Chapter 4: Dot Matrix LCD

Working with the dot-matrix LCD is probably the most challenging part of the entire MPG Level 2 course if you have never worked with one before. It is certainly the most data-intensive, which is why it may seem harder to work with initially. Writing a solid driver for an LCD takes a lot of careful thought and planning. It could be simplified if written specifically for the MPG Level 2 device, but with some extra effort can be written to support a wide variety of LCDs which is a much preferred solution and thus what we will build for the course.

4.1 Newhaven LCD Module

The LCD module for the course development board is from Newhaven Display, part number NHD-C160100DiZ-FSW-FBW. It is a 160x100 dot-matrix display with 4-bit grayscale capability (each pixel can have 16 varying levels of darkness between totally off and totally on). The pixel addressing has pixel 0, 0 in the top right corner and 159, 99 in the bottom left corner. There is a border of unused area around the edge of the LCD to allow the screen to be installed inside a plastic frame. Figure 4.1.1 shows a picture of the LCD.

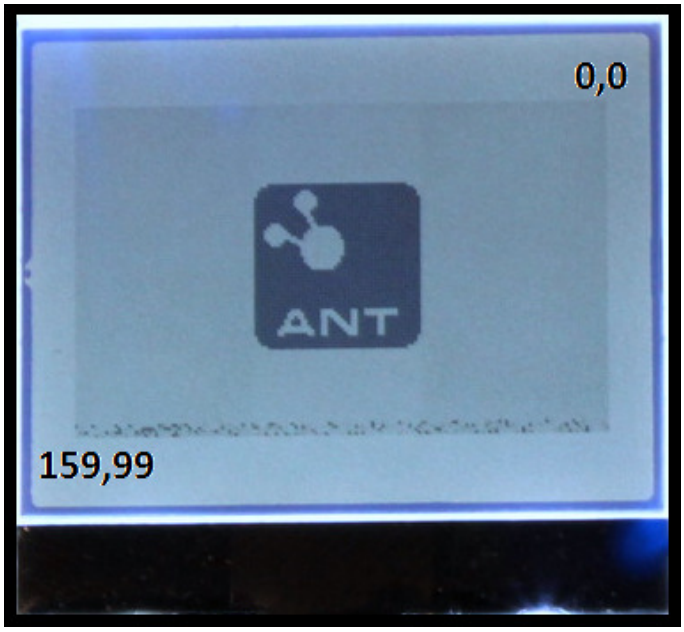

Figure 4.1.1: Picture of working LCD screen with pixel address shown

The display has a built-in LCD controller with an I²C interface for communication to a host microcontroller. The LCD controller part number is ST7528i from Sitronix – the data sheet is available on the MPG website or you can check it out at www.sitronix.com.tw/products.htm.

If you have ever wondered how an LCD works, rest assured there really is no magic behind it. Every pixel has a physical connection to it which allows a voltage to be applied to polarize the liquid crystal solution that is stored at the location. The ST7528i LCD controller chip has 385 pads even though the chip itself is a mere 12mm x 1.2mm in size. Most of the pads are just 47 x 108um in size! There is a matrix of very, very tiny wires running from each pad to each pixel. These wires may be segment wires (160 of them) or common wires (100 of them). Just like in MPG Level 1 where different grounds (commons) were discussed for multiplexing 7-segment displays, the LCD uses the same concept, though there are a lot more commons in this case!

The nice part about using an LCD module with an integrated controller is that you are not required to figure out how to map and route all the segments and common lines to all the pixels to make them work (if you ever design a custom LCD, this is exactly what you have to do, though the manufacturer can usually help a lot). Most of the ST7528 data sheet can be skipped since the LCD manufacturer has taken care of all the integration details. In fact, you do not even need to have the actual driver data sheet as the LCD data sheet has all the information needed to operate the display.

Each pixel has a corresponding memory location on board the controller. In this case, the memory locations are 4-bits wide to support 16 levels of gray scale. This memory is known as DDRAM – Display Data RAM. Since there are 160 x 100 pixels, each requiring 4-bits of data, the controller has to have at least 64kbits (8kB) of memory. This particular driver chip has slightly more than that, and can actually support LCDs up to 160 x 129 pixels.

To build an LCD driver, we have to make a system that can load pixel data 4-bits at a time to the correct location of the LCD RAM. All of the images on the LCDs are bitmaps, so whenever you think about what is happening on the screen, think in terms of individual pixels and how they compile into a bitmap to make an image. The entire LCD display is simply a two-dimensional array of 4-bit location addresses. Creating the LCD images is then just a brute-force process of building the image and then dumping it over to refresh the LCD. The image will be built by maintaining a copy of the LCD memory inside the processor memory. This will allow us to work with a buffered image that we can update and manipulate as required without impacting the real image on the LCD screen. The drawback is that 8k of microcontroller RAM is required to be permanently allocated to the LCD firmware, but that was planned for all along.

The easiest approach to refreshing the images on screen is to update the entire screen every time something changes. Not counting control byte overhead, we already know we have 8kB of data to send to update the entire screen. The I²C bus runs at 400 kHz, and each byte is part of a frame in the I²C protocol that has one bit of overhead. 400,000bps / 9bits = 44,444 frames per second. Factoring in the command overhead, we can probably safely update the LCD 5 times per second but that will keep the LCD communication bus (and thus our processor) running continuously. This type of approach will waste a lot of processor cycles re-writing memory locations that have not changed. Though the processor will not exactly get tired, it will waste power and could slightly impact our ability to handle

other interrupt signals that will be occurring periodically. It is not a very elegant solution, so it begs for a more clever solution which we will come up with momentarily.

A secondary part of the driver will be a mechanism to populate the local copy of the LCD RAM with characters (text and numbers), symbols (the Pong paddles and the Pong ball), and graphics (the Engenuics and ANT logos), which are all bitmaps defining the pixels required for each image. We will create a data table in the LPC175x flash memory that has all of these bitmaps and specific addresses. We will then build a function that will take the symbol name and symbol size as arguments to retrieve the images we need from flash and load them into local LCD RAM as they are required.

4.2 Bitmap Creation

If you ever wondered why ASCII LCD screens are so popular, one of the reasons is that most of them have a built-in font bitmap library. This allows a host controller to simply pass in an ASCII code for the character to display and the module automatically looks up the bitmap in its internal memory. Dot matrix displays do not typically have character information stored, so the bitmap for every character, symbol or image displayed on the LCD has to be defined manually. This applies to all images – even letters and numbers. Instead of passing an ASCII "A" to the LCD, the host must pass data to set each pixel that makes up the letter A to the LCD. The good thing is that this gives full control over character size and appearance to the programmer.

All of these required bitmaps will be stored in the processor flash, just like any other constant that occupies flash space except the stored bitmaps will be arrays of pixel data. Though the Pong game could be developed without any text, no doubt at some point text messages will be needed. Figuring out a way to build a font library can then be extended to some of the other symbols and images needed.

Managing all of these bitmaps is a challenge that must be handled intelligently. C source and header files will ultimately be used to store all the data that will be compiled into flash memory on the processor, but text-based code files do not provide a very intuitive mechanism for managing what will turn out to be several thousand bits of image information. What we will do instead is start with a higher-level tool that can be used to graphically edit bitmap information. The tool will automatically generate the code that can then be cut and paste into the C source files. All we have to do is make sure that the generated code will properly and easily integrate into the source functions that will make use of it.

The tool comes in the form of an Excel spreadsheet and uses a few simple formulas and conditional formatting to work. The spreadsheet is called LCD Worksheet and is included in the course material or available on the course website. Though it took several hours to set up, the resulting savings in time and frustration make it well worth it. Now would be a great time to take a preliminary look at the spreadsheet to gain a context for the next few pages of discussion. Note that the spreadsheet has several tabs including:

1. Single Color Small Characters
2. Single Color Big Characters
3. Grayscale Bitmaps
4. Images
5. Screen Large
6. Screen Small
7. Screen Start
8. Addressing Example

Brief instructions on using the tool are shown at the top of each page and are very straight forward. Generally speaking, bitmaps are defined by setting pixel information for each image. The code to implement the image is automatically generated next to the image and can be copied and pasted into the C source files. The only distinction to make is between single color and grayscale bitmaps that use slightly different methods for drawing the images and result in slightly different code to implement the images.

4.2.1 Character Bit Maps

Character images are the simplest to implement though the sheer number of them makes the exercise a bit tedious. For starters, fonts will be defined to be 7 pixels tall by 5 pixels wide. This size is chosen as it matches the MPG Level 1 ASCII LCD font size and the data sheet for that LCD has a bitmap font table on which the MPG Level 2 font table will be based. Only the ASCII characters from ! (33) to ~(126) will be defined which is 93 characters in total. If you are not familiar with ASCII, check out www.asciitable.com. Figure 4.2.1.1 shows the letter 'A' as an example.

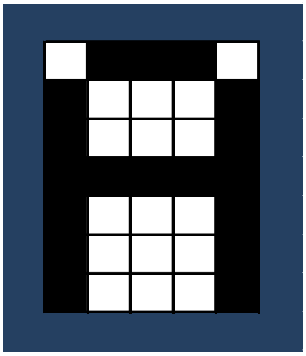

Figure 4.2.1.1: 7 x 5 pixel bitmap for the letter 'A'

For a basic font, the pixels should all be the same color, though you could plan for some nice bold or ghosting effects by using different gray levels. Thinking ahead to how this will all be implemented, each character can be stored with single bit-per-pixel data and as the character is written to the LCD memory a grayscale value can be assigned. The function to parse out the data will pull out each bit, apply the 4-bit grayscale level, and map that data into the LCD RAM.

For a small character, a single byte is used for each row of pixel data though only 5 bits have meaningful information since the character is only 5 pixels wide. The right-most column will be considered column 0 (LSB) and the left-most column is column 4. The top three bits are unused. For the letter 'A,' the binary representation of the top row is b'01110' which is 0x0E in hex. As the character is 7 rows in height, a total of 7 bytes is required to fully define the bitmap.

The spreadsheet tool uses '0' and '1' to set and clear pixels in the image. Enter the numbers into the pixel cells and conditional formatting will automatically set or clear the pixel. The code beside the image is generated in two steps:

1. The total value of a row is calculated by raising the pixel value in each column to a power of 2. For the first row of the letter 'A,' the value is $(0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 14$ which is 0x0E as previously mentioned. You can see the formula for this calculation in column J of the spreadsheet for Single Color Small Characters.

2. The value is converted to hex format and concatenated with the necessary C formatting that will be expected by the 2D array functions that will operate on the bitmaps.

Figure 4.2.1.2 shows the LCD Worksheet data for letter 'A' highlighted so the pixel data is visible. Take a moment to look at the values generated for each row of pixels to ensure you understand how the values in the code are generated.

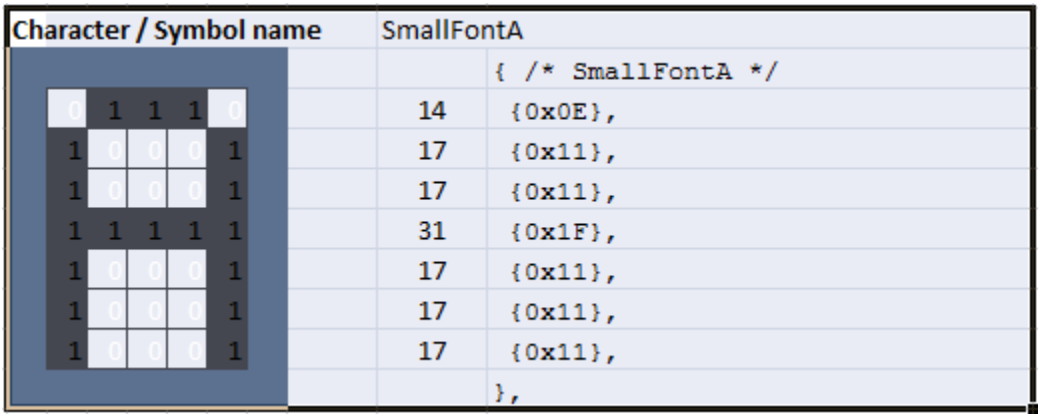

Figure 4.2.1.2: Pixel data and code for the letter A

Once these formulas are in place for a single character, the spreadsheet data can be copied for all the remaining characters in the font. Though the individual pixel values still must be manually set for each character, the rest of the code is automatically generated. It took only about 20 minutes to copy all of the character bitmaps into the LCD Worksheet. The total memory space required for the font library is 93 characters x (7 x (5 used pixels + 3 unused pixels)) x 1 bit/pixel = 5208 bits or 651 bytes. Clearly, as soon as you get into graphics, memory space becomes a factor!

4.2.2 Image Bitmaps

While characters can be optimized to 1 bit per pixel, grayscale images must store all color information within the bit map. Since this particular LCD uses 4-bits per pixel, a single byte can be used to store 2 pixels of information, where the values 0 through 15 (0x0 to 0xF) correspond to lightest to darkest. A representation is shown in Figure 4.2.2.1 with the grayscale colors as they correspond to their decimal, hex and grayscale RGB equivalents.

Dec	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Hex	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
RGB	0	17	28	41	51	77	95	119	128	150	178	192	221	234	248	255

Figure 4.2.2.1: Representation of 4-bit grayscale

The bit groups in the 2D array of Char map directly to the pixels that will be loaded to make up a character. Memory in the LCD is sequential from lowest to highest (little Endian), so be careful to get the nibbles of the bytes in the right order. The simplest image is a 1x1 pixel bitmap as shown in Figure 4.2.2.2.

Figure 4.2.2.2: A simple 1x1 pixel character

This would be represented in the LCD memory as 0x0f since the smallest memory element on the processor is a byte. The upper nibble 0x0 is white and the lower nibble 0xf is black. It is up to the LCD code to correctly parse the byte to extract the nibble for the single pixel.

The LCD Worksheet defines grayscale images in a very similar way to the single-color bit maps. Images are drawn using values 0-9 and A-F in the pixels which are again conditionally formatted to light up with the corresponding value of grayscale so you get a good representation of what the image will look like. An Excel formula is used to generate the code from the pixel information and put it into an array format that will be used by the LCD source code. Remember that for grayscale images, each pixel is defined by 4-bits of data. The MSB is on the left; be careful to get the nibble order correct.

Figure 4.2.2.3 shows a grayscale arrow symbol image highlighted so the pixel values are visible along with the corresponding code that is generated. If you look at the code carefully, you can see the pattern of bytes that correspond to the shape of the plus sign. Do you feel like you are in the Matrix?

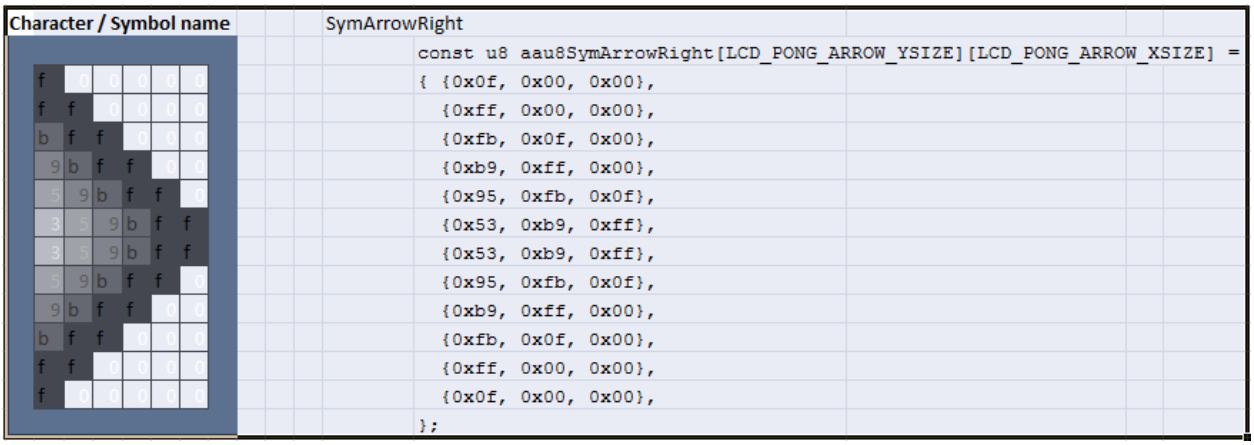

Figure 4.2.2.3: Grayscale image of an arrow with corresponding source code

If you examine the fifth row from the top for example, you can see the pixel values 5, 9, b, f, f, 0. The corresponding line of code is {0x95, 0xfb, 0x0f}. The source code that reads this data to get the bitmap information must know that the data is ordered in this way.

As with the character bitmaps, once the image tool is working properly it can be copied, pasted and resized as necessary to generate the images used in the program. Even the largest images are created this way by expanding the formulas. For each image that is created, there must be a corresponding row (YSIZE) and column (XSIZE) size definition which will be stored in the lcd_dotmatrix.h header file.

All of the image data is copied directly into lcd_bitmaps.h from the code columns in LCD Worksheet. Have a look at this file and be sure to understand where the data came from. Any changes to any of the bitmaps in the LCD Worksheet should be carried over to lcd_bitmaps.h. The changes will appear on the LCD once the code is rebuilt and reloaded onto the processor.

4.3 LCD Interface

Before getting to the really fun functions, we have to understand how to properly interface to the LCD controller. From the hardware chapter in the course, we know the microcontroller communicates to the LCD over the I^2C bus. We can get some clues for the kind of functions needed by examining the LCD Data sheet and looking at the example code that starts on page 11 showing how to initialize the LCD.

```
I2C_out(): sends a data byte equivalent to I2C_Write()
I2C_Start: sets a bus start condition like I2C_Start()
I2C_Stop: sets a bus stop condition like I2C_Stop()
Show(): sends data to the LCD following the correct protocol.
```

Chapter 3 has provided the low-level I^2C driver functions so all that is needed now are some functions to send commands and data to the LCD following the correct protocol. The LCD data sheet does not do the greatest job in explaining what exactly you must send to the LCD, but between the example code and

some of the other information it can be figured out with decent certainty. Figure 4.3.1 comes directly from the LCD datasheet and is one of the key pieces of information that explain how to use the LCD.

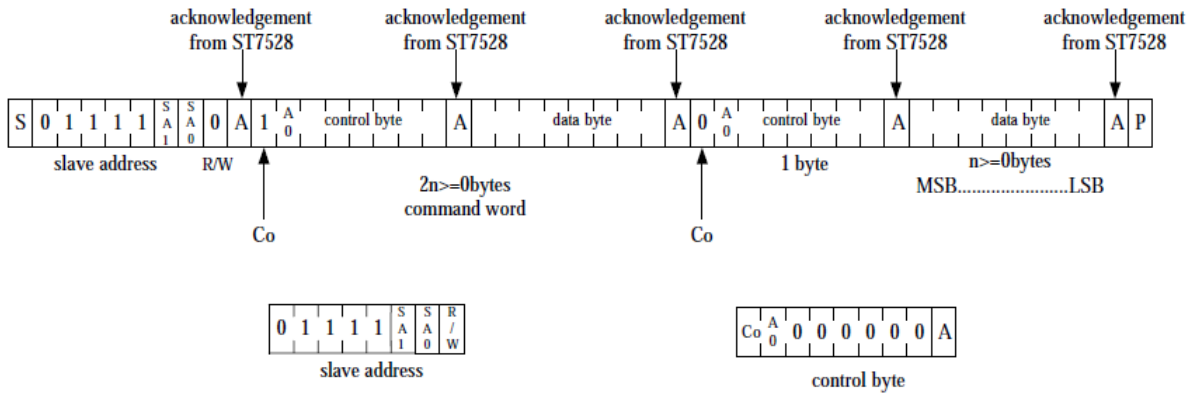

Figure 4.3.1: LCD data sequencing

In general, communication always starts with the LCD write address followed by a control byte and data byte(s). The control byte is mostly 0s, but has two bits, Co and A0 that change depending on what information is being sent. Bit Co is used to tell the LCD what to expect for data that will follow, either a single data byte followed by another control byte, or a stream of data bytes followed by a stop condition. In most situations, an LCD communication burst will use the latter case. In fact, the implementation of the LCD in the course code never sets Co to send multiple control bytes. This would only need to be done if you wanted to provide a command to set up something on the LCD followed immediately by a data stream.

Bit A0 is used regularly by the source code as it tells the LCD to distinguish between data that is meant as a command and data that is pixel information for display. Since this is used repeatedly throughout the code, there are constants defined in the LCD header file for the two different control bytes.

```
/* Communication type (command or data) byte */
#define LCD_CONTROL_CMD_SEND        (u8)0x00 /* The A0 bit clear for command */
#define LCD_CONTROL_DATA_SEND       (u8)0x40 /* The A0 bit set for data */
```

When sending information to the LCD it is critical that A0 is in the correct state. If not, commands that you think you are sending will be treated as data and will change pixel information, or vice-versa.

4.3.1 LCD Command Set

LCD commands tell the LCD controller how to behave. Commands do everything from setting contrast levels to specifying the write address where pixel data will be sent. The LCD data sheet has three pages of commands that the LCD controller will recognize. One of the first monotonous tasks to complete is setting up the preprocessor definitions in the lcd_dotmatrix.h header file. To be complete, every command byte should be included here even if you might not use them right away (or ever) in the code.

The definitions are made by referencing the Table of Commands that starts on page 8 of the LCD data sheet. Definitions are separated based on the "Ext" mode which is either 0 or 1. Notice that some commands have all bits completely defined, but others have variable bits that will need to be written on the fly. The preprocessor definition must be a complete byte, so any variable bit should be set to 0 so that updated values can simply be ORed in. The naming convention chosen to indicate this is the addition of a lowercase "x" in the definition name to remind the user to OR in some bits to complete the command.

```
#define LCD_ICONS_OFF               (u8)0xA2: a complete command
#define LCD_SET_PAGE_ADDRESSx       (u8)0xB0: a command with "x" to denote that
                                              additional information is required
```

Fortunately for you, all the definitions have already been made in lcd_dotmatrix.h. However, if you ever change LCD controllers, you will have to replace the current file with a new one that matches the new hardware.

4.3.2 LCD Data Transfer

The approach to writing to the LCD will be quite straightforward once all the data is prepared. Really all that is happening is moving bits from the copy of LCD memory in the MCU RAM to the display data RAM in the LCD. The addressing structure is the key here, so when defining the MCU LCD RAM it is imperative that the addressing is correctly mapped to the LCD RAM.

4.3.2.1 MCU LCD RAM Organization

Ideally, the LCD data addresses are organized as rows and columns. Remember that for the 4-bit grayscale LCD, each pixel address requires one nibble of data to define its color. Figure 4.3.2.1.1 shows the addressing used in the processor memory and how it relates to the actual pixel number on the LCD. Remember that the top right area of the screen is pixel 0, 0 and the bottom left is pixel 159, 59.

The addressing along the top shows the actual column numbers and the corresponding LCD RAM addresses where the pixel information for those columns will be stored. Since there are 4 bits per pixel, two pixels can be defined per byte in the LCD RAM to cut the amount of memory used in half – this is significant since so much memory is required to maintain the LCD RAM. It would be slightly easier to map and code the pixel data with a single byte per pixel column, but memory requirements would then be 16kB instead of only 8kB. In fact, the LPC2142 processor would run out of memory if this were the case!

The row addresses increment as expected and will map 1-to-1 to the LCD RAM on board the MCU. Notice the additional "Page" and "Bit" information that groups the rows into pages – this is for the LCD controller RAM and will be discussed further momentarily. For now, just note how all the addressing works. If you would like to see the full image with all of the addresses, it is available in the LCD Worksheet.

		LCD RAM Address	79	79	78	78	77	77	76	76	75	75	74	74	73	73	72	72	71	71	70	70	69	69	68
		High / Low Nibble	H	L	H	L	H	L	H	L	H	L	H	L	H	L	H	L	H	L	H	L	H	L	H
		Column (decimal)	159	158	157	156	155	154	153	152	151	150	149	148	147	146	145	144	143	142	141	140	139	138	137
		Column (hex)	9F	9E	9D	9C	9B	9A	99	98	97	96	95	94	93	92	91	90	8F	8E	8D	8C	8B	8A	89
Row (decimal)	Row (hex)	Page	Bit																						
0	00	0	0																						
1	01	0	1																						
2	02	0	2																						
3	03	0	3																						
4	04	0	4																						
5	05	0	5																						
6	06	0	6																						
7	07	0	7																						
8	08	1	0																						
9	09	1	1																						
10	0A	1	2																						
11	0B	1	3																						
12	0C	1	4																						
13	0D	1	5																						
14	0E	1	6																						
15	0F	1	7																						
16	10	2	0																						
17	11	2	1																						
18	12	2	2																						

Figure 4.3.2.1.1: The top left corner of the LCD addressing

4.3.2.2 LCD Controller RAM Organization

The LCD controller does not address rows directly, but rather groups of 8 rows in pages. The LCD memory on the controller is organized into 13 8-bit pages and 160 four-bit column addresses. There are commands to set the starting page address and column address for a write operation, but no commands to select a row address – you must determine what page the row you want to write is in and set the page address that includes that row. This also implies that if you want to write pixels on adjacent pages, you must address the first page and column(s), write the data, then increment the page address to write the pixels in the adjacent page.

The controller also happens to use what seems like a very bizarre way to load pixel data. Logically, once the address is set, you would write the 4-bit pixel data (or maybe write 2 pixels at a time with a single byte). However, since the addressing scheme addresses pages and not rows, this approach is not used. Instead, one bit of data for the entire addressed column of pixels (8 vertical pixels) in the current page is written with each byte sent to the LCD. Since each pixel requires 4 bits of data to define the grayscale level, 4 bytes of data must be sent per column to fully define the bits. While this data is being clocked in, the "main" column address does not change. Only after the fourth bit is written to a column is the main column address pointer automatically incremented inside the LCD controller so the next 4 bytes of data sent to the LCD will write the bit data for the next column.

To be complete, we must comment that the “column address” that you send to the LCD controller is actually bits 9:2 of a larger column address inside the controller. The bottom two bits 1:0 are “internal column” address bits. When the 4 bytes of data are written to the LCD controller by the MCU, the internal column address bits increment after each byte write. After the fourth byte is sent, the internal bits will roll from b’11’ to b’00’ and subsequently increase bit 2 of the column address. As you can probably guess, if you do not write LCD data to the controller in groups of 4 bytes, than you will not have complete data for the pixels and the internal LCD addressing will be corrupt.

The implication of all this is that you cannot ever write a single pixel on this LCD. The smallest number of pixels that can be refreshed is a 1 column x 8 row line and it will always take a minimum of four bytes sent to the LCD controller from the MCU to properly set those pixels. If you really do only want to update one pixel in the 1x8 column, you must be sure to write the same data that is already present for the other 7 pixels so their information does not change. This is where keeping a local copy of all pixel data in the LCD RAM becomes essential as individual pixels can be changed in the local RAM as the application requires it, and pixel data for unchanged pixels will still be present for the LCD refresh.

This is without a doubt confusing, but fear not, not all LCD controllers end up being this difficult. It will take some time to fully understand, so be patient. Since you have working code, you can step through an LCD refresh and see what gets updated as data is clocked out. Figure 4.3.2.2.1 attempts to graphically show the mapping that takes place if we wanted to write data to set a column of 8 pixels from totally black (intensity 15) down to totally white (intensity 0) in steps of 2 or 3. Take some time to study this to fully understand how the LCD data is written to the controller. The process that is carried out here is as follows:

1. The page address of the LCD is set to 0 and the column address is set to 0
2. Four bytes of data are sent to the LCD controller: 0x80, 0xAA, 0xCC, 0xF0.

When Page = 0 and Column = 0, writing the first byte (0x80 highlighted in red) writes the LSB of the 4-bit grayscale value for page 0-row 0-col 0, page 0-row 1-col 0, …, page 0-row 7-col 0. You can see the value 0x80 appears vertically highlighted in red. The internal column data pointer increments by one which means it now points to bit 1 of the 4-bit column 0 data nibble. Sending the next byte (0xAA highlighted in green) writes bit 1 of the 4-bit grayscale value for page 0-row 0-col 0, page 0-row 1-col 0, …, page 0-row 7-col 0 and the internal column pointer increments again. The next byte sent (0xCC highlighted in blue) writes the values for bit 2 and the fourth byte sent (0xF0 highlighted in purple) writes the values for bit 3. At this time when the internal column pointer increments, it will overflow thus incrementing the main column address. The internal bits are back to 0 so that the next 4 data bytes that the MCU sends will set bits 0 – 3 of the second column of pixels. Each pixel in “main” column 0 is now fully defined with 4-bits (read horizontally across all four highlighted colors for a single pixel).

Page	Page Row	LCD Row	Column Read/Write Address (Bits9:2). This is also the pixel number. 159 158 157 156 . . . . 2 1 0 / Column Internal Address (Bits1:0 incremented automatically with each data write) 3 2 1 0 … 3 2 1 0
0	0	0	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 1 1 1 1
0	1	1	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 1 1 0 0
0	2	2	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 1 0 1 0
0	3	3	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 1 0 0 0
0	4	4	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 1 1 0
0	5	5	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 1 0 0
0	6	6	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 1 0
0	7	7	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
1	0	8	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
1	1	9	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
1	2	10	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
1	3	11	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
1	4	12	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
1	5	13	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
1	6	14	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
1	7	15	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
.	.	.	. . . . .
.	.	.	. . . . .
.	.	.	. . . . .
11	0	88	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
11	1	89	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
11	2	90	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
11	3	91	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
11	4	92	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
11	5	93	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
11	6	94	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
11	7	95	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
12	0	96	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
12	1	97	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
12	2	98	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
12	3	99	0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0 … 0 0 0 0 0 0 0 0
12	4	x	
12	5	x	
12	6	x	x = Not applicable: nothing happens if data is written here

	Page Addr	Set Col Addr	Internal Col Addr
	x	x	x
Set Page Address = 0	0	x	x
Set Column Address = 0	0	0	0
First data byte 0x80	0	0	0
Second data byte 0xAA	0	0	1
Third data byte 0xCC	0	0	2
Fourth data byte 0xF0	0	1	3

Resulting LCD image (column 0, rows 0 - 7):

- b'1111' = (intensity 15)
- b'1100' (Intensity 12)
- b'1010' (Intensity 10)
- b'1000' (Intensity 8)
- b'0110' (Intensity 6)
- b'0100' (Intensity 4)
- b'0010' (Intensity 2)
- b'0000' (Intensity 0)

Figure 4.3.2.2.1: LCD controller addressing and data write mapping

The way the LCD controller is built causes the addressing scheme to be a bit complicated when mapping from the "normally" formatted row x column 2D matrix in MCU RAM to the page x column formatted LCD controller RAM, but such is life and with careful programming it is not that bad. The really nice thing about working with an LCD is that you have visual feedback if things are working or not!

4.4 LCD Driver Functions

If we were writing code in C++, this discussion would be about defining an LCD class. In some ways it would be nice as a base class could be written and classes for each different LCD could be derived. However, since we are writing in C, we will simply call this our LCD driver / application and include all the code for the course LCD here.

All the functions will be stored in the lcd_dotmatrix.c and lcd_dotmatrix.h files, though some interface functionality will be required in different locations to allow other applications to use the LCD. We want to create a driver set that is intuitive and easy to use and that is hopefully scalable anywhere from smaller, single-bit-per-pixel LCDs to full HD, 24-bit color displays.

There are not too many functions necessary in the LCD driver, though some will require very careful thought to implement correctly as alluded to by the discussion of the LCD pixel mapping. As with all applications, the LCD should be initialized and be ready for other applications to make use of it. Public functions to load single color and grayscale bitmaps are required along with functions to erase the screen or portions thereof. Since the LCD operates by setting a starting address and then clocking data in, there should be some private utility functions to manage lower level tasks like that.

4.4.1 LCD Message Sender

What we have not talked about yet is actually sending the data to the LCD. This will require using the I²C drivers that have already been built to talk to the LCD controller and feed it data in the correct format. Data transfer at the I²C level will be handled entirely by the MessageSender function that has also already been written. LCD Command messages will be queued using QueueTxMessage() just like with the UART functions from the previous chapter.

The message source is LCD_MSG_SRC which will queue MessageSender() to use the correct I²C bus that has been defined for the LCD. The correct format of the message content will be up to the LCD application to ensure just like with any other message sent. The data must be in an array and a pointer to the array is passed during the call to QueueTxMessage(). There is a slight inefficiency here because LCD command messages can be quite long so copying the data takes processor resources that could be avoided by a more direct approach. However, the modularity of the system would suffer. As it turns out, the long LCD command messages only happen during initialization so this becomes a non-issue.

4.4.1.1 QueueTxMessageLCDData ()

For LCD pixel data messages that can also be extremely long, a variation of QueueTxMessage will be written called QueueTxMessageLCDData(). This is also where the memory mapping translation that has

to occur will take place which is an unavoidable inefficiency anyway, so at least some work is being done as data is copied over. The structure of the function is shown in the listing below.

```
bool QueueTxMessageLCDData(u8 u8Page_, u16 u16ColumnStart_, u16 u16ColumnEnd_)
{

  /* Allocate memory for the new data and new message and check for fail */
  /* Copy data to the newly allocated message structure */
  /* Write the header bytes */
  /* Create the message data 1 LCD page column at a time */

  while(u16ColumnsToUpdate > 0)
  {
    /* Initialize the variables for this column of pixel data */
    /* Initialize 4-byte array that corresponds to one column of data in a page. */

    /* Create the 4 bytes of data required to write the current column of 8 pixels.
    This is done by getting the current nibble of data from LCDRam and mapping the
    bits into 4 successive chars in pu8Data.  This is repeated 8 times (once for each
    4-bit pixel nibble in the 8 rows of LCD RAM that correspond to the LCD page rows)
    to get data for all 8 rows in the LCD page. */

    for(u8 j = 0; j < LCD_PAGE_SIZE; j++)
    {
      /* Get the pixel information at the current Local RAM column / current row */

      /* Populate the current bits into the data arrays */
    }

    /* Pixel data arrays for the current column are now ready, move to next column */
  } /* end while(u16ColumnsToUpdate > 0) */

  /* All data for this page has been queued, so update the links in the message
  linked-list, update flags and exit.  */

  return(TRUE);

} /* end QueueTxMessageLCDData */
```

The mechanics of the function are not particularly difficult, but the implementation of the memory mapping is definitely challenging. Several techniques involving bit masking are used that allow the code to be written efficiently so a few nested loops can perform all of the functionality needed (there two FOR loops nested inside a WHILE loop). The best way to figure out what is happening is by drawing some pictures and following through the code to see how the data is moving about. A few of the key memory indexing lines of code are discussed below:

```
/* Allocate memory for the new data and new message and check for fail */
pu8Data = malloc( ( (u16ColumnsToUpdate * LCD_PIXEL_BITS) +
sizeof(au8LCDSendDataHeader) ) * sizeof(u8) );
```

Here an array of data is allocated that will contain the LCD controller formatted pixel information and be sent directly to the LCD. At a minimum, this array is 6 bytes long because a data message has two header bytes and must update a least one column of pixels which we know requires 4 bytes. In general, the size of this array will be 2 + 4n, where n is the number of columns that will be updated. The constant LCD_PIXEL_BITS is 4.

```
/* Get the pixel information at the current Local RAM column / current row */
u8CurrentLocalRAMNibble =
GGaau8LCDRAMImage[u16LocalRAMCurrentRow][u16LocalRAMCurrentColumn / 2];
if(u16LocalRAMCurrentColumn & 0x0001) /* modulo two without the math */
{
  /* For odd LCD RAM column numbers, need the high nibble */
  u8CurrentLocalRAMNibble  >>= 4;
}
```

This code grabs the current nibble of pixel data from the MCU LCD RAM that will be loaded in LCD RAM. The pixel address is straightforward beyond dividing the column address by two to handle the 2-pixel per byte storage. Integer division ensures that this will address correctly. Once the location of the nibble is known, the code selects the correct nibble based on the original column address using a bit mask to clear all but the LSB which determines which nibble to use.

```
u8CurrentLocalRAMPixelBitMask = 0x01;

/* Populate the current bits into the data arrays */
for(u8 i = 0; i < LCD_PIXEL_BITS; i++)
{
  /* It a bit in the nibble is set, then set the bit in the current byte of pu8Data*/
  if(u8CurrentLocalRAMNibble & u8CurrentLocalRAMPixelBitMask)
  {
    pu8Data[i] |= u8CurrentRowInLCDPageBitMask;
  }
/* Advance the bit mask to look at the next bit in the nibble */
u8CurrentLocalRAMPixelBitMask <<= 1;
}
```

Here the bits for the current pixel are spread out across four elements of the pu8Data array. Two tricks are used here:

1. The pointer / array duality is used so pu8Data can be indexed. pu8Data points to the start of a four-byte section in the array, so pu8Data[0] is the first byte which corresponds to the first internal column of pixel data, pu8Data[1] is the second byte, etc. The pointer is not moved until all pixel data has been written which takes 8 runs through the FOR J loop (LCD_PAGE_SIZE = 8).

2. The elements of pu8Data are all initialized to 0 at the start of a column iteration and a bit mask (u8CurrentLocalRAMPixelBitMask) is used to set bits if required by the pixel data. The bit mask

essentially addresses individual bits within a byte (since there is no way to use C syntax to address anything smaller than a byte).

If you can understand these three sections of code, then you should have a pretty good grasp on the entire LCD address translation between the local LCD RAM and the data that must be written to the LCD controller. Even if you are not entirely comfortable to the way the mapping works, QueueTxMessageLCDData() completely abstracts the mapping away from any other code that is written for the LCD application. You can ignore all of this discussion and simply focus on the row and column indexing as implemented in the LCD RAM (where the only mildly tricky part is managing the nibble addressing).

4.4.2 LCDInitialize()

Now that we have the mechanisms to send data reliably to the LCD, we can focus on the functions that organize the data and send the correct information. If you have ever worked with an LCD in the past, you know that before anything can be displayed the LCD must be initialized and properly configured. The course LCD is no exception and requires a fairly lengthy initialization sequence to be fully setup. There is nothing difficult about this initialization other than correctly formatting the data into an array to be passed off to the LCD controller.

The data to be sent comes from the LCD data sheet. There are four groups of data each separated by a delay to allow certain setup times inside the controller – this information also comes from the data sheet. Each of the four data groups are defined as arrays inside the LCDInitialize() function in lcd_dotmatrix.c. The mnemonic definitions are all defined in lcd_dotmatrix.h. As you can see, there is a fair amount of data to send, especially when it comes to defining the contrast levels. The default values are selected which seem to work well for most LCDs tested, but they can be adjusted for individual needs.

LCDInitialize() also provides some debug output to indicate what is going on, and adds a splash screen that first turns all of the pixels on as a pixel test, then displays a logo that provides visual clarification that the LCD is correctly operating. All of these elements are provided through function calls, which leaves nothing more to discuss.

4.4.3 LCD RAM functions

The majority of the code in lcd_dotmatrix.c sets and clears pixel data within the LCD RAM space. Remember that mapping data to the LCD controller is handled entirely by QueueTxMessageLCDData(), so we only have to worry about the LCD RAM as organized in an intuitive 2D array. The definition of that array is Global Global so it is accessible by any function in the source code.

```
u8 GGaau8LCDRAMImage[LCD_IMAGE_ROWS][LCD_IMAGE_COLUMNS];
```

If you look at this variable in the debugger, you will see how big it is, but also how it makes sense. Data can be written to this array to set and clear pixels but nothing will happen on the LCD screen until a

screen refresh occurs, which requires defining the pixels to be updated and providing the data to message sender. The area to be updated is always a rectangular box with minimum size 1x1 pixel. These parameters are stored in the Global Global GGstLCDUpdateDataParams which is of type UpdateDataParamsType. The typedef for UpdateDataParamsType is in lcd_dotmatrix.h

```
/* Structure type for current LCD update information */
typedef struct
{
  u16 u16RowStart;
  u16 u16RowEnd;
  u16 u16ColumnStart;
  u16 u16ColumnEnd;
} UpdateDataParamsType;
```

Whenever an LCD update is desired, the fields in GGstLCDUpdateDataParams must be set to correctly define the area of the LCD screen to be updated. It is through this definition that the corresponding locations within GGaau8LCDRAMImage[] are sent over to the LCD controller. The obvious assumption is that the data in GGaau8LCDRAMImage[] is properly setup with the pixel data to be transferred. The next few sections describe the functions used to set or clear pixel data in LCD RAM.

4.4.3.1 LCDClearRAMArea()

An obvious function needed is one that clears pixel data from LCD RAM. The arguments to this function are the row and column numbers that define the starting pixel of a rectangular area of memory to clear, and the size of the rectangular area to clear. Any values can be used with minimum 1 and maximum 159 for columns and 99 for rows. Currently there is no error checking for invalid data – this is something that you would want to add for production quality code.

This function is the probably the best example to use to ensure you understand how indexing works to the LCD RAM array. All of the functions that manipulate LCD RAM will use a pair of nested loops to index the desired pixels. Provisions must always be made to handle the fact that each byte in the array holds two pixel nibbles.

```
  for(u16 i = 0; i < u8RowSize_; i++)
  {
    for(u16 j = 0; j < u8ColumnSize_; j++)
    {
      /* Calculate the current pixel */
       u16Pixel = u8ColumnSize_ - 1 + u16Column_ - j;

      /* On odd column number, clear the high nibble */
      if( (u16Pixel) & 0x0001)
      {
        GGaau8LCDRAMImage[u16Row_ + i][u16Pixel / 2] &= 0x0F;
      }

      /* On even column number, clear the low nibble */
      else
```

```
      {
        GGaau8LCDRAMImage[u16Row_ + i][u16Pixel / 2] &= 0xF0;
      }
    }
  }
```

The function loops through each column in each row. The calculation of u16Pixel is the main operation to select the correct column index in LCD RAM. The nibble selection is based on this value. Clearing the low nibble is done by logically ANDing 0xF0; clearing the high nibble is done by logically ANDing 0x0F. Also notice that the bits are actually cleared left to right (highest index to lowest). The reason for this is to match the pixel setting functions that load bitmaps in the same order as it is a bit more intuitive.

4.4.3.2 LCDLoadSingleColorBitmap()

This function takes a single-color bit map (defined by 1 bit per pixel) and loads it into the LCD RAM. The arguments specify the top-right pixel location for the bitmap and the size of the image being loaded. These values come from bitmaps defined in the LCD Worksheet that were copied over to lcd_bitmaps.h. A pointer to the bitmap array is also provided as an argument along with the grayscale value desired for the image.

Reading a single color bitmap takes advantage of a sliding bit mask so that a nice looping structure can be used to parse through all the data. Each bit in the bitmap data is read by ANDing it with the current value of the bit mask that will mask off all but the pixel data bit of interest. If the bit of interest is set, then the u8Grayscale_ value is loaded into the corresponding pixel in LCD RAM. The pixel is always cleared first, so if a particular bit is not set, then the pixel is properly cleared.

For images that are wider than 8 column pixels, data is handled in “bit groups”. The number of bit groups equals 1 + the image width in pixels integer divided by 8. In general, the hierarchy of indexing that is occurring in this function is row > bit group > pixel in bit group. The indexing for the row loop (i) and column loop (j) should be easily understood along with the handling of the pixel nibbles. So the only new addressing that is tricky is the access to the correct pixel in LCD RAM and the operation to load the grayscale value or clear the pixel.

```
/* If odd column number in the LCD RAM, clear then load the high nibble */
if( ( j + u16Column_) & 0x0001 )
{
  GGaau8LCDRAMImage[u16Row_ + i][(j + u16Column_) / 2] &= 0x0F;
  if(aau8Bitmap_[ (i * u16BitGroups) + (j / 8)] & u8CurrentBitMask)
  {
    GGaau8LCDRAMImage[u16Row_ + i][(j + u16Column_) / 2] |= (u8Grayscale_ << 4);
  }
}
```

Indexing the row current row of GGaau8LCDRAMImage is straightforward: just the starting row plus the current index of the ‘i’ loop [u16Row_ + i]. To get the column index correct, the absolute column

starting address and current index of the 'j' loop is used, but then divided by two to access the nibble [(j + u16Column_) / 2].

Looking at the correct bit in the image bitmaps is done by carefully indexing what is actually a 2D array with a single array since that is the only way that C syntax allows the array pointer to be passed (there are ways to work around this but they become far uglier solutions than simply handling the 2D array as a pointer to a 1D array). The index of the bit group that contains the pixel data bit of interest is the current row index i multiplied by the number of bit groups in the image + the column offset j / 8. This value is ANDed with the current bit mask to pick out the single bit of interest.

The explanation makes it sound more confusing than it actually is. To fully understand it, it helps to draw a picture and trace through the code to see what is happening. The good news is that this is the end of the tricky LCD code!

4.4.3.3 LCDLoadGrayscaleBitmap()

Loading a grayscale bitmap is actually easier than loading a single color bit map because the grayscale bit maps store pixel nibbles that are copied directly into LCD RAM. The function arguments are identical to the single color bitmap except there is no need for a grayscale value since the color data is contained in the bitmap information.

Nested row and column data loops are used to index through the image and the LCD RAM. Management of the correct nibble of the bitmap and of the LCD RAM must be made independently since the image could be loaded to any location on the LCD and one cannot assume that the high nibble from the bitmap will load to the high nibble in LCD RAM (betting on a 50/50 chance is not wise!).

4.4.3.4 LCDLoadStringX()

Writing character strings to the LCD is an important function to have and the implementation should be very close to simply providing a C-string pointer to a function. There are currently two font sizes available in the course firmware, "small" and "big," though the "big" font only contains bitmaps for numerical digits 0 – 9. Separate functions were written to handle the two font sizes as it was less code than parameterizing a single function to handle both (and more in the future). If this were done, an additional "font" argument could be passed in, and all font bitmaps used would have to be fully defined.

Both functions take arguments to the top LEFT pixel of the string to be printed. This is different than the other image loading functions that use the top right pixel of the image. Using the top left pixel makes it much easier to get the string to end up where you want it to since almost all English text is left-justified. The characters are printed with repeated calls to LCDLoadSingleColorBitmap() using ASCII indexing to get the correct bitmap image. The font color is also added on the fly so different intensities of characters can be printed if desired.

There is currently no error checking for strings that extend past the limits of the LCD screen. If you do send a string that is too long to fit on the screen, you will encounter a Hard Fault memory access error

and the firmware will freeze because it jumps to a trap state for debugging purposes. Clearly you would want to change this for production code, but it remains as-is in the course code to flag that an error has occurred.

4.4.4 Other LCD Functions

A few more functions are required to tie in the LCD application to the LCD controller.

4.4.4.1 LCDSetStartAddressForDataTransfer()

Pixel data transfers do not contain any information about where the data should be loaded to the LCD. The LCD controller has address registers that hold the starting position for data writes that are set up prior to streaming the pixel data. This information includes the page number and the column number within the page and is set by an addressing command.

The LCD application stores the address information in the Global Global GGstLCDUpdateDataParams which is the variable that LCDSetStartAddressForDataTransfer() will always refer to when sending the starting address to the LCD. Therefore the function does not require any arguments but must assume the data in GGstLCDUpdateDataParams is always correct.

LCDSetStartAddressForDataTransfer also handles split-page or multi-page writes based on the starting and ending row values in GGstLCDUpdateDataParams. It will update a pair of Local Global variables, LGu8PagesToSend and LGu8CurrentPageToSend, which are used by the LCD application to correctly load page data.

4.4.4.2 LCDUpdateFullScreen()

A common function is to update the entire LCD screen. To save multiple instances of repeated code to set GGstLCDUpdateDataParams to write the entire screen, this function is written to carry out the task of setting GGstLCDUpdateDataParams for a full page update. No arguments are required.

4.5 LCD Application

Finally we can look at the LCD application that takes care of initiating updates to the LCD controller. The app is responsible for managing all screen update messages and does not care about the source of any message – it will simply do what it is told.

4.5.1 GGu32LCDFlags

Six flag bits are used to manage application data flow inside the LCD state machine.

```
/* GGu32LCDFlags */
#define _LCD_FLAGS_MESSAGE_IN_QUEUE /* An LCD message is in the MessageSenderQueue */
#define _LCD_FLAGS_MORE_PAGES /* There are more pages to send */
#define _LCD_FLAGS_SM_MANUAL /* Run the LCD SM in manual mode */
#define _LCD_FLAGS_UPDATE_IN_PROGRESS /* LCD update is currently in progress */
#define _LCD_FLAGS_LCD_BUSY /* An application has control of the LCD */
#define _LCD_UPDATE_SCREEN /* The LCD should refresh a portion of the screen */
```

These flags are set by various functions to queue the state machine advances or provide status information to the LCD application or other applications that want to use the LCD.

4.5.2 LCD Application States

The LCD application is relatively simple and implemented in just four short states. Generally speaking, the application is either waiting to update the screen, updating the screen, or in an error state. It uses the _LCD_FLAGS_MESSAGE_IN_QUEUE flag to manage state transitions and ensure that LCD messages are fully delivered to the LCD before new messages are sent.

4.5.2.1 LCDIdle()

When no updates are queued, the LCD application will do absolutely nothing and exit immediately. If _LCD_FLAGS_MESSAGE_IN_QUEUE is set, then the application will proceed to transfer data via MessageSender() to the LCD controller. Note that only LCDSetStartAddressForDataTransfer() ever sets _LCD_FLAGS_MESSAGE_IN_QUEUE which is a safety mechanism to ensure that the starting address in the LCD has been set (though it cannot guarantee that whomever called LCDSetStartAddressForDataTransfer() did so when the start address parameters were set correctly).

4.5.2.2 LCDSMWaitAddressTransfer()

The starting address must be fully sent to the LCD before any data is sent. LCDSMWaitAddressTransfer() will not do anything until _LCD_FLAGS_MESSAGE_IN_QUEUE is cleared. Once this occurs, the full data message is queued to the MessageSender using QueueTxMessageLCDData() and parameters that must correctly be set in GGstLCDUpdateDataParams. The flag _LCD_FLAGS_MESSAGE_IN_QUEUE is set again while the data message is in the queue.

4.5.2.3 LCDSMWaitDataTransfer()

Similar to waiting for the address transfer, the LCD state machine will hold states while data is being clocked out to the LCD. Data being sent is only ever as large as a complete single page of data, so the longest period of time in which this state will be active is about 640 bytes x 9 bits per word / 400kbps = 15ms. Remember that the state does not actually hold for that amount of time – it will simply run through without doing anything for 15 iterations of the main system loop for a total time of 15ms before advancing. If the state used a while loop that held the code until _LCD_FLAGS_MESSAGE_IN_QUEUE was cleared, then it would be in violation of the system design and the desired 1ms loop timing.

Once _LCD_FLAGS_MESSAGE_IN_QUEUE is cleared, the state makes calculations to decide if there is another page of data to be sent. In either case, the appropriate Local Global variables and status flags are updated and the application is pointed back to the Idle state.

4.6 Chapter Exercises

Creating the LCD application certainly involves some intricate code that challenges your knowledge of memory mapping and indexing. However, the final application ends up being very straight forward and all of the tricky software is sufficiently hidden under a simple API.

The goal of this chapter exercise is to make use of the API and prove that you are comfortable with the implementation and all that is required to make it work. We start off with a simple exercise to test that you can use the LCD Worksheet to generate an image and import it into the code, and finish with a standalone application to use the LCD functionality.

4.6.1 Exercise 1

There is a blank image at the bottom of the "Images" tab in the LCD Worksheet called MyLogo. Create a personalized image to give your board a unique startup splash screen. Copy the data of the image into lcd_bitmaps.h and write code in LCDInitialize() function to display it after the pixel test.

4.6.2 Exercise 2

Write an application that will print your name in the center of the screen, then allow you to move it around the screen with the trackball much like the Board Test program that you used back in Chapter 1 to test the LCD and trackball. Do not use an image for your name, use a call to LCDLoadStringSmall() where the starting pixel is determined by the current trackball position. Make sure to set the trackball limits so the message does not print off screen and crash the program.