MiniWin Embedded Window Manager
2 December 2018

1 Introduction

MiniWin is a generic open source overlapped window manager for small embedded systems with a touch screen. MiniWin is written in C 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.
- Six bitmapped fonts included – five fixed width and one proportional.
- Additional TrueType font rendering capability for fonts of your choice
- A clean easy to use API.
- Comprehensive documentation and example projects.
- Runs in bare metal systems or within a single thread of a RTOS.
- Example projects showing FatFS, USB host, FreeRTOS integration, and TrueType font rendering
- 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
- Arrows buttons in 4 directions
- Numeric keypad
- Keyboard supporting all ASCII characters
- Text box for rendering justified text using any TrueType font

These are the standard ones. All controls except the keyboard come in 2 sizes – standard and large. 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, resize 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

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 the following utility applications:

- Convert monochrome bitmap files in .bmp format to C99 source code.
- Convert 24 bit colour bitmap files in .bmp format to C99 source code.
- Convert TrueType font files to run-length encoded C99 source file for a single font point size

The C99 source code produced by the utilities can be dropped into your project and used straight away by routines in the graphics library. Each utility processes a single input file at a time, so must be run for each input file processed. You can create a script to do this or add it into your TrueStudio project/makefile as a pre-build step to automate the conversions.

Each utility is located in MiniWin under its own folder where source code and project files are found (and makefiles for Linux). These utilities can be built from source and run on either Windows or Linux (the font converter for Windows is currently provided as an executable only).

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

7 Drivers and Examples

MiniWin comes with all source code for the window manager, graphics libraries, sample drivers, user interface components and dialogs. It also comes with 6 example projects - a simple getting started example, a fully comprehensive project showing usage of all MiniWin's user interface controls, an example that integrates the FatFS file system and a USB host driver for a pen drive, an example that integrates MiniWin into FreeRTOS real-time operating system, a non-overlapped tiled windows example and a TrueType font justified-text rendering demonstration.

MiniWin comes with four 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 two simulated versions that allows the example applications to run within a Microsoft Windows window or a Linux X11 window. Each of the 6 example projects can be built for all four of the hal drivers. User interface development using the Windows or Linux simulation speeds up development time considerably.

All sample projects are built with the free compiler/IDE Atollic TrueStudio, currently version 9.0.1, which can be obtained for Microsoft Windows or Linux running on x-86. The Windows examples will build using the Windows version of Atollic TrueStudio only and similarly the Linux examples will build using the Linux version only.

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 skin 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. It is only necessary to implement this function if your user code is going need it. The MiniWin window manager, dialogs and user interface components do not require it. The example projects, however, do need it.

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 determines 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, or 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 a window’s message handler whenever it 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 to hold its state can be within the window source file as single instance static data, and the code within the window’s message handler and paint function 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 window’s message handler and paint functions. Therefore each window has associated with it a void pointer which is used to store instance data. 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 (gl) 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 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 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
- which bitmap font to use (not TrueType font)
- text rotation

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.

13.1 Fonts

MiniWin comes with a viariety of font capabilities as part of gl. Some of these are always available and others are optional which can be included or excluded by user configuration, depending on requiresments and available program memory on the target device. The font types fall into the following categories:

- Always available bit-mapped
- Optional bit-mapped
- Run-length encoded TrueType with anti-aliasing
- Run-length encoded TrueType without anti-aliasing

These are described in the sections below.

13.1.1 Always available bit-mapped

The 9 pixel high fixed width and the 15 pixel high proportional fonts are required by the MiniWin window manager and its components (dialogs and user interface controls), but can also be used from user code for simple text rendering. These are included in every MiniWin project by default and are always available. Text rendered in these fonts is always left justified. Text using these fonts can be drawn rotated up, right, down and left and transparently on the background. In transparent mode the pixels of each letter are drawn in the foreground colour but the pixels in a character's block that do not make up part of the character are not drawn leaving the background intact. In non-transparent mode a character's block is drawn first as a solid colour block in background colour then the character's pixels are drawn on top. These fonts are the fastest to render. The fonts contain all characters from the ASCII chracters set.

Before calling a bitmapped font fendering function in gl the font style, forground colour, background colour, transparency and text rotation must be set in the graphic's context. The function call to render the text is the following:

```
/**
 * Draw a horizontal string of small fixed width horizontal characters.
Foreground
 * colour, background colour and transparency controlled by gc.
 *
 * @param draw_info Reference frame origin coordinates and clip region rect
 * @param x Coordinate of the left edge of the rectangle containing the first
 *          character
 * @param y Coordinate of the top edge of the rectangle containing the first
 *          character
 * @param s Pointer to the null terminated string containing ASCII characters
 */
void mw_gl_string(const mw_gl_draw_info_t *draw_info, int16_t x, int16_t y,
const char *s);
```

All example projects use these fonts.

13.1.2 Optional bit-mapped

The other bit-mapped fonts are 12, 16, 20 or 24 pixel high and fixed width. These are optional and can be removed from the build if not used to save space. There are 4 `#defines` in miniwin_config.h for controlling the inclusion of these optional fonts. Comment out the lines to prevent the unused fonts being included in your build. Attempting to use an unavailable font will result in a run-time assert which will indicate the problem.

Like the always available bit-mapped fonts these can be rendered rotated and transparently if required, contain all the ASCII characters, and are always rendered left justified. The function call is the same as given in the previous section.

The example projects MiniWinTest uses these fonts.

13.1.3 Run Length Encoded TrueType Fonts

MiniWin has the capability to render any TrueType font using either monochrome pixel drawing or alternatively using anti-aliasing with the border pixels alpha-blended with the character's background colour. This method of font rendering gives a wide range of options of font size and typeface compared to the bitmapped fonts but at the expense of slower rendering and extra work by the programmer. Text rendered using TrueType fonts can be justified left, right, centre or full. However, this type of font rendering currently allows no rotation or transparent plotting over an existing background. The text is rendered on a solid colour background that is drawn in the font rendering routine as the text is rendered. The example projects MiniWinTTFonts uses these fonts.

TrueType font files are often large and real-time rendering of the font glyphs from the TrueType font data to alpha blended pixels as they are required takes significant processing power and program code space – too much for a small embedded device. Therefore in order to use TrueType fonts the font file is pre-processed before compiling to convert the data into C99 data structures containing run-length encoded font data. This increases rendering speed and reduces program space with the downside that the font sizes available must be chosen before compiling, and every different font size takes extra space.

The process of including TrueType text is described in the following sections.

13.1.3.1Obtaining your TrueType Font

There are many TrueType fonts (.ttf file) available on the internet. Those that come with Microsoft Windows are copyright and cannot be used in a commercial device. There are however many available for free without license restrictions.

13.1.3.2Building the Font Processing Tool

For Linux users obtain the `libfreetype6-dev` package. On Ubuntu the command is:

```
apt-get install libfreetype6-dev
```

Go to the MiniWin folder `FontEncoder_Linux` and type `make`. There is also a TrueStudio project to build this tool but it calls the same `makefile`.

For Windows users the font conversion tool is supplied pre-built under the `FontEncoder_Windows` folder under the MiniWin root folder, named `mcufont.exe`.

13.1.3.3Processing You TrueType Font

For this task you use the font processing tool `mcufont` which you built in the previous section. This tool is a command line application, so start a Linux shell or a Windows command prompt. It is easiest if the font's `.ttf` file you are processing is in the same folder as the application. You process one font file at one point size at a time.

Processing a TrueType `.ttf` file to a `.c` file that can be included in your embedded application is a four stage process, all done via different commands to the same font processing tool. At any time you can see the required command line by typing `mcufont` with no parameters.

The first command is to import the `.ttf` file which converts it to a `.dat` file. A typical command is…

```
mcufont import_ttf DejaVuSans.ttf 12
```

The 12 value is the point size the final font will be rendered in. This produces a font with anti-aliasing. To produce a font without anti-aliasing (which renders more quickly and saves program memory space) use…

```
mcufont import_ttf DejaVuSans.ttf 12 bw
```

The next command is to choose which characters from the font you wish to use in your application, as some TrueType fonts contain thousands of characters (currently MiniWin only supports single byte characters, so this range's endpoint should not be greater than 255). A numerical range (or ranges) is given to choose these values. To choose only ASCII characters for example use 0 – 128 range…

```
mcufont filter DejaVuSans12bw.dat 0-128
```

The next command is to optimize your converted font's datafile to the minimum size. This can take a few minutes. An example command is…

```
mcufont rlefont_optimize DejaVuSans12bw.dat
```

Finally export your font data to a run-length encoded C source file…

```
mcufont rlefont DejaVuSans12bw.dat
```

This produces a file called `DejaVuSans12.c` which can be dropped into your project's source folder and added to the project.

13.1.3.4Using your Font

If you look in your font file you have created near the bottom you will find a data structure declared of type `mf_rlefont_s`. For example, in the `DejaVuSans12.c` file it is called `DejaVuSans12`. You need to have an extern declaration to this data structure in the source file where you are calling the font rendering function, for example…

```
      extern const struct mf_rlefont_s mf_rlefont_DejaVuSans12;
```

This extern reference is how you specify the font you wish to use to the text rendering function. You can have multiple fonts and point sizes available in your application if you wish.

Before calling the text rendering function in gl you need to set the foreground and background colour in the graphics context. The rotation and transparency values do not need to be set as these are not available for TrueType font rendering in MiniWin. Unlike bitmapped font rendering TrueType fonts are rendered across multiple lines in a box. You therefore specify a containing rectangle rather than just a start position. The function call in gl as as follows:

```
/**
 * Render justified true type text in a box
 *
 * @param draw_info Reference frame origin coordinates and clip region rect
 * @param text_rect The rect in the window's client area that the text is to
                    be rendered into
 * @param justification The justification to use when rendering the text
 * @param rle_font The true type font to use
 * @param tt_text The text to render
 * @param vert_scroll How many pixel lines to scroll the text up
 */
void mw_gl_tt_render_text(const mw_gl_draw_info_t *draw_info,
            mw_util_rect_t *text_rect,
            mw_gl_tt_font_justification_t justification,
            const struct mf_rlefont_s *rle_font,
            const char *tt_text,
            uint16_t vert_scroll_pixels);
```

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.

All but one 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 is the keyboard control which comes only in the small size because of size limitations on the LCD display typically used for MiniWin projects. Progress bars and text boxes are created with a user specified size and there is no small or large concept for these control types.

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 each instance of a particular control in use. 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 in 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

Most controls and control like features of a window frame (menu bar, vertical and horizontal scroll bars) can be enabled and disabled. When a control is disabled it accepts no input and is drawn greyed out. For list boxes and the menu bar, 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 list boxes or menu bars 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.

Enabling and disabling text boxes and progress bars has no effect.

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 or closed except by pressing one of the controls, typically an OK and/or Cancel button. Data can be sent to any window from a dialog when the dialog is dismissed. The following sections describe the dialogs that exist within MiniWin.

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 Number Entry

This dialog allows the user to enter an integer number using an on-screen numeric keypad. In user code the initial number to show when the dialog appears is set and a flag indicating if negative numbers are allowed. If negative numbers are disallowed the – sign on the keypad is disabled. When the dialog is dismissed with the ok button a message is sent to a specified window with a string of the characters entered. It is up to the recipient to interpret this string.

16.5 Text Entry

This dialog allows the user to enter a text string of any ASCII characters up to a maximum length of 20 characters using an on-screen keyboard. In user code the initial text to show when the dialog appears is set. When the dialog is dismissed with the ok button a message is sent to a specified window with the entered text.

16.6 Date Chooser

This dialog allows the user to set a date in year (4 digit), month (1-12) and date (1-31). 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.

16.7 File Chooser

This dialog allows the user to select a file or a folder. Whether a file or folder is to be chosen is set in user code with a flag. The dialog shows the contents of the starting folder, either files and folders or just folders depending on the option chosen. The folder contents are shown in a list box control with icons representing files and folders alongside each name. If the user chooses a folder that folder is opened. A back arrow button allows the user to go back a folder level. This dialog is optional and is only built if `#define MW_FILE_CHOOSER` is defined in the project's `mw_config.h` header file. If this dialog is used then the following functions must be declared and implemented in your `app.h/c`:

```
uint8_t find_folder_entries(char *path,
             mw_ui_list_box_entry *list_box_settings_entries,
             bool folders_only,
             uint8_t max_entries,
             const uint8_t *file_entry_icon,
             const uint8_t *folder_entry_icon);

char *app_get_root_folder_path(void);
```

See `app.h/c` in the file handling example applications for details.

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.

Messages are processed by the window manager in the order they were posted. For example, if in user code a message is posted to change the text in a label and then the paint control function is called (which is just a utility to post a paint control message), the messages will be processed in that order and the text in the label will be changed before the label is repainted. The processing of the messages will happen asynchronously, i.e. at a later time to the post message function calls. A consequence of this is that debugging is sometimes not straightforward. In the example given the actual painting of the control will not happen within the paint control function call. The solution to this problem is to place a breakpoint on the code where the message is handled.

The message used by MiniWin are defined in `miniwin.h` and detailed in Appendix 1 in this document. They fall into 4 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 handler will post a message letting user code know what has happened. These messages are not posted from user code.

17.3 Messages posted to controls to change their state or appearance

These messages are posted from user code to controls to set their state or change their appearance. For example, a label can have its text changed from user code, or a progress bar have its progress state set.

17.4 Messages posted by standard dialogs

These messages are posted from from standard dialogs to user code. They normally indicate that the user has finished interacting with the dialog and has closed it. Some messages contain data of the user's choice in the dialog.

17.5 Utility messages

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 6 fields. These are described below:

Sender id: This is the window id or control id of the message poster 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. Sometimes this value is used as 2 16 bit integers, 4 bytes or a single precision float. It is up to the recipient code to do the appropriate shifting, masking and casting if required.

Message pointer: This is a void* value sent from the sender to the recipient. It can be a pointer to anything and cast appropriately by the message recipient. The memory pointed to must exist for the lifetime of the message, which means static or global data. Sending a message with a pointer to a local variable which has gone out of scope by the time the recipient receives the message will not end well.

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 both the message data/pointer field are 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`, but this is simply defined as zero.

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, 4 bytes or a single precision float. It should not be used to pass a pointer as this field is fixed at 32 bits and the pointer size on your system may be different.

As no dynamic memory allocations are used in MiniWin, when the pointer field in a message is used, the object pointed to must not be a local variable that goes out of scope after the message is posted - it must be either static if it's a local function scope variable, 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 5 general purpose user message identifiers defined in `miniwin.h` which user code can use for any purpose.

19 MiniWin Memory Pools and Handles

When MiniWin resources (windows, controls and timers) are created they are stored internally in static memory pools. No dynamic data is used. When a resource is no longer used and the MiniWin window manager is informed of this by deleting a window or control or cancelling a timer (or it expiring) the area of the memory pool is returned for later re-use. This all happens within the manager and no user action is required.

When a resource is created a handle to that resource is returned from the create function call. This handle is used to refer to the resource in all subsequent calls to the window manager from user code or to user code from the window manager. This handle is not a pointer – do not dereference it. All handles are unique and used only once. A handle is typedef'd as a `uin32_t` meaning that there can be over 4 billion of them. Handle numbers can grow large - for example as MiniWin timers are one shot every timer firing creates a new handle. If there is any risk of the more than 4 billion handles change the typedef in miniwin.h to a uint64_t.

19.1 Resource Handles and Id's – For Information Only

This section is a brief description of the internals of the MiniWin window manager and can be skipped.

When calling a MiniWin public function all MiniWin objects are referred to in the public API function call by their handle. However, in the window manager's internal code it is necessary to obtain the position in the array of a particular resource type (window, control or timer). This position in the array is called an id. Inside MiniWin handles are converted to id's. All variables within MiniWin that contain a handle end in the name _handle and all variables that contain an id end in the name _id.

20 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.

21 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: Rendering of window frames is done in window manager code, not in user code. 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.

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 the 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.

22 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 operator 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. An example of this is changing the text a label control displays. The user posts a message to change the text, then posts a message to paint the control. Forgetting this behaviour is the most common answer to the question “why has my control not changed after I updated one of its values?”

There are many different types of painting request. This is also to minimize computation and flicker. A window frame is requested to paint 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 painted in their entirety or a sub-section defined by a rectangle. Finally a paint all can be requested which paints everything currently displayed. Always use the minimum painting type for the situation. Requesting a paint all at all times may be easy and the least code to write, but it will not give a pleasing user experience.

23 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 can have their scroll position set programmatically and can be enabled and disabled via the MiniWin manager API.

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. Control scroll bars can have their scroll position set programmatically by sending them a message and can be enabled or disabled using the generic MiniWin window manager API for enabling and disabling controls.

Both types of scroll bars do no scroll anything on their own. When the user scrolls a scroll bar its scroll position is posted as a message as a proportion of its range (always 0 - 255) and nothing else. It is up to the owning window's paint function to handle the message and perform the correct drawing of its contents, shifted as appropriate according to a scroll bar's position.

23.1 Window Scrolling Techniques

There are 3 methods of scrolling window contents. These are listed below along with where examples of the technique can be found in the example projects.

23.1.1 Dynamic Repainting

In this method the window’s contents are redrawn immediately when a user moves a scroll bar. As the scroll bar is scrolled the contents are continually redrawn. The user code intercepts the `MW_WINDOW_VERT_SCROLL_BAR_SCROLLED_MESSAGE` or `MW_WINDOW_VERT_SCROLL_BAR_SCROLLED_MESSAGE` messages and kicks off a paint window request immediately. These messages are received repeatedly as the scroll bars are scrolled so that the window contents will be redrawn as the scroll bars are scrolled. This method is suitable for window contents that are quick to draw.

An example of this techniqe is found in `window_scroll.c` in the MiniWinTest example projects.

23.1.2 Delayed Repainting

In this method as the `MW_WINDOW_VERT_SCROLL_BAR_SCROLLED_MESSAGE` and `MW_WINDOW_VERT_SCROLL_BAR_SCROLLED_MESSAGE` messages are processed the scroll positions are saved in memory but no window contents redrawing is requested until a `MW_TOUCH_UP_MESSAGE` message is received. The effect of this is that the scrolled window contents do not get redrarn in their new position until the user has finished scrolling. This technique is suitable for slow to draw window contents like images read directly from a file.

An example of this technique is found in `window_image.c` in the MiniWinFile example projects.

23.1.3 Window Drags

In this method no scroll bars are used at all and instead window drag messages `MW_TOUCH_DRAG_MESSAGE` are handled in the window message handler and the window's contents redrawn according to the user's drag inputs on the display. This is the most intuitive method for moving arounf the screen from the user's point of view but is the most difficult to implement. Careful scaling, range checking and handling of window resizing need performing.

An example of this technique is found in `window_text.c` in the MiniWinFile example projects.

24 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 controls: MiniWin list box and text box controls can have more content than they can show in their client area at any one time. A solution to this is to make them scrolling. Neither of these controls can be created with its own scroll bar, but they can co-operate with a separate vertical scroll bar control that the user creates.

To implement this feasture these two control types are scroll aware. This means that their scroll position can be set from their parent window and the control will take this into account when it paints itself. In addition, these controls will inform their parent window if they actually need to be scrolled depending on their current content. For example, a list box with 4 entries by created able to show 6 lines does not need to be scrolled. A text box created 100 pixels high but currently showing text when rendered that is 400 lines high will need to be scrolled.

24.1 Scrolling Messages – List boxes

24.1.1 Messages received

MW_LIST_BOX_SCROLL_BAR_POSITION_MESSAGE

This message is sent from the list box's parent window to the list box control and contains the proportion (0 – 255) that a list box should scroll. This is useful when a vertical scroll bar is used to scroll a list box as it is the same value that a scroll bar returns. In the parent window's message handler intercept the message from the vertical scroll bar and send the data value on to the list box.

MW_LIST_BOX_LINES_TO_SCROLL_MESSAGE

This message is sent from the list box's parent window to the list box control and contains the absolute number of lines the list box should scroll (this is list box lines, not vertical pixels). For example, if a list box can show 4 lines but currently has 6 entries sending a value of 2 will display lines 2, 3, 4 and 5. This is useful when arrow buttons are used to scroll a list box.

24.1.2 Messages Sent

MW_LIST_BOX_SCROLLING_REQUIRED_MESSAGE

This message is sent from a list box to its parent window both when the list box control is created and when a list box has its array of entries changed. This message indicates if scrolling is necessary and the maximum number of lines the list box can be scrolled. Handle this message in the list box's parent window to enable/disable a scroll bar or arrow buttons and to record the maximum number of lines the list box can be scrolled if arrow buttons are used for scrolling.

24.2 Scrolling Messages – Text boxes

24.2.1 Messages received

MW_TEXT_BOX_SCROLL_BAR_POSITION_MESSAGE

This message is sent from the text box's parent window to the text box control and contains the proportion (0 – 255) that a text box should scroll. This is useful when a vertical scroll bar is used to scroll a text box as it is the same value that a scroll bar returns. In the parent window's message handler intercept the message from the vertical scroll bar and send the data value on to the text box.

MW_TEXT_BOX_LINES_TO_SCROLL_MESSAGE

This message is sent from the text box's parent window to the text box control and contains the absolute number of vertical pixel lines the text box should scroll . For example, if a text box can show 200 pixel lines but currently has text contents that when rendered is 400 pixel lines high sending a value of 50 will lines of rendered text from 50 to 250. This is useful when arrow buttons are used to scroll a list box.

24.2.2 Messages Sent

MW_TEXT_BOX_SCROLLING_REQUIRED_MESSAGE

This message is sent from a text box to its parent window both when the text box control is created and when a text box has its array of entries changed. This message indicates if scrolling is necessary and the maximum number of vertical pixel lines the text box can be scrolled. Handle this message in the text box’s parent window to enable/disable a scroll bar or arrow buttons and to record the maximum number of vertcal pixel lines the text box can be scrolled if arrow buttons are used for scrolling.

25 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.

26 MiniWin Example Projects

MiniWin comes with 6 example projects each of which has a variant for the 4 hal layers. Each project has its own `miniwin_config.h` header file in the `common` folder (but are shared amongst different hardware variants for each project). The sample projects are listed below.

26.1 MiniWinTest

This is a comprehensive example using most of MiniWin's user interface 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 most of the user interface controls 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.

26.2 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.

26.3 MiniWinFile

This project connects to a pen drive via USB (STM32F4xx examples) or the Windows/Linux file system (Windows/Linux examples) 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 and 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 on embedded hardware examples.

26.3.1 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.

26.3.2 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.

26.4 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.

26.5 MiniWinTTFonts

This project contains the code for a demonstration of MiniWin's TrueType fonts example. The two yellow background windows show the same font in two modes – one with anti-aliasing and alpha-blending, the other in plain pixel mode, so that a comparison can be made between the appearance and rendering speed of the two modes. The anti-aliased alpha blended mode gives a better appearance but at the expense of slower rendering, especially when scrolling text.

The other windows show the text box control which can render TrueType fonts. One window shows a fixed text box, a second with a scroll bar and a third with up/down arrows. The text displayed can be changed with a button from too big to all fit in the text box (requiring scrolling) to too small to overflow the text box.

26.6 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. On the Linux build keyboard scroll lock LED is flashed.
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 4 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 the 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.

   On the Linux build there is also no accelerometer or gyrometer but the arrow keys and the 'l' and 'r' keys are used to simulate rotations.

The version of FreeRTOS used in this integration example is 10.1. This 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 partly 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.

In order for the MiniWin/FreeRTOS example to run under Linux the FreeRTOS Linux port is *not* used as it was found to be based on a too old version of FreeRTOS. Instead a quick and dirty FreeRTOS simulation layer is used using Linux pthreads. Do not base any of your code on this sample, it's for example simulation purposes only!

26.7 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 draw 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.

26.8 Eclipse Linked Folders

None of the example projects contain the MiniWin source code within them. They all use the Eclipse linked folder 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 4 builds for each application (for the 4 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.

26.9 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.

26.10 Board Support Package Code

The embedded examples and the Windows/Linux FreeRTOS examples have a folder called BSP. For the embedded examples 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 and Linux example projects do not need this code as the services required are supplied by the operating system (apart from the FreeRTOS examples, where the BSP folder contains the FreeRTOS for Windows source or the FreeRTOS Linux simulation source).

27 Standard Controls Details

27.1 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.

27.2 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_CHECK_BOX_SET_CHECKED_STATE_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.

27.3 Keypad

Initialisation: User needs to set the is_only_positive flag to allow or disallow negative numbers.
Resources required: A timer is required to animate the flashing cursor.
Sizes: Small and large
Messages processed: `MW_CONTROL_CREATED_MESSAGE, MW_WINDOW_TIMER_MESSAGE, MW_TOUCH_DOWN_MESSAGE`
Messages sent: `MW_KEY_PRESSED_MESSAGE` Data contains the ASCII value of the pressed key (‘0’ to ‘9’, ‘-’ or ‘\b’).

27.4 Keyboard

Initialisation: None.
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_WINDOW_TIMER_MESSAGE, MW_TOUCH_DOWN_MESSAGE.
Messages sent: MW_KEY_PRESSED_MESSAGE Data contains the ASCII value of the pressed key (Any visible ASCII character, ' ' or '\b').

27.5 Label

Initialisation: User needs to set the label text.
Resources required: None
Sizes: Small and large
Messages processed: MW_LABEL_SET_LABEL_TEXT_MESSAGE. Data contains a pointer to the new label text that is copied into the control's memory.
Messages sent: None.

27.6 List Box

Initialisation: User needs to set the number of lines the list 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_TOUCH_DOWN_MESSAGE, MW_WINDOW_TIMER_MESSAGE to redraw a pressed line, MW_LIST_BOX_SCROLL_BAR_POSITION_MESSAGE to receive an associated scroll bar proportion scrolled message (0 – 255), MW_LIST_BOX_LINES_TO_SCROLL_MESSAGE to receive an absolute number of lines to scroll the text, or MW_LIST_BOX_SET_ENTRIES_MESSAGE to receive a pointer to new entries data to render.
Messages sent: MW_LIST_BOX_SCROLLING_REQUIRED_MESSAGE at start-up and also when new entries to display message is received to indicate to parent window that not all the entries fits in the list box and scrolling or arrow buttons will be required and the maximum number of lines the list box can be scrolled, MW_LIST_BOX_ITEM_PRESSED_MESSAGE. Data contains the number of the list box item pressed, starting at 0.

27.7 Progress Bar

Initialisation: User needs to set the progress percentage.
Resources required: None.
Sizes: User defined on creation
Messages processed: MW_PROGRESS_BAR_SET_PROGRESS_MESSAGE. Data contains the progress as a percentage.
Messages sent: None.

27.8 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_RADIO_BUTTON_SET_SELECTED_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.

27.9 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_SCROLL_BAR_SET_SCROLL_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.

27.10 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_SCROLL_BAR_SET_SCROLL_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.

27.11 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.

27.12 Text Box

Initialization: User needs to set the run-length encoded font, justification, foreground colour, background colour and text.
Resources required: The font data in program or data memory
Sizes: Not applicable

Messages processed: `MW_CONTROL_CREATED_MESSAGE`, `MW_TEXT_BOX_SCROLL_BAR_POSITION_MESSAGE` to receive an associated scroll bar proportion scrolled message (0 – 255), `MW_TEXT_BOX_LINES_TO_SCROLL_MESSAGE` to receive an absolute number of pixel lines to scroll the text, or `MW_TEXT_BOX_SET_TEXT_MESSAGE` to receive a pointer to new text data to render.
Messages sent: `MW_TEXT_BOX_SCROLLING_REQUIRED_MESSAGE` at start-up and also when new text to render message is received. To indicate to parent window that not all the text fits in the text box and scrolling or arrow buttons will be required and the maximum number of lines the list box can be scrolled.

28 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_Windows/Linux: This is a command line utility for converting a Windows 24 bit per pixel `.bmp` file to a 3 bytes per per pixel C99 source file. It uses EasyBMP.

BMPConvMono_Windows/Linux: This is a command line utility for converting a Windows 2 bit per pixel `.bmp` file to an 8 pixels per byte C99 source file. It uses EasyBMP.

FontEncoder_Windows/Linux: This is a command line utility for converting a TrueType font file to a run-length encoded C99 source file. It comes as part of the mcufonts library.

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

`docs`	All documentation
`bitmaps`	Bitmaps and their C99 file encodings used by MiniWin
`gl`	The graphics library incorporated in MiniWin including fonts
`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`	Templates of required application-specific files
`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 4 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, Windows and Linux 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 example 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.

MiniWinTTFonts: As above but for the TrueType font rendering example project. The application code and sub-project layout is as for the previous example.

MiniWinFreeRTOS: As above but for the FreeRTOS integration example project. The application code and sub-project layout is as for the previous example with the addiiton of the FreeRTOS source under the BSP folder. For the Windows and Linux examples a FreeRTOS simulation is provided.

29 User Configuration

MiniWin can be configured by changing default settings found in header file `MiniWin/user/miniwin_config_template.h`. For your project you should copy this file and rename it `miniwin_config.h` and put it in a folder where the compiler can find it. You can customise the values found in this file. There are 7 sections to this file:

29.1 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.

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.

29.2 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.

29.3 User Interface Colours

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

29.4 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.

29.5 Bitmapped Fonts

There are 6 bitmapped fonts available as standard in MiniWin – 5 fixed width and one proportional width. The smallest fixed width (9 pixels high) and the proportional width (15 pixels high) are required by the MiniWin window manager and are always built in. The other 4 fixed width (12, 16, 20 and 24 pixels high) are optional. These can be included in the build or excluded by commenting in or out #defines in the configuration header file.

The `#defines` are:

```
#define MW_FONT_12_INCLUDED
#define MW_FONT_16_INCLUDED
#define MW_FONT_20_INCLUDED
#define MW_FONT_24_INCLUDED
```

29.6 Dialogs

The file chooser dialog is an option. This is because it requires extra functions to be implemented to interact with the file system. See the dialogs section in this document for details. Inclusion of the file chooser dialog is controlled by this `#define`:

```
#define MW_DIALOG_FILE_CHOOSER
```

29.7 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.

30 MiniWin Asserts

The MiniWin window manager, supporting code and example projects all use the MiniWin assert macro. Under debug build when `NDEBUG` is defined (usually this is defined automatically for debug builds) 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), attempt to use an un-included font, or something bad is attempted via the public API (i.e. removing a window with a bad window id, specifying a null pointer when not allowed) 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.

31 MiniWin Naming Convention

All public identifiers (functions, types, globals and constants) in the MiniWin project are prefixed with mw_ or MW_. This is to help prevent name clashes with other user-code identifiers. Static functions and variables and file-local constants do not have the mw_ or MW_ prefix. This helps to distinguish between public and non-public identifiers.

32 Quick-start MiniWin Application 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, if you are not using one of the supplied hal versions, you need to implement your hal functions, as described earlier, for your board.

Once you have implemented you hal functions, or if you are using one of the supplied hal layers, 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).

You need to exclude from the build all the supplied sample layers (`stm32f407, stm32f429, windows` or `linux`) that you are not using. If you are using one of them, leave it included in the build. They are found under `MiniWin/hal`.

1. Copy `user/miniwin_config_template.h` to a project folder of yours where the compiler can find it and rename it `miniwin_config.h`. 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(mw_handle_t window_handle,
                                  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"

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;
```

Add these two extern declarations while you are here. They are handles to the controls you will be creating later.

```
extern mw_handle_t button_handle;
extern mw_handle_t label_handle;
```

3. Add stub functions to window_simple.c

```
void window_simple_paint_function(mw_handle_t window_handle,
                                  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 and local variables section, which are handles to the window and controls you are creating, and instance data for the controls:

```
#include "miniwin.h"
#include "window_simple.h"

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

mw_handle_t window_simple_handle;
mw_handle_t button_handle;
mw_handle_t label_handle;

/**********************
*** LOCAL VARIABLES ***
**********************/

static mw_ui_label_data_t label_data;
static 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_handle = 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_handle = mw_ui_label_add_new(100,
			5,
			84,
			window_simple_handle,
			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_handle = mw_ui_button_add_new(10,
			10,
			window_simple_handle,
			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_get_window_client_rect(window_handle).width,
            mw_get_window_client_rect(window_handle).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_handle);
      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_handle == button_handle)
      {
            mw_create_window_dialog_one_button(20,
                  50,
                  150,
                  "Title",
                  "This is a message",
                  "Yep",
                  false,
                  message->recipient_handle);
      }
      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 by posting it a message.

Posting messages is fundamental to how the user interacts with the window manager and other windows in MiniWin so the parameters to this post message example will be discussed in detail following the code snippet below.

```
case MW_DIALOG_ONE_BUTTON_DISMISSED_MESSAGE:
      mw_post_message(MW_LABEL_SET_LABEL_TEXT_MESSAGE,
            message->recipient_handle,
            label_handle,
            MW_UNUSED_MESSAGE_PARAMETER,
            (void *)"Hello world!",
            MW_CONTROL_MESSAGE);
      mw_paint_control(label_handle);
      break;
```

`MW_LABEL_SET_LABEL_TEXT_MESSAGE`
This is the message type you are sending to the label control. It is from miniwin.h where you will find it and many others. It tells the message recipient (the label control in this case) what the message is about and what the message's data fields contain.

`message->recipient_handle`
This is the sender of the message about to be posted. The sender of this new message is the recipient of the message just received that is being processed. It might seem confusing to see recipient in the sender field, but it's a common pattern in MiniWin. It's because in processing a just received message you are posting a new one.

`label_handle`
This is the recipient of the new message you are posting. In this case you are sending a message to your label control which you refer to by its handle returned by MiniWin when you created the control.

`MW_UNUSED_MESSAGE_PARAMETER`
You are not sending anything in the uint32_t data field in this example so this field can be anything but use this #define to indicate that it is unused.

`(void *)"Hello world!"`

This is the pointer field of the message which contains this message's data. Remeber that it is essential that whatever is pointed to by this field will still exist when the message is received, so no pointers to local variables. In this case it's a pointer to constant data.

`MW_CONTROL_MESSAGE`
This indicates to MiniWin that the recipient of this message is a control.

You need to get this label repainted after changing its text so call
`mw_paint_control(label_handle)`.

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

33 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

mcufont, Petteri Aimonen

FreeType, version 2.9.1 (required by mcufont font processing tool)

34 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.

Handle – a reference to a MiniWin resource created by MiniWin for the user – i.e. window, control or handle. The handle is used in all later API calls to MiniWin to refer to the resource. A handle is unique and never reused. It is not a pointer.

35 Appendix 1

This section lists all the messages defined within MiniWin and the contents of each message's data and pointer fields.

35.1 Messages posted by the window manager

MW_WINDOW_CREATED_MESSAGE
Message send to window as soon as it is created and before it is painted
message_data: Unused
message_pointer: Unused

MW_WINDOW_REMOVED_MESSAGE
Message sent to window just before it is removed
message_data: Unused
message_pointer: Unused

MW_WINDOW_GAINED_FOCUS_MESSAGE
Message sent to a window when it gains focus
message_data: Unused
message_pointer: Unused

MW_WINDOW_LOST_FOCUS_MESSAGE
Message sent to a window when it loses focus
message_data: Unused
message_pointer: Unuseded

MW_WINDOW_RESIZED_MESSAGE
Message to a window when it has been resized
message_data: Upper 16 bits = window new width, lower 16 bits = window new height
message_pointer: Unused

MW_WINDOW_MOVED_MESSAGE
Message to a window when it has been moved
message_data: Unused
message_pointer: Unused

MW_WINDOW_MINIMISED_MESSAGE
Message to a window when it has been minimised
message_data: Unused
message_pointer: Unused

MW_WINDOW_RESTORED_MESSAGE
Message to a window when it has been restored
message_data: Unused
message_pointer: Unused

MW_WINDOW_VISIBILITY_CHANGED_MESSAGE
Message to a window when its visibility has changed
message_data: true if made visible, false if made invisible
message_pointer: Unused

MW_WINDOW_VERT_SCROLL_BAR_SCROLLED_MESSAGE
Message to a window when a window vertical scroll bar has been scrolled
message_data: new vertical scroll position 0 - 255 as a proportion of scroll bar length
message_pointer: Unused

MW_WINDOW_HORIZ_SCROLL_BAR_SCROLLED_MESSAGE
Message to a window when a window horizontal scroll bar has been scrolled
message_data: new vertical scroll position 0 - 255 as a proportion of scroll bar length
message_pointer: Unused

MW_CONTROL_CREATED_MESSAGE
Message send to control as soon as it is created and before it is painted
message_data: Unused
message_pointer: Unused

MW_CONTROL_REMOVED_MESSAGE

Message sent to control just before it is removed
message_data: Unused
message_pointer: Unused

MW_CONTROL_GAINED_FOCUS_MESSAGE
Message sent to all controls in a window when parent window gains focus or control made visible
message_data: Unused
message_pointer: Unused

MW_CONTROL_VISIBILITY_CHANGED_MESSAGE
Message to a control when its visibility has changed
message_data: true if made visible, false if made invisible
message_pointer: Unused

MW_CONTROL_LOST_FOCUS_MESSAGE
Message sent to all controls in a window when parent window loses focus or control made invisible
message_data: Unused
message_pointer: Unused

MW_TOUCH_DOWN_MESSAGE
Message sent to a window or control when it receives a touch down event
message_data: Upper 16 bits = x coordinate, lower 16 bits = y coordinate
message_pointer: Unused

MW_TOUCH_HOLD_DOWN_MESSAGE
Message sent to a window or control when it receives a touch hold down event
message_data: Upper 16 bits = x coordinate, lower 16 bits = y coordinate
message_pointer: Unused

MW_TOUCH_UP_MESSAGE
Message sent to a window or control when it receives a touch up event
message_data: The handle of the original window or control where the touch down occurred
message_pointer: Unused

MW_TOUCH_DRAG_MESSAGE
Message sent to a window or control when it receives a drag event
message_data: Upper 16 bits = x coordinate, lower 16 bits = y coordinate
message_pointer: Unused

MW_MENU_BAR_ITEM_PRESSED_MESSAGE
Response message from a menu bar that an item has been pressed
message_data: The menu bar item selected, zero based
message_pointer: Unused

MW_TIMER_MESSAGE
Message sent to a window or control when a timer has expired
message_data: The handle of the timer that has just expired
message_pointer: Unusedd

35.2 Messages posted by controls in response to user interface input

MW_BUTTON_PRESSED_MESSAGE
Response message from a button that it has been pressed
message_data: Unused
message_pointer: Unused

MW_CHECKBOX_STATE_CHANGE_MESSAGE
Response message from a check box that its state has changed
message_data: true if check box checked, false if check box unchecked
message_pointer: Unused

MW_RADIO_BUTTON_ITEM_SELECTED_MESSAGE
Response message from a radio button that its state has changed
message_data: The selected radio button zero based
message_pointer: Unused

MW_LIST_BOX_ITEM_PRESSED_MESSAGE
Response message from a list box that an item has been pressed
message_data: The selected list box line zero based
message_pointer: Unused

MW_LIST_BOX_SCROLLING_REQUIRED_MESSAGE
Message posted by a list box to its parent window indicating if scrolling is required, i.e. too many lines to display at once
message_data: upper 16 bits: 1 if scrolling required, 0 if scrolling not required; lower 16 bits: the maximum lines that can be scrolled
message_pointer: Unused

MW_CONTROL_VERT_SCROLL_BAR_SCROLLED_MESSAGE
Response message from a vertical control scroll bar that it has been scrolled
message_data: Unused
message_pointer: Unused

MW_CONTROL_HORIZ_SCROLL_BAR_SCROLLED_MESSAGE
Response message from a horizontal control scroll bar that it has been scrolled

message_data: new horizontal scroll position from 0 to 255 as a proportion of the scroll bar
message_pointer: Unused

MW_ARROW_PRESSED_MESSAGE
Response message from a arrow that it has been pressed
message_data: The arrow direction
message_pointer: Unused

MW_KEY_PRESSED_MESSAGE
ASCII value of key pressed, can be backspace

message_data: The ASCII code of the pressed key
message_pointer: Unused

MW_TEXT_BOX_SCROLLING_REQUIRED_MESSAGE
Message posted by a text box to its parent window indicating if scrolling is required, i.e. too much text to display in the box at once
message_data: upper 16 bits: 1 if scrolling required, 0 if scrolling not required; lower 16 bits: the maximum lines that can be scrolled in pixels
message_pointer: Unused

35.3 Messages posted to controls from user code

MW_LABEL_SET_LABEL_TEXT_MESSAGE
Set the label's text by passing a pointer to a character buffer
message_data: Unuseded
message_pointer: Pointer to the label's new text

MW_CHECK_BOX_SET_CHECKED_STATE_MESSAGE
Set a check box's checked state
message_data: true to set check box checked or false to set it unchecked
message_pointer: Unused

MW_PROGRESS_BAR_SET_PROGRESS_MESSAGE
Set a progress bar's progress level as a percentage
message_data: Percentage to set progress bar's progress from 0 - 100
message_pointer: Unused

MW_SCROLL_BAR_SET_SCROLL_MESSAGE
Set a scroll bar's scroll position
message_data: Set a scroll bar's scroll position from 0 - 255
message_pointer: Unused

MW_LIST_BOX_LINES_TO_SCROLL_MESSAGE
Set how many lines to scroll a list box through the list box's lines
message_data: Number of lines to scroll zero based
message_pointer: Unused

MW_LIST_BOX_SCROLL_BAR_POSITION_MESSAGE
Position of a scroll bar associated with a list box
message_data: Scroll bar position, 0 - 255
message_pointer: Unused

MW_LIST_BOX_SET_ENTRIES_MESSAGE
Set new entries and entry count for a list box
message_data: Number of entries in new array of entries
message_pointer: Pointer to array of entries

MW_RADIO_BUTTON_SET_SELECTED_MESSAGE
Set a radio button's chosen button
message_data: The button to set zero based
message_pointer: Unused

MW_TEXT_BOX_SET_TEXT_MESSAGE
Set the scrollable text box's text by passing a pointer to a character buffer
message_data: Unused
message_pointer: Pointer to the scrollable text box's new text

MW_TEXT_BOX_SCROLL_BAR_POSITION_MESSAGE
Position of a scroll bar associated with a text box
message_data: Scroll bar position, 0 - 255
message_pointer: Unused

MW_TEXT_BOX_LINES_TO_SCROLL_MESSAGE
Set how many lines to scroll a text box
message_data: Number of lines to scroll in pixels
message_pointer: Unused

35.4 Messages posted by standard dialogs

MW_DIALOG_ONE_BUTTON_DISMISSED_MESSAGE
One button dialog has been dismissed
message_data: Unused
message_pointer: Unused

MW_DIALOG_TWO_BUTTONS_DISMISSED_MESSAGE
Two button dialog has been dismissed
message_data: 0 if left button pressed, 1 if right button pressed
message_pointer: Unused

MW_DIALOG_TIME_CHOOSER_OK_MESSAGE
Time chooser dialog has been dismissed by ok button
message_data: Mask with 0x00ff for minutes, mask with 0xff00 for hours
message_pointer: Unused

MW_DIALOG_TIME_CHOOSER_CANCEL_MESSAGE
Time chooser dialog has been dismissed by cancel button
message_data: Unused
message_pointer: Unused

MW_DIALOG_DATE_CHOOSER_OK_MESSAGE
Date chooser dialog has been dismissed by ok button
message_data: Mask with 0xffff0000 for 4 digit year, mask with 0x0000ff00 for month (1-12), mask with 0x000000ff for date (1-31)
message_pointer: Unused

MW_DIALOG_DATE_CHOOSER_CANCEL_MESSAGE
Date chooser dialog has been dismissed by cancel button
message_data: Unused
message_pointer: Unused

MW_DIALOG_FILE_CHOOSER_FILE_OK_MESSAGE
File chosen in file chooser dialog
message_data: Unused
message_pointer: Pointer to char buffer holding path and file name

MW_DIALOG_FILE_CHOOSER_FOLDER_OK_MESSAGE
Folder chosen in file chooser dialog
message_data: Unused
message_pointer: Pointer to char buffer holding path name

MW_DIALOG_FILE_CHOOSER_CANCEL_MESSAGE
File chooser dialog was canceled with no file chosen
message_data: Unused
message_pointer: Unused

MW_DIALOG_TEXT_ENTRY_OK_MESSAGE
Text entry dialog ok message
message_data: Unused
message_pointer: Pointer to char buffer holding entered text

MW_DIALOG_TEXT_ENTRY_CANCEL_MESSAGE
Text entry dialog cancel message
message_data: Unused
message_pointer: Unused

MW_DIALOG_NUMBER_ENTRY_OK_MESSAGE
Number entry dialog ok message
message_data: Unused
message_pointer: Pointer to char buffer holding entered number as text including '-' if entered by user

MW_DIALOG_NUMBER_ENTRY_CANCEL_MESSAGE
Number entry dialog cancel message
message_data: Unused
message_pointer: Unused

35.5 Utility Messages

MW_WINDOW_PAINT_ALL_MESSAGE
System message to paint everything
message_data: Unused
message_pointer: Unused

MW_WINDOW_FRAME_PAINT_MESSAGE
System message to get a window's frame painted
message_data: Combination of MW_WINDOW_FRAME_COMPONENT_TITLE_BAR,
MW_WINDOW_FRAME_COMPONENT_BORDER,
MW_WINDOW_FRAME_COMPONENT_MENU_BAR,
MW_WINDOW_FRAME_COMPONENT_VERT_SCROLL_BAR,
MW_WINDOW_FRAME_COMPONENT_HORIZ_SCROLL_BAR
message_pointer: Unused

MW_WINDOW_CLIENT_PAINT_MESSAGE
System message to call a window's client area paint function
message_data: Unused
message_pointer: Unused

MW_WINDOW_CLIENT_PAINT_RECT_MESSAGE
System message to call a window's client area paint rect function
message_data: Unused
message_pointer: Pointer to a mw_util_rect_t structure

MW_CONTROL_PAINT_MESSAGE
System message to call a control's paint function
message_data: Unused
message_pointer: Unused

MW_CONTROL_PAINT_RECT_MESSAGE
System message to call a control's paint rect function
message_data: Unused
message_pointer: Pointer to a mw_util_rect_t structure

MW_WINDOW_EXTERNAL_WINDOW_REMOVED
Message to a window when another window has been removed that is not the window receiving the message. Use this message when a window is removed as a result of user code and another window needs to know
message_data: Unused
message_pointer: Unused

MW_USER_1_MESSAGE
Message to a window for any user-defined purpose
message_data: Any user meaning
message_pointer: Any user meaning

MW_USER_2_MESSAGE
Message to a window for any user-defined purpose
message_data: Any user meaning
message_pointer: Any user meag

MW_USER_3_MESSAGE

Message to a window for any user-defined purpose
message_data: Any user meaning
message_pointer: Any user meag

MW_USER_4_MESSAGE
Message to a window for any user-defined purpose
message_data: Any user meaning
message_pointer: Any user mea

MW_USER_5_MESSAGE
Message to a window for any user-defined purpose
message_data: Any user meaning
message_pointer: Any user mea