Application Report

Developing LCD Applications for TM4C12x MCU

Charles Tsai

ABSTRACT

Several variants of the TM4C microcontroller (MCU) family have an integrated LCD controller that is capable of supporting a wide variety of external display modules. This application report walks through the capabilities of the integrated LCD controller as well as an introduction to TivaWare™ Graphic Library (grlib). Grlib offers a compact yet powerful collection of graphics functions for the development of user interfaces on displays attached to the TM4C microcontrollers. Grlib can be used for TM4C MCUs that have LCD controller to drive the displays directly or MCUs without the LCD controller that will interface with the displays via SPI interface.

Table of Contents

List of Figures

List of Tables

Trademarks

TivaWare™ is a trademark of Texas Instruments.
TrueType®, OpenType®, PostScript®, and Windows® are registered trademarks of Apple Inc., registered in the United States and other countries.
All trademarks are the property of their respective owners.

1 TFT LCD Overview

Thin Film Transistor (TFT) is a variant of LCD that uses TFT technology to improve image qualities. A TFT is an active matrix LCD, in contrast to passive matrix LCD or direct-driven LCD such as the character-based LCD with a few segments. The TFT LCD is a popular display type for small-to-medium size applications.

A typical TFT LCD module will consist of a TFT LCD panel, a chip-on-glass (COG) driver IC, a backlight unit, and an interface Flexible Printed Circuit (FPC). The TFT driver ICs are highly integrated chips that will combine the source driver (in a 320 x 240 display, driving 320 x 3 = 960 sources of the transistors for supporting Red, Green and Blue), gate driver (driving the 240 gates of the transistors), timing control and sometimes other circuits such as memory, power or image processing. The FPC provides an interface connector to the MCUs. Figure 1-1 shows the Kentec320 x 240 display that is mounted on the DK-TM4C129X EVM board shown in Figure 1-2. The driver IC on the Kentec320x240x16 is a SSD2119.

Figure 1-1. Kentec320x240x16 TFT LCD Display

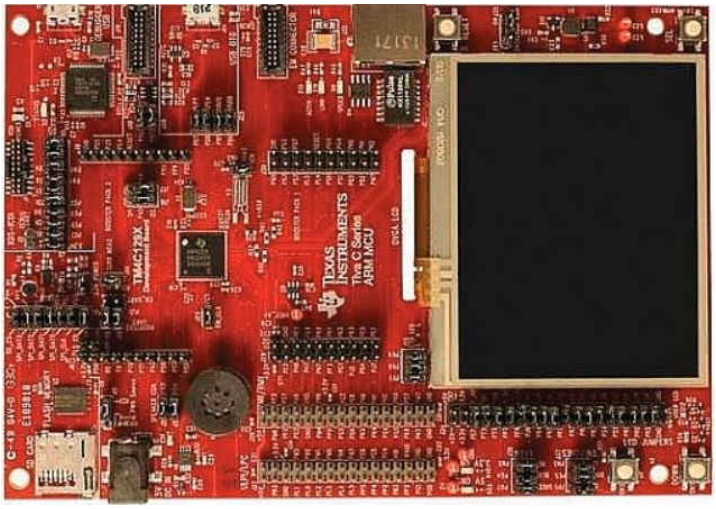

Figure 1-2. DK-TM4C129x EVM Board

1.1 Typical Interfaces

Depending on the particular TFT panel size and resolution, some driver ICs may support multiple interfaces to communicate with the host controller. The interfaces are normally selectable on the module FPC or through firmware initialization.

- SPI interface. This is the most simple and low pin-count interface which is normally reserved for character-based LCD or small TFT panels that require slow frame rate operation. Any MCU with a SPI interface can interact with the displays supporting this interface. The BOOSTXL-K350QVG-S1 BoosterPack is one such example that can interface with TI's various LaunchPad's through the SPI interface.

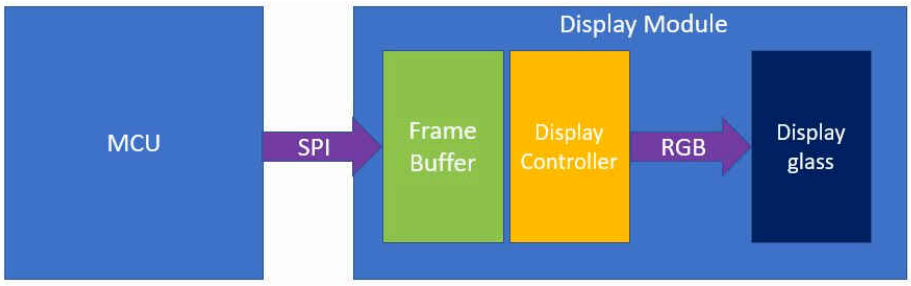

Figure 1-3. SPI Interface to the Display Module

- MPU interface. This is a parallel bus interface. This is normally the interface of choice when the display module contains an integrated display controller. For example, the Kentec320x240x16 QVGA Display has the built-in SSD2119 display controller IC. The MCU uses the parallel bus interface to write the frame buffer on the SSD2119 controller. There are two types of MPU interfaces – Motorola 6800 and Intel 8080. 8080 is a more popular interface than 6800. This interface consists 4/8/9/16 bits data, Chip-Select (CS) , Data/ Instruction Select (RS), Read-Write (RD) and Write-Enable (WR). Figure 2-3 shows the signal mapping on TM4C129 MCU.

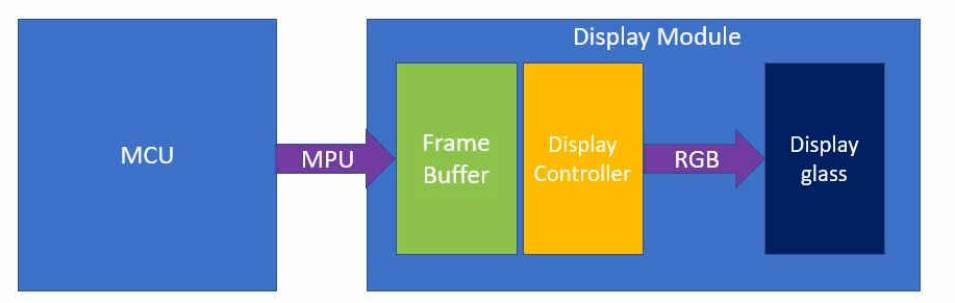

Figure 1-4. MPU Interface to the Display Module

- RGB interface. This is a parallel bus interface that works for displays without a frame buffer. The MCU is responsible for updating the display by providing the timing signals (HSYNC, VSYNC, OE, CLK) and the RGB sub-pix data (16 bits, 18 bits, 24 bits).
  - In active TFT mode, the HSYNC (LCDLP signal) acts as a horizontal line clock. HSYNC toggles after all pixels in a horizontal line have been transmitted to the LCD and a programmable number of pixel clock wait states have elapsed both at the beginning and end of each line.
  - In active TFT mode, the VSYNC (LCDFP signal) is the vertical frame clock. VSYNC toggles after all lines in a frame have been transmitted to the LCD and a programmable number of line clock cycles has elapsed at the beginning and end of each frame.
  - In active TFT mode, the OE (LCDAC signal) acts as an output enable signal. It is used to signal the external LCD that the data is valid on the data bus.

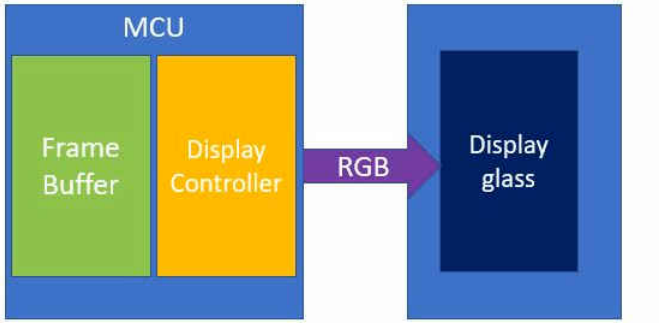

Figure 1-5. RGB Interface to the Display Glass

1.2 Frame Buffer

The LCD frame buffer is a contiguous memory block, storing enough data to fill a full LCD screen. The frame buffer contains the palette look-up table (palette RAM) and the frame data for a given source image. Operating in RGB Mode requires the LCD frame buffer. The LCD controller has the option to use the internal SRAM memory or external memory through the EPI interface to hold the frame buffer.

1.2.1 Frame Buffer Size Calculation

Take the Kentec 320x240x16 QVGA display module as an example, how much frame buffer memory would be needed to display one full screen? Assume that 16-bit color depth is needed. 16-bit color depth means that two bytes are needed for each pixel.

Frame buffer size (FB) = Number of pixels x color depth / 8

$$FB = 320 \times 240 \times 16 / 8 = 153.6KB \quad (1)$$

The TM4C129x MCU has 256KB of internal SRAM. Therefore, there is enough internal SRAM to be used as frame buffer for a 320x240x16 displays and perhaps enough SRAM left for system memory to support other MCU functions.

As the display resolution increases, the more frame buffer is needed. A 480 x 272 display with 16-bit color depth would require 261KB of frame buffer. As expected, the TM4C129 internal SRAM is not sufficient in this scenario. In this case, an external memory dedicated for the frame buffer and accessed through the EPI interface is required.

Using the internal RAM for frame buffers makes the read and write access as fast as possible by the MCU. It will normally translate to smoother display viewing. Having multiple frame buffers will further imply no visual artifacts like tearing will appear as one frame buffer is used for writing the next resulting image while the other frame buffer is used for transferring current image to the display. However, the internal RAM is usually a limited resource as it is used by many parts of a system. Therefore, using internal RAM may be limited to smaller display applications.

The LCD controller on the TM4C129x MCU can support a maximum resolution of 2048 x 2048 pixels. Therefore, the limitation on the maximum resolution of a display will depend on the amount of memory that can be dedicated for frame buffers.

1.3 Frame Rate (FPS)

Frame rate is the frequency at which consecutive frames appear on a display. The maximum frame rate is determined by the image size in combination with the pixel clock rate. More precisely, the pixel clock frequency, the size of the display, and the porch intervals determine the frame rate. The porch intervals that are expressed in terms of pixel clocks refer to the blanking period between each line and each frame. For the duration required for the porch intervals, see the device-specific data sheet. The FPS can be expressed as:

```
FPS = Fpixel_clock / ((HBP + Resolution Width + HFP) x (VBP + Resolution Height +
VFP))
```

Suppose the following example:

`Pixel clock = 60 MHz LCD clock` (This is the maximum LCD clock on TM4C129 MCU.)

`LCD Controller mode = parallel RGB interface (24-bit bus)`

`Color depth = 24-bit`

`Resolution = QXGA (2048 x 1536)`

`Porch intervals = Assume insignificant compared to the resolution width and height`

`FPS = 60 MHz / (2048 x 1536) = 19 or less if porch intervals is taken into consideration`

For video playback, normally a minimum of 24FPS is needed for human eyes to render the motion smoothly. Therefore, there is a compromise between FPS, display resolution, and color depth when choosing the optimal displays to use.

Suppose another example using a SPI interface:

`Pixel clock = 60 MHz SPI clock` (This is the maximum SPI clock on TM4C129 MCU.)

`LCD Controller mode = SPI Interface`

`Color depth = 24-bit`

`Resolution = QVGA (320 x 240)`

`Porch intervals = Assume insignificant compared to the resolution width and height`

`FPS = 60 MHz / (2048 x 2048 x 24) = 35 or less if porch intervals is taken into consideration`

Theoretically, it is possible to support a small display with video playback at greater than 24 FPS.

1.4 Touch Display

There are various touch screen technologies available, but going into the details about each one is outside of the scope of this application report. Wikipedia has a good introduction on how different touch screen technologies work and the pros and cons among them. This application report will focus on demonstrating the possible interfaces the TM4C MCU can use to sense the input from the touch sensors. Figure 1-6 illustrates the possible interfaces the TM4C MCU can sense the touch inputs.

For display modules (either resistive or capacitive) that have the integrated touch controller, the interface to the touch sensors can be either the I2C or SPI interface. For resistive touch displays that do not have an integrated touch controller such as the Kentec320x240x16 display as shown in Figure 1-6(B), the host MCU can sense the inputs via the 4 or 5-wire sensor interface using the on-chip ADC and GPIO. TM4C's TivaWare library provides examples on how this can be done. For capacitive touch displays, it is advised that customers choose one that comes with the integrated touch controller as TM4C MCUs have no such hardware capability or software to implement this function.

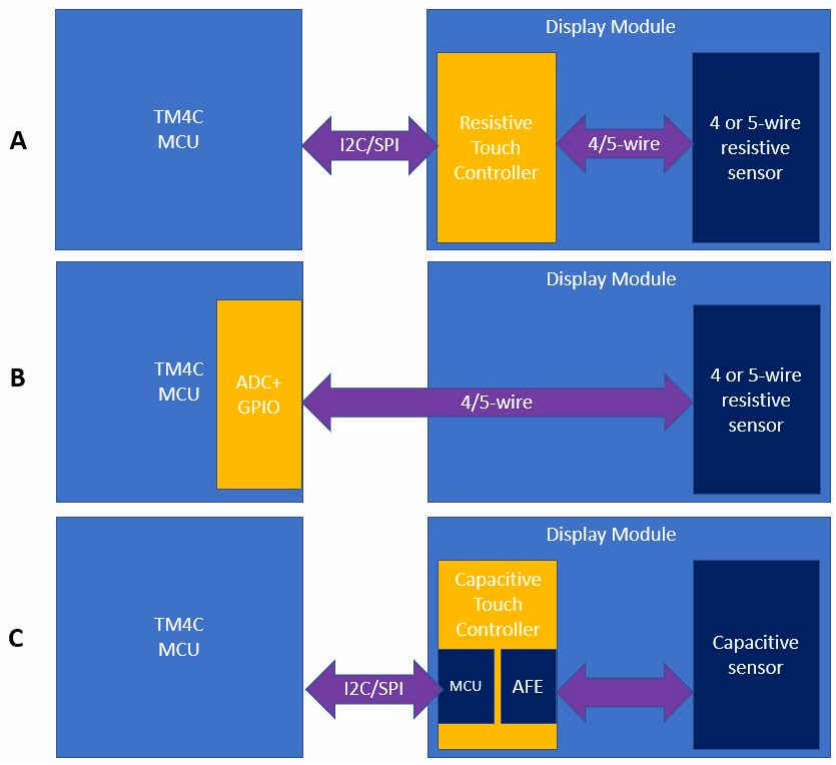

Figure 1-6. Touch Display Interface

2 LCD Controller Overview

The Liquid Crystal Display (LCD) Controller on TM4C MCUs provides support for a variety of LCD and OLED panels.

- Character-based panels:
  - Support for two character panels (CS0 and CS1) with independent and programmable bus timing parameters when in asynchronous Hitachi, Motorola and Intel modes.
  - Support for one character panel (CS0) with programmable bus timing parameters when in synchronous Motorola and Intel modes.
  - Can be used as a generic 16-bit address/data interleaved MPU bus master with no external stall.
- Passive matrix LCD panels:
  - Panel types including STN, DSTN, and C-DSTN
  - AC Bias Control
- Active matrix LCD panels:
  - Panel types including TN TFT
  - 1, 2, 4 or 8 bits per pixel with palette RAM and 16 or 24 bits per pixel without palette RAM
- OLED panels:
  - Passive Matrix (PM OLED) with frame buffer and controller IC side the panel
  - Active Matrix (AM OLED)
- Bus mastering capability from either SRAM or EPI memory

Note

While the LCD controller is capable of supporting a variety of display panels, this application report focuses on the support for the more popular active matrix LCD panels such as the TFT display panels.

2.1 Block Diagram

The LCD controller consists of two independent controllers, the Raster controller and the LCD Interface Display Driver (LIDD) controller. Each controller operates independently from the other and only one of them is active at any given time.

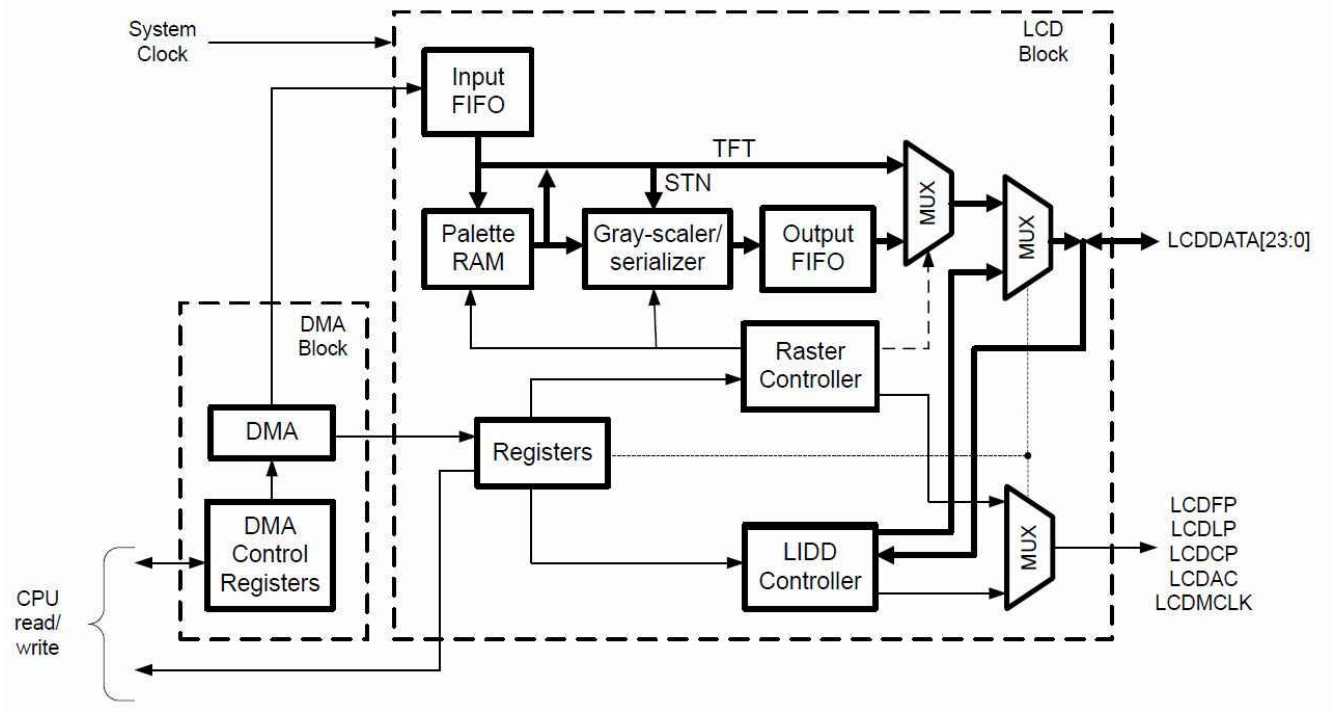

Figure 2-1. LDC Controller Block Diagram

2.1.1 Raster Controller

The Raster Controller provides a synchronous LCD interface. The Raster controller supports a wide variety of monochrome and full-color display types and sizes by use of programmable timing controls, a built-in palette, and a gray-scale/serializer. Graphics data is processed and stored in the frame buffer. A frame buffer is a contiguous memory block in the system.

The Raster controller should be used if the display panel does not have its own frame buffer nor its own controller. In this mode, the MCU is responsible for sending pixels to the display directly.

Note

When using the LCD controller with EPI to interface to external memory, the external code address space 0x1000_0000 must be selected by programming the ECADR field to 0x1 in the EPI Address Map (EPIADDRMAP) register at EPI offset 0x01C.

Figure 2-2 shows the various modes and interfaces the TM4C129 MCU Raster Controller supports. The Active TFT Color RGB interface can be supported when the RASTERCTRL[9,7,1] register is programmed with x10.

Interface	Data Bus Width	RASTERCTRL [9, 7, 1]	Signal Name	Description
Passive (STN) Mono 4-bit	4	001	LCDDATA[3:0]	Data Bus
			LCDCP	Pixel Clock
			LCDLP	Horizontal clock (line clock)
			LCDFP	Vertical clock (frame clock)
			LCDAC	AC Bias
			LCDMCLK	Not used
Passive (STN) Mono 8-bit	8	101	LCDDATA[7:0]	Data Bus
			LCDCP	Pixel Clock
			LCDLP	Horizontal clock (line clock)
			LCDFP	Vertical clock (frame clock)
			LCDAC	AC Bias
			LCDMCLK	Not used
Passive (STN) Color	8	100	LCDDATA[7:0]	Data Bus
			LCDCP	Pixel Clock
			LCDLP	Horizontal clock (line clock)
			LCDFP	Vertical clock (frame clock)
			LCDAC	AC Bias
			LCDMCLK	Not used
Active (TFT) Color	16	x10	LCDDATA[15:0]	Data Bus
			LCDCP	Pixel Clock
			LCDLP	Horizontal clock (line clock)
			LCDFP	Vertical clock (frame clock)
			LCDAC	AC Bias
			LCDMCLK	Not used

Figure 2-2. Raster Mode Support

2.1.2 LIDD Controller

The LIDD controller can offer either an asynchronous or a synchronous interface depending on the mode of operation. It provides full-timing programmability of control signals (chip select, read/write strobes, enable, direction) and output data.

The LIDD controller is mainly used for character-based LCD panels. In addition to supporting character-based LCD panel, it can be configured to provide a generic 8080 or 6800 Microprocessor Unit (MPU) parallel bus interface to the displays. The LIDD controller should be configured in generic 8080 or 6800 bus interface mode when the display has an integrated display controller and frame buffer. The LIDD controller updates the frame buffer in the display. Be aware that the integrated display controller in the display is responsible for handling all the display updates from this frame buffer. In this mode of operation, the LIDD controller only performs write operations whenever the frame changes.

The Kentec320x240x16 display panel found on the DK-TM4C129x EVM board is one example of a display panel that has an integrated SSD2119 display controller with built-in frame buffer. Figure 1-4 illustrates this.

The LIDD controller can be configured for different MPU interfaces by specifying the MODE bits in the LCDLIDDCTL register. Figure 2-3 highlights the LCD module pins mapping for MPU interface modes. Kentec320x240x16 display falls under 8080 family of interface type. Figure 2-4 highlights MODE = 0 x 3 that is used for interfacing to the Kentec320x240x16 display on the DK-TM4C129x EVM board.

Table 21-4. LIDD I/O Name Map

Display Type	Interface Type	Data Bits	LCDLIDDCTL[2:0]	Signal Name	LCD Display I/O Name	LCD Display I/O Description
Character Display	HD44780 Type	4	0x4	LCDDATA[7:4]	DATA[7:4]	LCD Data Bus (length defined by instruction)
				LCDAC	E (or E0)	Enable Strobe (first display)
				LCDLP	R/$\overline{W}$	Read/Write
				LCDFP	RS	Register Select (Data/not Instruction
				LCDMCLK	E1	Enable Strobe (second display optional)
Character Display	HD44780 Type	8	0x4	LCDDATA[7:0]	DATA[7:0]	LCD Data Bus (length defined by instruction)
				LCDAC	E (or E0)	Enable Strobe (first display)
				LCDLP	R/$\overline{W}$	Read/Write
				LCDFP	RS	Register Select (Data/not Instruction
				LCDMCLK	E1	Enable Strobe (second display optional)
Micro Interface Graphic Display	6800 Family	Up to 16	0x1	LCDDATA[15:0]	DATA[15:0]	LCD Data Bus (16 bits always available)
				LCDCP	E	Enable Clock
				LCDLP	R/$\overline{W}$	Read/Write
				LCDFP	A0	Address/Data Select
				LCDAC	CS (or CS0)	Chip Select (first display)
				LCDMCLK	CS1	Chip Select (second display optional)
			0x0	LCDMCLK	None	Synchronous Clock (optional)
Micro Interface Graphic Display	8080 Family	Up to 16	0x3	LCDDATA[15:0]	DATA[15:0]	LCD Data Bus (16 bits always available)
				LCDCP	RD	Read Strobe
				LCDLP	WR	Write Strobe
				LCDFP	A0	Address/Data Select
				LCDAC	CS (or CS0)	Chip Select (first display)
				LCDMCLK	CS1	Chip Select (second display optional)
			0x2	LCDMCLK	None	Synchronous Clock (optional)

Figure 2-3. LIDD IO Name Map

PIN	MODE = 0x0	MODE = 0x1	MODE = 0x2	MODE = 0x3	MODE = 0x4
LCDCP	RS/WS	RS/WS	RS	RS	N/A
LCDLP	DIR	DIR	WS	WS	DIR
LCDFP	ALE	ALE	ALE	ALE	ALE
LCDAC	CS0	CS0	CS0	CS0	E0
LCDMCLK	MCLK	CS1	MCLK	CS1	E1

Figure 2-4. LIDD Mode Signal Functions

3 TivaWare Graphics Library (grlib)

The TivaWare Graphics Library (grlib) offers a compact yet powerful collection of graphics functions which aid with the development of compelling user interfaces on small monochrome or color displays attached to TM4C microcontrollers. The grlib is included in all TivaWare firmware development packages supporting evaluation or development kits that include color displays or via add-on BoosterPack color displays. Latest TivaWare release for these kits can be downloaded from SW-TM4C.

Following installation of the TivaWare package, the graphics library source can be found in the C:\ti\TivaWare_C_Series-2.2.0.295\grlib directory (assuming the installation is in the default location) and various example applications using the library can be found in the below board specific directories:

- *C:\ti\TivaWare_C_Series-2.2.0.295\examples\boards\dk-tm4c129x*
- *C:\ti\TivaWare_C_Series-2.2.0.295\examples\boards\ek-tm4c1294xl-boostxl-kentec-s1*
- *C:\ti\TivaWare_C_Series-2.2.0.295\examples\boards\ek-tm4c123gxl-boostxl-kentec-s1*

Application examples for different kits are shown Table 3-1.

Table 3-1. Application Examples

Example	Kit	Description
fontview	All	This example displays the contents of a TivaWare graphics library font on the DK board's LCD touchscreen. By default, the application shows a test font containing ASCII, the Japanese Hiragana and Katakana alphabets, and a group of Korean Hangul characters. If an SDCard is installed and the root directory contains a file named font.bin, this file is opened and used as the display font instead. In this case, the graphics library font wrapper feature is used to access the font from the file system rather than from internal memory. The font used in this example contains ASCII, Hiragana, Katakana, Korean Jamo and a small number of Hangul syllables and Chinese ideographs. It is intended purely for illustration purposes and is unlikely to be of use in a real-world application.
grlib_demo	All	This application provides a demonstration of the capabilities of the TivaWare Graphics Library using both primitives and widgets. A series of panels show different features of the library. As each panel is traversed, different capability is presented ranging from printing simple text to drawing primitives: lines, circles and different shapes to advanced widgets creating check box, container, push button, radio button and slider functions.
lang_demo	All	This application provides a demonstration of the capabilities of the TivaWare Graphics Library's string table functions. Two panels show different implementations of features of the string table functions. For each panel, the bottom provides a forward and back button (when appropriate). The purpose of the string table and custom fonts is mainly for applications intending to display accented characters and Asian language ideographs. The font library containing the Asian language ideographs would be enormous for embedded applications. The string table and custom fonts allow only the needed characters for the application to be stored in a custom string table and fonts.
scribble	All	The scribble pad provides a drawing area on the screen. Touching the screen will draw onto the drawing area using a selection of fundamental colors (in other words, the seven colors produced by the three color channels being either fully on or fully off). Each time the screen is touched to start a new drawing, the drawing area is erased and the next color is selected.
grlib_driver_test	dk-tm4c129x	This application provides a simple, command-line tool to aid in debugging TivaWare Graphics Library display drivers. The tool is driven via a command line interface provided on UART0. Configure the terminal emulator on your host system 115200bps, 8-N-1. Commands allow a given low level graphics function to be executed with parameters provided by you. Commands provide the ability to read and write arbitrary memory locations and registers, and tests displaying test patterns intended to exercise specific display driver functions.
hello_widget	dk-tm4c129x	A very simple “hello world” example written using the TivaWare Graphics Library widgets. It displays a button which, when pressed, toggles display of the words “Hello World!" on the screen. This may be used as a starting point for more complex widget-based applications.

3.1 Graphics Library Structure

The grlib provides a set of graphics primitives and a widget set for creating graphical user interfaces on microcontroller-based boards that have a graphic display. The graphic library consists of three layers with each subsequent layer building upon the previous layer to provide more functionality.

- The display driver layer must be supplied by the application since it is specific to the display in use. For example, a driver that is specific to Kentec320x240x16.
- The graphics primitive layer is the low-level API that provides the abilities to draw individual items on the display, such as lines, circles, text, and so on.
- The widget layer provides an encapsulation of one or more graphic primitives to draw a user interface element on the display, along with the ability to provide application-defined responses to user interaction with the element.

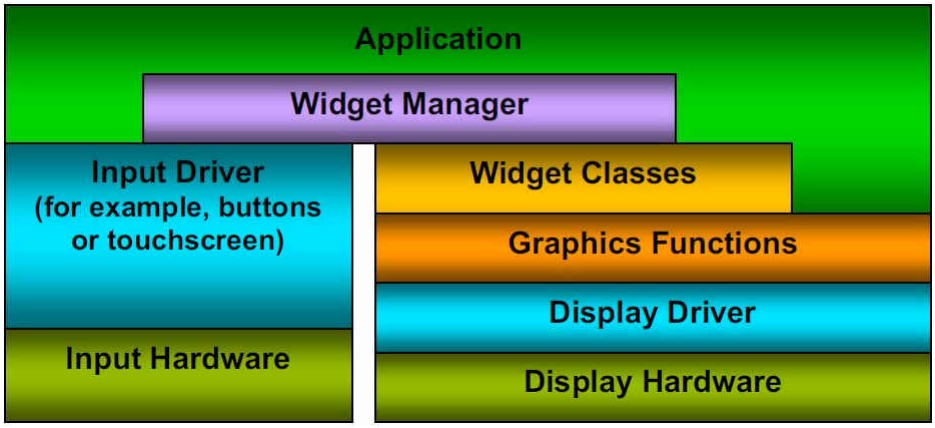

Figure 3-1. Organization of TivaWare Graphics Library

The right side of Figure 3-1 shows the layers of the graphic stack. APIs are provided at the widget level (the Widget API), the primitive graphics function level (the Low-Level Graphics API) and the display driver level (the Display Driver API). Additionally, a standard user-input driver interface (the input Driver API) is also provided. Depending upon the requirements of a given application, some portions of the library can be omitted if their functions are not required. All type definitions, labels, macros, and function prototypes for the graphics functions and display driver layers can be found in the grlib.h file. Definitions relating to the higher-level widget library can be found in widget.h and individual headers such as canvas.h and pushbutton.h contain definitions for each supported widget class.

3.1.1 Display Driver Overview

The display driver layer provides a standard programming interface to the graphics library code allowing it to draw actual pixels on the display. The API is simple (draw horizontal and vertical lines, copy a line of pixels to a position on the screen, plot a single pixel) and is not typically accessed directly by application since it is missing many of the graphics primitives that an application is likely to require such as slanted lines, rectangles, circles, text, and image support.

The display driver must be supplied by the application since it is specific to the display in use. For example, a driver that is specific to Kentec320x240x16.

3.1.2 Low-Level Primitive Graphics API Overview

The first API that is intended for application use is the low-level graphics API. This gives access to functions that draw the major graphics primitives: lines, rectangles, circles, text, and images. In addition, functions and macros are provided to perform coordinate checking and rectangle processing. For example, checking for intersection and overlap, determining whether a point lies within a rectangle, and so on.

The low-level primitive graphics API only handles drawing to the display and has no knowledge of user input or any high-level control.

3.1.3 Widget API Overview

Above the low-level graphics API and the input driver, the widget API offers a high-level interface that allows the programming to build a complex user interface that includes individual control such as buttons, sliders, checkboxes, and other high-level widgets (controls).

Table 3-2 shows the widget classes supported by TivaWare Graphic Library.

Table 3-2. Widget Classes

Widget	Source File	Description
Canvas	canvas.c	The canvas widget provides a simple drawing surface that provides no means for interaction with the user. The canvas has the ability to be filled with a color, outlined with a color, have an image drawn in the center, have text drawn within it, and allow the application to draw into the canvas.
Checkbox	checkbox.c	The checkbox widget provides a graphical element that can be selected or unselected, resulting in a binary selection (such as “on” or “off”). A checkbox widget contains two graphical elements; the checkbox itself (which is drawn as a square that is either empty or contains an “X”) and the checkbox area around the checkbox that visually indicates what the checkbox controls.
Container	container.c	The container widget provides means of grouping widget together within the widget hierarchy, most notably useful for joining together several radio button widgets to provide a single one-of selection. The container widget can also provide a visual grouping of the child widgets by drawing a box around the widget area.
Image Button	imgbutton.c	The image button widget provides a button that can be pressed, causing an action to be performed. An image button is defined using a background image, a pressed-state background image, a keycap image and, optionally, a text string. The use of independent background and keycap images can offer memory saving in some applications which wish to show many similar buttons.
ListBox	listbox.c	The listbox widget allows the user to select one from a list of several strings held by the widget. The touch screen can be used to select and deselect a string by tapping it or to scroll through the strings in the listbox by pressing and dragging on the screen. Whenever the selected element in the box changes, a message is sent to an application callback informing it of the new selection (or lack thereof).
Keyboard	keyboard.c	The keyboard widget allows the user to create an on-screen keyboard for entering text without an external keyboard. The touch screen can be used to handle the pointer for selecting keys. Whenever a key is pressed a message is sent to the application callback to allow the application to handle the newly pressed key. The keyboard widget does not handle printing any of the keys as they are pressed, leaving all processing of keys to the application.
Push Button	pushbutton.c	The push button widget provides a button that can be pressed, causing an action to be performed. A push button has the ability to be filled with a color, outlined with a color, have an image drawn in the center, and have text drawn in the center. Two fill colors and two images can be utilized to provide a visual indication of the pressed or released state of the push button.
Radio Button	radiobutton.c	The radio button widget provides a graphical element that can be grouped with other radio buttons to form a means of selecting one of many items. For example, three radio buttons can be grouped together to allow a selection between “low”, “medium”, and “high”, where only one can be selected at a time. A radio button widget contains two graphical elements; the radio button itself (which is drawn as a circle that is either empty or contains a filled circle) and the radio button area around the radio button that visually indicates what the radio button controls.
Slider	slider.c	The slider widget allows the user to drag a marker either horizontally or vertically to select a value from within an application-supplied range.

The widgets are organized in a tree structure, and can be dynamically added or removed from the active widget tree. The tree structure allows messages to be delivered in a controlled manner. Each message is delivered in either top-down or bottom-up order based on the semantics of the message.

Using hello_widget.c as an example, it has a widget tree with four levels starting from WIDGET_ROOT shown in Figure 3-2.

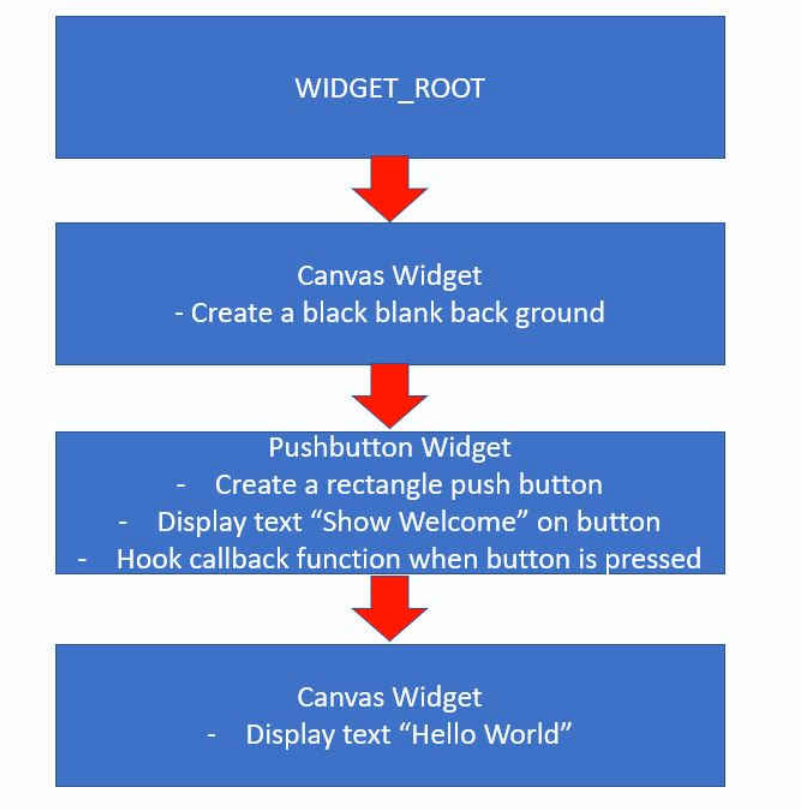

Figure 3-2. Example Widget Tree

The widget layer ties the graphic display to the input system. It manages the input and updates the displayed widgets according to button or touch screen presses made by the user. Application interaction with widgets is via callback functions provided during initialization. These callbacks are specific to the type of widget but would include functions called when a button is pressed or a slider is moved.

Figure 3-3 shows some of the attributes of the hello_widget.c example code on how widgets are declared and how the callback function is called when a touch event is detected.

1. Declare a canvas widget named g_sBackground. Its parent is WIDGET_ROOT. It has no sibling widget but has a child widget called g_sPushBtn.
2. Declared a push button widget named g_sPushBtn. Its parent is g_sBackground. It has no sibling and neither a child widget.
3. This g_sPushBtn push button widget will display text “Show Welcome”. A callback function OnButtonPress is hooked to this widget.
4. Declare a canvas widget named g_sHello. Its parent is g_sPushBtn. It is no sibling widget and neither a child widget.
5. The g_sHello canvas widget will display text “Hello World” on the display at the specified coordinates. This widget is not hooked into the active widget tree (by making it a child of the g_sPushBtn widget) yet since it is not intended for the widget to be displayed until the button is pressed.
6. When the button is pressed, the callback function OnButtonPress is called. In the function, it first adds the g_sHello as a child to the g_sPushBtn widget. It then changes the text attribute of the g_sPushBtn widget from “Show Welcome” to “Hide Welcome”. Lastly, a call to repaint the push button widget and all widgets beneath it (In this case, the g_sHello widget that will print “Hello World” on the display.

Figure 3-3. Example Code Analysis

The widget framework provides a generic means of dealing with a wide variety of widgets. Each widget has a message handler that responds to a set of generic messages; for example, the WIDGET_MSG_PAINT message is sent to request that the widget draw itself onto the screen.

3.1.4 Input Driver Overview

The input driver, like the display driver, is responsible for managing a block of hardware and translating user interaction into a standard format that the widget manager can understand. An application will not typically call the input driver other than during startup when a call is made to initialize the device.

For more information, see the call log in Figure 3-4 and Figure 3-5. Each box corresponds to a specific graphics library layer after a button is pressed on the display for the hello_widget example.

1. Input driver. For more information, see Figure 1-6(B) where the display module does not have its own touch controller. Resistive touchscreen displays are composed of multiple layers that are separated by thin spaces. Pressure applied to the surface of the display by fingers causes the layers to touch, which completes electrical circuits and tells the device where the user is touching. Refer to the source code of TouchScreenIntHandler() and TouchScreenDebouncer() drivers where a touch algorithm is implemented using GPIO and ADC to determine the position on the display that was touched.
2. Widget manager. A detection of a touch action causes ADC to sample the position of the touch and sends a message with a specific message type (WIDGET_MSG_PTR_UP message indicating the pointer/finger is now up/released) to the widget manager. Multiple messages could be added to the message queue for processing.
3. Widget manager. The widget manager constantly polls for messages to handle. When a message is received, it sends a message to the entire widget tree.

4. Push button widget class. In this call log, the widget manager routes the message to the push button widget.
5. Low-level primitive. A low-level primitive is called to render text on the screen. The hello_widget is designed to add new text on the display after the display is touched.
6. Display driver. The lowest level display driver function is invoked to render text using the PixelDraw function.

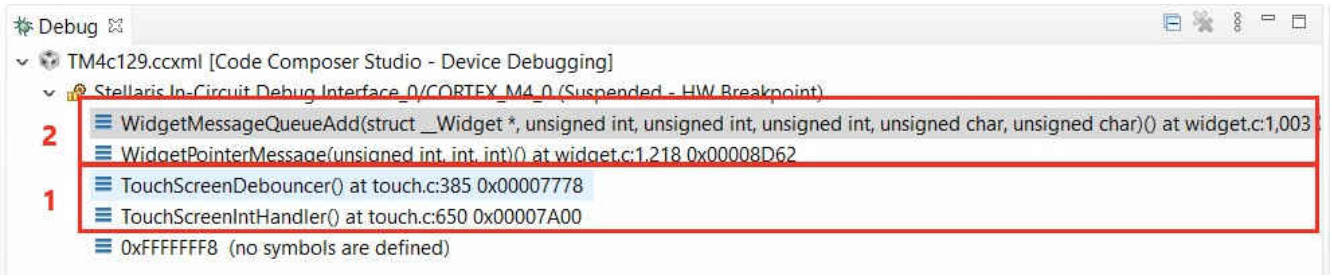

Figure 3-4. Input Driver Creating Message Request

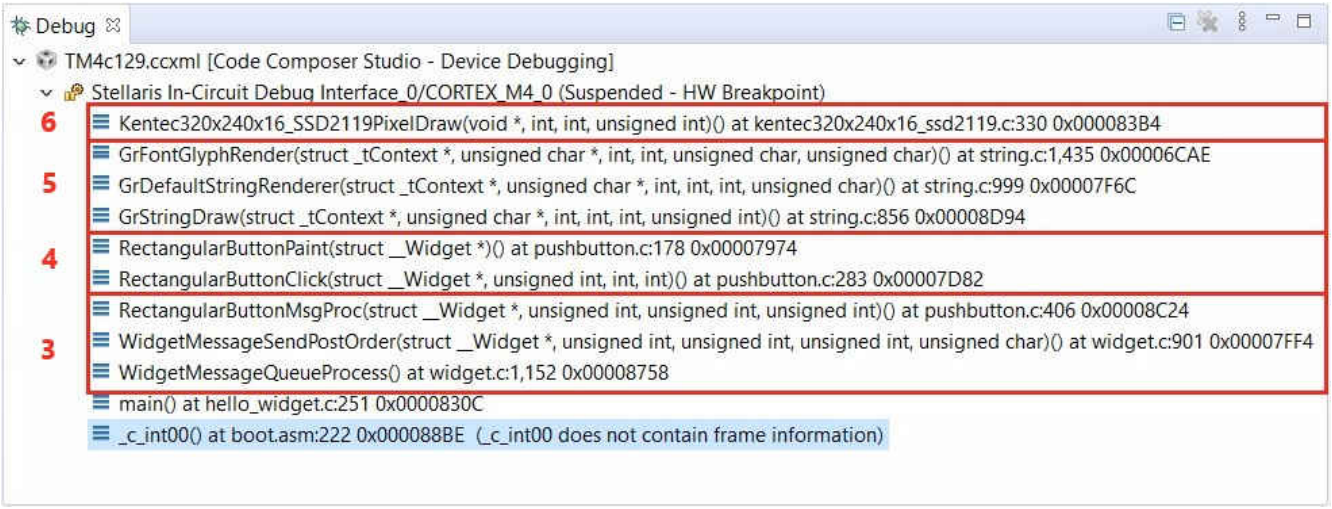

Figure 3-5. Widget Manager Responds to Message Request

4 Display Driver Adaptation

The lowest level of the graphics library stack is the display driver. Although the display driver API is specified by the graphics library, the source is specific to the board and display hardware and can be found in the C:\ti\TivaWare_C_Series-2.2.0.295\examples\boards\<board>\drivers directory. The driver source file name is typically derived from the display manufacturer, supported display controller part, display resolution, and bit depth. For example, the display driver for the dk-tm4c129x is named Kentec320x240x16_ssd2119.c since it supports a Kentec display using an SSD2119 controller and offers 320 x 240 resolution at 16 bits per pixel. The display driver for the SPI-based BOOSTXL-K350QVG-SI BoosterPack to be used on the EK-TM4C1294XL LaunchPad is named Kentec320x240x16_ssd2119_spi.c.

The display driver's responsibility is to translate calls made to the standard display driver API into orders to draw pixels or lines on the display. The interface to the driver is intended to offer the absolute minimum subset of drawing orders required to support the main graphics library and, as a result, make it extremely straightforward to develop a driver for a new display very quickly.

The display driver API includes the following basic functions that must be supported by every display driver:

```
static void
PixelDraw(void *pvDisplayData, int32_t i32X,
                               int32_t i32Y, uint32_t ui32Value)

static void
PixelDrawMultiple(void *pvDisplayData, int32_t i32X,
                                       int32_t i32Y, int32_t i32X0,
                                       int32_t i32Count, int32_t i32BPP,
                                       const uint8_t *pui8Data,
                                       const uint8_t *pui8Palette)

static void
LineDrawH(void *pvDisplayData, int32_t i32X1,
                               int32_t i32X2, int32_t i32Y,
                               uint32_t ui32Value)

static void
LineDrawV(void *pvDisplayData, int32_t i32X,
                               int32_t i32Y1, int32_t i32Y2,
                               uint32_t ui32Value)

static void
RectFill(void *pvDisplayData, const tRectangle *psRect,
                              uint32_t ui32Value)

static uint32_t
ColorTranslate(void *pvDisplayData, uint32_t ui32Value)

static void
2119Flush(void *pvDisplayData)
```

The actual names of these functions are not important since they are provided to the graphics library by means of a function pointer table. This can be found in the tDisplay structure that the display driver exports and that the applications uses when calling the graphics API function.

```
const tDisplay g_sKentec320x240x16_SSD2119 =
{
    sizeof(tDisplay),
#if defined(PORTRAIT) || defined(PORTRAIT_FLIP)
    240,
    320,
#else
    320,
    240,
#endif
    Kentec320x240x16_SSD2119PixelDraw,
    Kentec320x240x16_SSD2119PixelDrawMultiple,
    Kentec320x240x16_SSD2119LineDrawH,
    Kentec320x240x16_SSD2119LineDrawV,
    Kentec320x240x16_SSD2119RectFill,
    Kentec320x240x16_SSD2119ColorTranslate,
    Kentec320x240x16_SSD2119Flush
};
```

Additionally, the display driver typically provides an initialization function (Kentec320x240x16_SSD2119Init()) that the application is expected to call prior to initializing the graphics library. This call is used to initialize the underlying graphics hardware and clear the screen.

Notice that the driver API contains significantly fewer graphics primitives than the low-level graphics API. Most graphics primitives are broken down by the higher-level code and passed to the driver in pieces. For example, an unfilled rectangle is drawn using two calls to the LineDrawV function and two calls to the LineDrawH function. Similarly, text is rendered using multiple calls to the PixelDraw and LineDrawH functions. This model works well with small, low-cost displays which do not typically include any graphics acceleration hardware but do often include the ability to choose drawing direction and copy lines of pixels.

The other higher-level feature carried out by the low-level graphics layer on behalf of the display driver is clipping. No coordinates that are outside the bounds of the display are ever passed to the display driver since this is checked for and handled in the layer above. Using this approach, it becomes quick and easy to produce a new graphics driver since only a small number of simple functions need to be developed.

4.1 Off-Screen Display Drivers

Although most display drivers are intended to allow specific hardware displays to be used with the TivaWare Graphics Library, three special drivers are included within the library itself. These drivers are intended for off-screen graphics rendering in 1 bpp (bit per pixel), 4 bpp, and 8 bpp formats and are typically used in combination with a driver which supports the physical display. These drivers support the standard display driver interface and may be used alongside other display drivers.

The main use for an off-screen display driver is to support applications which require smooth animation or which render an image slowly. In these cases, an image is drawn into a memory buffer using the offscreen display driver and, once the image is completed, it is transferred to the physical display in one operation. Since the rendering of the image takes many steps and may include erasing the entire buffer before starting to redraw, using an off-screen display driver allows flicker-free operation. The physical display continues to show the previous image until a new one is ready for display, at which point the image is updated so quickly that the user does not see any of the intervening graphic operations that were required to generate the new image.

Source for the off-screen display drivers can be found in the offscr1bpp.c, offscr4bpp.c, and offscr8bpp.c files in the C:\ti\TivaWare_C_Series-2.2.0.295\grlib directory.

4.2 Individual Display Driver Functions

This section describes the individual display driver functions in detail. Note that the first parameter to each function, pvDisplayData, is a pointer that the driver itself provides in the tDisplay structure it exports. The driver does not need to use this parameter, but it is provided to support drivers which must maintain state data.

4.2.1 Init

The prototype for the driver initialization function is driver-specific. An application calls this function directly prior to initializing the low-level graphics API layer and the function then initializes the display hardware and blanks the screen in preparation for receiving other calls.

4.2.2 ColorTranslate

```
unsigned long ColorTranslate(void *pvDisplayData,
                             unsigned long ulRGBColor);
```

The higher-level graphics driver APIs make use of a standard 24-bit RGB color description with the color described in a single, unsigned long value with the red component in bits 16 to 23, green in bits 8 to 15, and blue in bits 0 to 7. Different displays, however, describe color in different ways so the ColorTranslate function allows the graphics library to obtain a representation of a given RGB24 color in the native format supported by the display.

For monochrome displays, the returned value should represent the brightness of the supplied RGB color. For color displays, the returned value should represent the original color as closely as possible given the constraints of the display. If the display supports 16-bit RGB, for example, the returned color value truncates, masks, and shifts the supplied 8-bit R,G, and B samples into a correctly packed 16-bit value.

All other calls to the display driver will be passed pre-translated colors which are colors that have been returned from a previous call to the ColorTranslate function, so the overhead of color translation is kept to a minimum.

4.2.3 PixelDraw

```
void PixelDraw(void *pvDisplayData, long lX, long lY,
       unsigned long DispColor);
```

The simplest function that a display driver must support is the ability to plot a single pixel at a given position on the display. This function plots a pixel using color ulDispColor at position (lX, lY) on the screen. Note that the color passed has already been translated into the display-dependent format using a previous call to the ColorTranslate function.

4.2.4 PixelDrawMultiple

```
void PixelDrawMultiple(void *pvDisplayData, long lX, long lY,
              long lX0, long lCount, long lBPP,
              const unsigned char *pucData,
              const unsigned char *pucPalette);
```

The PixelDrawMultiple function is used when displaying images. A block of pixel data representing a given horizontal span is passed to the display driver, which renders the pixel data onto the display at the specified position. In this case, the driver must support 1 bpp, 4 bpp, and 8 bpp pixel formats. Displays supporting 16 bpp formats may also support native 16 bpp pixels.

For the 1 bpp pixel format, the pucPalette points to a 2-entry array containing pre-translated colors for background and foreground pixels. For 4 bpp and 8 bpp formats, the pucPalette parameter points to a color table containing RGB24 colors which the driver must translate to the native color format during the drawing process.

The palette is ignored for the 16 bpp formats since it is assumed that the pixels passed are in the native color format of the display.

When using 1 bpp and 4 bpp formats, the lX0 parameter indicates where the first pixel to draw is within the first byte of supplied pixel data. For 1 bpp, valid values are 0 through 7 and for 4 bpp, values 0 or 1 may be used. In each case, pixels are packed with the leftmost pixel in the most significant bit or nibble of the byte. Taking 1bpp as an example, if lX0 is 5 this indicates that the 5 leftmost pixels are skipped in the first byte passed and will draw 3 pixels from that byte. These will be taken from bits 2 , 1 and 0 of the byte.

4.2.5 LineDrawH

```
void LineDrawH (void *pvDisplayData, long lX1, long lX2
        long lY, unsigned long ulDispColor);
```

This function draws horizontal lines using the supplied, display-dependent color. Note that the line drawn includes both the first and last pixels specified by parameters lX1 and lX2, which means that the number of pixels written is ((lX2 – lX1) + 1). The graphics library ensures that lX2 is always greater than lX1, so no parameter sorting is required in the display driver.

4.2.6 LineDrawV

```
void LineDrawV (void *pvDisplayData, long lX1, long lX2
        long lY, unsigned long ulDispColor);
```

This function draws vertical lines using the supplied, display-dependent color. As for LineDrawH, the line drawn includes both the first and last pixels specified by parameters lY1 and lY2 meaning that the number of pixels written is ((lY2 – lY1) + 1). The graphics library ensures that lY2 is always greater than lY1 so no parameter sorting is required in the display driver.

4.2.7 RectFill

```
void RectFill(void *pvDisplayData, const tRectangle *pRect,
      unsigned long ulDispColor);
```

This function fills a rectangle on the display with the solid color provided in the ulDispColor parameter.

Note that the tRectangle type uses a bottom-right inclusive definition so the width of the rectangle to draw is given by ((pRect->sXMax – pRect->sXMin) + 1) and the height is ((pRect->sYMax – pRect->sYMin) +1). This is different from Windows and various other graphics libraries that use a bottom-right exclusive rectangle definition.

4.2.8 Flush

```
void Flush(void * pvDisplayData);
```

The flush function is provided to support display hardware that does not contain an integrated frame buffer and where the display driver must keep the display contents in a local RAM buffer. In this model, the drawing functions provided by the driver update the contents of the RAM buffer instead of updating the display. These changes are flushed to the actual display using the Flush API.

In drivers that update the display on each call to any of the driver drawing APIs, this call can be a stub that returns without performing any action.

Since the widget classes do not currently make use of the Flush() driver API, if a driver performs an off-screen rendering with the widget layer, it must update the widget classes that are used in order to call Flush() at the appropriate points in their paint functions

5 Fonts

Understanding the graphics library text handling functions will be easier if the following terminology is known:

- ASCII

  American Standard Code for information Interchange. ASCII is a 7-bit code-page with codepoints in the range 0x00-0x7F. It contains the basic upper and lower-case Latin alphabet, numeric digits, common punctuation marks and terminal control codes. It is in common use in English-speaking countries but offers no way to encode accented characters or non-Latin alphabets.
- codepage

  A character encoding scheme mapping between codepoints and glyphs within a font. The codepage determines which character a given codepoint (character number) represents. For example, when using the ASCII codepage, codepoint 0x20 represents the space character.
- codepoint

  A single entry in a codepage. A number identifying a character in a font. Knowing the codepage in use, the codepoint (or character code) defines a single character.
- glyph

  A graphical representation of a single character in a font.
- font

  A collection of character glyphs in a particular typeface and size each represented by a codepoint.
- UTF-8

  Unicode Transformation Format(8). A variable length encoding system for Unicode text where any given character can be represented by 1 to 6 bytes depending upon the character. UTF-8 has the advantage that it is backwards compatible with ASCII and is commonly used in text file processing.

The are a large range of fonts supplied with the grlib that can be used for rendering text on the screen. Additional fonts can be created by using the ftrasterize utility to compress font files into the format required by the grlib. For a full list of available fonts, see the *Font Reference* section in the *TivaWare™ Graphics Library User's Guide*.

Below is an example of specifying an 18-pt computer modern sans-serif font when drawing the “hello world” string using the grlib primitive functions.

```
GrContextFontSet(psContext, g_psFontCmss18b);
GrStringDrawCentered(psContext, "hello world", -1, 160, 8, false);
```

Step 5 in Figure 3-3 shows another example specifying a 40-pt computer modern serif font for a widget declaration.

5.1 Creating Custom Fonts for Different Languages

Western languages such as English uses letters to compose words. The number of letters or characters in Western languages is generally small and can be easily encoded using 7-bit character encoding scheme such as ASCII. However, languages such as Chinese consists of tens of thousands of characters. A typical font set for Chinese would have taken up enormous amount of memory in the range of megabytes. Certainly, this is not practical for embedded applications.

An application requiring Chinese language may only need to use a small subset of the Chinese characters. For example, if an application only needs to use 100 unique Chinese characters and if these 100 characters are known ahead of time then the programmer can create a custom font that only supports these 100 characters which will result in a small memory footprint. This is the purpose of creating custom fonts for non-Latin languages.

Examine the code snippet of a simple example shown in Figure 5-1 to demonstrate the effect of using custom font. The full source code can be found in Appendix A.

1. Start by setting the grlib to use a 20-pt custom font. In this 20-pt custom font, the only supported characters will be “W”, “e”, “l”, “c”, “o”, “m”, “欢”, “迎”, “您”. The last three characters are the Chinese characters for the word “Welcome” in English.

   For how to generate this custom font: see Section 5.1.
2. Render the string “Welcome” by calling the grlib primitive function. Cross reference to the first row in the display shown in Figure 5-2.
3. Render the string “abcdefghijklmnopqrstuvWxyz” by calling the grlib primitive function. Cross reference to the second row in the display. It may appear that something is broken but it is not. This is the intention of this example. As mentioned in step 1, a custom font was created that will support only English characters “W”, “e”, “l”, “c”, “o”, “m”. This means the rest of the 20 characters are not displayable as they are not in the font.
4. Render the string “欢迎您”. Cross reference to the third row in the display. As the font will only support these three Chinese characters, there is no surprise that the string is displayed properly. Any other characters will not be displayed. In order to display other Chinese characters, a new custom font must be generated.
5. This demonstrates another way to enter the three Chinese characters “欢”, “迎”, “您” by their corresponding UTF-8 character code. Each Chinese character is encoded by a three-byte codepoint in UTF-8. The character “欢” corresponds to UTF-8 “0xe6,0xac,0xa2”. There are various online tools that will convert any characters (Chinese, Japanese and any accented Latin characters) into their corresponding UTF-8 code. For an example, see Figure 5-3.

```
int
main(void)
{
    tContext sContext;
    uint32_t ui32SysClock;
    char string[10] = {0xe6, 0xac, 0xa2, 0xe8, 0xbf, 0x8e, 0xe6, 0x82, 0xa8, 0x00}; // UTF-8 code for "欢迎您"

    //
    // Run from the PLL at 120 MHz.
    //
    ui32SysClock = MAP_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL | SYSCTL_CFG_VCO_240), 120000000);
    //
    // Configure the device pins.
    //
    PinoutSet();
    //
    // Initialize the display driver.
    //
    Kentec320x240x16_SSD2119Init(ui32SysClock);
    //
    // Set graphics library text rendering defaults.
    //
    GrLibInit(&GRLIB_INIT_STRUCT);
    //
    // Initialize the graphics context.
    //
    GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
    //
    // Render white background
    //
    GrContextForegroundSet(&sContext, ClrWhite);
    //
    // Set font to a new custom 20pt font that supports only below characters:
    // "W", "e", "l", "c", "o", "m" and
    // "欢", "迎", "您"
    //
    GrContextFontSet(&sContext, FONT_20PT);
    //
    // Render the string "Welcome" on the display
    //
    GrStringDrawCentered(&sContext, "Welcome", -1, 160, 8, false);
    //
    // Render the string "abcdefghijklmnopqrstuvWxyz" on the display
    //
    GrStringDrawCentered(&sContext, "abcdefghijklmnopqrstuvWxyz", -1, 160, 56, false);
    //
    // Render the Chinese string "欢迎您" which means "Welcome" in English
    //
    GrStringDrawCentered(&sContext, "欢迎您", -1, 160, 32, false);
    //
    // Render the same Chinese string by using UFT-8 code
    //
    GrStringDrawCentered(&sContext, string, 8, 160, 80, false);
```

1

2

3

4

5

Figure 5-1. Custom Font Demonstration

Figure 5-2. Custom Font Display

Figure 5-3. Unicode Converter

6 Useful Utilities

There are several utility applications can be used to produce the data structures required by the graphics library for fonts and images since trying to produce these structures by hand would be a difficult process.

6.1 Pnmtoc

The pnmtoc utilities converts a NetPBM image file into the format that is recognized by the graphics library. The input image must be in the raw PPM format (in other words, with the P6 tag). The NetPBM image format can be produced using GIMP, NetPBM, ImageMagick, or numerous other open source and proprietary image manipulation packages.

Use an example to demonstrate the usage of this utility:

1. Load a JPEG image file into GIMP, see Figure 6-1.
2. Scale the image to 320x240 resolution.
3. Convert the image to indexed mode (Image->Mode->Indexed). Select “Generating optimum palette” and select either 2, 16, or 256 as the maximum number of colors (for a 1 BPP, 4 BPP, or 8 BPP image respectively). If the image is already in indexed mode, it can be converted RGB mode (image->Mode->RGB) and then back to indexed mode.
4. Save the file as a PNM image (File->Save As). Select raw format when prompted.
5. Use pnmtoc to convert the PNM image into c array as in:

   > pnmtoc -c <input_image.pnm> output_image.c
6. To test the new image, replace the g_pucLogo array in the image.c file for the grlib_driver_test example with the new array generated in the output_image.c file.
7. Recompile the grlib_driver_test project and run. You should see the new image rendered on the display, see Figure 6-2.

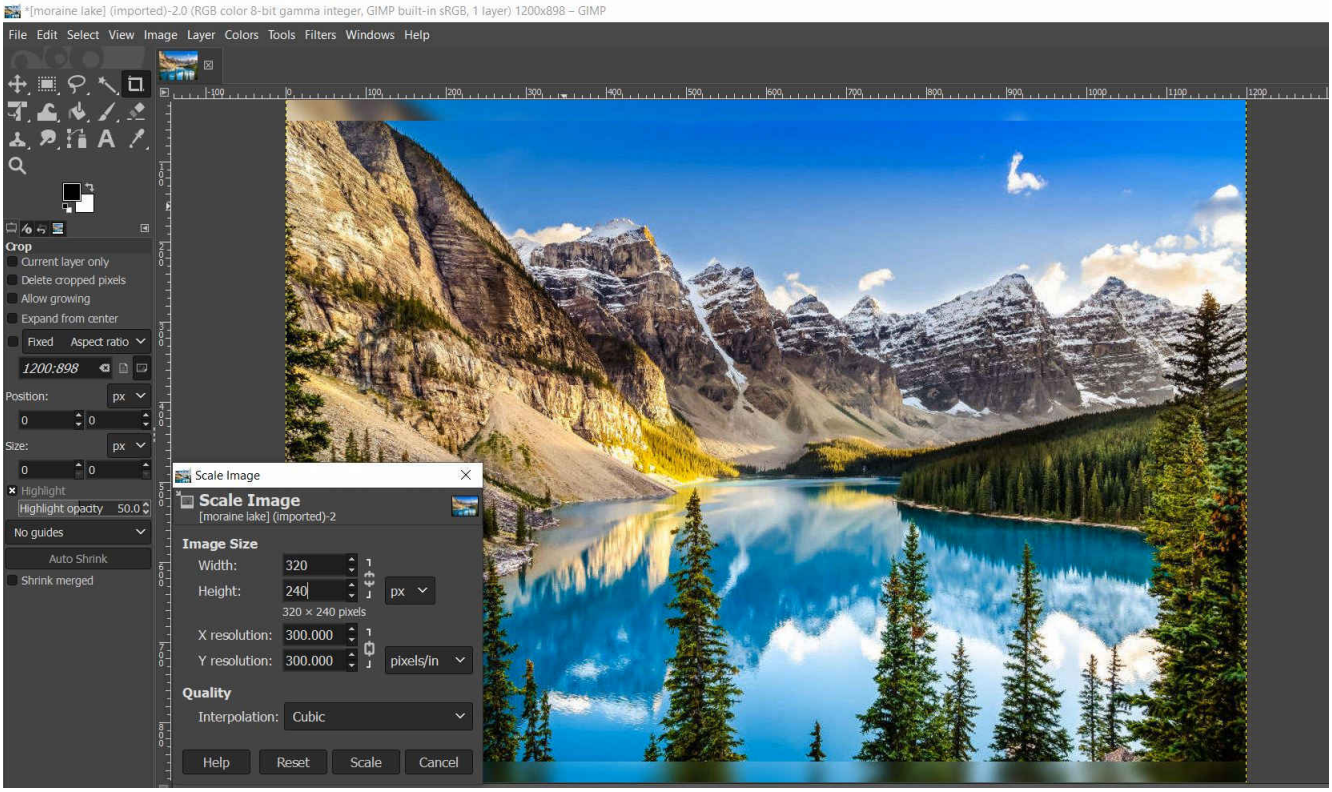

Figure 6-1. Gimp Scaling the Image to 320x240 Resolution

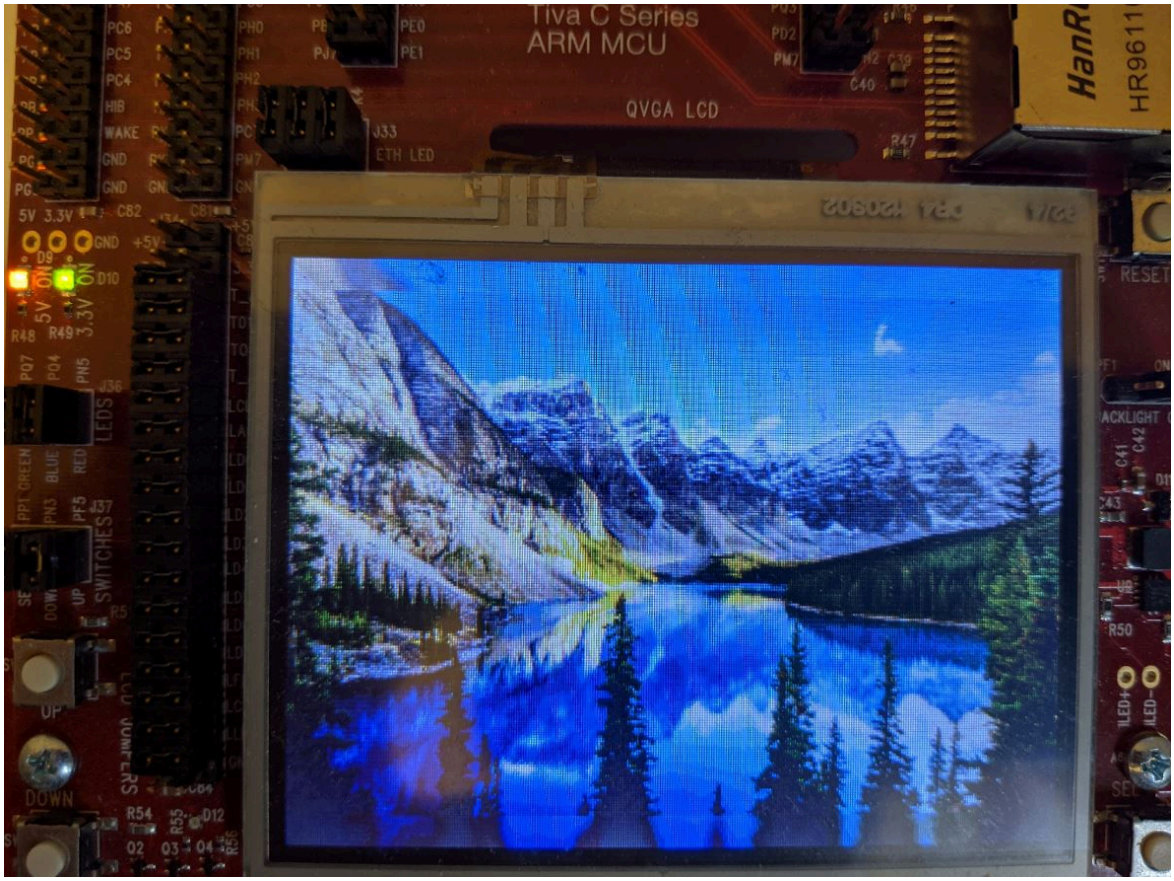

Figure 6-2. Rendering New Image on The Display

6.2 mkstringtable and ftrasterize

The strings for the application example described in Section 5.1 is saved in the language.csv file (encoded in UTF8 format) to allow accented characters and Asian language ideographs to be included. The mkstringtable tool is used to generate two versions of the string table, one which remains encoded in UTF8 format and the other one that is remapped to a custom codepage allowing the table to be reduce in size compared to the original UTF8 text. The tool also produces character map files listing each character used in the string table. These are then provided as input to the ftrasterize tool which generates two custom fonts for the application, one indexed using Unicode and a smaller one indexed using the custom codepage generated for this string table. The command line parameters required for mkstringtable and ftrasterize can be found in the Makefile in C:\ti\TivaWare_C_Series-2.2.0.295\third_party\fonts\lang_demo.

The ftrasterize utility uses the FreeType font rendering package to convert a font into the format that is recognized by the graphics library. Any font that is recognized by FreeType can be used, which includes TrueType®, OpenType®, PostScript® Type 1, and Windows® FNT fonts. A complete list of supported font formats can be found on the FreeType web site at http://www.freetype.org.

There is an already a language.csv file in the existing \third_party\fonts\lang_demo directory which is created for the lang_demo example described in Table 3-1. The file can be modified for the string table to be used in the example shown in Section 5.1.

1. Open a command window and go to \third_party\fonts\lang_demo directory.
2. Create or modify the existing string table source language.csv file and save it in a comma separated values (CSV) file format, which is a plain text file that contains a list of data.

Figure 6-3. String Table in CSV Format

3. At the command line, run the make command. If needed, view the Makefile in the directory for details. The Makefile calls the mkstringtable tool to generate the string table and then use the generated string table as an input to the ftrasterize tool. The ftrasterize tool generates a subset custom fonts based on the string table and the complete font sets (fireflysung.ttf for Chinese characters and NanumMyeongjo for Korean). The original Makefile is intended for the lang_demo example that display the various characters for different languages. For this simple example that is intended to display only English ASCII characters and Chinese characters, only the fireflysung.ttf is required.

```
#
# Build our string table from the CSV file.
#
language.c: language.csv
     ../../../tools/bin/mkstringtable -u -f language.csv -b language -s utf8 -t

#
# Build our remapped string table from the CSV file.
#
langremap.c: language.csv
     ../../../tools/bin/mkstringtable -u -c 0x8000 -f language.csv -b langremap -s utf8 -r

#
# Build the custom font used by the application based on the output of the
# mkstringtable step.  We build 2 fonts, one for use with the remapped
# string table and the other indexed using normal Unicode codepoints.
#
fontcustom14pt.c: language.c AndBasR.ttf fireflysung.ttf  NanumMyeongjo-Regular.ttf sazanami-
gothic.ttf
    ../../../tools/bin/ftrasterize -u -c language.txt -r -f custom -s 14 AndBasR.ttf
fireflysung.ttf NanumMyeongjo-Regular.ttf sazanami-gothic.ttf
```

```
fontcustomr14pt.c: langremap.c AndBasR.ttf fireflysung.ttf  NanumMyeongjo-Regular.ttf sazanami-
gothic.ttf
    ../../../tools/bin/ftrasterize -u -c langremap.txt -r -z 0x8000 -f customr -s 14
AndBasR.ttf fireflysung.ttf NanumMyeongjo-Regular.ttf sazanami-gothic.ttf

fontcustom20pt.c: language.c AndBasR.ttf fireflysung.ttf  NanumMyeongjo-Regular.ttf sazanami-
gothic.ttf
    ../../../tools/bin/ftrasterize -u -c language.txt -r -f custom -s 20 AndBasR.ttf
fireflysung.ttf NanumMyeongjo-Regular.ttf sazanami-gothic.ttf


fontcustomr20pt.c: langremap.c AndBasR.ttf fireflysung.ttf  NanumMyeongjo-Regular.ttf sazanami-
gothic.ttf
    ../../../tools/bin/ftrasterize -u -c langremap.txt -r -z 0x8000 -f customr -s 20 AndBasR.ttf
fireflysung.ttf NanumMyeongjo-Regular.ttf sazanami-gothic.ttf
```

4. For the simple welcome example, copy only the files language.h, language.c, fontcustom14pt.c, and fontcustom20pt.c to the project directory.

7 References

- Texas Instruments: *TivaWare™ Graphics Library User's Guide*
- Texas Instruments: *Tiva™ TM4C1297NCZAD Microcontroller Data Sheet*
- Texas Instruments: *TivaWare™ Peripheral Driver Library User's Guide*

A Appendix A

```
#include <stdint.h>
#include <stdbool.h>
#include "driverlib/sysctl.h"
#include "driverlib/rom.h"
#include "driverlib/rom_map.h"
#include "grlib/grlib.h"
#include "drivers/kentec320x240x16_ssd2119.h"
#include "drivers/pinout.h"


//*****************************************************************************
// Define the custom fonts
//
//*****************************************************************************
#include "language.h"

extern const unsigned char g_pui8Custom14pt[];
extern const unsigned char g_pui8Custom20pt[];

#define FONT_20PT (const tFont *)g_pui8Custom20pt
#define FONT_14PT (const tFont *)g_pui8Custom14pt
#define GRLIB_INIT_STRUCT g_sGrLibDefaultlanguage


//*****************************************************************************
//
// A simple demonstration to show custom fonts
//
//*****************************************************************************
int
main(void)
{
    tContext sContext;
    uint32_t ui32SysClock;
    char string[10] = {0xe6, 0xac, 0xa2, 0xe8, 0xbf, 0x8e, 0xe6, 0x82, 0xa8, 0x00}; // UTF-8 code
for "欢迎您"

    //
    // Run from the PLL at 120 MHz.
    //
    ui32SysClock = MAP_SysCtlClockFreqSet((SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN | SYSCTL_USE_PLL |
SYSCTL_CFG_VCO_240), 120000000);
    //
    // Configure the device pins.
    //
    PinoutSet();
    //
    // Initialize the display driver.
    //
    Kentec320x240x16_SSD2119Init(ui32SysClock);
    //
    // Set graphics library text rendering defaults.
    //
    GrLibInit(&GRLIB_INIT_STRUCT);
    //
    // Initialize the graphics context.
    //
    GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
    //
    // Render white background
    //
    GrContextForegroundSet(&sContext, ClrWhite);
    //
    // Set font to a new custom 20pt font that supports only below characters:
    // "W", "e", "l", "c", "o", "m" and
    // "欢", "迎", "您"
    //
    GrContextFontSet(&sContext, FONT_20PT);
    //
    // Render the string "Welcome" on the display
    //
    GrStringDrawCentered(&sContext, "Welcome", -1, 160, 8, false);
    //
    // Render the string "abcdefghijklmnopqrstuvWxyz" on the display
    //
    GrStringDrawCentered(&sContext, "abcdefghijklmnopqrstuvWxyz", -1, 160, 32, false);
```

```
    //
    // Render the Chinese string "欢迎您" which means "Welcome" in English
    //
    GrStringDrawCentered(&sContext, "欢迎您", -1, 160, 56, false);
    //
    // Render the same Chinese string by using UFT-8 code
    //
    GrStringDrawCentered(&sContext, string, 8, 160, 80, false);

    while(1)
    {

    }

}
```

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), 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, regulatory 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 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.

TI objects to and rejects any additional or different terms you may have proposed.

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