IBEX
MONOCHROME GRAPHIC DISPLAY SOURCE CODE DRIVER PROJECT

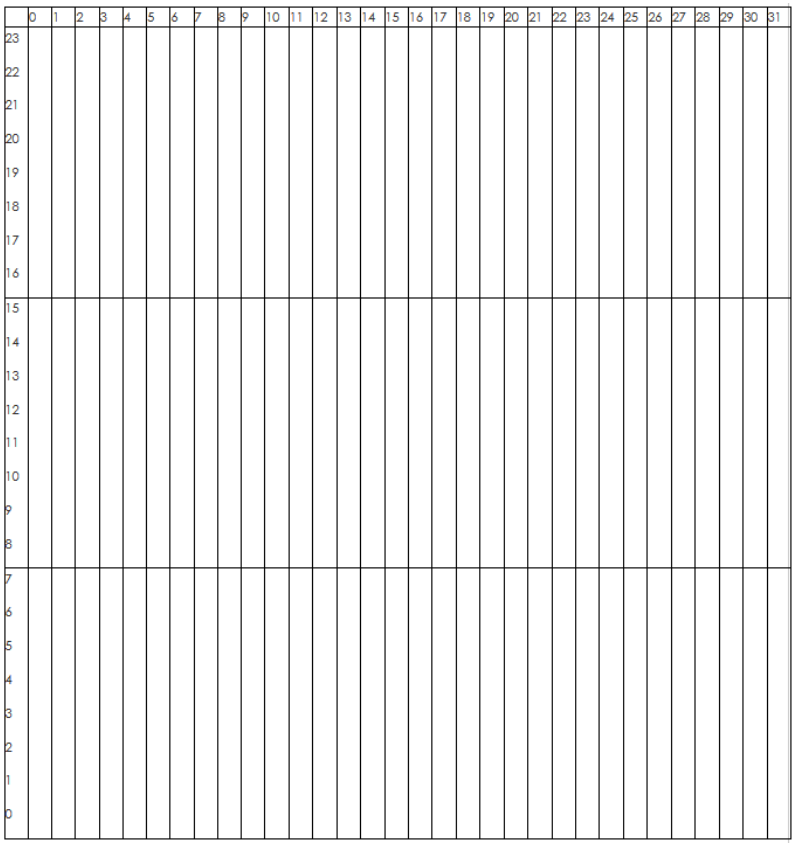
CONTENTS

DRIVER OVERVIEW

GRAPHIC SCREENS OVERVIEW

A typical monochrome graphic screen, which may use LCD, E Ink, LED, OLED, Vacuum Fluorescent or some other screen technology, consists of a large array of pixels that are triggered using a X and Y axis matrix of connections. The actual drive of each pixel is provided by a display controller IC. This may already be built into your screen, may need to be provided by you externally or may be built into the particular microcontroller or processor you are using. This display controller IC will constantly scan the matrix of pixels, turning on or off each pixel based on the data held in the IC's memory. In its basic form the IC will have a data buffer to which you write data to tell the controller IC which pixels you want on and which you want off. This driver is designed to be used with any screen which can be controlled in this way, allowing its use with the simplest and also more sophisticated controllers.

The problem with displaying graphics and text on a monochrome screen is that you typically need to transfer pixel data as bytes which means that either the X or the Y axis of your screen is being addressed in 8 bit multiples. For example:-



In the above example the layout of a 32 x 24 pixel screen is shown. The controller IC has a data memory buffer that is laid out say as follows:-

Address, Pixels
0 Column 0, Rows 0:7
1 Column 0, Rows 8:15
2 Column 0, Rows 16:23
3 Column 1, Rows 0:7
||
95 Column 31, Rows 16:23

If you want to turn on the top left pixel you would write 0x01 to address 2 – easy enough. If you want to display a bitmap that you have in processor memory that is 8 bits high by 10 bits wide to be displayed with its bottom left corner 8 rows up and 5 columns in then no problem, your working with the screen and its layout. However, this is of course very limiting and as soon as you want to be able to display a bitmap that isn't sized in 8 bit multiples in the byte based axis, or want to be able to display bitmaps at any position on the screen then things get a whole lot more complex. Even worse, what if you need to use the screen rotated by 90º? As you can see, having to work with a graphic screen and this byte addressing can severely limit what you are able to do with the screen. This driver solves this problem and allows you to display bitmap graphics and text of any size anywhere on the screen.

DRIVER INTRODUCTION

Graphic screens look impressive and allow you to display images that a simple character based screen can't provide. However this improvement comes at the cost of a great deal of complexity. Once configured for your screen, this driver removes that complexity for you and allows you to simply display bitmaps of any size and ASCII text strings using user adjustable fonts anywhere on a screen. Monospace or proportional width text with left, centre or right alignment and scrolling functions provides the complete toolbox of text display capabilities. Once you have these capabilities using a graphic screen is easy and it's a simple matter to add any special user interface features you require for your application, such as menus, animated graphics, graphs and more. One of the great advantages of this driver is that it provides you with very simple but very

powerful control over everything you display on the screen, allowing you to implement your display, and any user interface, exactly as you want it down to the individual pixel.

This driver provides complete bitmap and string display capabilities on any standard mono graphic display with a built in or external 8 bit display controller IC. It may also be used with virtually any mono pixel display system, such as LCD, E Ink, LED, OLED, Vacuum Fluorescent, or others. Once configured, the driver removes you from all of the problems of byte updating of displays. The driver can optionally use the display controller IC memory buffer for the display data rather than processor memory, resulting in a very small RAM memory requirement for the driver for screens which provide read as well as write access.

The included bitmap converter PC application will read all of your source monochrome .bmp bitmap files, created in a standard graphics application, and convert them into the selected format ready for use in your application. It will also convert special font bitmap files to be used to display ASCII text strings. Options allow you to rotate your bitmap images and fonts before converting, allowing you to use a screen in any of the 4 possible orientations.

As well as displaying ASCII text strings in standard 'monospace' fixed pitch mode (each character takes the same horizontal space regardless of its actual width), this driver also allows you to display text in proportional mode. This makes lines of text much more attractive and natural to read and typically allows you to fit additional characters across a screen as each character only uses as much width as is required to display it. Left, centre and right text alignment options remove you from the pain of having to manually calculate text offsets to make your text appear where you want it on the screen. Scrolling text lets you create impressive effects or display more information on a small display than would fit as a static line.

This driver can't quite remove all of the complexity for you as you will need to configure the basic screen access functions to suit the particular screen / display controller IC you're using. This has been made as simple as possible, and some sample screen models are included for you to copy and modify as required.

The driver code has been designed using ANSI compliant C compliers. Using the driver with other ANSI compliant C compliers and with other processors / microcontrollers should not present significant problems, but you should ensure that you have sufficient programming expertise to carry out any modifications that may be required to the source code. Embedded-code.com source code is written to be very easy to understand by programmers of all levels. The code is very highly commented with no lazy programming techniques. All function, variable and constant names are fully descriptive to help you modify and expand the code with ease.

The driver and associated files are provided under a licence agreement. Please see www.embedded-code.com/licence.php for full details.

The remainder of this manual provides a wealth of technical information about the driver as well as useful guides to get you going. We welcome any feedback on this manual and the driver.

FEATURES

Display bitmaps of any size anywhere on the screen without having to worry about 8 bit size multiples or 8 bit boundaries.

Use your display in any of the 4 possible orientations.

Display monospace or proportional width text anywhere on the screen.

Left, centre or right text alignment and scrolling options.

Use the included PC Bitmap Converter Application to convert your source bitmap (.bmp) files ready for use with the driver. Create your bitmaps using a standard graphics application.

Create your own fonts simply by creating a line of characters in a single bitmap file.

Output your bitmap source files as a C compliant header file or as a binary file using the Bitmap Converter Application. Use the header file option to include all of your bitmaps as part of the program memory of your

microcontroller / processor – convert the files and they are immediately ready to use in your project. Use the binary file option to store your bitmaps as a single data block in external flash memory and provide the driver with a function to read the binary data byte by byte. Use both methods if desired to split storage between on chip and external.

No requirements to use your display in pre-defined ways – use your display exactly as you want with your user interface implemented exactly as you want it, down to the individual pixel.

Use display memory instead of microcontroller / processor memory to buffer the display data, resulting in a very low ram requirement. Use a local ram buffer for displays with no read capability or to improve speed.

Small program memory requirement.

ADDING THE DRIVER TO YOUR PROJECT

NOTES ABOUT OUR SOURCE CODE FILES

How We Organise Our Project Files

There are many different ways to organise your source code and many different opinions on the best method! We have chosen the following as a very good approach that is widely used, well suited to both small and large projects and simple to follow.

Each .c source code file has a matching .h header file. All function and memory definitions are made in the header file. The .c source code file only contains functions. The header file is separated into distinct sections to make it easy to find things you are looking for. The function and data memory definition sections are split up to allow the defining of local (this source code file only) and global (all source code files that include this header file) functions and variables. To use a function or variable from another .c source code file simply include the .h header file.

Variable types BYTE, WORD, SIGNED_WORD, DWORD, SIGNED_DWORD are used to allow easy compatibility with other compilers. Our projects include a 'main.h' global header file which is included in every .c source code file. This file contains the typedef statements mapping these variable types to the compiler specific types. You may prefer to use an alternative method in which case you should modify as required. Our main.h header file also includes project wide global defines.

This is much easier to see in use than to try and explain and a quick look through the included sample project will show you by example.

Please also refer to the resources section of the embedded-code.com web site for additional documentation which may be useful to you.

Modifying Our Project Files

We may issue new versions of our source code files from time to time due to improved functionality, bug fixes, additional device / compiler support, etc. Where possible you should try not to modify our source codes files and instead call the driver functions from other files in your application. Where you need to alter the source code it is a good idea to consider marking areas you have changed with some form of comment marker so that if you need to use an upgraded driver file its as easy as possible to upgrade and still include all of the additions and changes that you have made.

STEP BY STEP INSTRUCTIONS

Move The Main Driver Files To Your Project Directory

The following files are the main driver files which you need to copy to your main project directory:

display.c – The main driver functions

display.h

display-model.c – The lower level functions tailored to a specific screen

display-model.h

If the driver includes the display-model files for your specific screen they should be copied from that screens sub-directory, or alternatively see later in this manual for details on creating your own version for a new screen model.

Move The Generic Global Defines File To You Project Directory

The generic global file is located in each driver sample project directory. Select the most suitable sample project based on the compiler used and copy the following file to your main project directory:

main.h – The embedded-code.com generic global file

Check Driver Definitions

Check the definitions in each of the following files to see if any need to be adjusted for the microcontroller / processor you are using, and your hardware connections.:-

display.h

display-model.h

Check the definitions in the following file and adjust if necessary for your compiler-

main.h

Application Requirements

In each .c file of your application that will use the driver functions include the 'display.h' file.

Your application needs to call the following function as part of its power-up initialisation:

```
display_initialise();
```

The display functions may now be called whenever you wish to update the display.

Bitmap Files

If using a C header file to store your bitmap files as part of the program memory then convert the files using the included bitmap converter application and store the cbitmaps.h output file in the following sub directory off your projects directory:

[your project directory]\bitmaps\output\

(You can simply store all of your source bitmap files in the subdirectory called [your project directory]\bitmaps\ and the bitmap converter application will then automatically create and store the output file in the \output\ sub directory)

If using a binary file to store your bitmap files then convert the files using the included bitmap converter application and provide the means to store them in your flash memory device. You will also need to provide the DISPLAY_FLASH_READ_FUNCTION function to allow the driver to read the files.

See later in this manual for details of how to create the bitmap output file from your source bitmaps.

SAMPLE SCREEN FILES AND SAMPLE PROJECTS

GENERAL NOTES

Sample display-model.h screen files are included for several screens. If using a matching screen they can be used directly or if using a different screen type they can be used as a basis for your new screen. See later in this manual for details on creating your own version for a new screen model.

Sample projects are included with the driver for specific devices and compilers. The example schematics at the end of this manual detail the circuit each sample project is designed to work with. You may use the sample projects with the circuit shown or if desired use them as a starting block for your own project with a different device or compiler. To use them copy all of the files in the chosen sample project directory into the same directory as the driver files and then open and run using the development environment / compiler the project was designed with.

SAMPLE SCREEN FILES INCLUDED

Sample display_model files are included for each of the following screens:-

BATRON BTHQ128064 AVE FETF

White on Blue 128 x 64 pixel LCD screen.

Uses driver SED1565D0B

Parallel interface

Available from Farnell InOne Stock Code 122-0418

BATRON BTHQ128064 AVD COG

Black on white 128 x 64 pixel LCD screen.

Uses driver S1D10605D04B

Parallel interface

Available from Farnell InOne Stock Code 944-9396

BATRON BTHQ240064AVB-EMN-06-LED White-COG

240 x 64 pixel LCD screen.

Uses 2x driver S1D10605D04B

SPI bus transmit only serial interface

Available from Farnell InOne Stock Code 944-9434

BATRON BTHQ100032V

100 x 32 pixel LCD screen.

Uses 2x driver SED1520

Parallel interface

Available from Farnell InOne Stock Code 944-9329

See later in this manual for details on creating your own version for a new screen model.

SAMPLE BITMAP FILES INCLUDED

Sample Font Table Bitmaps

font_05x05.bmp

font_05x07.bmp

font_06x10.bmp

font_07x13.bmp

font_23x26.bmp

Sample General Bitmaps

ec_logo_64x37.bmp

Embedded-code.com logo 64 x 37 pixels

ec_logo_128x64.bmp

Embedded-code.com logo 128 x 64 pixels

graph_pixel.bmp

Single pixel bitmap used by the sample graph display function

graph_xy.bmp

128 x 64 pixel graph background used by the sample graph display function

SAMPLE PROJECTS FUNCTIONS

The 2 switch inputs provide the following functions:-

Switch 1

Each press cycles through the following:

Displays the embedded-code.com logo

Displays the embedded-code.com logo with all pixels inverted

Displays the standard fonts included with the driver

Displays the standard fonts included with the driver with all pixels inverted

Display a simple graph

Switch 2

Hold to cycle the contrast value from min to max.

The projects may be used as a starting point to write a new application or just as a reference for including the driver in your own project.

SAMPLE PROJECTS INCLUDED

Microchip C18 Compiler

Compiler:

Microchip C18 MPLAB C Compiler for PIC18 family of 8 bit microcontrollers

Device:

PIC18F4620

Notes:

The C18 project uses a modified version off the Microchip standard linker script for the PIC18F4620. This is required as the C18 compiler does not support data buffers over 256 bytes without a modification to the linker script to define a larger bank of microcontroller ram. The optional DISPLAY_USE_LOCAL_RAM_BUFFER define requires a large ram buffer if included, but if using the screen as the display ram buffer (DISPLAY_USE_LOCAL_RAM_BUFFER is commented out) then this special linker script doesn't need to be used. Check the USING_3V3_POWER define in main.h for your hardware.

You will see in the sample linker script that several consecutive gpr banks have been removed and instead replaced with:-

DATABANK NAME= display_local_buffer_ram_section START=0x400 END=0xBFF

where '0x#00' is the start address of the first removed bank and '0x#FF ' is the end address of the last removed bank. If modifying other device linker scripts ensure that you also check the bank used by the stack and change it to another bank if it conflicts.

Microchip C30 Compiler

Compiler:

Microchip C30 MPLAB C Compiler for PIC24 family of 16 bit microcontrollers and dsPIC digital signal controllers

Device:

PIC24FJ128GA010

Notes:

Remove the default 'rom' from the Bitmap Converter PC application 'C Constant Declaration String' as this compiler does not use the special qualifier 'rom'.

Microchip C32 Compiler

Compiler:

Microchip C32 MPLAB C Compiler for PIC32 family of 32 bit microcontrollers

Device:

PIC32MX360F512L

Notes:

Remove the default 'rom' from the Bitmap Converter PC application 'C Constant Declaration String' as this compiler does not use the special qualifier 'rom'.

USING THE DRIVER IN YOUR PROJECT

UPDATING THE DISPLAY

Displaying Bitmaps

Use the 'display_bitmap' function. For example:-

```
display_bitmap(0, ec_logo_128x64,               //Bitmap (flash / program
memory)
            DISPLAY_BITMAP_INVERT_PIXELS_OFF,   //Options
            0, 0);                              //X, Y
```

Available options:

DISPLAY_BITMAP_INVERT_PIXELS_OFF

DISPLAY_BITMAP_INVERT_PIXELS_ON

Displaying Text

Use the 'display_const_string' and 'display_variable_string' functions to display null terminated strings. Individual strings may be displayed as monospace or proportional width text with left, centre or right alignment. For example:

```
      display_const_string(0, font_05x07, //Bitmap (flash / program memory)
                              (DISPLAY_STRING_ON_X_AXIS |
                              DISPLAY_STRING_INVERT_PIXELS_OFF |
                              DISPLAY_STRING_PROPORTIONAL |
                              DISPLAY_STRING_CENTER_ALIGN),
      //Options
                              20, 20,
      //X, Y
                              (CONSTANT BYTE*)"Hello World");     //String


CONSTANT BYTE sample_const_string [] = {"Hello World"};

      display_const_string(0, font_05x07,       //Bitmap (flash/program memory)
                              (DISPLAY_STRING_ON_X_AXIS |
                              DISPLAY_STRING_INVERT_PIXELS_OFF |
                              DISPLAY_STRING_MONOSPACE |
                              DISPLAY_STRING_LEFT_ALIGN),
      //Options
                              20, 20,
      //X, Y
                              sample_const_string);
      //String


BYTE sample_variable_string[12] = {"Hello World"};

      display_variable_string(0, font_05x07, //Bitmap (flash/program memory)
                              (DISPLAY_STRING_ON_X_AXIS |
                              DISPLAY_STRING_INVERT_PIXELS_OFF |
                              DISPLAY_STRING_MONOSPACE |
                              DISPLAY_STRING_LEFT_ALIGN),         //Options
                              20, 20,                                   //X, Y
                              sample_variable_string);                  //String
```

Available options:

```
      DISPLAY_STRING_INVERT_PIXELS_OFF
      DISPLAY_STRING_INVERT_PIXELS_ON

      DISPLAY_STRING_ON_X_AXIS
      DISPLAY_STRING_ON_Y_AXIS

      DISPLAY_STRING_MONOSPACE
      DISPLAY_STRING_PROPORTIONAL

      DISPLAY_STRING_LEFT_ALIGN
      DISPLAY_STRING_CENTER_ALIGN
      DISPLAY_STRING_CENTRE_ALIGN
      DISPLAY_STRING_RIGHT_ALIGN
```

Displaying Scrolling Text

The following examples demonstrate how to use the scrolling text function.

Displaying a scrolling constant string (string stored in program memory) example

```
//Define the string (always include a trailing space)
CONSTANT BYTE my_string[] = {"        Hello World "};

//Call this function initially with display_scroll_text_start_pixel = 0xff, and
then every time you want to move the text on 1 place
void display_scrolling_text (void)
{
      static CONSTANT BYTE *p_start_character = &my_string[0];

      display_scroll_text_start_pixel += 1;     //1 for faster scrolling)
      display_scroll_text_display_pixels_count = 40;  //Width of the display
area
 //to display within, in pixels

      display_const_string(0, font_05x07,       //Bitmap (flash / program
memory)
                           (DISPLAY_STRING_ON_X_AXIS |
DISPLAY_STRING_INVERT_PIXELS_OFF |
DISPLAY_STRING_SCROLL), //Options
                           10, 10,                //X, Y start coordinate
                           p_start_character);    //String

      if (display_scroll_1st_char_done_skip_pixels)
      {
            //FIRST CHARACTER HAS FALLEN OFF THE START OF THE DISPLAY
//MOVE TO NEXT ONE
            p_start_character++;
            if (p_start_character >= (&my_string[0] + sizeof(my_string) - 1))
                  p_start_character = &my_string[0];

            //Reset the number of pixels to skip
            if (display_scroll_1st_char_done_skip_pixels < 0xff)  //(0xff =
reset to zero)
                display_scroll_text_start_pixel =
display_scroll_1st_char_done_skip_pixels;
            else
                display_scroll_text_start_pixel = 0;
      }
}
```

Displaying a scrolling variable string (string stored in ram) example

```
//Call this function initially with display_scroll_text_start_pixel = 0xff, and
then every time you want to move the text on 1 place
void display_scrolling_text (void)
{
      static BYTE my_string[] = "Hello World ";
      BYTE *p_source;
      BYTE *p_dest;
      BYTE first_character;

      display_scroll_text_start_pixel += 1;     //1 for faster scrolling)
      display_scroll_text_display_pixels_count = 60;  //Width of the display
area
//to display within, in pixels

      display_variable_string(0, font_07x13,    //Bitmap (flash / program
memory)
                              (DISPLAY_STRING_ON_X_AXIS |
DISPLAY_STRING_INVERT_PIXELS_OFF |
DISPLAY_STRING_SCROLL), //Options
                              2, 15,                        //X, Y start
coordinate
                              my_string);             //String

      if (display_scroll_1st_char_done_skip_pixels)
      {
            //FIRST CHARACTER HAS FALLEN OFF THE START OF THE DISPLAY
//MOVE IT TO THE END
            p_source = &my_string[1];
            p_dest = &my_string[0];
            first_character = *p_dest;
            while (*p_source != 0x00)
                  *p_dest++ = *p_source++;

            *p_dest++ = first_character;
            *p_dest++ = 0x00;

            //Reset the number of pixels to skip
            if (display_scroll_1st_char_done_skip_pixels < 0xff)  //(0xff =
reset to zero)
                display_scroll_text_start_pixel =
display_scroll_1st_char_done_skip_pixels;
            else
                display_scroll_text_start_pixel = 0;
      }

}
```

This will display the string which will continuously scroll, automatically moving each character to the end of the string each time they fall off the start, so that the scrolling never resets.

Scrolling Details

The implementation of scrolling is deliberately provided in this quite raw fashion so that you can fine tune it exactly as you wish and also use the functionality to do special things if desired. These are the variables that provide the scrolling functionality when any string is displayed with the DISPLAY_STRING_SCROLL option selected:

display_scroll_text_start_pixel

> This variable defines how many pixels at the start of the displayed string to ignore. You would typically incremented it each time the string is displayed.

display_scroll_text_display_pixels_count

Set to the required width you want the text to be displayed within (any extra trailing characters or part characters will not be displayed). Must be set each time the string is displayed.

display_scroll_1st_char_done_skip_pixels

This variable will contain 0 after a scrolling string has been displayed if the first character of the string is still partly displayed. If not zero then the first character of the string can be removed and the value indicates the value to reset display_scroll_text_start_pixel to (0xff = set it to zero). Although it is best for fastest processing to remove characters from the start of strings and reset display_scroll_text_start_pixel, doing this is optional and you could simply keep on incrementing display_scroll_text_start_pixel all the way to the end of the string if you wished.

Manipulation of display_scroll_text_start_pixel and display_scroll_text_display_pixels_count variables gives you the flexibility to create your own special scrolling or bespoke string display capabilities should you wish.

Scrolling Notes

Displaying fast scrolling text and large scrolling text, requires enough processing speed and a fast enough communications interface to your display to avoid scrolling being slowed. Most other display functions don't require your hardware to be very fast, as a slight delay while a bitmap or string is displayed is barely noticed by the eye. However when moving text along pixel by pixel slowness of updating can become an issue if your hardware is not fast enough. If scrolling is too slow the following may help:

Slow the scrolling speed

Reduce the size of the text

Use fonts that are stored in a C header file (instead of an external binary file)

Enable the DISPLAY_USE_LOCAL_RAM_BUFFER define so that internal microcontroller / processor memory is used to buffer the display data, avoiding the driver having to read from the display.

Clear The Screen

```
display_clear_screen(0);     //Clear the screen

display_clear_screen(1);     //Clear the screen with all pixels on
```

How To Create Graphical Elements

Creating User Interface Menu Bars

Create bitmap images of each of the menu elements. Display these and then use text strings to overwrite on top of each menu item. Using the 'invert pixels' option allows you to display text over bitmap blocks without needing to create a new inverted font.

Creating Windows With Variable Content

Create a bitmap image of the whole window and then overwrite it with text strings and small bitmap images in the required places to create the variable content inside.

Creating Flashing Or Animated Icons

Create two or more bitmap images of the same size and then display each one over the top of the last using a timer

Creating Progress Bars

Create a bitmap of the empty progress bar and then use a single bitmap of one row of pixels to update the progress bar for each successive step, moving it on one column or row at a time.

Creating Lines
For variable length lines use a single pixel bitmap and display repeatedly to make the line, or use bitmaps of complete lines.

Creating Graphs
Create a bitmap for your graph X and Y axis markers and then use a single pixel bitmap to create each of the points of a graph line.

Updating Screen In A Single Operation
If you want to be able to update the contents of your display in local ram and then write to the screen in a single bulk operation this is possible by modifying the driver when using the DISPLAY_USE_LOCAL_RAM_BUFFER define. With this define included, so that a local ram buffer is used to store a copy of the display output, you could remove the actual output to the screen code in the function display_write_bitmap_byte. Instead this function would update the local ram buffer as usual but wouldn't attempt to write to the display. When you wanted to update the display you could then provide your own function to write the complete display_copy_buffer to your screen.

USING DRIVER WITH A NEW SCREEN MODEL

Using the driver with a new screen model requires adjusting each of the functions in the display-model.c file to suit your screen and its driver (these files contain just the functions that are screen specific and don't contain any of the general display processing driver functions). Unfortunately due to the massive amount of screens and screen driver IC's available there is no generic way of doing this, but following this guide should make the process straightforward.

Copy A Pair Of Sample Files
First copy the sample display-model.c and display-model.h pair of files for the included screen types that is most similar to your screen to use as a reference. These can then be modified as necessary for your screen and its controller IC.

If your screen includes 2 controller IC's (many do to drive half of the screen each) then select sample files that are for a screen that also uses 2 controllers.

Screen Specific Defines
Go through each of the defines in the display-model.h file and update as required for your screen.

DISPLAY_WIDTH_PIXELS

> The width of the screen in pixels (X coordinate)

DISPLAY_HEIGHT_PIXELS

> The height of the screen in pixels (Y coordinate – this is used for the page address – divide this value by 8 to give the number of pages). This axis must match the axis that the screen addresses in bytes (for instance it may be the real world X axis even though its called height in the driver). As the driver allows screens to be used in all 4 possible orientations width and height will actually refer to the Y and X axis in 2 of the possible orientations.

DISPLAY_WIDTH_START_OFFSET

> Offset to first actual display position (0 if not applicable – required if screen has a number of unused rows that need to be skipped past).

DISPLAY_DEFAULT_CONTRAST

> The default contrast value for the screen (only required if screen has digital contrast control)

DISPLAY_USE_LOCAL_RAM_BUFFER

> The driver provides complete functionality without requiring a local ram buffer of the outputted display data. It will read the current display state of individual bytes back from the display when it

needs to. However if your microcontroller has sufficient ram available the display functions can be speeded up if a local copy of the display output is stored, especially if the display has a serial interface. If you are using a display that does not provide the means to read back display data (for instance serial displays which have a 'SI' serial in pin and no 'SO' serial out pin) then you will have to use this use local ram buffer option for the driver to function correctly. Your local ram buffer will need to be big enough to hold every pixel of the display which is typically (X resolution * Y resolution) / 8. If the resolution of either axis is not a multiple of 8 then increase it up to the next multiple of 8 before carrying out this calculation.

ORIENTATION_IS_0
ORIENTATION_IS_90
ORIENTATION_IS_180
ORIENTATION_IS_270

Enabling 1 of these defines (comment out the other 3) selects the orientation your screen is to be used in. (Use the same setting for the Bitmap Converter Application).

INVERT_Y_AXIS_COORDINATES

Should be commented out normally, but include if you need to reverse the direction of display on the Y axis (only applicable when in screen orientations when the Y axis is horizontal). Typically required for one of the four possible orientations.

IO Defines

Alter the IO defines to provide all of the input and output pins required by your screen and serial port access if your screen uses a serial rather than parallel interface.

Bus Access Delay Defines

DISPLAY_BUS_ACCESS_DELAY

This define may be used with parallel interface screens to add in one or more null (no operation) instructions to allow data bus signals to stabilise during read or write access. 2 layer PCB's (no ground plane) and PCB's with long track lengths are more likely to require this. If you are experiencing problems setting up a new screen it is worth using this define to give a reasonable delay and then remove it later once you have the screen working.

User Options

INCLUDE_DISPLAY_BYTE_TEST_SEQUENCE_FUNCTION

The test sequence function is useful when configuring the driver for a new screen. Comment this define out to remove the function to save program memory space. An example of its use is included in the sample projects. See the description of this function in this manual.

Screen Specific Functions

void display_model_initialise(void)

Adjust this to provide the initialisation sequence required by your screen. This can often be found in the screens data sheet or by contacting the screen manufacturer to request the initialisation sequence required. Controller IC's used in the screens are often quite generic and can therefore be confusing to know exactly the configuration that is required for the particular screen model they have been used in. Therefore it is reasonable to expect the screen manufacturer to provide this if they have failed to include it in a screen datasheet.

void display_model_set_contrast(BYTE contrast_value)

If your screen includes digital contrast control then include the command sequence in this function to set the contrast to a new level. If there is no digital contrast command just leave this as an empty function.

void display_write_bitmap_byte (BYTE bitmap_mask, BYTE bitmap_data, WORD x_byte_coord, WORD y_byte_coord)

This function is called to output each byte to the display. It first reads the current byte value from the display (if DISPLAY_USE_LOCAL_RAM_BUFFER is not defined), and you may need to adjust this to suit your display. It then modifies the byte with the new data to be written, which you shouldn't modify. Finally it writes the new byte value to the display, which you may need to adjust to suit your display.

void display_set_address (WORD x_coord, WORD y_coord_page)

This function outputs the address of the byte being output to the display and may need to be altered to suit your display.

void display_write_command (BYTE data)

This function outputs individual command bytes to the display and may need to be altered to suit your display.

Getting A New Screen To Work

Screen data sheets don't always provide very good information to help you get a new screen working. Some datasheets simply provide the mechanical details of the screen, the electrical characteristics and specify which display controller IC is used in the screen. Here's some tips:-

Beware of screen mechanical drawings – just because a data sheet shows a screen in a particular orientation doesn't mean that this is the correct orientation to use the screen! If the datasheet doesn't specify the correct viewing angle the best way to check is to get the screen working and see which orientation provides the best viewing angle for the intended application. If you can't do this ask the manufacturer to confirm the correct orientation or find a photo of the screen in use. The bitmap converter application allows you to rotate your bitmap images and fonts into all 4 orientations, allowing you to use a screen any way round.

Get the datasheet for the display controller IC being used. Also check to see if there is a technical manual as some IC datasheets provide little helpful information on how to get a screen working because this is provided in a separate technical manual or application note.

Your screen may use 2 off the same display driver IC. This makes the screen access functions slightly more complicated as each IC has to be individually addressed. Sample screen drivers which have 2 driver IC's are included with this driver which you can copy and modify when configuring for your screen.

Some screens, such as LCD screens, have specific voltage requirements. If you are lucky your screen will use a controller IC that includes generation of these voltages. You will still need to configure the driver to provide the voltages required by the screen, but you will not need to provide the voltages externally. If not then you will need to provide the required voltage(s). For LCD screens there are many IC's available designed to provide bias voltages, often with a digital interface to provide the contrast adjustment (variation of the bias voltage). Otherwise a simple potentiometer may be used. Be aware that sometimes the bias voltage needs to be negative, so you may have to use a DCDC converter if your application has no negative voltage rail.

A controller IC provides the actual drive of the display pixels and a standard data bus connection to your processor / microcontroller. Some controller IC's provide advanced features such as more than 1 'layer' of display output. This driver uses a single layer so any other layers can be ignored. If your screen has no built in controller IC you will need to select one and fit it between the screen and your processor / microcontroller. This can be a complex task and is outside the scope of this manual. A good solution is to see if there is an evaluation board available for the screen and copy the controller IC used on this, or find another screen that includes a built in controller IC!

Tips on configuring the driver for your screen

If your screen has built in contrast control a good way to check that your initialisation sequence is working is to use a routine to cycle through the contrast values. If all is well you should see the contrast change on the screen regardless of what the screen might be displaying. This confirms that you are at least communicating correctly with the screen. If there is no contrast command maybe your screen driver has special commands to say turn all pixels on, again providing a means to confirm that you are communicating correctly.

Ask your screen supplier or manufacturer if there is an example code sequence available for your screen or an application note. Often screen datasheets just specify the controller IC used and then leave it to you to work out how that generic controller IC needs to be configured for the screen it is being used with. Ask the manufacturer for this information as its often available but not included in the datasheet.

Using the ‘display_byte_ordering_test_sequence’ function is a very useful way to discover how your screen addresses bitmap data. See the description of this function in this manual. This will also confirm that you are able to write to the correct area of the controller IC ram to output bitmap data. Sometimes controller IC’s have larger display ram areas than are needed for the screen you are using and will therefore need to adjust the ‘display_set_address’ function to allow for this.

All addressing is done using X and Y coordinates. However these are software based coordinates and will not necessarily match the logical directions you would assume for the orientation you are using your screen in. Therefore try displaying a bitmap in the centre of the screen and see which way it is orientated, or try it at coordinates 0, 0 and then max x, max y (e.g. 127, 63 for a 128 x 64 pixel screen). Initially displaying it at a position that is a multiple of 8 for the Y coordinate is a good idea in case some adjustment is needed to deal with bit position offsetting.

The driver writes 1 byte of display data at a time and it does this on the Y axis. Therefore ensure you have specified you resolution the correct way round for DISPLAY_WIDTH_PIXELS and DISPLAY_HEIGHT_PIXELS. You may need to reverse these values to match your screen (remember don’t assume X and Y will match the way you think it ought to be). Using the ‘display_byte_ordering_test_sequence’ function will show you in which coordinate your screen is byte addressed.

Remember that you will also need to convert your source bitmap files using the bitmap converter application to the orientation that matches the orientation you have set in the header file.

If your display is working in the reverse direction modify the ‘display_set_address’ function to correct this. You may also need to use the INVERT_Y_AXIS_COORDINATES define (see explanation above). Note that often what appears as a garbage display of a bitmap is simply caused by reverse X or Y addressing.

If text doesn’t display in the direction you expect change the DISPLAY_STRING_ON_#_AXIS option when calling the string display function (its an option when displaying a string as sometimes you may wish to display in the other axis for vertical text)

The X and Y addressing can seem daunting but it is usually relatively simple to solve by a little bit of trial and error. Once this is set correctly for a new screen you are ready to use the full power of this driver.

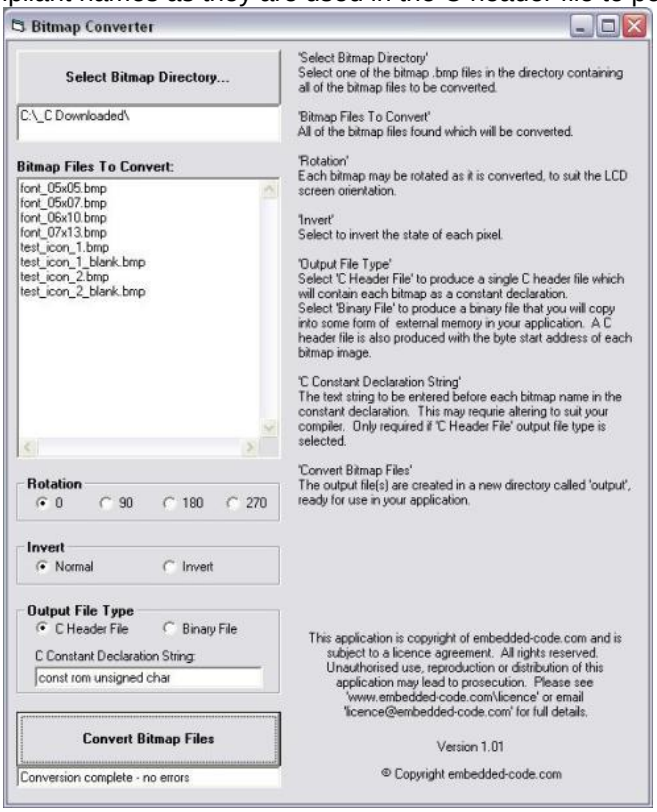
BITMAP CONVERTER APPLICATION

OVERVIEW

The included Bitmap Converter Application will read all of your bitmap files and convert them into the selected format ready for use in your project.

Source bitmap files need to be saved as standard .bmp bitmap files, in 'Black and White 1 bit' format. Use a standard graphics program to do this. (The embedded-code.com sample graphics we're created using Corel Photo-Paint™ and Adobe Photoshop™).

Filenames must use C compliant names as they are used in the C header file to point to that bitmap.



Full instructions are shown on the application main screen.

BITMAP FILE FORMATS

A project may use bitmaps converted to a C header file, to be stored along with the program in the program memory, or converted to a binary file to be stored in external flash memory, or both.

Bitmaps Converted To C Header File Format

The file 'cbitmaps.h' is created.

Each bitmap is saved as a constant. Example:-

```
const rom unsigned char bitmap_1[]={      //The filename was bitmap_1.bmp
0x0,0xA,0x0,0x4,0x0           //Bitmap WidthH:WidthL:HeightH:HeightL:Flags
0x55,0x40,                    //Line 0
0xAA,0x80,                    //Line 1
0x55,0x40,                    //Line 2
0xAA,0x80,                    //Line 3
};
```

```
const rom unsigned char logo_main[]={     //The filename was logo_main.bmp
0x0,0xA,0x0,0x3,0x0           //Bitmap WidthH:WidthL:HeightH:HeightL:Flags
0xFF,0xC0,                    //Line 0
0x0,0x0,                      //Line 1
0xFF,0xC0,                    //Line 2
};
```

The ‘const rom unsigned char‘ before each file name may be changed in the Bitmap Converter application to suit your compiler.

The ‘Flags’ field is provided for future upgradeability and is not currently used.

Bitmaps Converted To Binary File Format

A C header file (bbitmaps.h) is created with the start address of each bitmap contained in the binary file (starting at address 0). Example:-

```
#define bitmap_1 0x0
#define logo_main 0xd
```

A binary (bbitmaps.bin) file is created containing the bitmap data. Example:-

```
Byte 0:     0x00
Byte 1:     0x0A
Byte 2:     0x00
Byte 3:     0x04
Byte 4:     0x00
Byte 5:     0x55
Byte 6:     0x40
Byte 7:     0xAA
Byte 8:     0x80
Byte 9:     0x55
Byte 10:    0x40
Byte 11:    0xAA
Byte 12:    0x80
Byte 13:    0x00
Byte 14:    0x0A
Byte 15:    0x00
Byte 16:    0x03
Byte 17:    0x00
Byte 18:    0xFF
Byte 19:    0xC0
Byte 20:    0x00
Byte 21:    0x00
Byte 22:    0xFF
Byte 23:    0xC0
```

Note that the .bin file extension is not a standard file extension – the file format is not readable by any standard application.

The binary file may be copied to external flash memory in your application and accessed using the C ‘bbitmaps.h’ header file. Note that you will need to provide the means of copying the binary data to your flash memory to do this, and a ‘BYTE flash_read (DWORD address)’ function to allow the driver to read the flash memory.

‘Bootloader’ memory is a particularly good choice of memory for this type of application as it is low cost, fast to read and the block erase requirements are not usually a problem when just storing bitmap data. Other non volatile memory types are of course also suitable.

Font Bitmaps

The bitmap converter will also convert special font bitmap images, to be used for ASCII text output. A font bitmap file needs to be created in the following way:-

Say you wish to create a font that will be 5 pixels wide and 7 pixels high.

Create a bitmap image 5 pixels wide by 1016 high ((character height + 1) x 127)

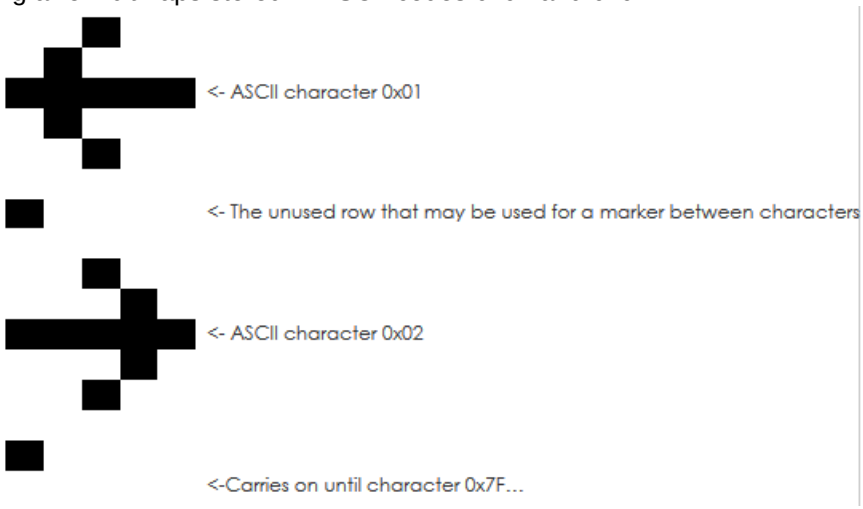
Then create your font as follows:-

The top 5 x 7 pixel block will be the character that represents ASCII code 0x01

Then there is an unused row pixels, which may be used to provide say a single pixel marker between each character to make it easier to navigate through the bitmap (it is ignored by the bitmap converter application).

Then the next block of 5 x 7 pixels is the character that represents ASCII code 0x02, and so on.

An example showing arrow bitmaps stored in ASCII codes 0x01 and 0x02:-



Save the file using the following naming format:-

font####07.bmp

The important fields are the ‘font’ at the beginning (non case sensitive) and the height value in characters 9 and 10. The following are examples of valid file names:-

font_05x07.bmp

Font_05x07.bmp

Font_05x07_style_1.bmp

FontTyp107_style_1.bmp

The bitmap converter application will convert this type of file into a special font bitmap. This is the same as a normal bitmap, except that the height field will contain the character height, rather than the overall bitmap height. The display bitmap function will use the character height to locate the required ASCII character within the bitmap.

Note that .bmp bitmap images are stored from the bottom row upwards. The bitmap file bottom row must be the final unused marker row (as this is the first row that will be read by the bitmap converter application and will be stored as the first row of the bitmap image).

Should you not wish to use all 127 available ASCII character locations a file of a smaller overall height may be used and the bitmap converter application will only store the characters that are available (e.g. say you only want ASCII characters from 0x30 to 0x7F).

The following sample font bitmap files are included with the driver:-

5 x 5 pixel

5 x 7 pixel

6 x 10 pixel

7 x 13 pixel

23 x 26 pixel

The sample fonts contain the following characters:-

HEX	Character
0x01	Left arrow
0x02	Right arrow
0x03	Up arrow
0x04	Down arrow
0x05	
0x06	
0x07	
0x08	
0x09	
0x0A	
0x0B	
0x0C	
0x0D	
0x0E	
0x0F	
0x10	
0x11	
0x12	
0x13	
0x14	
0x15	
0x16	
0x17	
0x18	
0x19	
0x1A	
0x1B	
0x1C	
0x1D	
0x1E	
0x1F	
0x20	
0x21	!
0x22	"
0x23	#
0x24	$
0x25	%
0x26	&
0x27	'
0x28	(
0x29	)
0x2A	*
0x2B	+
0x2C	,
0x2D	–
0x2E	.
0x2F	/

HEX	Character
0x30	0
0x31	1
0x32	2
0x33	3
0x34	4
0x35	5
0x36	6
0x37	7
0x38	8
0x39	9
0x3A	:
0x3B	;
0x3C	<
0x3D	=
0x3E	>
0x3F	?
0x40	@
0x41	A
0x42	B
0x43	C
0x44	D
0x45	E
0x46	F
0x47	G
0x48	H
0x49	I
0x4A	J
0x4B	K
0x4C	L
0x4D	M
0x4E	N
0x4F	O
0x50	P
0x51	Q
0x52	R
0x53	S
0x54	T
0x55	U
0x56	V
0x57	W
0x58	X
0x59	Y
0x5A	Z
0x5B	[
0x5C	\
0x5D	]
0x5E	^
0x5F	_

HEX	Character
0x60	`
0x61	a
0x62	b
0x63	c
0x64	d
0x65	e
0x66	f
0x67	g
0x68	h
0x69	i
0x6A	j
0x6B	k
0x6C	l
0x6D	m
0x6E	n
0x6F	o
0x70	p
0x71	q
0x72	r
0x73	s
0x74	t
0x75	u
0x76	v
0x77	w
0x78	x
0x79	y
0x7A	z
0x7B	{
0x7C	|
0x7D	}
0x7E	~
0x7F	Vertical bar (* required)

Unused characters can be used to add your own special characters if required.

(* required) The character at location 0x7F must be a single vertical bar. This special character is used for the space between each character and the width of the bar determines the number of blank pixels to be placed between each character.

INFORMATION

FREQUENTLY ASKED QUESTIONS

Where are the bitmaps and fonts stored?

The included PC Bitmap Convert Application allows you to output all of your bitmap graphic and font files as a C compliant header file, so that the data is stored in the program memory, or as a binary file so that you can store in external memory, or a combination of both.

Is The Driver Difficult To Use

Setting a new screen up for the first time can be difficult. Due to the huge number of screens available it would be impossible for us to provide individual drivers for every screen. Therefore we have made this driver as generic as possible and you will need to alter the display-model.c and display-model.h files to work with your particular screen. See earlier in this manual for details of how to do this. Once this is done you then only need to select the orientation you want to use the screen in and the driver will then be working correctly for your screen. That's the hard bit out of the way – with your screen setup correctly using the driver is a breeze and the flexibility of being able to put what you want wherever you want on the screen makes designing user interfaces really enjoyable.

SPECIFICATIONS

Maximum screen size

5000 x 5000 pixels

Using The Driver With a RTOS or Kernel

The stack / driver is implemented as a single thread so you just need to make sure it is always called from a single thread (it is not designed to be thread safe).

CODE AND DATA MEMORY REQUIREMENTS

Code Size Example

The following tests we're carried out using a 128 x 64 pixel 8 bit parallel interface screen without using faster internal memory (DISPLAY_USE_LOCAL_RAM_BUFFER define disabled) and with all compiler optimisations turned off.

PIC18 Sample Project:

> Approximately 7440 program memory bytes (3720 x 16 bit instructions) compiling just the driver functions with the Microchip C18 MPLAB C Compiler for PIC18 family of 8 bit microcontrollers.

PIC24 Sample Project:

> Approximately 3334 program memory words (1677 x 24 bit instructions) compiling just the driver functions with the Microchip C30 MPLAB C Compiler for PIC24 family of 16 bit microcontrollers and dsPIC digital signal controllers.

PIC32 Sample Project:

> Approximately 7540 program memory bytes (1855 x 32 bit instructions) compiling just the driver functions with the Microchip C32 MPLAB C Compiler for PIC32 family of 32 bit microcontrollers.

Variables Memory Space

Approximately 16 bytes of static RAM (DISPLAY_USE_LOCAL_RAM_BUFFER define disabled)

The driver does not require a great deal of temporary variable storage space from the stack as bitmap data is read and displayed one byte at a time.

Speed

This will depend on your screen and processor / microcontroller. As a basic guide, using the driver with the BATRON BTHQ128064 AVD COG 128 x 64 pixel sample screen and a PIC18 microcontroller running with a 6 MIPS instruction clock (3V3 powered) , internal program memory used to store bitmaps and DISPLAY_USE_LOCAL_RAM_BUFFER define disabled:-

Displaying a single 128 x 64 pixel bitmap takes approximately 85mS.

Displaying a 20 character left aligned ASCII text string, 5×7 pixel font, monospaced characters takes approximately 25ms.

Displaying a 20 character left aligned ASCII text string, 5×7 pixel font, proportional characters takes approximately 29ms.

With a 10 MIPS instruction clock (5V powered)

Displaying a single 128 x 64 pixel bitmap takes approximately 51mS.

Displaying a 20 character left aligned ASCII text string, 5×7 pixel font, monospaced characters takes approximately 15ms.

Displaying a 20 character left aligned ASCII text string, 5×7 pixel font, proportional characters takes approximately 17ms.

How The Driver Works

The Driver Functions and Defines

Note – this section of the manual is for information should you wish to gain an understanding of how each of the driver components works.

Generic Driver and Screen Specific Files

The ‘display.c’ and ‘display.h’ files contain all of the driver generic functions and defines.

The ‘display-model.c’ and display-model.h’ files contain functions and defines that are particular to a specific screen and controller IC.

Delay Function

display_delay_ms(delay_ms)

Provides a simple means for the drive to delay while initialising the screen. Accuracy is not important as long as a value of 1 = a delay of at least 1mS (i.e. longer delays don’t matter).

Initialise The Screen

void display_initialise (void)

This function calls a screen specific function in the ‘display-model.c’ file. The function contains the screen initialisation sequence.

If configuring the driver for a new screen then copy sample ‘display-model.c’ and ‘display-model.h’ files for a screen that is most similar and then modify this function to suit the screen in use.

Set the Contrast

void display_set_contrast (BYTE contrast_value)

This function is only used for screens that have built in digital contrast adjustment. It calls a screen specific function in the ‘display-model.c’ file. The function contains the screen initialisation sequence.

If configuring the driver for a new screen then copy sample ‘display-model.c’ and display-model.h’ files for a screen that is most similar and then modify this function to suit the screen in use.

Clear Screen

void display_clear_screen (BYTE invert_pixels)

This function writes 0x00 or 0xFF to every screen output byte location to clear it.

Usage examples:-

display_clear_screen(0); //Normal clear screen

display_clear_screen(1); //Set all screen pixels on

This is a universal function – modification is not normally required

Display Bitmap

void display_bitmap (DWORD image_flash_address, const rom BYTE *p_image_memory_address, WORD image_options, WORD x_start_coord, WORD y_start_coord)

image_flash_address

Start address of bitmap in flash memory (set to 0 if bitmap is in program memory))

*p_image_memory_address

Pointer to bitmap in program memory (set to 0 if bitmap is in flash memory)

image_options

Bit 15:13 Unused

Bit 12 Font table image: 1 = Scroll text, 0 = static text

Bit 11:10 If either is high then do not output to the screen (used for centre and right aligned text)

Bit 9 Font table image: 1 = using proportionally spaced characters

Bit 8 Font table image: 0 = displayed along X axis, 1 = display along Y axis

Bit 7 1 = Invert pixels (i.e. a pixel on becomes off and vice versa)

Bit 6:0

0 = image is a normal bitmap.

>0 = Image is from a font table. The address given is the start of the font table. The value of bits 6:0 is the ASCII character required (0x01 – 0x7F)

x_start_coord

0 to DISPLAY_WIDTH_PIXELS. (If = 0xFFFF then the ‘ display_auto_x_coordinate’ variable is used for the bitmap x coordiante. This variable is updated with the next x coordinate each time this function is called to allow successive display across a screen).

ui_y_start_coord

0 to DISPLAY_HEIGHT_PIXELS. (If = 0xFFFF then the ‘display_auto_y_coordinate’ variable is used for the bitmap y coordiante. This variable is updated with the next y coordinate each time this function is called to allow successive display across a screen).

image_options standard defines:

DISPLAY_BITMAP_INVERT_PIXELS_OFF

DISPLAY_BITMAP_INVERT_PIXELS_ON

This is a large complex function that carries out all of the bitmap display functions. At a basic level it is simply reading bitmap data from memory and displaying each column in order. It is complex because it is dealing with all the problems having to read bitmaps in bytes of 8 pixels each and then write to the screen in bytes of 8 pixels but without there necessarily being any alignment between the two, and in as efficient a manor as possible. This function is also used to display each character of a string and when displaying text in proportional mode it deals with calculating how each character bitmap should be shifted and masked for any of the 4 possible screen orientations that may be used.

Usage example:-

```
display_bitmap(0, ec_logo_128x64, //Bitmap

DISPLAY_BITMAP_INVERT_PIXELS_ON, //Options

0, 0); //X, Y
```

This is a universal function – modification is not normally required.

Special Note

The X and Y start coordinates are screen based, not screen orientation based. Depending on how your particular screen is addressed and orientated in your application the X and Y values may relate to either actual real world axis and their values may run in either direction. An easy way of determining this for a new application is to display a bitmap or an ASCII text string in the centre of the screen and then alter the X and Y values to see how they affect the position of the bitmap or string.

Display String

Version for displaying constant strings:-

void display_const_string (DWORD image_flash_address, const rom BYTE *p_image_memory_address, WORD image_options, WORD x_start_coord, WORD y_start_coord, const rom BYTE *p_ascii_string)

Version for displaying variable strings:-

void display_variable_string (DWORD image_flash_address, const rom BYTE *p_image_memory_address, WORD image_options, WORD x_start_coord, WORD y_start_coord, BYTE *p_ascii_string)

image_flash_address

Start address of font table bitmap in flash memory (set to 0 if bitmap is in program memory))

*p_image_memory_address

Pointer to font table bitmap in program memory (set to 0 if bitmap is in flash memory)

image_options

Bits15:10 Unused

Bit9 0 = Monospace / fixed pitch characters, 1 = proportionally spaced characters

Bit8 0 = Display string along X axis, 1 = Display string along Y axis

Bit7 1 = Invert pixels (i.e. a pixel on becomes off and vice versa)

Bits6:0 Not available (used by display bitmap function)

x_start_coord

0 to DISPLAY_WIDTH_PIXELS. (If = 0xFFFF then the ‘ display_auto_x_coordinate’ variable is used for the bitmap x coordiante. This variable is updated with the next x coordinate each time this function is called to allow successive display across a screen).

y_start_coord

0 to DISPLAY_HEIGHT_PIXELS. (If = 0xFFFF then the ‘ display_auto_y_coordinate’ variable is used for the bitmap y coordinate. This variable is updated with the next y coordiante each time this function is called to allow successive display across a screen).

*p_ascii_string

Pointer to the string to be displayed (must be null terminated)

image_options standard defines:

DISPLAY_STRING_INVERT_PIXELS_OFF

DISPLAY_STRING_INVERT_PIXELS_ON

DISPLAY_STRING_ON_X_AXIS

DISPLAY_STRING_ON_Y_AXIS

DISPLAY_STRING_MONOSPACE

DISPLAY_STRING_PROPORTIONAL

DISPLAY_STRING_LEFT_ALIGN

DISPLAY_STRING_CENTER_ALIGN

DISPLAY_STRING_CENTRE_ALIGN

DISPLAY_STRING_RIGHT_ALIGN

These functions are relatively straightforward and they simply read the string one character at a time calling the display_display_bitmap function to display each character and to display the space between each character. A bitmap is displayed for the space between each character so that any underlying display pixels are overwritten as a string is displayed and so that the number of pixels required between each character can be effectively set by the special gap character in each font table bitmap (character 0x7f).

Constant string usage example:-

```
const rom BYTE sample_const_string[] = {“Hello World”};

    display_const_string(0, font_05x07, //Bitmap

    (DISPLAY_STRING_ON_X_AXIS | DISPLAY_STRING_INVERT_PIXELS_OFF |

    DISPLAY_STRING_MONOSPACE |

    DISPLAY_STRING_LEFT_ALIGN), //Options

    0, 0, //X, Y

    sample_const_string); //String
```

Variable string usage example:-

```
BYTE sample_variable_string[15] = {“Hello World”};

    display_variable_string (0, font_05x07, //Bitmap

    (DISPLAY_STRING_ON_X_AXIS | DISPLAY_STRING_INVERT_PIXELS_OFF |

    DISPLAY_STRING_MONOSPACE |

    DISPLAY_STRING_LEFT_ALIGN), //Options

    0, 0, //X, Y

sample_variable_string); //String
```

This is a universal function – modification is not normally required.

Special Note

The X and Y start coordinates are screen based, not screen orientation based. Depending on how your particular screen is addressed and orientated in your application the X and Y values may relate to either actual real world axis and their values may run in either direction. An easy way of determining this for a new application is to display a bitmap or an ASCII text string in the centre of the screen and then alter the X and Y values to see how they affect the position of the bitmap or string.

Text Alignment

Left aligned text is the default way that strings are displayed. If you select centre or right aligned text the driver has automatically to calculate the required string start position as if the string was left aligned. It does this by simply displaying the string once, but with the data not actually being output to the screen, and then using the completed string end position to calculate the required left aligned string start position. The string is then displayed again, this time being outputted to the display. You don't need to worry about the complexities of how the driver does this, but bear in mind that there is additional time required for this double processing of the string to take place. The double display approach is required because when centre or right aligning proportional text (which centre and right aligning is so useful for from a programmers point of view) each character has to be completely read and analysed to determine its width.

Byte Ordering Test Sequence

void display_byte_ordering_test_sequence (BYTE flags)

uc_flags

bit 0 set = reset to byte 0

bit 1 clear = move forwards 1 byte, set = move back 1 byte

This function may be used when configuring a new screen, to test how each sequential byte gets displayed on the screen. Call the function the first time with bit 0 set. The function will send a byte to the display at address 0x00, 0x00. Using push buttons or a timer, repeat calling the function with bit 0 and 1 clear and the function will write to the next byte, then the next, then the next…

The test byte sent is formatted as follows:-

This is a really useful way to discover how the screen is mapped and if there are bytes at the beginning of the address space that are unused (i.e. you have to call the function several times before you see the first byte). In this instance update the DISPLAY_WIDTH_START_OFFSET constant with the required offset to correct this.

This is a universal function – modification is not normally required.

Write Bitmap Byte

void display_write_bitmap_byte (BYTE bitmap_mask, BYTE bitmap_data, WORD x_byte_coord, WORD y_byte_coord);

bitmap_mask

bit high = use the bitmap_data value, bit low = use the existing screen value

bitmap_data

The data to write, controlled by the bitmap_mask bits

x_byte_coord

The X coordinate to write to

y_byte_coord

The Y coordinate to write to

This function first reads the current display value for the byte being addressed, then applies the bitmap data AND'd with the mask, before writing the byte back to the display.

If configuring the driver for a new screen then copy sample 'display-model.c' and display-model.h' files for a screen that is most similar and then modify this function to suit the screen in use.

Set Byte Address

void display_set_address (WORD x_coord, WORD y_coord_page)

This functions deals with the complexities of how a screen is mapped and sends the address to be accessed to the screen. Displays with non logical or reverse orientation addressing are corrected using this function.

If configuring the driver for a new screen then copy sample ‘display-model.c’ and display-model.h’ files for a screen that is most similar and then modify this function to suit the screen in use.

Write Command

void display_write_command (BYTE data)

This function is usually that same as the instructions in the write part of display_write_bitmap_byte. The only difference may be setting the DC data / command line. However to allow for screens that aren’t so normal this separate write command function is used.

If configuring the driver for a new screen then copy sample ‘display-model.c’ and ‘display-model.h’ files for a screen that is most similar and then modify this function to suit the screen in use.

TROUBLESHOOTING

GENERAL TROUBLESHOOTING NOTES

Getting a new screen to work can be frustrating as until you get the control signals and initialisation sequence right you won't see anything on the screen. Here are a few tips to help:-

Double check IO pin definitions in the driver header file.

Verify with a scope that all of the control and data pins to the screen are working correctly.

Add additional null execution steps to the DISPLAY_BUS_ACCESS_DELAY define in the display.h file in case more time is required for signals to stabilise on your PCB. This is more likely to be an issue for 2 layer PCB's (no ground plane) with long tack lengths or screens with long ribbon / FFC cable connections.

Check that no other device on the data bus is outputting while the driver is trying to communicate with the screen.

If your screen controller IC includes a status register try reading the status from the screen and confirm that you get the expected response.

If you're using output latches for some of the screen control pins, instead of pins connected directly to your processor, check your output latch function restores the previous output on the data bus when it exits, to avoid destroying the data the driver function is writing to the screen.

Check that your microcontroller is not resetting due to a watchdog timer timeout.

Check that you have enough stack space allocated. This driver does not use an excessive amount of ram from the stack, but if your application is already using large amounts of the stack before calling driver functions this may be causing a stack overrun?

Re-read the datasheet to find out what you're doing wrong!

REVISION HISTORY

CHANGES TO THE MONOCHROME SCREEN DRIVER FILES

V1.00

Original release

V1.01

Changes to deal with long variable names that are cropped by some compilers.

lcd_delay_ms changed from a define to a function to simplify for compilers that don't provide standard delay library functions or macros)

V1.02

Implemented centre and right align options for displaying strings

Changed the example lcd_display_bitmap call to use DISPLAY_BITMAP_INVERT_PIXELS_OFF to avoid confusion for new users (was previously DISPLAY_BITMAP_INVERT_PIXELS_ON).

V1.03

Significant changes to the overall driver code.

Added scrolling text functionality.

Moved many of the defines to the lcd-model.h file to make using the driver with new screens more straightforward.

Added USE_LOCAL_RAM_BUFFER define for a local ram buffer of display data to optionally be used to increase speed where microcontroller / processor ram is available, or when required for displays that do not provide the capability to read display data back from the screen.

Added CONSTANT define to main.h and replaced all 'const rom' to CONSTANT in the code, to allow easy dealing with compilers require an additional qualifier such as 'rom' and those that don't.

Added Microchip C30 and C32 compiler compatibility for 16 and 32 bit Microchip devices.

Added new screen model sample files for Batron BTHQ240064AVB-EMN-06-LED with SPI interface.

CHANGES TO THE BITMAP CONVERTER PC APPLICATION

V1.00

Original release

V1.01

Added processing of font bitmap files

Added flags field to bitmap header

V1.02

Some graphics programs sometimes use an opposite colouring of the pixels (an opposite palette). We now read the palette of each image and invert pixels if the opposite pallet colouring is used.

V1.03

LCD_C define changed to DISPLAY_C to match new define used by the V1.03 driver

Message box confirmation of conversion complete added.

Designed by:

IBEX UK Limited
32A Station Road West
Oxted
Surrey
RH8 9EU
England
Tel: +44 (0)1883 716 726
E-mail: info@ibexuk.com
Web: www.ibexuk.com