1 Trademarks

MSP430, MSP430Ware are trademarks of Texas Instruments.
Stellaris is a registered trademark of Texas Instruments.
All other trademarks are the property of their respective owners.

Application Report
SLAA548–October 2012

Design Considerations When Using MSP430 Graphics Library

Michael Stein

ABSTRACT

LCDs are a growing commodity in today's market with products as diverse as children's toys to medical devices. Modern LCDs, along with the graphics displayed on them, are growing in complexity. A graphics library can simplify and accelerate development while creating the desired user experience. TI provides the MSP430 Graphics Library for use in developing products with the MSP430™ MCU. This application report describes design considerations when using the MSP430 Graphics Library, and provides an example of implementation and optimization. Project collateral discussed in this application report can be downloaded from the following URL: www.ti.com/lit/zip/SLAA548.

Contents

List of Figures

List of Tables

2 Introduction to the MSP430 Graphics Library

Texas Instruments' (TI) MSP430 Graphics Library is an open source set of graphics primitives for creating graphical user interfaces. The MSP430 Graphics Library is built into TI's MSP430Ware™ software suite. The graphics primitives include functions to draw individual pixels, lines, rectangles, circles, text, and images.

The MSP430 Graphics Library provides the flexibility to interface any dot matrix LCD to any MSP430. It is compatible with a wide variety of LCDs through customization of a low-level abstraction layer. Up to 16-bit color is supported, as well as grayscale. There is no inherent limit to resolution; an MSP430 running the graphics library performs well on a QVGA, and this is the display used in this document's example.

Interfacing with dot matrix displays is made possible by the faster processing speeds of the MSP430 along with the efficiency of the MSP430 Graphics Library. As an integral part of an MSP430 Graphics Library application, efficiency must be maintained throughout the design. This efficiency has a direct impact on the application's drawing speed.

This document provides guidelines for designing an application with the MSP430 Graphics Library. An example implementation is provided to help illustrate some of the key decisions and their impact on the performance.

3 System Overview

Dot matrix LCD displays are often controlled with an embedded LCD driver or controller, typically called a chip-on-glass. These LCD modules have embedded RAM for the display and handle synchronization, pixel control, and pixel signal conversion. The chip-on-glass LCD controllers support standard communication protocols for easy connection with microcontrollers. The MSP430 Graphics Library is intended to interface to LCD display modules equipped with a chip-on-glass LCD controller.

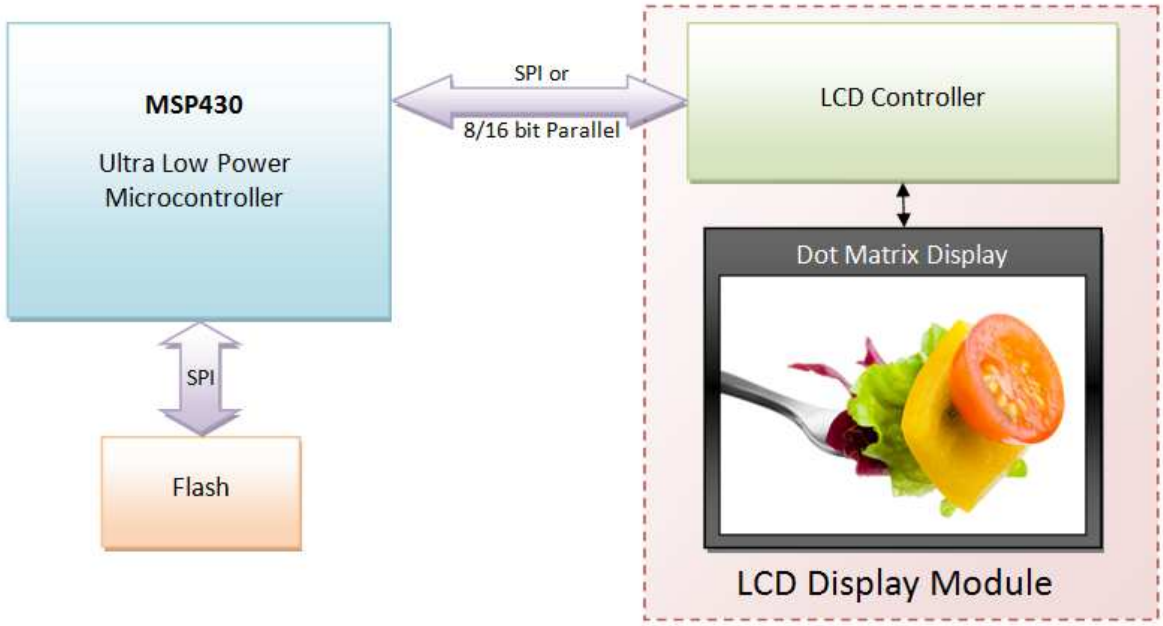

Figure 1. System Overview Block Diagram

The MSP430 Graphics Library is split into two main layers: the LCD display driver and the graphics primitives. The LCD display driver layer is the hardware abstraction layer that controls how the MSP430 communicates with the LCD. The graphics primitives layer contains functions to draw pixels, lines, rectangles, circles, text, and images.

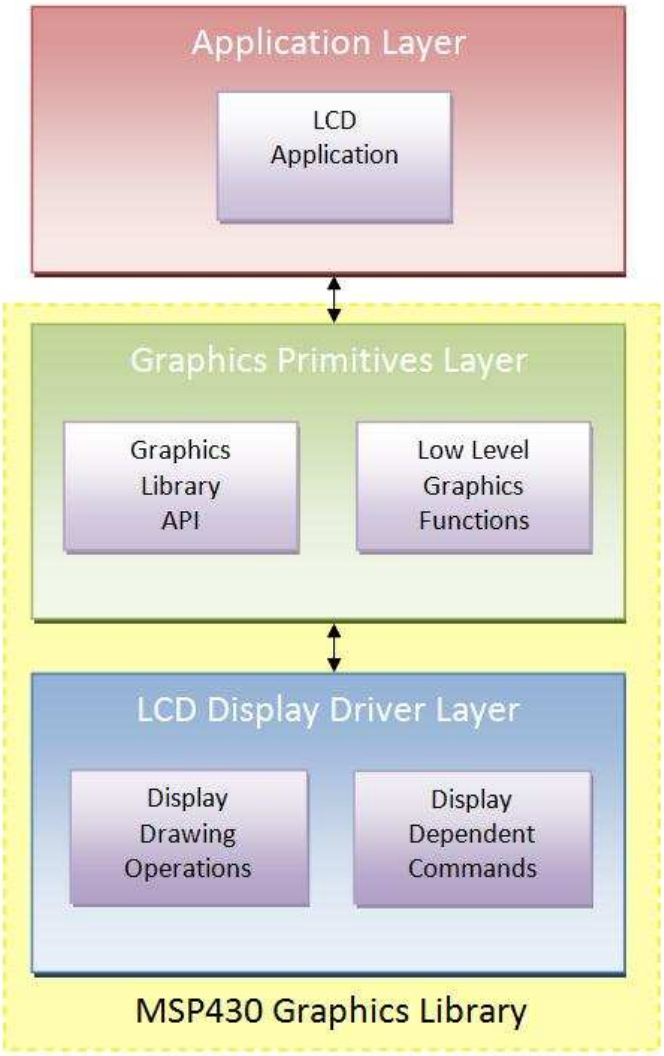

Figure 2. System Overview Stack Diagram

4 Hardware Implementation - LCD Bus Type

Selecting the bus connection of the LCD is an important step. There are several LCD types supporting multiple methods of connectivity, but the most common bus types are parallel and serial peripheral interface (SPI). Understanding the tradeoffs allows the bus type to be chosen to fit the application.

4.1 Parallel Bus

The parallel bus will most likely be the best option for speed if fast drawing is required. This is due to the bit banging operations that allow for rapid drawing. The Motorola 6800 and Intel 8080 buses have a specific signal to indicate if a command or data is being sent. This allows for faster data transactions to occur because there is often no built in protocol around a data write requiring additional bytes to be transferred.

If performance is not essential, a parallel bus may not be the best choice. It requires a lot of general-purpose input/output (GPIO) pins to implement, and LCD controllers that accept a parallel bus require more board space to connect to the MSP430.

4.2 SPI Bus

The SPI bus pairs well with smaller LCDs because they contain far less pixels and, therefore, can tolerate lower data throughput. This type of interface requires significantly less GPIO resources from the MSP430 and can use integrated SPI hardware modules on the MSP430. The SPI bus is flexible as it can be implemented on a number of various SPI capable ports on the MSP430.

The SPI bus has limitations depending on the selected LCD and corresponding LCD controller. The limitations include the clock speed and protocol. For instance, a very fast SPI bus does not increase performance if the protocol requires a larger number of bytes to be transferred. The protocol is very important; in particular the number of bytes that need to be transferred for a data write. The data write is the operation called most frequently for all functions, and the speed at which it can be executed will significantly impact data throughput.

Table 1. Tradeoffs of Using Parallel and SPI Bus

	SPI Bus	Parallel Bus
Advantages	• Fewer I/O required • Flexible	• Fast write speeds • Standard data protocol
Disadvantages	• Varying LCD command protocols • Varying write speeds	• Requires more I/O pins

5 Software Implementation- LCD Display Driver Layer

Customization of the LCD display layer is what allows the graphics library to be used with a wide variety of LCD modules, accounting for the variety in interface timings, resolutions, command protocols, and individual features. Each LCD requires careful customization of this layer. The efficiency of the code in this layer has perhaps more impact on draw times than even the library itself does, because the low-level functions it contains are executed for each pixel drawn. For this reason, it is important to get to know the LCD controller well at an early stage so that the best choices can be made in implementing these functions.

5.1 Driver Layer Components

The LCD display driver layer should be made up of several basic functions required by the MSP430 Graphics Library: color translation, draw pixel, draw horizontal line, draw vertical line, draw filled rectangle, draw image, draw compressed image, and flush. The upper levels of the graphics library derive all functions from these basic LCD operations. Additionally the display driver should provide display dependent operations such as initialization, and may include operations for backlight control or contrast control.

Many LCDs represent a pixel using less than a full byte, for example one, two, or four bits per pixel. This type of interface means that 1 byte of data, or 8 bits, represents multiple pixels on the LCD display. To alter a single pixel only part of the byte must be altered such that the other pixels represented in the data byte are not corrupted. In these cases there are two available options: read out the data before altering it and writing it back or keep a global frame buffer of the current LCD state. The frame buffer requires significant memory resources, but is easier to implement and often maintains better performance. There is built in support for a frame buffer in the LCD display driver layer for use with the flush functionality.

5.2 Creating New LCD Driver Files

There are several example LCD display driver files provided in the MSP430 Graphics Library for use or general reference. Among these drivers is a template driver (Template_Driver.c) that is only missing the lowest level of the customizable hardware abstraction layer. The template driver functions are all derived from the draw pixel function. This is not how an optimized driver should be constructed, but it allows for the user to rapidly test the functionality of the entire LCD driver file by only writing the color translation and draw pixel functions. Once these functions are implemented, the rest of the pixel level driver operations will be functional because they are all derived from the draw pixel function.

The template driver accelerates development by creating an easy initial implementation. If fast draw speeds are not required, the template driver provides a rapid solution to creating a functioning LCD driver file. In most cases, once the driver is functioning correctly, it should then be optimized for faster drawing speeds.

5.3 Optimizing the LCD Display Driver Layer for Speed

5.3.1 Utilizing LCD Controller Features

To fully optimize the LCD display driver layer, a good understanding of the different options and features of the LCD controller is required. This step is especially important when interfacing the MSP430 to more complex graphic LCDs. The efficiency of the LCD display driver has a direct impact on the draw speeds of every graphics primitive in the MSP430 Graphics Library. Eliminating even a single instruction can have a significant impact on drawing speed because the LCD driver layer functions are called for every pixel drawn.

An effective technique in optimizing the LCD driver layer is to take advantage of auto incrementing features of the LCD controller. Not every LCD controller has this functionality, but it is very common.

With the auto increment feature of the LCD controller enabled, the microcontroller can simply stream data rather than resetting the pixel location before each data write. This sounds like a small change but it has a substantial impact on the draw speed of functions that take advantage of it.

Looking at a simplified example illustrates how important the auto increment feature is. When drawing a horizontal line, a simple option would be to sit in a loop and repeatedly call the draw pixel function. This method works, but the draw pixel function requires setting the cursor location followed by the pixel data write to the LCD. This implementation is displayed on the left side below. Alternatively, the auto increment feature can be used and the cursor location is set only once, as it is automatically incremented by the LCD controller following a pixel data write. This implementation is displayed on the right side below.

```
DrawHorizLine(x1, x2, y, color)
{
   While(x1++ <= x2)
   {
      SetCursorLocation(x1, y);
      WriteDataToLCD(color);
   }
}
```

```
DrawHorizLineOptimized(x1, x2, y, color)
{
   SetCursorLocation(x1, y);
   While(x1++ <= x2)
   {
      WriteDataToLCD(color);
   }
}
```

For this example, it is assumed that *SetCursorLocation()* requires 20 clock cycles and *WriteDataToLCD()* requires 5 clock cycles.

Total clock cycles = (x2 – x1 + 1) * 25 Total clock cycles = 20 + (x2 – x1 + 1) * 5

Neglecting the small overhead of the initial *SetCursorLocation()*, the right side is five times as fast as the left side. With a small and simple change, all horizontal lines now draw at five times the speed. Although this is a simplified example, it illustrates realistic gains that can be made by taking advantage of the auto increment feature of the LCD controller. This feature should be used throughout the LCD display driver whenever possible.

The auto increment feature is a great optimizing tool, but many LCDs have other built in features to take advantage of. Additional features include clearing the screen, flipping the pixel polarity, turning the screen orientation, and many more. These features are built into some LCD controllers because their common use provides functionality that would otherwise need to be written into the application or driver layer, thus consuming more cycles per pixel and slowing drawing times.

5.3.2 Coding Optimizations

The built in features of the LCD controller are a powerful speed optimizing tool, but there are other simple coding techniques to increase speed of the low-level driver functions. Using macros instead of functions increases performance by reducing overhead. This is especially apparent for functions that are called most often. In the example shown above, making *SetCursorLocation()* and *WriteDataToLCD()* macros will reduce the overhead of repeatedly calling a function and instead place the necessary code directly in the loop. This method increases code size, but if macros are only used for the simple functions, little code size is sacrificed for the increase in performance.

Modern compilers also offer features to help shape the application. If fast draw times are desired, compiler settings can be changed to use high amounts of optimization for speed. After all optimizations are made, the disassembly can be viewed to verify that all functions are optimized to the highest possible level. Instruction cycle counts can be found in the device-specific MSP430 Family User's Guide.

5.4 Images in the MSP430 Graphics Library

5.4.1 Image Format

Images can be presented in many different formats. For images to be drawn onto any LCD screen, they must first be converted into a format that is read by the MSP430 Graphics Library. The library is accompanied by a GUI, the MSP430 Image Reformer, which does this automatically; the output is formatted as C code and can be added to the application project. As a result, the process is automated. However, it is advantageous for the developer to understand the format to assist in optimizing performance.

The library uses a palette-based approach where each pixel in an image is represented by an index to a common color palette, rather than containing the data for the color itself. This approach divides the image into an information section, color palette section, and a pixel data section, each requiring a specific format to be read properly.

The information section of the image contains information for the graphics library about the image. This section contains six elements that describe: bits per pixel (BPP) and compression, x size, y size, number of colors in the palette, a pointer to the color palette section, and a pointer to the pixel data section. This section is interpreted by the library for proper image drawing no matter what image options are selected.

The color palette section of data contains all colors used in a particular image. The MSP430 Graphics Library supports palette sizes of 2, 16, and 256 colors. These palette sizes correspond to the number of bits per pixel required to store the index to the palette, or 1BPP, 4BPP, and 8BPP, respectively. The color of each pixel is represented in a 24-bit form with 8 bits each for red, green, and blue. This is a typical method of conveying pixel color information and looks like 0xRRGGBB. For instance, a blue pixel would be represented by 0x0000FF while a red pixel would be represented by 0xFF0000.

The pixel data section is comprised of information for each pixel in the image. This data is organized according to the size of the color palette or bits per pixel. In an 8BPP image each pixel data byte indexes to a single pixel. In the other BPP configurations, multiple pixels represent one pixel data byte.

The coordinate system for pixel matrices is different than a standard Cartesian coordinate system. Pixel data for images is ordered across rows from left to right, starting with the top row and working downward. Figure 3 shows the difference between the standard x-y Cartesian coordinate system and the pixel matrix coordinate system where r and c denote rows and columns, respectively.

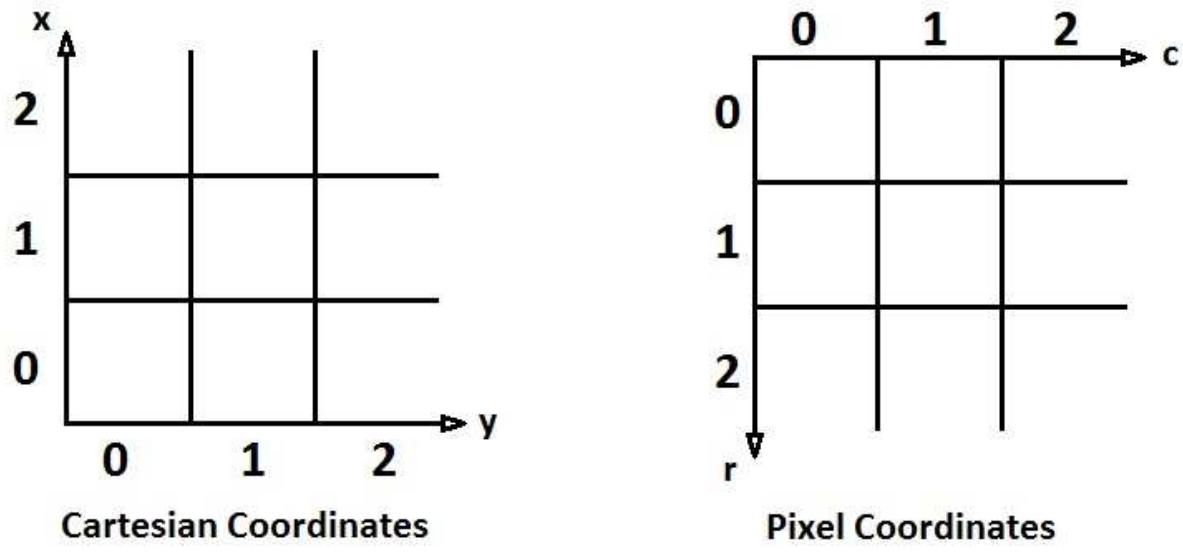

Figure 3. Image Format – Coordinate Systems

Figure 4 illustrates how images are converted into an acceptable format for the MSP430 Graphics Library. This 16 pixel image consists of 4 pixels each of blue, green, red and white.

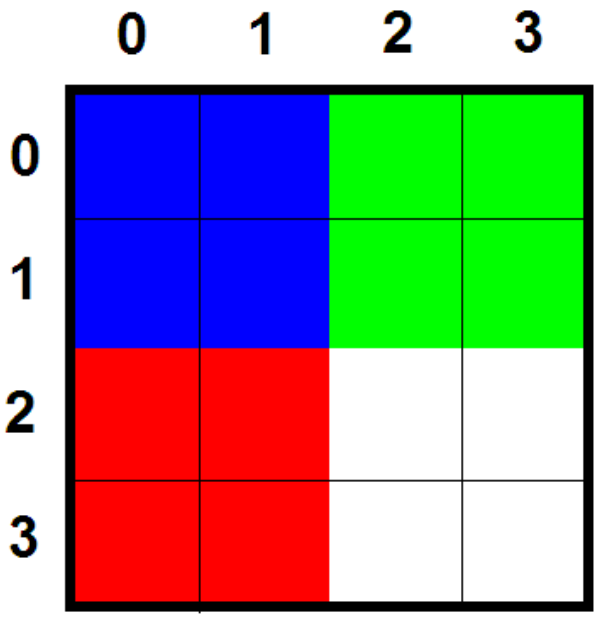

Figure 4. Image Format - Example 16 Pixel Figure

In a generic bitmap form, this image is represented by a color string of 48 total bytes.

```
Blue  = 0x0000FF
Green = 0x00FF00
Red   = 0xFF0000
White = 0xFFFFFF

0x0000FF, 0x0000FF, 0x00FF00, 0x00FF00,
0x0000FF, 0x0000FF, 0x00FF00, 0x00FF00,
0xFF0000, 0xFF0000, 0xFFFFFF, 0xFFFFFF,
0xFF0000, 0xFF0000, 0xFFFFFF, 0xFFFFFF
```

After converting this image to the palette and index format used by the graphics library, the format will look like the following palette and pixel sections. In this form, the image saves space by using the indices to repeatedly represent the same colors.

```
0x0000FF,
0x00FF00,
0xFF0000,   } Palette
0xFFFFFF

0x00, 0x00, 0x01, 0x01,
0x00, 0x00, 0x01, 0x01,
0x02, 0x02, 0x03, 0x03,   } Per-Pixel Palette Indices
0x02, 0x02, 0x03, 0x03
```

The image above has only four colors and could use 4BPP storage instead of 8BPP storage. This would merge the 8BPP indices together so that 1 byte represents 2 pixels of 4 bits each. The transition from 8BPP to 4BPP converts the first and second rows of pixels 0x00, 0x00, 0x01, 0x01 to 0x00, 0x11. The third and fourth rows of pixels 0x02, 0x02, 0x03, 0x03 become 0x22, 0x33. This alters the pixel data section as shown below.

```
0x0000FF,
0x00FF00,
0xFF0000,   } Palette
0xFFFFFF

0x00, 0x11,
0x00, 0x11,
0x22, 0x33,   } Per-Pixel Palette Indices
0x22, 0x33
```

The palette and index method saves a considerable amount of space over the direct color string method, especially as the image grows in size. The image is now fully optimized for storage size and has reduced from 48 bytes down to 20 bytes, a 42% size savings.

5.4.2 Image Conversion

Most images imported into the MSP430 Graphics Library undergo some lossy compression because the largest palette size is 256 colors. The difference between an image containing thousands of colors and 256 colors is quite palatable as shown in Figure 5 and Figure 6.

Figure 5. Image Conversion – Original Leaf Image

Figure 6. Image Conversion – Converted Leaf Image With 256 Color Palette

After the image transformation, some loss of color information is evident; however, the image remains vibrant and distinguished. The converted image utilizes 256 colors in the palette to emulate the thousands of colors present in the original image.

When the amount of memory required to store an image is an important consideration, simpler images can be converted to a 4BPP format to utilize only 16 colors. These 4BPP images have even less color information but only require half of the storage space of the 8BPP image.

Figure 7. Image Conversion – Original Frog Image

Figure 8. Image Conversion – Converted Frog Image With 16 Color Palette

In this converted image, some fine details are lost and color banding is prevalent, but this image is stored in a 4BPP format at a small fraction of the size of the original. The frog is still easily discernible with only 16 unique colors in the image.

5.4.3 MSP430 Image Reformer

The MSP430 Image Reformer is a PC application provided by TI to convert images (.bmp, .jpg, .gif, .tif) to an appropriate format for use with the MSP430 Graphics Library. The Image Reformer tool is designed to be simple and imports images and converts them directly as is. Resizing of images is supported but other image manipulation, such as cropping, is not supported. If some image manipulation is required, image editing software must be used before importing the image into the MSP430 Image Reformer tool.

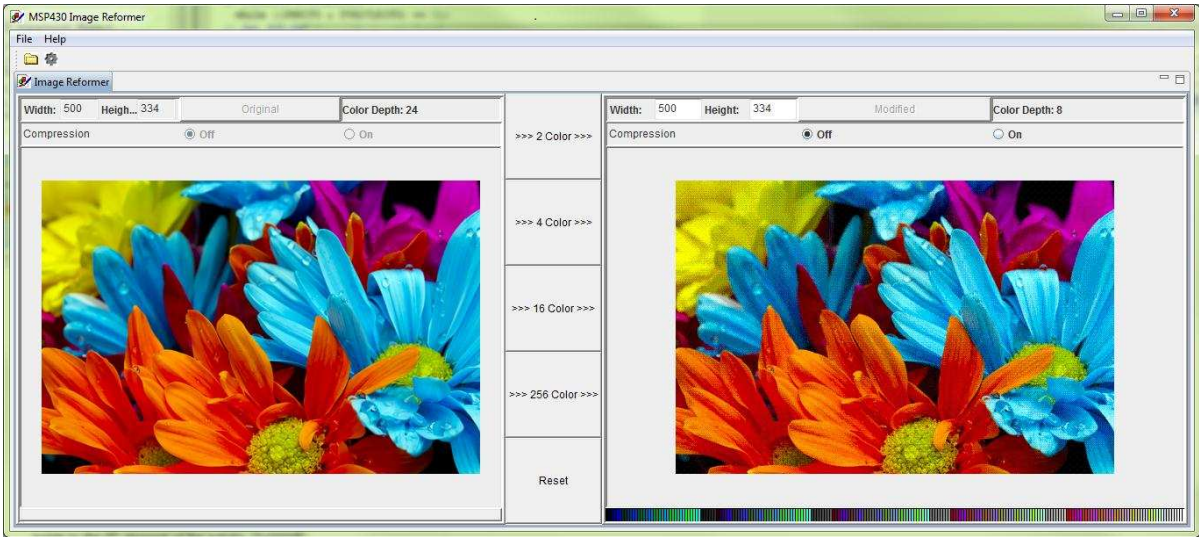

Figure 9. MSP430 Image Reformer – Image Reformer Screenshot

The two main steps in the tool to convert an image are palette size selection and image compression. The palette size determines the image quality and has a large impact on the image size and draw speed. The image compression helps fine-tune the image size and draw speed to an application.

5.4.4 Palette Conversion

The initial image conversion is lossy if the image contained more colors than the palette it is being transformed to. Once this step is complete, the exact bytes of the image are finalized and the converted image is displayed on the right hand side of the tool. The original image is kept on the left hand side for direct reference.

Image conversion offers the choice of 2, 16, or 256 colors in the palette. Depending on the amount of color content, the image size scales as the colors do. Given the same image, the uncompressed 8BPP option will be 8 times larger than the uncompressed 1BPP option. The 1BPP option requires less storage space, but all of the logical operations required to read each individual pixel from a byte cause an inverse relationship between palette size and draw times.

Table 2. Tradeoffs of Palette Sizing Options

	1BPP	4BPP	8BPP
Advantages	• Smallest storage size • Supports RLE4 and RLE8 compression	• Small storage size for images of moderate complexity • Fast drawing speeds • Supports RLE4 and RLE8 compression	• Fastest drawing speed (byte and sec) • Can display the most complex images
Disadvantages	• Slowest draw speed (byte and sec) • Only two colors in image	• Complex images cannot be represented in 16 colors	• Largest storage size • Only supports RLE8 compression

In most cases, the palette size is easily determined by the quality of the image needed for the application. If a simple black and white image is needed, 1BPP is used, and if a complex colorful image is needed, 8BPP is used. The Image Reformer tool provides the output of what is shown on the LCD so that image quality can easily be assessed and palette size quickly determined.

5.4.5 Compression Types

Run length encoding (RLE) is a type of compression that thrives when long runs of pixels are present. The algorithm is simple to understand and can drastically reduce storage size and draw speeds. There are two different types of run length encoding used in the GUI to compress images: 4-bit run length encoding (RLE4) and 8-bit run length encoding (RLE8).

Run length encoding an image results pixel data being compressed into two components: run length and pixel index. This replaces the string of consecutive pixel data bytes with encoded bytes. This compression is not lossy, meaning it does not change the contents of the image data, it simply stores the data in a different format. Storing the data differently can allow for various size and speed tradeoffs for the image to be drawn.

The difference between RLE4 and RLE8 is the number of bits reserved for both the run length and the pixel value. RLE4 uses 4 bits for run length and 4 bits for the pixel index. RLE4 encoding limits run length to 16 pixels and can only be used for 1BPP or 4BPP images with 16 pixels or less in the palette. Run length has a minimum length of 1 pixel, so the value 0x00 for run length indicates a run of 1, 0x01 indicates a run of 2, and so on.

Figure 10 is a 7x8 pixel image consisting of five different colors. This image is used to compare multiple types of compression supported in the MSP430 Graphics Library.

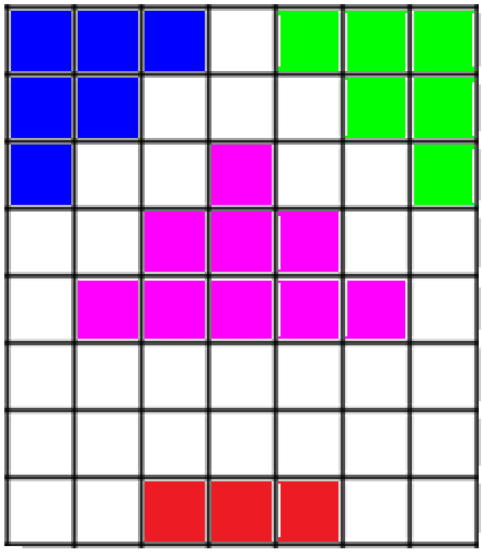

Figure 10. MSP430 Image Reformer – Example 7x8 Pixel Image

Image data with all available compression options is presented below. The palette remains the same for all types of image compression.

For uncompressed pixel data each line of the image must end on an even byte boundary. The uncompressed 4BPP pixel data below illustrates how the bytes are padded with extra zeroes on the last byte of the line to achieve this. This padding occurs when 4BPP images have a width that is not a multiple of 2, and 1BPP images have width that is not a multiple of 8. Run length encoded images do not require a line to end on an even byte boundary, and runs can extend onto the next line if the pixel color is still the same.

```
                            Color Palette
                           0x0000FF,  (Blue)
                           0x00FF00,  (Green)
                           0xFF0000,  (Red)
                           0xFF00FF,  (Purple)
                           0xFFFFFF   (White)

0x00, 0x04, 0x11, 0x10,                          0x20, 0x04, 0x21, 0x10, 0x24, 0x11,
0x00, 0x44, 0x41, 0x10,                          0x00, 0x14, 0x03, 0x14, 0x01, 0x14,
0x04, 0x43, 0x44, 0x10,                          0x23, 0x24, 0x43, 0xF4, 0x04, 0x22,
0x44, 0x33, 0x34, 0x40,                          0x14
0x43, 0x33, 0x33, 0x40,
0x44, 0x44, 0x44, 0x40,
0x44, 0x44, 0x44, 0x40,
0x44, 0x22, 0x24, 0x40

Uncompressed 4BPP Pixel Data                     RLE4 Compressed Pixel Data
```

```
                             Color Palette
                            0x0000FF,  (Blue)
                            0x00FF00,  (Green)
                            0xFF0000,  (Red)
                            0xFF00FF,  (Purple)
                            0xFFFFFF   (White)

0x00, 0x00, 0x00, 0x04, 0x01, 0x01, 0x01,        0x02, 0x00, 0x00, 0x04, 0x02, 0x01,
0x00, 0x00, 0x04, 0x04, 0x04, 0x01, 0x01,        0x01, 0x00, 0x02, 0x04, 0x01, 0x01,
0x00, 0x04, 0x04, 0x03, 0x04, 0x04, 0x01,        0x00, 0x00, 0x01, 0x04, 0x00, 0x03,
0x04, 0x04, 0x03, 0x03, 0x03, 0x04, 0x04,        0x01, 0x04, 0x00, 0x01, 0x01, 0x04,
0x04, 0x03, 0x03, 0x03, 0x03, 0x03, 0x04,        0x02, 0x03, 0x02, 0x04, 0x04, 0x03,
0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,        0x10, 0x04, 0x02, 0x02, 0x01, 0x04
0x04, 0x04, 0x04, 0x04, 0x04, 0x04, 0x04,
0x04, 0x04, 0x02, 0x02, 0x02, 0x04, 0x04

Uncompressed 8BPP Pixel Data                     RLE8 Compressed Pixel Data
```

Each type of run length encoding has tradeoffs. RLE4 is better with shorter runs because it requires 1 byte to encode the run + data where RLE8 requires 2 bytes to encode. However, RLE8 is much better with longer runs because it supports runs up to 256 pixels long where RLE4 supports runs up to only 16 pixels long. In this example image, there were several short runs leading the RLE8 method to require twice the number of bytes. There was one long run of 17 white pixels where RLE4 required two separate runs to produce. The amount the image pixel data is compressed is very image dependent because of these tradeoffs.

Table 3. Tradeoffs of RLE4, RLE8, and Uncompressed Image Formats

	1BPP	4BPP	8BPP
Advantages	• Typically best compression with 4BPP images	• Much faster byte and second drawing speeds than RLE4 (approximately 1.5-4x faster)	• Only option that supports image clipping for images extending beyond the bounds of the LCD • Works best with very complex images such as photographs
Disadvantages	• Can become larger than uncompressed image • Contains unused bits with 1BPP images • Cannot use with 8BPP images • Image must stay within LCD boundary	• Can become larger than uncompressed image • Contains unused bits with 1BPP and 4BPP images • Image must stay within LCD boundary	• Sometimes larger and slower than its compressed counterpart

5.4.5.1 Compressing Images With 16 Colors or Less

Images that are 4BPP (16 colors) or 1BPP (two colors) can be compressed with RLE4 or RLE8. In most cases, the RLE4 compression method compresses to a smaller image size. Given the variability of images, there will be cases in which RLE8 will be the optimal choice. These images will be very simple, lending themselves to long runs.

5.4.5.2 Compressing 256 Color Images

8BPP images can only be compressed with RLE8. The image draw time is very fast, but the storage size can become larger than the uncompressed format for more complex images that contain short pixel runs. Typically images that require the 256 color palette are considered complex. Simple images can be used and stored in this format for the fastest draw speeds.

5.4.5.3 Choosing a Compression Type

Due to the nature of run length encoding, the compressed image may require more storage space than an uncompressed image. This is important to keep in mind so that the uncompressed image option is not overlooked. Uncompressed images also make use of image boundary clipping. This functionality is not supported on compressed images and will draw incorrectly if the image is not placed completely inside the bounds of the LCD.

Even though the RLE4 method typically compresses images into less total bytes does not mean it is always the best choice. All image decisions are a tradeoff of image size vs. drawing speed. The RLE8 images draw much faster than the RLE4 images. This varies with the LCD interface and how the LCD display driver is written, but an RLE8 image generally draws around twice as fast on a byte per second basis. An application optimized for storage size or speed often chooses the RLE4 or RLE8 options, respectively. These options tailor the image to the application.

The MSP430 Image Reformer is made to be easy to use and incorporate the output file into a project. If the application is constrained for storage space or drawing speed, multiple schemes can be tried to see which creates the best results specific to the application.

6 Design Example

The goal of this design was to interface a large color LCD to be driven by the MSP430. This design was to be optimized for speed so that the draw times on the large graphic LCD display would be acceptable. The schematics and associated gerber files, along with software demo are attached in the associated files folder that can be downloaded from http://www.ti.com/lit/zip/slaa548.

6.1 Hardware Implementation

The Kitronix K350QVG-V2-F, which is featured in the Stellaris® Graphics Library demo was chosen as the display. The Kitronix display is a full color QVGA LCD screen with 320 x 240 pixels capable of displaying over 250,000 unique colors.

To meet the design goals that require acceptable drawing times on the Kitronix LCD, an MSP430F5529 was selected. The MSP430F5529 supports system clock speeds of up to 25 MHz for fast drawing operations, and has enough GPIO pins to implement a large parallel interface. The MSP430F5529 also has 8KB of RAM along with 128KB of Flash memory to store large images for drawing to the display.

6.2 Bus Comparison

The SSD2119 LCD controller IC integrated into the Kitronix display supports several different bus types including multiple types of parallel and SPI. The SPI bus has slower bus speeds and requires loops to wait and check for status flags. The parallel bus was chosen for this application because it has fast and deterministic write times. The parallel bus was implemented with jumper selectable 8-bit parallel bus or 16-bit parallel bus. This allowed for the evaluation of both types of parallel buses.

Table 4. Performance of Parallel Bus

	8-Bit Parallel Bus Speed	16-Bit Parallel Bus Speed
8 BPP Uncompressed Full Screen Image	120 ms	77 ms
4 BPP Uncompressed Full Screen Image	126 ms	83 ms
Full Screen Filled Rectangle	82 ms	53 ms

For 320 x 240 pixel full screen writes, the 16-bit bus requires about 65% of the drawing time compared to the 8-bit bus. This is a significant decrease in drawing time, but this needs to be evaluated against the need for additional GPIO pins.

6.3 Software Implementation

The LCD display driver was written with speed as the main consideration. Macros were used frequently to deliver faster draw times. The auto incrementing feature was used wherever possible, including changing the incrementing direction for use with vertical lines. Compiler directives were written to support rotating the display into any orientation for flexibility of use. This LCD display driver can be found in the associated application demo for this design.

The overall optimization of the hardware and software on this design took the original drawing speed of a full screen uncompressed image from over 700 ms to under 100 ms. Careful optimization can have an overwhelming impact on the application.

7 References

SSD2119 LCD Controller Data Sheet: http://www.crystalfontz.com/controllers/SSD2119.pdf

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES “AS IS” AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265