MiniWin Embedded Window Manager
Version 1.0
30 January 2018

1   Introduction

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

2   Who is MiniWin For?

- Open source projects that need a quick-start user interface

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

3 Features

- Written specifically for small embedded systems with a LCD display and a touch screen
- Apart from a small hardware abstraction layer, platform independent
- Supports multiple overlapped windows with Z ordering
- Incorporates a flexible graphics library
- Comes with a basic set of user interface controls
- No dynamic memory used - all data structures allocated at compile time
- Two fonts included - fixed width and proportional
- A clean easy to use API
- Comprehensive documentation and example code
- Runs in bare metal systems or within a single thread of a RTOS
- Requires minimal memory - no off-screen buffering used
- Compiles without warning with GCC
- Doxygen documentation for every function and type

4 User Interface Controls

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

These are the standard ones. You can easily add more.

5 MiniWin Windows

- Optional title bar, title and border
- Overlapped with unique Z order
- Movable
- Resizable
- Maximise, minimise and close icons
- Minimisable to icon on desktop

- Customisable desktop colour or bitmap
- Can have a single system modal window
- Window-aware graphics library for drawing in a window

6   Developer Utility Applications

MiniWin includes applications (including source that can be compiled for Windows or Linux) to convert monochrome bitmaps in Microsoft Windows .bmp file format to C99 source code and similarly to convert 24 bit .bmp format files to 16 bit C99 source code.

A third utility converts back a text file containing 16 bit hex numbers to a Microsoft Windows 24 bit .bmp file.

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

7   Drivers and Examples

MiniWin comes with all source for the window manager, graphics libraries and user interface components. It also comes with 2 example projects - a simple getting started example and a fully comprehensive project showing usage of all MiniWin's features.

MiniWin comes with two hal layer samples - and embedded one for the STM32F407 and another that allows MiniWin to run within a Microsoft Windows window. Further hal layer examples are under development. Sample projects are built with the free compiler/IDE Atollic TrueStudio.

8   Customisability

Because MiniWin is fully open source its look and feel is fully customisable. Don't like the look of a user interface component? Simple. Modify its paint function and draw it how you want it to look.

This is the easy part. The difficult part of what to repaint and when is done for you by the window manager and won't need changing.

9   Implementing the Hardware Abstraction Layer (HAL)

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

9.1   Delay Driver

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

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. Do not use these functions from within your user code as you will block the responsiveness of your user interface. Use the MiniWin tick timer and message handlers instead.

Function: `mw_hal_delay_init`

Any hardware initializations for the delay driver are here.

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

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

9.2   Init Driver
Source file: `hal_init.c, hal_init.c`

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

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

9.3   Timer Driver
Source file: `hal_timer.c, hal_timer.c`

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

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

9.4   Non-Volatile Storage Driver
Source file: `hal_non_vol.c, hal_non_vol.h`

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

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

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

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

9.5   Touch Driver
Source file: `hal_touch.c, hal_touch.c`

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

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

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

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

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

9.6 LCD Driver
Source file: `hal_lcd.c, hal_lcd.h`

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

Type: `mw_hal_lcd_colour_t`
This typedef represents the colour format of the display device, for example a 16 bit colour depth device uses a uint16_t colour typedef, a monochrome display device a bool.

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

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

Function: `mw_hal_lcd_init`
This function initializes the display device hardware.

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

Function `mw_hal_lcd_filled_rectangle_clip`

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

Function `mw_hal_colour_bitmap_clip`
This function fills a full colour rectangle on the screen with data from bitmap specified by the caller. The bitmap data must be in the format of the colour type, for example if the colour type is 16 bit then the data format must be two bytes per pixel. The image data must start at the top left and proceed across a full row before starting the next row. The co-ordinates of all points within the specified rectangle are clipped to the screen maximum co-ordinates and also to a caller specified rectangle width and height.

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

10 Paint Algorithm

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

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

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

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

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

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

11 Z Orders

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

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

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

12 Graphics Library

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

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

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

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

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

be drawn where the origin is within the area being painted but the end point is outside the boundary the line will be drawn from the origin up to the boundary.

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

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

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

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

13 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.

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 message handler and paint functions have the same function parameters as their window functions equivalents.

Each control has its own data structure storing fields common to all controls, for example visibility, currently used and enabled status. 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 extra data unique to that control. For example, a button has a single label but a list box has an array of labels and the array size. Each extra data structure is declared in the control's header file and the standard control common data structure contains a void pointer to reference it. The fields in the control-specific data structure are initialized when the control is created. Some fields are user modifiable and some are not; the non-user-modifiable fields contain control state and should not be changed by the user. Each

control-specific data structure indicates which fields are user modifiable and which should not be changed in the structure's declaration it the control's header file.

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

14 Enabling and Disabling Controls

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

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

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

15 MiniWin Messages

The MiniWin window manager works using a message queue for requests for actions from user code or reporting of events to user code. These actions and events happen asynchronously. This means that when a request is made or an event occurs a message is posted to the window manager's message queue and the window manager processes this message at a later time. User code must continually call the window manager's message processing function to get anything done. If the user fails to call the message processing function the user interface will not respond to input and not repaint any windows or controls. The reason for this design pattern is to keep MiniWin running in a single thread and the code simple. Programmers with long memories will remember that this is the design pattern used by 16 bit Microsoft Windows up to Windows 3.1.

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

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

15.1 Messages posted by the window manager

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

15.2 Messages posted by controls in response to user interface input

When the user interacts with a control the control code will post a message letting user code know what has happened. These messages are not posted from user code unless the user needs to create new types of controls.

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

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

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

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

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

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

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

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

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

16 Transferring Data to Message Handler Functions

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

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

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

17 Touch Events

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

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

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

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

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

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

18 Painting and Client Areas

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

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

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

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

19 Automatic Painting and User Painting

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

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

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

20 Scroll Bars

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

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

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

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

21 Combining Controls

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

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

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

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

22 User Window Interaction

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

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

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

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

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

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

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

23 MiniWin Example Projects

MiniWin comes with 3 example projects. These are:

23.1 MiniWinTest

This is a comprehensive example using all of MiniWin's features. It has an implemented hal layer for a STM32F407 processor with an IL9325 LCD QVGA colour LCD and an ADS7843 touch screen controller.

23.2 MiniWinTest_Win

This uses the same code base as the previous example apart from the hal layer which simulates touchscreen/LCD panel hardware in Microsoft Windows. This project contains no source code apart from its own `main.c`. Instead it uses links to the code in the previous example.

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

The Windows example project is a console application even though it draws graphics to the console window. To run it on Windows 10 you need to start a cmd command prompt, under properties untick all the Edit Options and Text Selection check boxes, re-start your command prompt and then run the example application from the command line. Running the example application from Windows Explorer or Atollic TrueStudio will not work.

If you wish to debug this Windows application build it for debug, run it as described and then in Atollic TrueStudio set up a debug profile to attach to process.

23.3 MiniWinSimple

This project contains the example code as described in detail later in this document. It uses the same hal layer as the first example.

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

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

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

window_ok_cancel.c
This window demonstrates a modal window with 2 buttons which can be used as a pattern for a simple warning or question dialog box. The window cannot be closed from the title bar or any other window switched to while it is showing.

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

window_scroll.c

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

window_yield.c

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

window_test.c

This window demonstrates all user interface controls (except buttons, see window_ok_cancel.c) 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.

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

All the example projects share the same `user/MiniWin_config.h` header file for convenience. For your projects you can use your own customized version.

24 Standard Controls Details

24.1 Button

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

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.

24.2 Check Box

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

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

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

24.3 Integer Number Chooser

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

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

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

24.4 Keyboard

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

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

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

24.5 Label

Initialisation: User needs to set the label text.
Resources required: None

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

24.6 List Box

Initialisation: User needs to set the number of lines the scroll box has, the array of label strings and the bit field identifying which items in the list box are enabled or disabled.

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.

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

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

24.7 Progress Bar

Initialisation: User needs to set the progress percentage.
Resources required: None.

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

Messages sent: None.

24.8 Radio Button

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

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

24.9 Horizontal Scroll Bar

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

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

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

24.10 Vertical Scroll Bar

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

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

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

25 Source Code Layout

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

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

BMPConv16Colour: This is a command line utility for converting a Windows 24 bit per pixel .bmp file to a 16 bit per pixel C source file. It uses EasyBMP.

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

BMPConvBack16Colour: This is a command line utility for converting a text file containing pixels in hexadecimal 16 bit per pixel to a Windows 24 bit per pixel .bmp file. It uses EasyBMP

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

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

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

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

calibrate.h/c	Third party touch screen calibration routines
MiniWin_debug.h/c	Implementation of assert functionality, debug build only
MiniWin_message_queue.h/c	Simple message queue code for MiniWin messages
MiniWin_settings.h/c	Simple non-volatile storage routines for settings
MiniWin_touch.h/c	Interface between touch driver code under hal and the touch client code in MiniWin
MiniWin_utilities.h/c	Generic utility routines
MiniWin.h/c	The main window manager code

MiniWinTest: This contains the main example project targeted for a STM32F407. It does not contain its own copy of the MiniWin source tree, instead it links to it.

MiniWinSimple: This contains the sample code described later in this document targeted for a STM32F407. It does not contain its own copy of the MiniWin source tree, instead it links to it.

MiniWinTest: This contains the same example project as MiniWinTest but instead targeted for Microsoft Windows. It does not contain its own copy of the MiniWin source tree, instead it links to it. It also does not contain source files for example code, instead linking to the sample files in MiniWinTest project.

26 User Configuration

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

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

26.2 Sizes

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

26.3 User Interface Colours

Customize the look of your windows here. The colours are defined in the LCD HAL layer as different systems might use different colour depth displays.

26.4 Timings

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

26.5 Other

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

27 Quick Start Guide

This section describes how to create a single window with 2 standard controls: a button and a label. Pressing the button changes the text of the label and touching the window's client area outside the controls draws a circle using gl on the client area. Before you can do this you need to implement you HAL functions.

Once you have implemented you hal functions you need to create a new project and point it to the MiniWin source.

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

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

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

2. Create a new source file for your window called `window_simple.c`. Add this data structure declaration and definition to your source file. It stores data used by the window:

```
#include <stdint.h>
#include <stdbool.h>
#include "MiniWin.h"
#include "ui_common.h"

typedef struct
{
    uint16_t circle_x;              // x coordinate of where to draw circle
    uint16_t circle_y;              // y coordinate of where to draw circle
    bool draw_circle;       // if to draw circle
    uint8_t timer_id;        // timer id for button animation
    char transfer_buffer[10];     // buffer to send data to label
} window_simple_data_t;

static window_simple_data_t window_simple_data;

extern uint8_t button_id;
extern uint8_t label_id;
extern mw_window_t mw_all_windows[MW_MAX_WINDOW_COUNT];
```

3. Add stub functions to window_simple.c

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

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

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

```
#include "window_simple.h"

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

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

5. Add the following root paint function to MiniWin_user.c:

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

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

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

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

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

    mw_util_safe_strcpy(label_data.label,
                    MW_UI_LABEL_MAX_CHARS, "NOT SET");
    label_id = mw_ui_label_add_new(100,
                5,
                2 * MW_UI_LABEL_X_OFFSET + 11 *
MW_GL_STANDARD_CHARACTER_WIDTH,
```

```
                window_simple_id,
                MW_CONTROL_FLAG_IS_VISIBLE |
            MW_CONTROL_FLAG_IS_ENABLED,
                &label_data);

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

        mw_paint_all();
}
```

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

```c
#include "MiniWin.h"

int main(void)
{
        mw_init();

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

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 like this, in window_simple_paint_function(). As for the root window paint function just pass on the mw_gl_draw_info_t parameter to any gl function that needs it:

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

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

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

9. Now start adding some message handling in your message handler, 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_id);
      break;
```

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

11. Finally add code to respond to the button press message. When this message is received you want to send a new text string to the label control to change its text and then get it repainted:

```
case MW_BUTTON_PRESSED_MESSAGE:
      if (message->sender_id == button_id)
      {
            mw_ui_common_post_pointer_to_control(label_id, "HELLO WORLD");
            mw_paint_control(label_id);
      }
      break;
```

12. That's it. Build and run. You'll find this example's source code under the MiniWinSimple folder.

28 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

29 Glossary of Terms

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

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.