 Time Chooser

This dialog allows the user to set a time in hours and minutes. The initial time to show when the dialog appears can be set. The user can dismiss the dialog with ok or cancel buttons. When the dialog is dismissed with the ok button a message is sent to a specified window with the user selected time.

16.4 Date Chooser

This dialog allows the user to set a date in year (4 digit), month (1-12) and date (1-28, 29, 30 or 31). The initial date to show when the dialog appears can be set. The user can dismiss the dialog with ok or cancel buttons. When the dialog is dismissed with the ok button a message is sent to a specified window with the user selected date.

17 MiniWin Messages

The MiniWin window manager works using a message queue for requests for actions from user code or reporting of events to user code. These actions and events happen asynchronously. This means that when a request is made or an event occurs a message is posted to the window manager's message queue and the window manager processes this message at a later time. User code must continually call the window manager's message processing function to get anything done. If the user fails to call the message processing function the user interface will not respond to input and not repaint any windows or controls. The reason for this design pattern is to keep MiniWin running in a single thread and the code simple.

The MiniWin message queue and message processing function is part of MiniWin. The user code's interaction with this process is to handle messages in their window's message handler and to call the message processing function repeatedly and often from their main loop. User code must handle messages to respond to events and can also post messages of their own to the message queue, either to coordinate with the window manager or to pass messages to other windows and controls in the system.

The message used by MiniWin are defined in `miniwin.h.` They fall into 3 groups described below:

17.1 Messages posted by the window manager

These messages can be handled by user code in window and control message handlers but should never be posted by user code as they are posted automatically by the window manager. For example, if user code calls the window manager function to add a new window or the user shuts a window from the user interface then the window manager will automatically post the window created or window removed message respectively.

17.2 Messages posted by controls in response to user interface input

When the user interacts with a control the control code will post a message letting user code know what has happened. These messages are not posted from user code.

17.3 Messages that can be posted from user code or user code called utility functions

These messages can be posted from user code. Some can also be posted from window manager code as well. The recipient of these messages can be other windows, other controls or the window manager. Some of the messages, for example paint all windows, are posted from utility function wrappers in the window manager but called directly from user code. In effect, these are posted from user code.

A MiniWin message has 5 fields. These are described below:

Sender id: This is the window id or control id of the posted for messages posted from window or control code.

Recipient id: This is the window id or control id of the recipient of messages posted to windows or controls.

Message id: This is the message number from the list in `miniwin.h`

Recipient type: This is the type of the recipient and can be a window, a control, the window manager or a special value indicating that the message has been cancelled and should be removed.

Message data: This is a 32 bit data value sent from the sender to the recipient. It can be a pointer to a larger data type but if used as this then the memory pointed to must exist for the lifetime of the message.

It is not necessary to fill in all of the fields for all messages. The context will indicate when this is not necessary, for example a message to the window manager does not have a recipient id, and the message data field is often not needed. There is a pre-defined value which can be used to indicate an unused field in `miniwin.h` called `MW_UNUSED_MESSAGE_PARAMETER`.

18 Transferring Data to Message Handler Functions

All communication between windows and controls with each other is done by passing messages. The MiniWin message structure contains a 32 bit data field. This is general purpose and can be used to pass a 32 bit values, 2 16 bit values or 4 bytes. It can also be used to pass a pointer to anything with appropriate casting. If the data field is used to pass a pointer then the object pointed to must exist when the message is posted and still exist when the message is processed.

As no dynamic memory allocations are used in MiniWin this only means that the object pointed to must not be a local variable that goes out of scope after the message is posted - it needs to either static if it's a local function scope variable or else be a file scope variable or a global variable, or a constant. This restriction does not apply to non-pointer data if the single 32 bit value is used as data as the value is copied into the message.

There are two types of data transfer messages that can be posted - one that indicates that the data is a 32 bit value and the other that it is a pointer. For each type there are numbered variants from 0 to 5 so that 5 different means of each type can be used. This number can be extended in `miniwin.h` or else the user could create their own message identifiers.

There are also 5 general purpose user message identifiers defined in `miniwin.h` which user code can use for any purpose.

19 Touch Events

WiniMin creates 3 different touch events from user input on the touch screen. Each message passes the coordinates of the touch location in the corresponding message's data parameter. The x coordinate is in the upper 16 bits and the y coordinate in the lower 16 bits.

Touch down: this event has a new location, the point on which the screen has been touched, so this is passed in the message.

Touch up: only the last known touch screen location is known, so this is passed in the message.

Touch drag: this also has a new touch location which is passed in the message.

For all touch events there is a time difference which must pass between two touch events being created. This prevents multiple messages being passed for a single user touch, a touch screen equivalent of key de-bouncing. For drag messages there is a distance difference which must be exceeded between the posting of two touch events. This is to prevent a flood of drag messages being created from the touch point oscillating between two adjacent locations.

Both the touch event time difference and the drag distance difference are customizable in the configuration header file.

When a touch down is made in a window that does not have focus the window receiving the touch down event is given focus and brought to the front. By default, after this, the touch event has been consumed by the giving of focus to the window and the window message function does not receive the touch down event. This behaviour is unsuitable for non-overlapped fixed windows where windows without title bars do not show any sign of having focus to the user. In this case it is desirable if the touch event in a non-focused window is also passed on to the window's message handler as a standard touch down event. It is possible to configure this behaviour when creating the fixed window by specifying the `MW_WINDOW_FLAG_TOUCH_FOCUS_AND_EVENT` flag. See the fixed windows example project which uses this flag.

20 Painting and Client Areas

There are 3 different types of client area in MiniWin - window frames, window client areas and control client areas. Each area has its own frame of reference, which is the coordinate (0, 0) at the top left corner of the respective area. Painting is different for each client area as described below:

Window frames: Painting of window frames is done by the window manager, not the user. Window frames have some configurable parts and some non-configurable parts. Optional parts are a title bar, a border, a menu bar, a vertical scroll bar and a horizontal scroll bar. If a title bar is specified then on the title bar are a resize control, a title, a minimize icon, a maximize icon and a close window icon. The close window icon is drawn enabled or disabled depending on if the window is created as closeable or not. A message can be sent to the window manager to repaint the window frame. A parameter specifies which components of the window frame to repaint. A window frame's area cannot be repainted in part, only completely.

Window client area: Painting of the window client area is done in a window's paint function programmed by the user. The user sets values in the gl library's graphics context then calls functions in the gl library passing to the gl call the `draw_info` received in the paint function parameter. The client area is painted before the controls belonging to a window so that the controls will always appear on top. In user code a request can be made to paint a client area completely or only a sub-section of the client by calling a different function that takes a rectangle area to repaint as a parameter.

Control client area painting: A control has no frame and the client area covers the whole of the control's area. The painting method is the same as for a window client area except a control cannot contain sub-controls. Like a window client area a request can be made to paint a control completely or only a part.

21 Automatic Painting and User Painting

The MiniWin window manager does minimal automatic repainting of windows, client areas and controls. Much of the repainting is under control of user code. The reason for this is that repainting is expensive computationally and needs to be minimized to achieve a smooth responsive flicker free user interface. This is especially important in a window manager like MiniWin that has no display buffering.

The general rule in MiniWin user code development is that when a repaint is required because of something the user has done then MiniWin will instigate the required repaint. Examples of this are moving, resizing, closing, minimizing and restoring a window. All repaints needed as a result of user code will need to be instigated in user code by calling a MiniWin utility function to post the required repaint message.

There are many different types of repainting request. This is also to minimize computation and flicker. A window frame is requested to repaint according to its features (title bar, border, menu bar, 2 scroll bars). A window client area and a control client area can be requested to be repainted in their entirety or a sub-section defined by a rectangle. Always use the minimum repainting type for the situation. Requesting a repaint all at all times may be easy and the least code to write, but it will not give a pleasing user experience.

22 Scroll Bars

There are two types of scroll bars - window frame and control - and each type comes in horizontal and vertical versions. Each type is described below.

Window frame scroll bars: these scroll bars are part of the window frame. They are always at the bottom edge (for horizontal scroll bars) or right edge (for vertical scroll bars) of the window just inside the border, if the window has one, and are always the height/width of the window client area. As a window is resized these scroll bars will resize and reposition themselves to remain at their respective edge and full window client area width/height. Window scroll bars cannot have their scroll position set programmatically and they are always enabled.

Control scroll bars: these are user interface controls just like any other. They have a location where they are drawn on a window's client rect and that's where they remain. If a window is resized so that they are no longer within the client rect they are no longer visible. A control scroll bar can have its position set programmatically and can be enabled or disabled. It can be used as an input (when enabled) or output (when disabled).

Both types of scroll bars do no scroll anything on their own. Their scroll position can be read as a percentage and nothing else. It is up to the owning window's paint function to read the scroll position and perform the correct drawing of its contents, shifted as appropriate according to a scroll bar's position.

23 Combining Controls

MiniWin contains only a limited set of simple controls to keep its code base small. The user developer can of course add more controls for more complex functionality, but an alternative is to combine controls. Two examples are described here:

Cascading menus: a MiniWin window frame can have a menu bar as part of its window frame but not cascading menus. However, a menu bar item can in its message handler cause a list box to appear immediately below the menu bar item. Each item in this list box could pop up a further list box, giving the impression of cascading menus.

Scrolling list box: a MiniWin list box must be made large enough to display all its items. However, by combining a standard list box with a scroll bar immediately to its right a list box can be made scrolling.

In both the above examples no changes are required to the standard window or control code; all functionality can be implemented in the user code for the containing window. Examples can be found in the example code.

24 User Window Interaction

If a window has a title bar the user can move, resize, minimize, restore, maximize and close windows (if enabled).

Moving a window: a window is moved by dragging the window's title bar in an area away from the icons.

Closing a window: a window is closed by tapping the close icon. If a window is created as non-closeable then the close icon is greyed out.

Maximizing a window: a window is maximized by tapping the maximize icon on the window's title bar. This makes the window the same size as the display.

Minimizing a window: a window can be minimized by tapping the minimize icon on the window's title bar. This reduces the window to an icon at the next location along the bottom of the screen. A closed window icon is a simple grey box with the window's title written across it. Icons cannot be moved on the screen. A user has no choice where an icon is located; MiniWin chooses the location automatically.

Restoring a window: an iconized window can be restored by tapping its icon.

Resizing a window: a window can be resized by dragging the resize arrow icon at the left side of the window's title bar. Only this corner can be used to resize a window. A window's borders cannot be used to resize it as in MiniWin they are too thin to be easy to locate the pointer on.

When a window is being moved or resized a guide box is drawn to show the effect of the operation. The guide box is drawn as a dashed patterned. When the operation is terminated and the move or resize is complete only then is the window repainted completely.

25 MiniWin Example Projects

MiniWin comes with 5 example projects each of which has a variant for the 3 hal layers. All projects share the same `user/MiniWin_config.h` header file for convenience. (For your projects you should use your own customized version). The sample projects are listed below.

25.1 MiniWinTest

This is a comprehensive example using all of MiniWin's features. The MiniWinTest example project creates a variety of windows to test and demonstrate the MiniWin features. Below is a description of the windows created in the example project and what they demonstrate.

window_drag.c
This window demonstrates capturing drag events. It stores the 15 most recent drag points received and draws a line between them all.

window_gl.c
This window demonstrates all the features of the functions found in the gl library. The features are separated out into multiple paint sections each of which is changed after a second by a timer. The test cycles forever.

window_paint_rect.c
This window shows how a window can be partially repainted rather that repainting the whole window every time a section needs repainting.

window_scroll.c
This window demonstrates window frame scroll bars by allowing a text pattern to be scrolled around horizontally and vertically.

window_yield.c
This window shows how a long task (plotting 1000 circles) can be split over multiple window message function calls. Restoring, moving or resizing the window restarts the task which shows how to capture these events.

window_test.c
This window demonstrates all user interface controls (except buttons which is shown when a standard 2 button message box dialog is shown) and the menu bar. It shows how controls with multiple items (menu bar and list box) can have some items enabled and disabled. There are two scroll bars, one for input and one for output with the position of the input one reflected to the output one. It shows how touch events on the client area can be captured. It shows how to simulate cascading menus by

appropriately place a list box and also how a list box and a scroll bar can be combined to simulate a scrolling list box.

25.2 MiniWinSimple

This project contains the example code as described in detail later in this document to demonstrate a minimalistic application with one window, one pop-up standard dialog and 2 controls.

25.3 MiniWinFile

This project connects to a pen drive via USB (STM32F4xx examples) or the Windows file system (Windows example) to show folder listings. This allows directories to be traversed and `.txt` text file and `.bmp` image files to be opened and their contents displayed. Multiple windows (up to a fixed limit) are allowed for each file type.

When the app starts tap the button to open the file system. Folders abd files are shown, each type with their own icon. If there are more file/folder entries than can be fitted in to the list box visible lines you can scroll the list box with the up and down arrows. Tap a folder name to open that folder and tap the left arrow to go up a folder. Tap a `.bmp` or a `.txt` file name to open that file. A new window pops up showing the file's contents. Filenames are shown in the 8-dot-3 format.

25.3.1 File System Integration

Under the `BSP/middleware` folder are 2 sub-projects containing third party libraries – the FatFS code for file system handling and the ST USB Host library with MSP support to link the FatFS module with the USB driver code in the ST HAL.

25.3.2 Multiple Window Instances

This project is an example of multiple window instances of the same window code – for displaying text files and for displaying image files. Multiple text and image windows can be shown at the same time. These windows contain no data in their source code files but instead use external instance data to store window state data.

25.4 MiniWinFixedWindows

This project contains the code for a fixed tiled windows example. The main window displays 6 windows, each as an icon and with no title bar or border. These windows accept only one event: a touch down. Some of the windows then bring up a further full screen window with no border or title bar. The 6 windows on the home screen have the flag set upon their window creation that a touch down event in a window that does not have focus gives the window focus and is also passed on to the window as a touch down event.

25.5 MiniWinFreeRTOS

This project integrates MiniWin into FreeRTOS. In these examples MiniWin runs in a single thread and 2 other threads are instantiated to perform other duties below:

1) Flash a LED. On the ST Discovery boards builds one of the board LED's is used for this purpose. On the Windows build a small red square to the right of the MiniWin root window is shown in the Windows window.

2) Read accelerometer values from the on-board MEMS accelerometer and display angle arrows and text for X, Y and Z, each in their own window, along with an offset setting button. The way the example projects work for the 3 hardware variants differs slightly because of hardware differences.

   The STM32F429 Discovery board has a gryrometer which senses angular accelerations. These can be integrated over time to give angular rotations about X, Y and Z axes. However, there is no code to calibrate the gyroscope output and hence the arrows will drift over time. Pressing the button in each window will reset the reading to zero.

   The STM32F407 Discovery board has a linear accelerometer which senses linear accelerations, including gravity. These can be used to calulate angular rotations about X and Y but not Z axes when the board is not accelerating relative to the earth. The Z axis rotation cannot be calculated and therefore the Z axis arrow does not move.

   On the Windows build there is no accelerometer or gyrometer but mouse is moved up and down across the Windows desktop to simulate X and Y rotations and the keyboard left and right arrows are used to simulate the Z rotation.

The version of FreeRTOS used in this integration example is 10.1. The is a few versions on from the FreeRTOS integration examples found in the STM32 Cube package from ST. The FreeRTOS code was obtained from the FreeRTOS website and the port part changed slightly to work on the ST Discovery boards.

The FreeRTOS examples all use fully static memory allocation by setting the apprpriate values in the configuration files and providing the required hook functions needed for a static memory implementation. All FreeRTOS dynamic memory allocation code is removed when using this configuration.

In order for the MiniWin/FreeRTOS example to run under Windows the FreeRTOS Windows port is used. See the FreeRTOS website for details.

25.6 Example Projects for Microsoft Windows

The example project built to run on Microsoft Windows are not proper well behaved Microsoft Windows application. MiniWin runs more slowly on Microsoft Windows compared to real hardware. Also the contents of the window in Windows do not properly repaint themselves if the window is covered and re-exposed.

The Windows example projects are console applications even though they draws graphics to the console window. To be able to click in these windows and the event to be sent to the MiniWin application, under the command window's properties untick all the Edit Options and Text Selection check boxes.

If you wish to debug the Windows applications build it for debug then when the console window appears (after breaking in main) move it to a part of your screen where it will not be covered by any other window, not even your IDE window,(as once it is covered, when it is re-exposed it is, not repainted and you will end up with a black window). Then hit go in your IDE to start debugging.

25.7 Eclipse Linked Folders

None of the example projects contain the MiniWin source code within them. They all use the Eclipse link feature to link to the MiniWin shared source folder. The C source files for the hal implementation not used by a particular example are excluded from the build, as is the empty sample `user/mini_user.c`, instead including or linking to an implemented version.

Similarly, the common application code for each application appears only once even though there are 3 builds for each application (for the 3 hal driver implementations). The application code appears under the common folder under each project parent directory. This folder is linked to in the project files using the Eclipse linked folder feature.

25.8 Other Required Project Files

All projects have a `src` folder which contains files particular to that target build of that project. All projects have an `app.c/h` pair of files. In these files it is needed to declare/define the following 2 functions:

```
void app_init(void);
```

This function is called by the `main` function in `main.c` (in the `common` folder). In this function's implementation perform application initializations that are not part of MiniWin, for example setting the system clock for the embedded builds.

```
void app_main_loop_process(void);
```

This function is called repeatedly from main in an endless loop and is used to implement main loop processing required by your particular embedded application.

In the file example file handling functions are implemented in `app.c` in order to separate out file handling code from the common application code because file handling is code is target platform dependent.

25.9 Board Support Package Code

The embedded examples have a folder called BSP. This folder contains all the board support code required for start-up, drivers from the ST HAL and middleware for file handling and USB hosting for the file example project. The Windows example projects do not need this code as the services required are supplied by the operating system.

26 Standard Controls Details

26.1 Button

Initialisation: User needs to set the label text.
Resources required: A timer for the short period between a button being drawn as down and redrawn as up. This timer is then released.
Sizes: Small and large

Messages processed: `MW_CONTROL_CREATED_MESSAGE`, `MW_WINDOW_TIMER_MESSAGE` to redraw a pressed button, `MW_TOUCH_DOWN_MESSAGE` when a button is pressed.

Messages sent: `MW_BUTTON_PRESSED_MESSAGE` when the button is pressed. No data is returned in the button pressed message.

26.2 Check Box

Initialisation: User needs to set the label text. The check box state is automatically set to false on creation.
Resources required: None
Sizes: Small and large

Messages processed: `MW_CONTROL_CREATED_MESSAGE`, `MW_TOUCH_DOWN_MESSAGE`, `MW_TRANSFER_DATA_1_MESSAGE`. Data contains the state of the check box, 0 or 1.

Messages sent: `MW_CHECKBOX_STATE_CHANGE_MESSAGE` when check box state changes. Data contains 1 for checked or 0 for unchecked.

26.3 Integer Number Chooser

Initialisation: User needs to set the is_only_positive flag to allow or disallow negative numbers. The number chooser's content is automatically set to zero and non-negative on creation.
Resources required: A timer is required for as long as the containing window has focus to animate the flashing cursor.
Sizes: Small only

Messages processed: `MW_CONTROL_CREATED_MESSAGE`, `MW_CONTROL_GAINED_FOCUS_MESSAGE` starts cursor animation, `MW_CONTROL_LOST_FOCUS_MESSAGE` halts cursor animation, `MW_WINDOW_TIMER_MESSAGE`, `MW_TOUCH_DOWN_MESSAGE`, `MW_TRANSFER_DATA_1_MESSAGE` to receive a new number to display in the number chooser. Data contains the new number.

Messages sent: `MW_INT_NUMBER_CHOOSER_CANCEL_MESSAGE` if the user presses the cancel button. `MW_INT_NUMBER_CHOOSER_OK_MESSAGE` if the user presses the ok button. Data contains the entered number.

26.4 Keyboard

Initialisation: None. The buffer is automatically cleared on creation and the keyboard set to upper case letters.
Resources required: A timer is required for as long as the containing window has focus to animate the flashing cursor.
Sizes: Small only

Messages processed: `MW_CONTROL_CREATED_MESSAGE`, `MW_CONTROL_GAINED_FOCUS_MESSAGE` starts cursor animation, `MW_CONTROL_LOST_FOCUS_MESSAGE` halts cursor animation, `MW_WINDOW_TIMER_MESSAGE`, `MW_TOUCH_DOWN_MESSAGE`, `MW_TRANSFER_DATA_1_PTR_MESSAGE` to receive new text to display in the keyboard. Data contains a pointer to the new text which is copied into keyboard storage.

Messages sent: `MW_KEYBOARD_CANCEL_MESSAGE` if the user presses the cancel button. `MW_KEYBOARD_OK_MESSAGE` if the user presses the ok button. Data contains a pointer to the entered text.

26.5 Label

Initialisation: User needs to set the label text.
Resources required: None
Sizes: Small and large

Messages processed: `MW_TRANSFER_DATA_1_PTR_MESSAGE`. Data contains a pointer to the new label text that is copied into the control's memory.
Messages sent: None.

26.6 List Box

Initialisation: User needs to set the number of lines the scroll box has, the array of list box entries and the bit field identifying which items in the list box are enabled or disabled. A list box entry is a structure containing a pointer to a string and a pointer to an icon data structure. The icon is a monochrome 8x8 pixel for a small list box or a 16x16 pixel for a large list box. If no icon is needed for a list box line then the icon pointer should be `NULL`.
Resources required: A timer for the short period between a list box item being drawn as down and redrawn as up. This timer is then released.
Sizes: Small and large

Messages processed: `MW_CONTROL_CREATED_MESSAGE`, `MW_WINDOW_TIMER_MESSAGE`, `MW_TOUCH_DOWN_MESSAGE`.

Messages sent: `MW_LIST_BOX_ITEM_PRESSED_MESSAGE`. Data contains the number of the list box item pressed, starting at 0.

26.7 Progress Bar

Initialisation: User needs to set the progress percentage.
Resources required: None.
Sizes: User defined on creation

Messages processed: `MW_TRANSFER_DATA_1_MESSAGE`. Data contains the progress as a percentage.

Messages sent: None.

26.8 Radio Button

Initialisation: User needs to set the number of buttons and labels array. The chosen button is automatically set to zero on creation.
Resources required: None
Sizes: Small and large

Messages processed: `MW_CONTROL_CREATED_MESSAGE`, `MW_TOUCH_DOWN_MESSAGE`, `MW_TRANSFER_DATA_1_MESSAGE`. Data contains the number of the selected radio button starting from 0.
Messages sent: `MW_RADIO_BUTTON_ITEM_SELECTED_MESSAGE`. Data contains the number of the selected radio button starting from 0.

26.9 Horizontal Scroll Bar

Initialisation: None. The scroll position is automatically set to 0 on creation.
Resources required: None.
Sizes: Small and large

Messages processed: `MW_CONTROL_CREATED_MESSAGE`, `MW_TOUCH_DOWN_MESSAGE`, `MW_TOUCH_DRAG_MESSAGE`, `MW_TRANSFER_DATA_1_MESSAGE`. Data contains the scroll position scaled to be a range of 0 - 255.

Messages sent: `MW_CONTROL_HORIZ_SCROLL_BAR_SCROLLED_MESSAGE`. Data contains the scroll position scaled from 0 to 255.

26.10 Vertical Scroll Bar

Initialisation: None. The scroll position is automatically set to 0 on creation.
Resources required: None.
Sizes: Small and large

Messages processed: `MW_CONTROL_CREATED_MESSAGE`, `MW_TOUCH_DOWN_MESSAGE`, `MW_TOUCH_DRAG_MESSAGE`, `MW_TRANSFER_DATA_1_MESSAGE`. Data contains the scroll position scaled to be a range of 0 - 255.

Messages sent: `MW_CONTROL_VERT_SCROLL_BAR_SCROLLED_MESSAGE`. Data contains the scroll position scaled from 0 to 255.

26.11 Arrow Button

Initialisation: User needs to set the arrow direction.
Resources required: A timer for the short period between an arrow button being drawn as down and redrawn as up. This timer is then released.
Sizes: Small and large

Messages processed: `MW_CONTROL_CREATED_MESSAGE`, `MW_WINDOW_TIMER_MESSAGE` to redraw a pressed button, `MW_TOUCH_DOWN_MESSAGE` when an arrow button is pressed, `MW_TOUCH_HOLD_DOWN_MESSAGE` or `MW_TOUCH_DRAG_MESSAGE` when an arrow button is held down.

Messages sent: `MW_ARROW_PRESSED_MESSAGE` when the arrow button is pressed. The data returned in the arrow button pressed message is the arrow button's direction.

27 Source Code Layout

The source code tree is quite complicated so is explained here:

EasyBMP: This is a third party Windows `.bmp` library source. It is used by the utility applications, not MiniWin.

BMPConv32Colour: This is a command line utility for converting a Windows 24 bit per pixel `.bmp` file to a 3 bytes per per pixel C source file. It uses EasyBMP.

BMPConvMono: This is a command line utility for converting a Windows 2 bit per pixel `.bmp` file to an 8 pixels per byte C source file. It uses EasyBMP.

MiniWin: This contains the MiniWin embedded window manager source code. It contains the following sub-folders:

`docs`	All documentation
`bitmaps`	Bitmaps and their C file encodings used by MiniWin
`gl`	The graphics library incorporated in MiniWin
`hal`	The hardware abstraction layer of drivers that need modification for the user's particular hardware
`ui`	Standard user interface controls. Users can add to this for further controls if required
`user`	Example code of sample windows
`dialogs`	Standard dialogs provided by MiniWin

Folder **`hal`** contains source files common to all hal implementations and then further source files in sub-folder for different hardware choices.

In addition the following source files are in the `src/miniwn` folder:

`calibrate.h/c`	Third party touch screen calibration routines
`miniwin_debug.h/c`	Implementation of assert functionality, debug build only
`miniwin_message_queue.h/c`	Simple message queue code for MiniWin messages
`miniwin_settings.h/c`	Simple non-volatile storage routines for settings
`miniwin_touch.h/c`	Interface between touch driver code under hal and the touch client code in MiniWin
`miniwin_utilities.h/c`	Generic utility routines
`miniwin.h/c`	The main window manager code

MiniWinTest: This contains code and project files for the comprehensive MiniWinTest example. It has 3 sub-projects, one for each hal layer. The application code is under the common folder and is used by all the sub-projects. Under the STM32F407, STM32F428 and Windows folders are project files and source code specific to different hal builds. Note that the actual hal driver source files are under the `MiniWin/hal/`*`<target>`* folder and in the projects in TrueStudio the driver source files for the non-used targets are excluded from the respective build.

MiniWinSimple: This contains the sample code described later in this document. The application code and sub-project layout is as for the previous example.

MiniWinFixedWindows: As above but for the fixed windows example project. The application code and sub-project layout is as for the previous example.

MiniWinFile: As above but for the file example project. The application code and sub-project layout is as for the previous example.

28 User Configuration

MiniWin can be configured by changing default settings found in header file `src/MiniWin/user/MiniWin_config.h`. There are 4 sections to this file:

28.1 Memory Allocation

All data structures in MiniWin are statically allocated at compile time. This section configures how many items to allocate memory for. Use this section to define the maximum number of windows, controls, message queue entries and timers. The memory used by a window or control can be reused if the item is no longer required.

Timers can be reused after they expire. Timers are used by animated controls (for example a button press/release animation, but only for a very short timer) and flashing cursors (required for as long as the owning window has focus).

Dialogs need one window slot and one to many control slots when they are showing, but these resources are released when the dialog is dismissed.

The number of messages required depends on the complexity of the system being developed and needs experimentation to find out the required size.

In debug mode any occurrence of a resource running out will trigger an assert failure enabling instant discovery of what went wrong.

28.2 Sizes

Set the display size here and the window title bar size. The title bar must be big enough to contain the title text and icons. If the title bar size is changed from default then the icons and text may need to be redesigned.

28.3 User Interface Colours

Customize the look of your windows here. The colours are defined in the LCD hal layer.

28.4 Timings

Customize the feel of your touch screen response here to prevent touch bounce, and the look of animated controls and window minimization/restoration. The MiniWin system timer is also defined here. Changing the system timer value will alter the effect of all other timings which are defined in units of system ticks.

28.5 Other

The drag threshold prevents slight movement on touch screen taps sending drag events. The busy text is displayed when the user interface is locked because of a long processing task.

29 MiniWin Asserts

The MiniWin window manager, supporting code and example projects all use the MiniWin assert macro. Under debug build when `NDEBUG` is defined a failed assert will show a blue screen of death with details of the function and line number where the assert failed along with a simple text message. On release builds the assert macro compiles to nothing.

In debug builds if you run out of resources (message queue space, window or control slots and timers) or something bad is attempted via the public API (i.e. removing a window with a bad window id) you will get an assert with an explanation making it easy to solve. On release builds these failures that can be ignored by the window manager will be, and will not cause an assert, allowing the window manager to continue.

If any asserts failures are seen that are not caused by running out of resources, or bad user code calling the public API, please report the problem via the website.

30 Quick Start Guide

This section describes how to create a single window with 2 standard controls: a button and a label. Pressing the button brings up a one button message box. Dismissing the dialog box changes the text of the label. Touching the window's client area outside the controls draws a circle using gl on the client area. Before you can implement this example you need to implement your hal functions as described earlier for your board.

Once you have implemented you hal functions you need to create a new project and copy in or point to the folder of the MiniWin source. Add the path to the `MiniWin` folder to your list of include folders to search in your IDE (or add it to the command line list if you are building on the command line).

1. Create a new header file for your window called `window_simple.h`. Add these 2 function prototypes to your header file:

```
void window_simple_paint_function(uint8_t window_ref,
                                  const mw_gl_draw_info_t *draw_info);

void window_simple_message_function(const mw_message_t *message);
```

2. Create a new source file for your window called `window_simple.c`. Add this data structure declaration and definition to your source file. This window is designed not to have multiple instances simultaneously showing so therefore the window's data is stored in the window's source file.

```
#include "MiniWin.h"
#include "ui/ui_common.h"
#include "dialogs/dialog_common.h"

typedef struct
{
      uint16_t circle_x;            // x coordinate of where to draw circle
      uint16_t circle_y;            // y coordinate of where to draw circle
      bool draw_circle;             // if to draw circle
} window_simple_data_t;

static window_simple_data_t window_simple_data;
```

```
extern uint8_t button_id;
extern uint8_t label_id;
```

3. Add stub functions to window_simple.c

```
void window_simple_paint_function(uint8_t window_ref,
                                  const mw_gl_draw_info_t *draw_info)
{
}

void window_simple_message_function(const mw_message_t *message)
{
}
```

4. Copy `miniwin_user.c` example file from the `MiniWin/user` folder to your source folder. Exclude the original from the build or delete it because you have your own copy now. Add your new header file to the lists of includes and the following variable declarations under the global variables section:

```
#include "window_simple.h"
#include "ui/ui_common.h"

/***********************
*** GLOBAL VARIABLES ***
***********************/

uint8_t window_simple_id;
uint8_t button_id;
uint8_t label_id;
mw_ui_label_data_t label_data;
mw_ui_button_data_t button_data;
```

5. Add the following root paint function to miniwin_user.c which draws the root window with solid purple:

```
void mw_user_root_paint_function(const mw_gl_draw_info_t *draw_info)
{
	mw_gl_set_solid_fill_colour(MW_HAL_LCD_PURPLE);
	mw_gl_clear_pattern();
	mw_gl_set_border(MW_GL_BORDER_OFF);
	mw_gl_set_fill(MW_GL_FILL);
	mw_gl_rectangle(draw_info, 0, 0, MW_ROOT_WIDTH, MW_ROOT_HEIGHT);
}
```

In all gl functions that take a `mw_gl_draw_info_t` parameter just pass on to the gl function the parameter that you have received.

6. In function `mw_user_init()` in `miniwin_user.c` add the following code. This creates the window and controls and adds the controls to the window.

```
void mw_user_init(void)
{
	mw_util_rect_t r;
```

```
	mw_util_set_rect(&r, 15, 100, 220, 210);
	window_simple_id = mw_add_window(&r,
			"SIMPLE",
			window_simple_paint_function,
			window_simple_message_function,
			NULL,
			0,
			MW_WINDOW_FLAG_HAS_BORDER | MW_WINDOW_FLAG_HAS_TITLE_BAR |
				MW_WINDOW_FLAG_CAN_BE_CLOSED |
				MW_WINDOW_FLAG_IS_VISIBLE,
				NULL);

	mw_util_safe_strcpy(label_data.label,
			MW_UI_LABEL_MAX_CHARS, "Not yet set");
	label_id = mw_ui_label_add_new(100,
			5,
			2 * MW_UI_LABEL_X_OFFSET + 11 *
				MW_GL_STANDARD_CHARACTER_WIDTH,
			window_simple_id,
			MW_CONTROL_FLAG_IS_VISIBLE | MW_CONTROL_FLAG_IS_ENABLED,
			&label_data);

	mw_util_safe_strcpy(button_data.button_label,
			MW_UI_BUTTON_LABEL_MAX_CHARS, "TEST");
	button_id = mw_ui_button_add_new(10,
			10,
			window_simple_id,
			MW_CONTROL_FLAG_IS_VISIBLE | MW_CONTROL_FLAG_IS_ENABLED,
			&button_data);

	mw_paint_all();
}
```

7. In `main.c` add code like this:

```
#include "MiniWin.h"
#include "app.h"

int main(void)
{
	app_init();
	mw_init();

	while(true)
	{
		app_main_loop_process();
		mw_process_message();
	}
}
```

8. Create a new header file called `app.h` and add these function prototypes:

```
void app_init(void);
void app_main_loop_process(void);
```

Create a new source file called `app.c` and add these two functions:

```
void app_init(void)
{
}

void app_main_loop_process(void)
{
}
```

Function `app_init()` is for any initialisations that need to be performed for your particular hardware, for example setting the clock. Function `app_main_loop_process()` is for any other code your embedded application needs to call to implement its behaviour. In this example it can be empty.

This is a good time to build and test your application. You should see a single window appear with an empty client area other than the 2 controls. You will be able to move, minimize, maximize, resize, restore and close this window. The window's client area won't be repainted at this stage.

The next stage is to implement the message handling and paint functionality in your window.

9. Give your window's client area a background, as by default it has none. You need to draw a filled rectangle with no border like this, in function `window_simple_paint_function()`. As for the root window paint function just pass on the `mw_gl_draw_info_t` parameter to any gl function that needs it:

```
mw_gl_set_fill(MW_GL_FILL);
mw_gl_set_solid_fill_colour(MW_HAL_LCD_WHITE);
mw_gl_set_border(MW_GL_BORDER_OFF);
mw_gl_clear_pattern();
mw_gl_rectangle(draw_info,
            0,
            0,
            mw_all_windows[window_ref].client_rect.width,
            mw_all_windows[window_ref].client_rect.height);
```

While you're here add the code to draw the circle at the touched point in the same function after the background drawing code above, although at the moment you are not handling the touch message in your message handler. That's next.

```
mw_gl_set_fg_colour(MW_HAL_LCD_BLACK);
if(window_simple_data.draw_circle)
{
      mw_gl_set_solid_fill_colour(MW_HAL_LCD_YELLOW);
      mw_gl_set_line(MW_GL_SOLID_LINE);
      mw_gl_set_border(MW_GL_BORDER_ON);
      mw_gl_circle(draw_info, window_simple_data.circle_x,
            window_simple_data.circle_y, 25);
}
```

10. Now start adding some message handling in your message handler functtion `window_simple_message_function()`. First of all create a switch statement. You want to handle

the `MW_WINDOW_CREATED_MESSAGE`. It's called once when a window is created, and is a good place for initializations:

```
switch (message->message_id)
{
case MW_WINDOW_CREATED_MESSAGE:
	window_simple_data.draw_circle = false;
	break;

default:
	break;
}
```

11. Add code to handle a touch down event and store the touch point. The touch point comes in the message's data parameter, x coordinate in the left 2 bytes, y in the right 2 bytes:

```
case MW_TOUCH_DOWN_MESSAGE:
	window_simple_data.circle_x = message->message_data >> 16;
	window_simple_data.circle_y = message->message_data;
	window_simple_data.draw_circle = true;
	mw_paint_window_client(message->recipient_id);
	break;
```

As the window's client area has been touched we want to redraw it to draw the circle. Don't call your paint routine directly, call the utility function that posts a repaint message to the message queue.

12. Now add code to respond to the button press message. When this message is received you want to pop up a single button dialog with customised text, as done below. You can check that the sender of the button pressed message is our button, although we only have one button in this example application, so it's not strictly necessary as it couldn't be any other button.

```
case MW_BUTTON_PRESSED_MESSAGE:
	if (message->sender_id == button_id)
	{
		mw_create_window_dialog_one_button(20,
			50,
			150,
			"Title",
			"This is a message",
			"Yep",
			false,
			message->recipient_id);
	}
	break;
```

This function call to create the pop up dialog is non-blocking, i.e. it returns straightaway. The specified window (in this case your only window) gets a message when it's dismissed, so catch that and use it to change the label's text on your window. You need to get this label repainted after changing its text so call `mw_paint_control(label_id)`.

```
case MW_DIALOG_ONE_BUTTON_DISMISSED_MESSAGE:
	mw_ui_common_post_pointer_to_control(label_id, "Hello world!");
```

```
mw_paint_control(label_id);
break;
```

13. That’s it. Build and run. You’ll find this example’s source code under the `MiniWinSimple/common` and hardware target folders.

31 Third Party Software

MiniWin uses the following open source third party software:

Touch screen calibration, copyright Carlos E. Vidales 2001

EasyBMP version 1.06 , Paul Macklin 2006

ST HAL driver libraries and BSP.

FatFS version R0.12c, copyright ChaN 2017

FreeRTOS version 10.1 published by Amazon

32 Glossary of Terms

User - the person using the application you develop by interacting with the windows via the touch screen.

HAL/hal – hardware abstraction layer. HAL in capitals refers to the set of drivers produced by ST for STM32 ARM processors; hal in lower case refers to the MiniWin interface to the hardware on the board it is running on and for the STM32 examples uses the ST HAL.

BSP – Board Support Package. All the code required to support the system on the embedded examples. Contains drivers, start-up code and middleware.

User Code - code you the developer writes as opposed to code that is supplied as part of the MiniWin window manager.

Window - rectangular area on the screen comprising a window frame, the client area within the window frame and controls within the client area.

Window Frame - The parts of the window outside of the client area comprising title bar, border, menu bar and scroll bars. All parts are optional.

Icons - Small bitmaps drawn on a window's title bar for window control purposes. Also a minimized window shown as a rectangle at the bottom of the root window.

Root Window - the background behind all the windows. Also called the desktop.

Z Order - the position of a window relative to all others. A high Z order window is drawn on top of a lower one. The root window always has Z order 0.

Client Area - The area inside the window frame that the user code draws on and which receives touch down, up and drag events. All controls in a window are limited to the client area. Also a control has a client area but no frame, so it is the same as the limits of the control.

Clipping - ignoring a pixel location that is requested to be drawn if it falls outside the client area.
Dialog – a standard window that is part of the MiniWin window manager for simple common user interaction, for example a message box with a dismiss button.

Modal – an optional property of a window that means that while it is showing no other window can receive user input. All MiniWin standard dialogs are modal.