MiniWin Embedded Window Manager
16 January 2019

1 Introduction....4
2 Who is MiniWin For?....5
3 Features....5
4 User Interface Controls....5
5 MiniWin Windows....6
6 Developer Utility Applications....6
7 Drivers and Examples....7
8 Customisability....8
9 Display Size and Rotation....8
10 Implementing the Hardware Abstraction Layer (hal)....9
  10.1 Delay Driver....9
  10.2 Init Driver....9
  10.3 Timer Driver....9
  10.4 Non-Volatile Storage Driver....10
  10.5 Touch Driver....10
  10.6 LCD Driver....11
11 Paint Algorithm....13
12 Z Orders....13
13 Multiple Instances of the Same Window....14
14 Graphics Library....14
  14.1 Fonts....15
    14.1.1 Always Available Bit-Mapped....16
    14.1.2 Optional Bit-Mapped....16
    14.1.3 Run Length Encoded TrueType Fonts....17
15 Standard Controls Overview....19
16 Enabling and Disabling Controls....20
17 Standard Dialogs....20

17.1 Single Button Message Box....21
17.2 Double Button Message Box....21
17.3 Time Chooser....21
17.4 Number Entry....21
17.5 Text Entry....21
17.6 Date Chooser....21
17.7 File Chooser....22
18 MiniWin Messages....22
18.1 MiniWin Message Types....23
18.1.1 Messages Posted by the Window Manager....23
18.1.2 Messages Posted by Controls in Response to User Interface Input....23
18.1.3 Messages Posted to Controls to Change Their State or Appearance....23
18.1.4 Messages Posted by Standard Dialogs....23
18.1.5 Utility Messages....23
18.1.6 User Code Defined Messages....23
18.2 MiniWin Message Fields....23
19 Transferring Data to Message Handler Functions....24
20 MiniWin Memory Pools and Handles....24
20.1 Resource Handles and Id's – For Information Only....25
21 Touch Events....25
22 Painting and Client Areas....26
23 Automatic Painting and User Code Painting....26
24 Scroll Bars....27
24.1 Window Scrolling Techniques....28
24.1.1 Dynamic Repainting....28
24.1.2 Delayed Repainting....28
24.1.3 Window Drags....28
25 Combining Controls....28
25.1 Scrolling Messages – List Boxes....29
25.1.1 Messages Received....29
25.1.2 Messages Sent....29
25.2 Scrolling Messages – Text Boxes....30
25.2.1 Messages Received....30
25.2.2 Messages Sent....30
26 Operator Window Interaction....30
27 MiniWin Example Projects....31

27.1 MiniWinTest....32
27.2 MiniWinSimple....33
27.3 MiniWinRoot....33
27.4 MiniWinFile....33
27.4.1 File System Integration....33
27.4.2 Multiple Window Instances....33
27.5 MiniWinFixedWindows....33
27.6 MiniWinTTFonts....34
27.7 MiniWinFreeRTOS....34
27.8 Example Projects for Microsoft Windows and Linux....35
27.9 Atollic TrueStudio/NXP MCUXpresso Linked Folders....35
27.10 Other Required Project Files....36
27.11 Board Support Package Code....36
28 Standard Controls Details....36
28.1 Button....36
28.2 Check Box....36
28.3 Keypad....37
28.4 Keyboard....37
28.5 Label....37
28.6 List Box....37
28.7 Progress Bar....38
28.8 Radio Button....38
28.9 Horizontal Scroll Bar....38
28.10 Vertical Scroll Bar....38
28.11 Arrow Button....39
28.12 Text Box....39
29 Source Code Layout....39
30 User Code Configuration....41
30.1 Memory Allocation....41
30.2 Display Rotation....41
30.3 Sizes....42
30.4 User Interface Colours....42
30.5 Timings....42
30.6 Bitmapped Fonts....42
30.7 Dialogs....42
30.8 Other....43

31 MiniWin Asserts....43
32 MiniWin Naming Convention....43
33 Quick-Start MiniWin Application Guide....43
34 MiniWin Code Generator....49
34.1 Building the Code Generator....49
34.2 Running the Code Generator....49
34.3 Example Code Generator Configuration Files....51
34.3.1 Building and Running Code Generator Examples for Linux....51
34.3.2 Building and Running Code Generator Examples for Windows....51
34.4 Code Generator Configuration File Format....51
34.5 Configuration File Format Overview....52
34.6 Configuration File Format Details....54
34.6.1 General Application Settings....54
34.6.2 Window Settings....55
34.6.3 Control Settings....58
34.7 Code Generator Error Messages....62
35 Known Issues....63
36 Third Party Software....63
37 Glossary of Terms....64
38 Appendix 1....65
38.1 Window Flags....65
38.1.1 User Settable Flags....65
38.1.2 Non-User Settable Flags....66
38.2 Control Flags....66
38.2.1 User Settable Flags....66
38.2.2 Non-User Settable Flags....66
39 Appendix 2....66
39.1 Messages Posted by the Window Manager....66
39.2 Messages Posted by Controls in Response to User Interface Input....69
39.3 Messages Posted to Controls from User Code....70
39.4 Messages Posted by Standard Dialogs....71
39.5 Utility Messages....73

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 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.
- Supports display rotation
- Incorporates a flexible graphics library.
- Comes with a 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 (with USB host or SDIO interface), 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.
- Built touch screen calibration capability the first time the window manager is started.
- Code generator to create all your windows and controls from a simple configuration file in JSON format.
- Simulators allowing your projects to be run and debugged on Windows or Linux speeding up the development of the user interface part of your embedded code.

4 User Interface Controls

- Button

- Check box
- Scrollable 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 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, scroll bars 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
- Standard or large sized features

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 operator 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:

- Converter for monochrome bitmap files in .bmp format to C99 source code.
- Converter for 24 bit colour bitmap files in .bmp format to C99 source code.
- Converter for TrueType font files to run-length encoded C99 source file for a single point size
- Code generator to create your window and control code from a JSON format configuration file.

The C99 source code produced by the font/bitmap 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.

The code generator creates the complete source for a buildable MiniWin app and makefiles to build it for Windows and Linux simulators. By changing the driver layer the generated code can be targeted at your hardware.

Each utility is located in MiniWin under its own folder where source code and makefiles are found. 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. The code generator is described in its own section later in this document.

These utility applications are all simple and do not contain any IDE project files. Instead they are all built on the command line using the makefiles provided. The command line build tools installed along with Atollic TrueStudio can be used for building these applications, or you can use any other installation of gcc, for example MinGW. For Windows only if using the the Atollic TrueStudio install for your command line compiler you need to edit your `PATH` environment variable and add the path to the folder containing `mingw32-make`, which if installed in the standard location is this...

```
C:\Program Files (x86)\Atollic\TrueSTUDIO for STM32 9.2.0\PCTools\bin
```

Following this, to build any application for Windows use this command line:

```
mingw32-make -f makefile_windows
```

To build any application for Linux use this command line:

```
make -f makefile_linux
```

There have been issues reported when using the converter for monochrome bitmaps to C99 source files. The input bitmap file when using this tool *must* be a monochrome bitmap. It will not work if the bitmap file has a colour depth of more than 1 bit per pixel (bpp). If you are having difficulty creating a 1 bpp file use an online converter to convert your image file to a 1 bpp file, for example...

https://online-converting.com/image/convert2bmp/

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 7 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 or a SD card, an example

that integrates MiniWin into FreeRTOS real-time operating system, a non-overlapped tiled windows example, a TrueType font justified-text rendering demonstration and an example demonstrating root window capabilities.

MiniWin comes with five 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, a hardware accelerated embedded example for the LPC54628 development board (NXP part number OM13098), and two simulator versions that allows applications to run within a Microsoft Windows window or a Linux X11 window. Each of the 7 example projects can be built for all five of the hal drivers. User interface development using the Windows or Linux simulators reduces application development time.

All sample projects are built with the free compiler/IDE Atollic TrueStudio or NXP MCUXpresso, currently version 9.2.0 and 10.3.0 respectively, either of which can be obtained for Microsoft Windows or Linux running on x-86. The Windows simulator examples will build using the Windows version of Atollic TrueStudio only and similarly the Linux simulator examples will build using the Linux version of Atollic TrueStudio 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 Display Size and Rotation

MiniWIn allows you to rotate the display as a compile time configuration (see configuration file section later in this document). It is not possible currently to rotate the display at run time to a different layout although this can be added if there is any demand.

The pixel transformations required to implement display rotation are required in the lcd driver. All the supplied platform implementations that MiniWin comes with implement this feature and therefore all example projects can be rotated. The default layout for the supplied hal layers is the natural screen layout. For the displays used by the embedded ST STM32 examples this is portrait, for the NXP LPC example it is landscape, and for the Windows and Linux simulators it is portrait.

When display rotation is chosen for the Windows and Linux simulators the window size is adjusted to reflect the change from portrait to landscape or vice-versa but the displayed window is not actually rotated on the computer screen as this would make development awkward.

Each display’s size for different displays needs to be specified in the `hal_lcd.c` hal source file. This value is the physical size in pixels of the display in use (or being simulated). It is not the size after any rotation is applied. So, for example, the displays attached to the STM32F429DISC1 board has a width of 240 and a height of

320. These vales remain the same in this source file regardless of the display rotation value chosen in the configuration file.

The touch screen needs recalibration each time the display rotation in configuration is changed. This is described later in this document.

10 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.

10.1 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 and touch screen drivers need delays during initialisation. Do not use these functions from within your application 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.

10.2 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.

10.3 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: `mw_hal_timer_fired`
This function is called by a timer interrupt handler and increments the MiniWin timer.

Function: `varies, timer interrupt handler`
This interrupt handler function, which the interrupt routine calls 20 times per second, calls `mw_hal_timer_fired` which increments the MiniWin tick counter on each call.

10.4 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.

10.5 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. It also contains details about if and when touch screen calibration is required as not all touch screens require calibration.

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.

Function: `mw_hal_touch_is_calibration_required`
This function tells MiniWin whether the user has requested a touch screen recalibration by some hardware methid, for example a push-button. If recalibration is required return true. This function is called only once by the MiniWin window manager at start-up. This functionality is demonstrated in the 2 ST hal and the LPC layers where at start-up the user button is checked and if pressed the touch screen calibration routine is run even if a previous calibration has been performed.

10.6 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. It also returns details of the screen size in pixels when requested to the MiniWin window manager or application code.

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_get_screen_width`
This function returns the width of the attached display in pixels. When using the Windows or Linux simulator the values returned by this function can be changed to simulate different sized displays. This value must be the physical width of the display in pixels regardless of any display rotation requirements. Display rotation is set in the configuration file (described later).

Function: `mw_hal_lcd_get_screen_height`
This function returns the height of the attached display in pixels. When using the Windows or Linux simulator the values returned by this function can be changed to

simulate different sized displays. This value must be the physical height of the display in pixels regardless of any display rotation requirements. Display rotation is set in the configuration file (described later).

Function: `mw_hal_lcd_pixel`
This function writes a single pixel with the specified colour to the display device. The x and y coordinates passed to this function will be within the display's maximum coordinate bounds and do not need further checking here.

If you wish to use the display rotation functionality of MiniWin then you need to implement the coordinate transformations required to achieve that here. If you do not wish to rotate your display but use the physical layout of your display then no coordinate transformation is required here. See the example projects for how to implement coordinate transformation if you require it.

Function `mw_hal_lcd_filled_rectangle_clip`
This function fills a rectangle on the screen with a single colour. The x and y and x + width and y + height coordinates passed to this function will be within the display's maximum coordinate bounds and do not need further checking here.

If you wish to use the display rotation functionality of MiniWin then you need to implement the coordinate transformations required to achieve that here. If you do not wish to rotate your display but use the physical layout of your display then no coordinate transformation is required here. See the example projects for how to implement coordinate transformation if you require it.

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.

The latter 3 functions can all be implemented using the pixel plotting function above. This reduces the amount of work needed porting the driver to your hardware but at the expense of running more slowly. This technique is used for the STM32F407 hal driver layer.

11 Paint Algorithm

MiniWin uses a painting algorithm that is optimized for use on a system without a shadow screen buffer or dual buffers, 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.

12 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 windows with no gaps in the numbering. This is done automatically by MiniWin; no user code 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 code 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.

13 Multiple Instances of the Same Window

Most windows are used with only a single instance of that window type. 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 type 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 same window type. 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`.

The example project MiniWinFile uses multiple instances of the same window type to show multiple images and text files, each in its own instance of the same window type.

14 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 code 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 code can keep a local copy and use `memcpy()` to copy the local copy into gl's copy as an alternative.

14.1 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

code 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.

14.1.1 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. Fore-
ground
 * 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.

14.1.2 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.

14.1.3 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.

14.1.3.1 Obtaining 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.

14.1.3.2 Building 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` and type

```
make -f makefile_linux.
```

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

14.1.3.3 Processing You TrueType Font

For this task you use the font processing tool `mcufont` (Linux) or `mcufont.exe` (Windows) which you built in the previous section. (Examples below use the Linux variant of `mcufont`. Replace this with `mcufont.exe` for Windows). 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 but doesn't look so sharp) 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.

14.1.3.4 Using 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 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);
```

15 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 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. Progress bars and text boxes are created with a user code 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 code modifiable and some are not; the non-user-code-modifiable fields contain control state and should not be changed by user code. Each control-specific data structure indicates which fields are user code 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

Controls can only be placed in windows, not in other controls. However, controls can be placed on the root window, and with suitable code, made pop-up. The MiniWinRoot example project shows this usage.

16 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.

17 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 system modal – that is the operator must respond to them and dismiss them before continuing. Because of this only one 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. They can be closed only 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.

All dialogs can be created as standard or large size. When created as large size the controls within them are also all created large size. A dialog must be completely on the screen when it is created or else the creation will fail and it won't be shown. The text entry dialog needs care in this respect as the keyboard control it creates is large and its width does not fit within a QVGA small dimension of 240 pixels.

17.1 Single Button Message Box

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

17.2 Double Button Message Box

This dialog shows a user code configurable message in a window with a user code configurable title and contains two buttons with user code 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.

17.3 Time Chooser

This dialog allows the user code to set a time in hours and minutes. The initial time to show when the dialog appears can be set. The operator 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 operator selected time.

17.4 Number Entry

This dialog allows the operator 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.

17.5 Text Entry

This dialog allows the operator 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.

17.6 Date Chooser

This dialog allows the operator 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 operator 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 operator selected date.

17.7 File Chooser

This dialog allows the operator 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 operator chooses a folder that folder is opened. A back arrow button allows the operator 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 MiniWinFile example project for details.

18 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 code 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 debugging problem is to place a breakpoint on the code where the message is handled.

18.1 MiniWin Message Types

The message used by MiniWin are defined in `miniwin.h` and detailed in Appendix 1 in this document. They fall into 6 groups described below:

18.1.1 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 operator shuts a window from the user interface then the window manager will automatically post the window created or window removed message respectively.

18.1.2 Messages Posted by Controls in Response to User Interface Input

When the operator 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.

18.1.3 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.

18.1.4 Messages Posted by Standard Dialogs

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

18.1.5 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.

18.1.6 User Code Defined Messages

These are free form messages that user code can use for any reason that are posted from user code in one window to user code in another. The message id’s of these messages are declared in `miniwin.h`.

18.2 MiniWin Message Fields

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.

19 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.

20 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 window manager and no user code action is required.

When a resource is created a handle to that resource is returned from the resource 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`.

20.1 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`.

21 Touch Events

WiniMin creates 4 different touch events from operator 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 hold down: this event has a 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 operator 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 operator. 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.

22 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 in user code. The user code 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. The window manager does not paint the window client area background. It is up to the user code to do this. You will see this being done in all the example project window paint functions.

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.

23 Automatic Painting and User Code Painting

The MiniWin window manager does minimal automatic repainting of window frames, 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 code 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 or 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 operator experience.

24 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 operator 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.

24.1 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.

24.1.1 Dynamic Repainting

In this method the window's contents are redrawn immediately when an operator moves a scroll bar. As the scroll bar is scrolled the contents are continually redrawn. The user code intercepts the `MW_WINDOW_HORIZ_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.

24.1.2 Delayed Repainting

In this method as the `MW_WINDOW_HORIZ_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 redrawn in their new position until the operator 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.

24.1.3 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 operator's drag inputs on the display. This is the most intuitive method for moving around the screen from the operator'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.

25 Combining Controls

MiniWin contains only a limited set of simple controls to keep its code base small. The user code 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 code creates.

To implement this feature 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 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.

25.1 Scrolling Messages – List Boxes

25.1.1 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.

25.1.2 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.

25.2 Scrolling Messages – Text Boxes

25.2.1 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.

25.2.2 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.

26 Operator Window Interaction

If a window has a title bar the operator can move, resize, minimize, restore, maximize and close windows (if enabled). A window can be created with a large sized flag set. When this flag is set the title bar and its icons, the menu bar and the window scroll bars (if selected) are all drawn with a larger size.

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. An operator 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.

27 MiniWin Example Projects

MiniWin comes with 7 example projects each of which has a variant for the 5 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).

All example projects go into the touchscreen calibration routine on first start-up. If the operator completes this successfully the calibration settings are stored in non-volatile memory and calibration does not need to be repeated on subsequent starts.

It is possible for all projects on all platforms to force a recalibration on any start-up. This is always required when changing the rotation of the display in the configuraton file (see the configuration section later). The techniques for forcing a recalibration are listed below for each platform:

Windows and Linux
Delete file settings.bin. If running in debug mode through TrueStudio or MCUXpresso this file is found in the project’s root folder. If running the executable file directly this file is found in the same folder as the executable file.

STM43F429DISC1 Development board
Hold down USER button on start-up

STM32F407 Discovery Development board
Hold down User button on start-up

LPC54628 OM13098 development board
Hold down User button on start-up.

The sample projects are listed below.

27.1 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. Iy displays details of the touch and drag events in the window.

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.

27.2 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.

27.3 MiniWinRoot

This project shows how to use the root window. A single button is created on the root window and has its message handler in the root window message handler. A normally invisible window with no border and no title bar is created that is completely filled by a list box. When the operator touches the root window this window is moved to that touch point and made visible. This simulates a pop-up menu on the desktop.

In addition the root window background is drawn with a bitmap to demonstrate this capability.

27.4 MiniWinFile

This project connects to a pen drive via USB for the STM32F4xx examples), a SD card via the SDIO interface for the LPC54628 example, or the Windows/Linux file system for the 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 and then the Open button 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.

27.4.1 File System Integration

Under the `BSP` folder are sub-projects containing third party libraries – the FatFS code for file system handling, SDMMC layer for the LPC54628 example, and the ST USB Host library with MSP support to link the FatFS module with the USB driver code in the ST HAL for the STM32F4xx examples.

27.4.2 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.

27.5 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.

Currently although this project builds and runs for the LPC54628 example the displayed elements do not fit correctly on the screen as the code is supplied because this device's board's LCD has a vertical resolution of only 272 pixels. The fix is to change the display rotation for this project from 0 to 90 degrees in this project's configuration file. This fixes the problem but then causes the same problem for the ST examples as the configuration file is shared. You can one or the other working at the same time but not both!

27.6 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.

27.7 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 and the LPC54628 development board builds one of the board user LED's is used for this purpose. On the Windows and Linux builds a small yellow square to the right of the MiniWin root window is shown in the MiniWin window.
2) Read accelerometer values from the on-board MEMS accelerometer and display angle arrows and text for X, Y and Z, each in their own window, along with an offset setting button. The way the example projects work for the 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 and the LPC54628 boards hve linear accelerometers which sense 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 the STM32F4xx integration examples is 10.1.1 and in the LPC54628 example is 10.0.1. This is a few versions on from the FreeRTOS integration examples found in the STM32 Cube package from ST. Later FreeRTOS code was obtained from the FreeRTOS website and the port partly changed slightly to work on the ST Discovery boards. The LPC54628 example is unchanged from that supplied with the MCUXpresso SDK.

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!

27.8 Example Projects for Microsoft Windows and Linux

The example project built to run on Microsoft Windows or Linux are not proper well behaved Microsoft Windows/X11 applications. MiniWin runs more slowly on Microsoft Windows/Linux compared to real hardware. Making an event driven window manager (MiniWin) run within an event driven window manager (Microsoft Windows or X11) is not a simple task, and various dodgy hacks are required to make it appear to work. Do not base any Windows or X11 code you may write on how things are done in the simulators!

27.9 Atollic TrueStudio/NXP MCUXpresso Linked Folders

None of the example projects contain the MiniWin source code within them. They all use the linked folder feature in Atollic TrueStudio/NXP MCUXpresso to link to the MiniWin shared source folder (Atollic TrueStudio and NXP MCUXpresso are based on Eclipse so the feature derives from there).

Similarly, the common application code for each application appears only once even though there are 5 builds for each application (for the 5 hal driver implementations).

The application code appears under the "ProjectName"_Common folder under the root MiniWin installation folder. These folders are linked to in the project files using the linked folder feature.

27.10 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 "ProjectName"_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.

27.11 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/NXP SDK and middleware for file handling and USB or SDIO 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).

28 Standard Controls Details

28.1 Button

Initialisation: User code 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.

28.2 Check Box

Initialisation: User code 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.

28.3 Keypad

Initialisation: User code 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’).

28.4 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 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 (Any visible ASCII character, ‘ ‘ or ‘\b’).

28.5 Label

Initialisation: User code 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.

28.6 List Box

Initialisation: User code 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.

28.7 Progress Bar

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

28.8 Radio Button

Initialisation: User code 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.

28.9 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.

28.10 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.

28.11 Arrow Button

Initialisation: User code 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.

28.12 Text Box

Initialization: User code 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.

29 Source Code Layout

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

MCUXpresso: This contains project files, BSP and application code specific to example applications that run on NXP processors and are built using the NXP MCUXpresso IDE. Under this folder is a sub-folder for each application example and under each of those a further sub-folder for each targeted NXP processor, currently only the LPC54628. In this folder are MCUXpresso project files, the required BSP for that application and under the `src` folder source files that are both application and hardware variant specific.

MiniWin: This contains the MiniWin embedded window manager source code which is common to all example applications for all platforms and processors. It contains the following sub-folders:

`bitmaps`	Bitmaps and their C99 file encodings used by MiniWin
`dialogs`	Standard dialogs provided by MiniWin
`docs`	All documentation

gl	The graphics library incorporated in MiniWin including fonts
hal	The hardware abstraction layer of drivers for all the currently supported processors and platforms
templates	Templates of required application files
ui	Standard user interface controls. User code can add to this for further controls if required

Folder **hal** contains source files common to all hal implementations and then further source files in sub-folder for different currently supported platforms and processors. Each set of drivers for a particular target builds only for that target. Other different target files compile to nothing through the use of #defines.

In addition the following source files are in the `MiniWin` 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_Common: This contains application source files for the comprehensive MiniWinTest example that are common to all platform/processor variants.

MiniWinSimple_Common: This contains the application example source files described later in this document.

MiniWinFixedWindows_Common: As above but for the fixed windows example project.

MiniWinRoot_Common: As above but for the root window example project.

MiniWinFile_Common: As above but for the file example project.

MiniWinTTFonts_Common: As above but for the TrueType font rendering example project.

MiniWinFreeRTOS: As above but for the FreeRTOS integration example project.

TrueStudio: This contains project files, BSP and application code specific to example applications that run on STM32 processors and are built using the Atollic TrueStudio IDE. Under this folder is a sub-folder for each application example and under each of those a further sub-folder for each targeted STM32 processor, currently the STM32F407 and STM32F429. In this folder are TrueStudio project files, the required BSP for that application and under the `src` folder source files that are both application and hardware variant specific.

In addition under the Atollic folder are the tool applications that are part of MiniWin that are built for Linux or Windows using Atollic TrueStudio IDE. These are all found under the `Tools` folder.

BMPConv24Colour: 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: 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.

CodeGen: This contains the MiniWin code generator utility. It is described later in this document.

FontEncoder: 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.

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

30 User Code Configuration

MiniWin can be configured by changing default settings found in header file `MiniWin/templates/miniwin_config.h_template`. 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 8 sections to this file:

30.1 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. User code can use timers for its own purposes as well.

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.

30.2 Display Rotation

In this section you can select the display rotation. There are 4 #defines for the supported 4 rotations. One and only one must be selected and the other 3

commented out. For example, the following configuration rotates the display 90 degrees to the right:

```
/* #define MW_DISPLAY_ROTATION_0 */
#define MW_DISPLAY_ROTATION_90
/* #define MW_DISPLAY_ROTATION_180 */
/* #define MW_DISPLAY_ROTATION_270 */
```

30.3 Sizes

Set 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.

30.4 User Interface Colours

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

30.5 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.

30.6 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
```

30.7 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
```

30.8 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.

31 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 handle, 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.

32 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.

33 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 paths to the `MiniWin` and `MiniWin/gl/fonts/truetype/mcufont` folders to your list of include folders to search in your IDE (or add it to the command line list if you are building on the command line).

1. Copy `/MiniWin/templates/miniwin_config.h_template` 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_template` example file from the `/MiniWin/templates` folder to your source folder and rename it `miniwin_user.c`. 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. Copy the appropriate `app.c_template, main.c_template` and `app.h_template` example files from the `/MiniWin/templates` folder to your source folder. Rename them `app.c, main.c` and `app.h`.

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.

8. Give your window's client area a background, as by default it has none. You need to draw a filled rectangle with no border 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);
}
```

9. 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;
}
```

10. 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.

11. 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 code 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)`.

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

34 MiniWin Code Generator

MiniWin contains a code generator that takes much of the legwork out of creating the user interface part of your project. Instead of creating the files for your windows and writing the code to create your controls you can instead specify what user interface components you want in a configuration file, run the generator, and you have a complete project created for you, including build files, the hal layer and any platform specific code for the Linux or Windows simulators. Once you are ready to move your project on to your embedded hardware you replace the Windows/Linux hal and platform specific code with that for your target device. The user interface code the generator creates is platform independent, so it will look and work the same on your target hardware as it does in the simulators.

The configuration file is written in JSON, a simple, widely used and human readable data transfer language (it's like a simpler XML). The JSON configuration file is parsed using an open source JSON parser and the generator produces well documented standard C99 code identical to that found in the example projects. A makefile is also produced either for Linux or Windows which can be used to build the generated code. There are many example JSON configuration files to look at, copy and build on. You can get your first MiniWin application running in the simulator to your design with your layout of controls in minutes. The generated code is clear and commented where you need to add your code to implement your application's behaviour. There are no opaque #defines in the generated code (remember the hideous code that MFC generated?) - it's all straight forward easy to understand C99.

34.1 Building the Code Generator

The code generator comes as source code, not binaries. The source code is found in the `CodeGen` folder. In this folder along with the source code are 2 makefiles, one for Windows and another for Linux.

If you have installed Atollic TrueStudio you have all the utilities necessary to build the code generator. For Windows only you must have your `PATH` environment variable set to the folder containing the `mingw32-make` application. See section 6 for details.

Following this, to build the CodeGen application for Windows use this command line:

```
mingw32-make -f makefile_windows
```

To build the application for Linux use this command line:

```
make -f makefile_linux
```

This will create you `CodeGen` for Linux or `CodeGen.exe` for Windows.

34.2 Running the Code Generator

The code generator is run with the following command lines for Linux…

```
./CodeGen <config_file.json>
```

and this command line for Windows...

```
CodeGen <config_file.json>
```

The code generator creates a new project folder (`<NewProjectName>` in the diagram below) parallel to the existing project folders under the TrueStudio folder. In this there will be a `Windows` or a `Linux` folder depending if you specified to build for the Windows or Linux simulator. There will be a `src` folder containing non-platform specific source code common to either a Linux or a Windows simulator build, and a common folder (`<NewProjectName>_Common` in the diagram below) folder containing source code specific to each build and a makefile.

The generated makefile expects the `MiniWin` folder to be available as in the diagram below. If you have a different arrangement you will need to modify the makefile paths yourself.

```
MiniWin/
      ...
TrueStudio/
      Tools/
            CodeGen/
                  CodeGen.exe or CodeGen
                  confile_file.json
            ...
      MiniWinFile/
      MiniWinDixedWindows/
      ...
      <NewProjectName>/
            Linux/ or Windows/
                  src/
                        app.c
                        app.h
                  makefile
            <NewProjectName>_Common/
                  main.c
                  ...
                  ...
```

After running the code generator using one of the example JSON file (or your own), if using the same output folder, it is better to delete the generator's output folder completely before generating code for a different example configuration file (for the supplied example configuration files the output folder is always called `MiniWinGen`). The reason for this is that the generated makefile for building your generated project uses wildcards to find its C source files. If there are any C source files left over from the previous generation that are not required for the subsequent one you may get build errors when building the generated code.

The generation process overwrites any previous output files when running the generator again to the same output folder. If you have hand edited any of the

generated files and want to keep your changes move your code somewhere else, or otherwise it will be overwritten when you run the generator again.

34.3 Example Code Generator Configuration Files

In the CodeGen folder is a collection of example configuration files. These are all in JSON format, which is human readable. These example files all generate a project by running the code generator followed by the JSON configuration file name, as described earlier in this section. All the examples create their output under a folder named `MiniWinGen`.

34.3.1 Building and Running Code Generator Examples for Linux

To build the project go to generated folder `MiniWinGen/Linux` and type

```
make
```

To run the example under the Linux simulator in folder `MiniWinGen/Linux` type...

```
./MiniWinGen
```

34.3.2 Building and Running Code Generator Examples for Windows

To build the examples for Windows you need to modify the example configuration JSON files and replace this line...

```
“TargetType”: “Linux”,
```

with this line, which specifies the Windows target type...

```
“TargetType”: “Windows”,
```

To build these examples go to the generated folder `MiniWinGen\Windows` and type

```
mingw32-make
```

To run the example under the Windows simulator in folder `MiniWinGen\Windows` type...

```
MiniWinGen
```

34.4 Code Generator Configuration File Format

Important: the JSON configuration file you pass to the code generator must be error-free valid JSON. A handy tip is before running the code generator with any JSON you have written, check that your JSON is well formed and error free by validating it with an on-line JSON validator. Here is an example that you can copy and paste your JSON in to and press validate...

```
https://jsonlint.com/
```

If you validation fails have a look at the Common Errors section of that webpage. The most common error is an extra comma where it should not be. You can get away with that in C but not in JSON.

If you are new to JSON, it is very simple and you will be able to grasp most of it in 10 minutes. It was invented for transferring data between webpages and servers, but ignore that bit. It's great for simple configuration files too. Here's a quick tutorial…

```
https://beginnersbook.com/2015/04/json-tutorial/
```

34.5 Configuration File Format Overview

This section describes the layout of the JSON configuration file. These examples are in pseudo JSON for clarity and are not valid JSON.

The top-level layout of the JSON configuration file is this…

```
{
	project wide setting
	array of windows
	[
		…
	]
}
```

There are multiple project wide settings and they all come at the same level.

Within the array of windows you can have multiple entries, one for each window in your project. Each window entry has multiple settings particular to that window…

```
{
	project wide setting 1,
	project wide setting 2,
	array of windows
	[
	{
		window 1 setting 1,
		window 1 setting 2,
		…
	},
	{
		window 2 setting 1,
		window 2 setting 2,
		…
	}
	]
}
```

You can also specify controls that belong to your windows (as in window 1 below), or have no controls for a particular window (window 2 below). Each window has its own list of controls for each type of control, for example buttons or labels, and you can have one or multiple controls of each type in a window (i.e. a window with 3 buttons), so these appear as arrays too…

```
{
	project wide setting 1,
	project wide setting 2,
	array of windows
	[
	{
		window 1 setting 1,
		window 1 setting 2,
		array of buttons
		[
		{
			...
		},
		{
			...
		}
		],
		array of labels
		[
		{
			...
		}
		]
	},
	{
		window 2 setting 1,
		window 2 setting 2
	}
	]
}
```

Each instance of each control type has settings too. These vary by control type. For example, a label just has location and text but a list box will have an array of entries as well as location and size. This completes the configuration file layout...

```
{
	project wide setting 1,
	project wide setting 2,
	array of windows
	[
	{
		window 1 setting 1,
		window 1 setting 2,
		array of buttons
		[
		{
			button 1 in window 1 setting 1,
			button 1 in window 1 setting 2
		},
		{
```

```
                button 2 in window 1 setting 1,
                button 2 in window 1 setting 2
        }
        ],
        array of labels
        [
        {
                label 1 in window 1 setting 1,
                label 1 in window 1 setting 2
        }
        ]
    },
    {
        window 2 setting 1,
        window 2 setting 2
    }
    ]
}
```

34.6 Configuration File Format Details

All JSON entries in the configuration file start with a key name in quotes followed by a colon. After the colon is an object for that key. The object may be a direct value (string, integer or boolean), an array of direct values, a sub-object in { } containing more key/object pairs, or an array of sub-objects each in { } containg more key/object pairs.

In the following sections each entry is described in the following format:

Key name
Optional or mandatory, with default value if optional
Setting purpose
Object type (direct value, array of direct values, sub-object or array of sub-objects)
Allowed values and any further information.

34.6.1 General Application Settings

`“TargetType”`
Mandatory
The target to generate the project makefiles for and which will be used to run the MiniWin simulator when the generated code is built. This also names the folder the target specific files will be in.
Direct value string.
`“Windows”` or `“Linux”` for the appropriate target operating system to build and run the MiniWin simulator.

`“TargetName”`
Mandatory
The name of the project to generate. This name is used for the folder where the generated files are placed.
Direct value string.
This must be a valid folder name for the target operating system.

“LargeSIze”
Optional defaulting to “false”
If the controls and window scroll bars for all windows and controls are to be created large sized or standard sized.
Direct value boolean.
“true” or “false”.

“MaxWindowCount”
Optional
The number of spaces to reserve in the array of windows.
Direct value number.
Defaults to number of specified windows + 2. Must be at least 1 greater than the number of windows that will be in concurrent use as root window always takes 1 space. Any dialog shown will take an additional 1 window.

“MaxControlCount”
Optional
The number of spaces to reserve in the array of controls.
Direct value number.
Defaults to number of specified controlss + 5. Any dialog shown will take additional control spaces, typically less than 5.

“MaxTimerCount”
Optional
The number of spaces to reserve in the array of timers.
Direct value number.
Defaults to 8. User interface components that are animated or show a cursor will need a timer each while they are shown.

“MaxMessageCount”
Optional
The number of spaces to reserve in the message queue.
Direct value number.
Defaults to 80. Larger systems may need more.

“CalibrateText”
Optional
The text to show on the start-up calibration screen.
Direct value string.
Defaults to “Touch centre of cross”.

“BusyText”
Optional
The text to show on when displaying that the user interface is busy.
Direct value string.
Defaults to “BUSY...”.

34.6.2 Window Settings

“Windows”
Mandatory

Details of MiniWin windows to generate code for.
Array of sub-objects.
This array must contain at least one entry which must be a sub-object.

“Name”
Mandatory
Name of this window which is used to create this window’s variable names.
Direct value string.
This must be a legal C99 variable name and must be different for every window.

“Title”
Mandatory
Title string displayed in the this window’s title bar.
Direct value string.
Any text. Text too long will be truncated when the window is created.

“X”
Mandatory
X position on the display the window will be created at.
Direct value integer.
Any positive integer less than the width of the display in pixels.

“Y”
Mandatory
Y position on the display the window will be created at.
Direct value integer.
Any positive integer less than the height of the display in pixels.

“Width”
Mandatory
Width in pixels the window will be created with.
Direct value integer.
Any positive integer but if the value is too small the window will fail to be created when the code runs. The minimum value depends on whether the window’s optional features like borders and title bar.

“Height”
Mandatory
Height in pixels the window will be created with.
Direct value integer.
Any positive integer but if the value is too small the window will fail to be created when the code runs. The minimum value depends on whether the window’s optional features like borders and title bar.

“Border”
Optional defaulting to “false”
If the window is created with a border.
Direct value boolean.
“true” or “false”.

“TitleBar”

Optional defaulting to “false”
If the window is created with a title bar.
Direct value boolean.
“true” or “false”.

“CanClose”
Optional defaulting to “false”
If the window is created with a greyed out window close icon.
Direct value boolean.
“true” or “false”.

“VerticalScrollBar"
Optional defaulting to “false”
If the window is created with a window vertical scroll bar.
Direct value boolean.
“true” or “false”.

“HorizontalScrollBar”
Optional defaulting to “false”
If the window is created with a window horizontal scroll bar.
Direct value boolean.
“true” or “false”.

“VerticalScrollBarEnabled”
Optional defaulting to “false”
If the window vertical scroll bar (if enabled by previous setting) is created enabled or disabled.
Direct value boolean.
“true” or “false”.

“HorizontalScrollBarEnabled”
Optional defaulting to “false”
If the window horizontal scroll bar (if enabled by previous setting) is created enabled or disabled.
Direct value boolean.
“true” or “false”.

“MenuBar”
Optional defaulting to “false”
If the window is created with a menu bar.
Direct value boolean.
“true” or “false”.

“MenuBarEnabled”
Optional defaulting to “false”
If the window is created with a menu bar this controls the menu bar’s global enable flag on window creation.
Direct value boolean.
“true” or “false”.

“MenuItems”

Mandatory if menu bar is enabled for this window, else superfluous.
Labels for the menu bar items.
Direct value string array.
The number of entries in the array specifies the size of the menu.

“Visible”
Optional defaulting to “false”
If the window is created visible or hidden.
Direct value boolean.
“true” or “false”.

“Minimised”
Optional defaulting to “false”
If the window is created minimised.
Direct value boolean.
“true” or “false”.

“Buttons”, “Labels”, “ScrollBarsVert”, “ScrollBarsHoriz”, “RadioBoxes”, “ListBoxes”, “ScrollingListBoxes, “CheckBoxes”, “Arrows”, “ProgressBars”, “CrollingListBoxes” or “TextBoxes”
Optional defaulting to no entries
Button controls belonging to this window.
Array of sub-objects.
See description of sub-object in next section.

34.6.3 Control Settings

All controls have the following settings so these are not itimised in the individual descriptions.

“Name”
Mandatory
The name of this control.
Direct value string.
This value is used to create the variable names used in the C99 code to refer to this control. The value must be a legal C99 variable name and must be unique for all controls of this type across the project.

“X”
Mandatory
X position on the display the control will be created at.
Direct value integer.
Any positive integer less than the width of the display in pixels.

“Y”
Mandatory
Y position on the display the control will be created at.
Direct value integer.
Any positive integer less than the height of the display in pixels.

“Visible”
Optional defaulting to “false”

If the control is created visible or hidden.
Direct value boolean.
“`true`” or “`false`”.

“`Enabled`”
Optional defaulting to “`false`”
If the control is created enabled or greyed out.
Direct value boolean.
“`true`” or “`false`”.

34.6.3.1 Button

“`Label`”
Mandatory
Text of the label displayed within the button.
Direct value string.
If the text is too long it is truncated when the button is created.

34.6.3.2 Label

“`Label`”
Mandatory
Text displayed within the label.
Direct value string.
If the text is too long for the label width it is truncated when the label is created.

“`Width`”
Mandatory
Width in pixels of the space for the label to be displayed in.
Direct value integer.
If the width is too short for the label the label it is truncated when the label is created.

34.6.3.3 Vertical Scroll Bar

“`Height`”
Mandatory
Height in pixels of a vertical scroll bar.
Direct value integer.
This is for a control scroll bar, not a window scroll bar.

34.6.3.4 Horizontal Scroll Bar

“`Width`”
Mandatory
Width in pixels of a horizontal scroll bar.
Direct value integer.
This is for a control scroll bar, not a window scroll bar.

34.6.3.5 Radio Buttons

“`Width`”
Mandatory
Width in pixels for the buttons and their labels.
Direct value integer.

Labels too long for the width will be truncated when the control is created.

“Labels”
Mandatory
The labels for the radio buttons.
Direct value string array
The number of entries in the array specifies the number of radio buttons to show.

34.6.3.6 List Box

“Width”
Mandatory
Width in pixels for the list box.
Direct value integer.
Labels too long for the width will be truncated when the control is created.

“Lines”
Mandatory
Number of lines hight the list box will be created. This value is not in pixels.
Direct value integer.
The number of lines should be greater than or equal to the number of labels.

“Labels”
Mandatory
The labels for the list box lines.
Direct value string array
The number of entries in the array can be more or less than the number of lines but excess labels will not be shown.

34.6.3.7 Scrolling List Box

“Width”
Mandatory
Width in pixels for the list box.
Direct value integer.
Labels too long for the width will be truncated when the control is created.

“Lines”
Mandatory
Number of lines hight the list box will be created. This value is not in pixels.
Direct value integer.
The number of lines can be less than the number of labels as scrolling is provided.

“Labels”
Mandatory
The labels for the list box lines.
Direct value string array
The number of entries in the array can be more or less than the number of lines as the entries can be scrolled.

34.6.3.8 Arrow

“Direction”

Mandatory
Direction of arrow to draw with the arrow button.
Direct value string.
“Up”, “Left”, “Down” or “Right”.

34.6.3.9 Check Box

“Label”
Mandatory
Text displayed alongside the check box.
Direct value string.
If the text is too long for the allowed width it is truncated when the check box is created.

34.6.3.10 Progress Bar

“Width”
Mandatory
Width in pixels of a progress bar.
Direct value integer.
No further details.

“Height”
Mandatory
Height in pixels of a progress bar.
Direct value integer.
No further details.

34.6.3.11 Text Box and Scrolling Text Box

“Width”
Mandatory
Width in pixels of a text box.
Direct value integer.
No further details.

“Height”
Mandatory
Height in pixels of a text box.
Direct value integer.
No further details.

“Justification”
Mandatory
Justification used to render the text.
Direct value string.
“Left”, “Right”, “Centre” or “Full”.

“BackgroundColour”
Mandatory
The background colour the text will be rendered using.
Direct value string.

This can be a value defined in `hal_lcd.h` or a C99 hex constant, for example `0xFF00FF`.

`“ForegroundColour”`
Mandatory
The foreground colour the text will be rendered using.
Direct value string.
This can be a value defined in `hal_lcd.h` or a C99 hex constant, for example `0xFF00FF`.

`“Font”`
Mandatory
The TrueType font data structure used to render the text.
Direct value string.
This must be a the C99 name of a data structure already available in your project as created by the font encoder tool.

34.7 Code Generator Error Messages

`Usage: CodeGen <config file>`
You have typed the command line incorrectly.

`Could not open input file`
The JSON configuration file specified could not be opened.

`There was a JSON parsing error: xxx`
The JSON configuration file contains badly formed JSON. Use a JSON verifier website to find out the problem.

`No target type specified`
The JSON configuration file does not specify a target type.

`Target type not supported`
The JSON configuration file does not specify the target types as Windows or Linux.

`No target name specified`
The JSON configuration file does not specify a target name. This value is used to create the generated files output folder.

`No windows found`
The JSON configuration file does not specify at least one window.

`Not enough space for all the specified windows.`
You have specified more windows than the value in maximum window count.

`Could not make ...`
Could not make an output folder. Either the target name is illegal for the OS or you do not have permission.

`Could not create file ...`
Could not make an output file. Either the target name is illegal for the OS or you do not have permission.

`Could not find ... Continuing anyway. You will need to supply your own ...`
The MiniWin source tree is not in a folder parallel to where you are running the code generator and hence the template files could not be found. Generation will continue but you will have to provide or manually copy the template files.

`No font name specified for text box ...`
Mandatory text box vaue missing

`No (or blank) ... value for window`
Mandatory value for a window is missing, blank or in the wrong format.

`Menu bar specified for window ... but no menu items specified`
You specified a menu bar for a window but didn't include the menu bar items setting.

`No (or blank) ... for ... in window ...`
Mandatory value for a control is missing, blank or in the wrong format.

`Unknown/Unrecognised value for ...`
A control value was supplied, in the correct format, not blank but is not an allowed value.

`Duplicate identifier found`
You have given 2 windows the same name or two controls of the same type the same name.

35 Known Issues

Some of the example applications are laid out badly for the LPC54628 hardware variant development board, in particular the fixed windows examples. While this example compiles and runs for this hardware variant it can't really be used. If you are using this development board the fix is to change display rotation from 0 degrees to 90 degrees in this project's configuration file.

Only ASCII characters supported at the moment. This will be fixed soon.

The font handling is a bit disorganised with functionality being added in a manner that has made the API inconsistent, for example TrueType fonts cannot be rotated or drawn transparently.

The gl library requires features from ISO C99 that are supported in newlib but not in redlib. This means that the larger newlib library has to be used to build MiniWin applications. This will be fixed if there is any demand for it.

36 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.

NXP SDK driver library and BSP

FatFS, version R0.12c, copyright ChaN 2017

FreeRTOS, version 10 published by Amazon

mcufont, Petteri Aimonen

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

JSON11

37 Glossary of Terms

Operator - 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 and start-up code.

Middleware – A library that is used by an application but does not interact with the hardware directly, instead using the BSP to access the hardware. An example is the FatFS file system library.

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 operator 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 operator input. All MiniWin standard dialogs are modal.

Handle – a reference to a MiniWin resource created by MiniWin, 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.

JSON – a simple human readable text based configuration file format.

38 Appendix 1

This section lists flags that the user can set when creating windows or controls. These flags should only be set when creating a window or control. Some of the flags are not intended to be set by the user. These are also listed in the following sections.

38.1 Window Flags

38.1.1 User Settable Flags

```
MW_WINDOW_FLAG_HAS_BORDER
Indicates that a window is to be created with a border.

MW_WINDOW_FLAG_HAS_TITLE_BAR
Indicates that a window is to be created with a title bar.

MW_WINDOW_FLAG_CAN_BE_CLOSED
Indicates that a window is to be created with an enabled close icon.

MW_WINDOW_FLAG_IS_VISIBLE
Indicates that a window is to be created visible.

MW_WINDOW_FLAG_IS_MINIMISED
Indicates that a window is to be created minimised.

MW_WINDOW_FLAG_IS_MODAL
Indicates that a window is to be created modal.

MW_WINDOW_FLAG_HAS_VERT_SCROLL_BAR
Indicates that a window is to be created with a vertical scroll bar.

MW_WINDOW_FLAG_HAS_HORIZ_SCROLL_BAR
Indicates that a window is to be created with horizontal scroll bar.
```

MW_WINDOW_FLAG_HAS_MENU_BAR
Indicates that a window is to be created with a menu bar.

MW_WINDOW_FLAG_MENU_BAR_ENABLED
Indicates that a menu bar, if existing, is to be created enabled.

MW_WINDOW_FLAG_VERT_SCROLL_BAR_ENABLED
Indicates that a vertical scroll bar, if existing, is to be created enabled.

MW_WINDOW_FLAG_HORIZ_SCROLL_BAR_ENABLED
Indicates that a horizontal scroll bar, if existing, is to be created enabled.

MW_WINDOW_FLAG_TOUCH_FOCUS_AND_EVENT
Indicates that a touch in a non-focused window gives it focus and generates a touch down event.

MW_WINDOW_FLAG_LARGE_SIZE
Indicates that a window's menu bar and scroll bars are to be created large sized.

38.1.2 Non-User Settable Flags

MW_WINDOW_FLAG_IS_USED
Indicates that a window is used.

MW_WINDOW_FLAG_MENU_BAR_ITEM_IS_SELECTED
Indicates that a menu bar, if existing, has an item selected.

38.2 Control Flags

38.2.1 User Settable Flags

MW_CONTROL_FLAG_IS_VISIBLE
Indicates that a control is to be created visible.

MW_CONTROL_FLAG_IS_ENABLED
Indicates that a control is to be created enabled.

MW_CONTROL_FLAG_LARGE_SIZE
Indicates that a control is to be created large size.

38.2.2 Non-User Settable Flags

MW_CONTROL_FLAG_IS_USED
Indicates that a control is used. This flag is controlled by the window manager.

39 Appendix 2

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

39.1 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: Unused

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: Unused

39.2 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

39.3 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: Unused
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

39.4 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

39.5 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 meaning

MW_USER_3_MESSAGE
Message to a window for any user-defined purpose
message_data: Any user meaning
message_pointer: Any user meaning

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

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