DK-TM4C129X Firmware Development Package

USER'S GUIDE

SW-DK-TM4C129X-UG-2.2.0.295

Copyright

Texas Instruments
108 Wild Basin, Suite 350
Austin, TX 78746
www.ti.com/tiva-c

Revision Information

This is version 2.2.0.295 of this document, last updated on April 02, 2020.

Table of Contents

1 Introduction

The Texas Instruments® Tiva™ DK-TM4C129X development board is a platform that can be used for software development and prototyping a hardware design. It can also be used as a guide for custom board design using a Tiva microcontroller.

The DK-TM4C129X includes a Tiva ARM® Cortex™-M4-based microcontroller and the following features:

- Tiva™ TM4C129XNCZAD microcontroller
- TFT display (320x240 16 bpp) with capacitive touch screen overlay
- Ethernet connector
- USB OTG connector
- 64 MB SPI flash
- MicroSD card connector
- Temperature sensor
- Speaker with class A/B amplifier
- 3 user buttons
- User LED
- 2 booster pack connectors
- EM connector
- On-board In-Circuit Debug Interface (ICDI)
- Power supply option from USB ICDI connection or external power connection
- Shunt for microcontroller current consumption measurement

This document describes the board-specific drivers and example applications that are provided for this development board.

2 Example Applications

The example applications show how to utilize features of the DK-TM4C129X development board. Examples are included to show how to use many of the general features of the Tiva microcontroller, as well as the feature that are unique to this development board.

A number of drivers are provided to make it easier to use the features of the DK-TM4C129X. These drivers also contain low-level code that make use of the TivaWare peripheral driver library and utilities.

There is an IAR workspace file (`dk-tm4c129x.eww`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy-to-use workspace for use with Embedded Workbench version 5.

There is a Keil multi-project workspace file (`dk-tm4c129x.mpw`) that contains the peripheral driver library project, along with all of the board example projects, in a single, easy-to-use workspace for use with uVision.

All of these examples reside in the `examples/boards/dk-tm4c129x` subdirectory of the firmware development package source distribution.

2.1 AES CBC Decryption Demo (aes_cbc_decrypt)

Simple demo showing an decryption operation using the AES128 and AES256 modules in CBC mode. A number of blocks of data are decrypted.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.2 AES CBC Encryption Demo (ae_cbc_encrypt)

Simple demo showing an encryption operation using the AES128 and AES256 modules in CBC mode. A number of blocks of data are encrypted.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.3 AES128 and AES256 CCM Decryption Demo (aes_ccm_decrypt)

Simple demo showing an decryption operation using the AES128 and AES256 modules in CCM mode. A set of test cases are decrypted.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.4 AES128 and AES256 CCM Encryption Demo (aes_ccm_encrypt)

Simple demo showing an encryption operation using the AES128 and AES256 modules in CCM mode. A set of test cases are encrypted.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.5 AES128 and AES256 CMAC Demo (aes128_cmac)

Simple demo showing an authentication operation using the AES128 and AES256 modules in CMAC mode. A series of test vectors are authenticated.

This module is also capable of CBC-MAC mode, but this has been determined to be insecure when using variable message lengths. CMAC is now recommended instead by NIST.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.6 AES128 and AES256 ECB Decryption Demo (aes_ecb_decrypt)

Simple demo showing an decryption operation using the AES128 and AES256 modules in ECB mode. A single block of data is decrypted.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.7 AES128 and AES256 ECB Encryption Demo (aes_ecb_encrypt)

Simple demo showing an encryption operation using the AES128 and AES256 modules in ECB mode. A single block of data is encrypted.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.8 AES128 and AES256 GCM Decryption Demo (aes_gcm_decrypt)

Simple demo showing authenticated decryption operations using the AES module in GCM mode. The test vectors are from the gcm_revised_spec.pdf document.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.9 AES128 and AES256 GCM Encryption Demo (aes_gcm_encrypt)

Simple demo showing authenticated encryption operations using the AES module in GCM mode. The test vectors are from the gcm_revised_spec.pdf document.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.10 Bit-Banding (bitband)

This example application demonstrates the use of the bit-banding capabilities of the Cortex-M4 microprocessor. All of SRAM and all of the peripherals reside within bit-band regions, meaning that bit-banding operations can be applied to any of them. In this example, a variable in SRAM is set to a particular value one bit at a time using bit-banding operations (it would be more efficient to do a single non-bit-banded write; this simply demonstrates the operation of bit-banding).

2.11 Blinky (blinky)

A very simple example that blinks the on-board LED.

2.12 Ethernet Boot Loader Demo (boot_demo_emac_flash)

An example to demonstrate the use of remote update signaling with the flash-based Ethernet boot loader. This application configures the Ethernet controller and acquires an IP address which is displayed on the screen along with the board's MAC address. It then listens for a "magic packet" telling it that a firmware upgrade request is being made and, when this packet is received, transfers control into the boot loader to perform the upgrade.

This application is intended for use with flash-based ethernet boot loader (boot_emac_flash).

The link address for this application is set to 0x4000, the link address has to be multiple of the flash erase block size(16KB=0x4000). You may change this address to a 16KB boundary higher than the

last address occupied by the boot loader binary as long as you also rebuild the boot loader itself after modifying its bl_config.h file to set APP_START_ADDRESS to the same value.

The boot_demo_flash application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

2.13 Flash-based Boot Loader Demo (boot_demo_flash)

An example to demonstrate the use of a flash-based boot loader. At startup, the application will configure the UART, USB and Ethernet peripherals, and then branch to the boot loader to await the start of an update.

This application is intended for use with flash-based ethernet boot loader (boot_emac_flash). Although it isn't neccessary to configure USB and UART interface in this example since only ethernet flash-based boot loader is provided, the code is there for completeness so that USB or UART flash-based boot loader can be implemented if needed. The link address for this application is set to 0x4000, the link address has to be multiple of the flash erase block size(16KB=0x4000). You may change this address to a 16KB boundary higher than the last address occupied by the boot loader binary as long as you also rebuild the boot loader itself after modifying its bl_config.h file to set APP_START_ADDRESS to the same value.

The boot_demo_emac_flash application can be used along with this application to easily demonstrate that the boot loader is actually updating the on-chip flash.

2.14 Boot Loader UART Demo (boot_demo_uart)

This example demonstrates the use of the ROM-based UART boot loader. When the application runs, it will wait for the screen to be pressed (to simulate an application waiting for a trigger to begin a firmware update) and then branch to the UART boot loader in the ROM. The UART is configured for a baud rate of 115,200 and does not require the use of auto-bauding.

2.15 Boot Loader USB Demo (boot_demo_usb)

This example application is used in conjunction with the USB boot loader in ROM and turns the development board into a composite device supporting a mouse via the Human Interface Device class and also publishing runtime Device Firmware Upgrade (DFU) capability. Dragging a finger or stylus over the touchscreen translates into mouse movement and presses on marked areas at the bottom of the screen indicate mouse button press. This input is used to generate messages in HID reports sent to the USB host allowing the development board to control the mouse pointer on the host system.

Since the device also publishes a DFU interface, host software such as the dfuprog tool can determine that the device is capable of receiving software updates over USB. The runtime DFU protocol allows such tools to signal the device to switch into DFU mode and prepare to receive a new software image.

Runtime DFU functionality requires only that the device listen for a particular request (DETACH) from the host and, when this is received, transfer control to the USB boot loader via the normal

means to reenumerate as a pure DFU device capable of uploading and downloading firmware images.

Windows device drivers for both the runtime and DFU mode of operation can be found in `C:/TI/TivaWare_C_Series-x.x/windows_drivers` assuming you installed TivaWare in the default directory.

To illustrate runtime DFU capability, use the `dfuprog` tool which is part of the Tiva Windows USB Examples package (SW-USB-win-xxxx.msi) Assuming this package is installed in the default location, the `dfuprog` executable can be found in the `C:/Program Files/Texas Instruments/Tiva/usb_examples` or `C:/Program Files (x86)/Texas Instruments/Tiva/usb_examples` directory.

With the device connected to your PC and the device driver installed, enter the following command to enumerate DFU devices:

```
dfuprog -e
```

This will list all DFU-capable devices found and you should see that you have one or two devices available which are in “Runtime” mode.

If you see two devices, it is strongly recommended that you disconnect ICDI debug port from the PC, and power the board either with a 5V external power brick or any usb wall charger which is not plugged in your pc. This way, your PC is connected to the board only through USB OTG port. The reason for this is that the ICDI chip on the board is DFU-capable device as well as TM4C129X, if not careful, the firmware on the ICDI chip could be accidently erased which can not restored easily. As a result, debug capabilities would be lost!

If IDCI debug port is disconnected from your PC, you should see only one device from above command, and its index should be 0, and should be named as “Mouse with Device Firmware Upgrade”. If for any reason that you cannot provide the power to the board without connecting ICDI debug port to your PC, the above command should show two devices, the second device is probably named as “In-Circuit Debug interface”, and we need to be careful not to update the firmware on that device. So please take careful note of the index for the device “Mouse with Device Firmware Upgrade”, it could be 0 or 1, we will need this index number for the following command. Entering the following command will switch this device into DFU mode and leave it ready to receive a new firmware image:

```
dfuprog -i index -m
```

After entering this command, you should notice that the device disconnects from the USB bus and reconnects again. Running “`dfuprog -e`” a second time will show that the device is now in DFU mode and ready to receive downloads. At this point, either LM Flash Programmer or dfuprog may be used to send a new application binary to the device.

2.16 Ethernet flash-based Boot Loader (boot_emac_flash)

The boot loader is a small piece of code that can be programmed at the beginning of flash to act as an application loader as well as an update mechanism for an application running on a Tiva microcontroller, utilizing either UART0, USB, or Ethernet. The capabilities of the boot loader are configured via the bl_config.h include file. For this example, the boot loader uses Ethernet to load an application.

The configuration is set to boot applications which are linked to run from address 0x4000 in flash. This is minimal address since the flash erase size is 16K bytes.

Please note that LMFlash programmer version number needs to be at least 1588 or later. Older LMFlash programmer doesn't work with this Ethernet boot loader.

2.17 Calibration for the Touch Screen (calibrate)

The touch screen driver (../drivers/touch.c) has default parameters, represented by the array g_pi32TouchParameters, to align the touch screen's cooordinates to the display coordinates. But these parameters might need calibration to achieve a better alignment, of the touch and display coordiantes.

The raw sample interface of the touch screen driver is used to compute the calibration matrix required to convert raw samples into screen X/Y positions. The produced calibration matrix can be inserted into the touch screen driver to map the raw samples into screen coordinates.

The touch screen calibration is performed according to the algorithm described by Carlos E. Videles in the June 2002 issue of Embedded Systems Design. It can be found online at `http://www.embedded.com/story/OEG20020529S0046`.

2.18 CRC-32 Demo (crc32)

Simple demo showing an CRC-32 operation using the CCM module.

Please note that the use of uDMA is not required for the operation of the CRC module. It is for demostration purposes only.

2.19 Ethernet-based I/O Control (enet_io)

This example application demonstrates web-based I/O control using the Tiva Ethernet controller and the lwIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, a static IP address will be chosen using AutoIP. The address that is selected will be shown on the QVGA display allowing you to access the internal web pages served by the application via your normal web browser.

Two different methods of controlling board peripherals via web pages are illustrated via pages labeled "IO Control Demo 1" and "IO Control Demo 2" in the navigation menu on the left of the application's home page. In both cases, the example allows you to toggle the state of the user LED on the board and set the speed of a graphical animation on the display.

"IO Control Demo 1" uses JavaScript running in the web browser to send HTTP requests for particular special URLs. These special URLs are intercepted in the file system support layer (io_fs.c) and used to control the LED and animation. Responses generated by the board are returned to the browser and inserted into the page HTML dynamically by more JavaScript code.

"IO Control Demo 2" uses standard HTML forms to pass parameters to CGI (Common Gateway Interface) handlers running on the board. These handlers process the form data and control the animation and LED as requested before sending a response page (in this case, the original form) back to the browser. The application registers the names and handlers for each of its CGIs with the HTTPD server during initialization and the server calls these handlers after parsing URL parameters each time one of the CGI URLs is requested.

Information on the state of the various controls in the second demo is inserted into the served HTML using SSI (Server Side Include) tags which are parsed by the HTTPD server in the application. As with the CGI handlers, the application registers its list of SSI tags and a handler function with the web server during initialization and this handler is called whenever any registered tag is found in a .shtml, .ssi, .shtm or .xml file being served to the browser.

In addition to LED and animation speed control, the second example also allows a line of text to be sent to the board for display on the LCD panel. This is included to illustrate the decoding of HTTP text strings.

Note that the web server used by this example has been modified from the example shipped with the basic lwIP package. Additions include SSI and CGI support along with the ability to have the server automatically insert the HTTP headers rather than having these built in to the files in the file system image.

Source files for the internal file system image can be found in the “fs” directory. If any of these files are changed, the file system image (io_fsdata.h) should be rebuilt by running the following command from the enet_io directory:

../../../../tools/bin/makefsfile -i fs -o io_fsdata.h -r -h -q

For additional details on lwIP, refer to the lwIP web page at: `http://savannah.nongnu.org/projects/lwip/`

2.20 Ethernet with lwIP (enet_lwip)

This example application demonstrates the operation of the Tiva Ethernet controller using the lwIP TCP/IP Stack configured to operate as an HTTP file server (web server). DHCP is used to obtain an Ethernet address. If DHCP times out without obtaining an address, AUTOIP will be used to obtain a link-local address. The address that is selected will be shown on the QVGA display.

The file system code will first check to see if an SD card has been plugged into the microSD slot. If so, all file requests from the web server will be directed to the SD card. Otherwise, a default set of pages served up by an internal file system will be used. Source files for the internal file system image can be found in the “fs” directory. If any of these files are changed, the file system image (enet_fsdata.h) should be rebuilt using the command:

../../../../tools/bin/makefsfile -i fs -o enet_fsdata.h -r -h -q

For additional details on lwIP, refer to the lwIP web page at: `http://savannah.nongnu.org/projects/lwip/`

2.21 Ethernet with uIP (enet_uip)

This example application demonstrates the operation of the Tiva C Series Ethernet controller using the uIP TCP/IP Stack. DHCP is used to obtain an Ethernet address. A basic web site is served over the Ethernet port. The web site displays a few lines of text, and a counter that increments each time the page is sent.

UART0, connected to the ICDI virtual COM port and running at 115,200, 8-N-1, is used to display messages from this application.

For additional details on uIP, refer to the uIP web page at: `http://www.sics.se/~adam/uip/`

2.22 Font Viewer (fontview)

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.

2.23 GPIO JTAG Recovery (gpio_jtag)

This example demonstrates changing the JTAG pins into GPIOs, along with a mechanism to revert them to JTAG pins. When first run, the pins remain in JTAG mode. Pressing the touchscreen will toggle the pins between JTAG and GPIO modes.

In this example, four pins (PC0, PC1, PC2, and PC3) are switched.

2.24 Graphics Library Demonstration (grlib_demo)

This application provides a demonstration of the capabilities of the TivaWare Graphics Library. A series of panels show different features of the library. For each panel, the bottom provides a forward and back button (when appropriate), along with a brief description of the contents of the panel.

The first panel provides some introductory text and basic instructions for operation of the application.

The second panel shows the available drawing primitives: lines, circles, rectangles, strings, and images.

The third panel shows the canvas widget, which provides a general drawing surface within the widget hierarchy. A text, image, and application-drawn canvas are displayed.

The fourth panel shows the check box widget, which provides a means of toggling the state of an item. Three check boxes are provided, with each having a red "LED" to the right. The state of the LED tracks the state of the check box via an application callback.

The fifth panel shows the container widget, which provides a grouping construct typically used for radio buttons. Containers with a title, a centered title, and no title are displayed.

The sixth panel shows the push button widget. Two rows of push buttons are provided; the appearance of each row is the same but the top row does not utilize auto-repeat while the bottom row does. Each push button has a red "LED" beneath it, which is toggled via an application callback each time the push button is pressed. While holding down any of auto-repeat buttons, the "LED" for that button should be toggled as long as the button is being held down.

The seventh panel shows the radio button widget. Two groups of radio buttons are displayed, the first using text and the second using images for the selection value. Each radio button has a red "LED" to its right, which tracks the selection state of the radio buttons via an application callback. Only one radio button from each group can be selected at a time, though the radio buttons in each group operate independently.

The eighth and final panel shows the slider widget. Six sliders constructed using the various supported style options are shown. The slider value callback is used to update two widgets to reflect the values reported by sliders. A canvas widget near the top right of the display tracks the value of the red and green image-based slider to its left and the text of the grey slider on the left side of the panel is update to show its own value. The slider on the right is configured as an indicator which tracks the state of the upper slider and ignores user input.

2.25 Graphics Driver Test Tool (grlib_driver_test)

This application provides a simple, command-line tool to aid in debugging TivaWare Graphics Library display drivers. As shipped in TivaWare, it is configured to operate with a DK-TM4C129X board using its QVGA display. The code is written, however, to allow easy retargeting to other boards and display drivers via modifications to the driver_config.h header file.

The tool is driven via a command line interface provided on UART0. Use a terminal emulator on your host system and set the serial port to use 115200bps, 8-N-1. To see a list of supported commands, enter “help” on the command line and to see extended help on a specific command enter “help [command]”.

The commands in the tool fall broadly into three categories:

- commands allowing a given low level graphics function to be executed with parameters provided by the user,
- commands providing the ability to read and write arbitrary memory locations and registers, and
- tests displaying test patterns intended to exercise specific display driver functions.

The first group of commands includes “r” to read a word from a memory location or register, “w” to write a word to a memory location or register, “dump” to dump a range of memory as words and “db” to dump a range of memory as bytes. Note that no checking is performed to ensure that addresses passed to these functions are valid. If an invalid address is passed, the test tool will fault.

The second group of commands contains “fill” which fills the screen with a given color, “rect” which draws a rectangle outline at a given position on the display, ”hline“ which draws a horizontal line, “vline” which draws a vertical line, “image” which draws an image at provided coordinates, and “text” which renders a given text string. The output of these commands is also modified via several other commands. “fg” selects the foreground color to be used by the drawing commands and “bg” selects the background color. “setimg” selects from one of four different test images that are drawn in response to the “image” command and “clipimg” allows image clipping to be adjusted to test handling of the *i32X0* parameter to the driver’s PixelDrawMultiple function.

Additional graphics commands are “pat” which redraws the test pattern displayed when the tool starts, “colbar” which fills the display with a set of color bars and “perf” which draws randomly positioned and colored rectangles for a given number of seconds and determines the drawing speed in pixels-per-second.

All driver function test patterns are generated using the “test” command whose first parameter indicates the test to display. Tests are as follow:

- “color” tests the driver’s color handling. The test starts by splitting the screen into two and showing a different primary or secondary color in each half. Verify that the correct colors are

displayed. After this, red, blue and green color gradients are displayed. Again, verify that these are correct and that no color other than the shades of the specific primary are displayed. If any color is incorrect, this likely indicates an error in the driver's pfnColorTranslate function or the function used to set the display palette if the driver provides this feature.

- "pixel" tests basic pixel plotting. A test pattern is drawn with a single white pixel in each corner of the display, a small cross comprising 5 white pixels in the center, and small arrows near each corner. If any of the corner dots are missing or any of the other pattern elements are incorrect, this points to a problem in the driver's pfnPixelDraw function or, more generally, a problem with the display coordinate space handling.

- "hline" tests horizontal line drawing. White horizontal lines are drawn at the top and bottom and a right-angled triangle is constructed in the center of the display. If any line is missing or the triangle is incorrect, this points to a problem in the driver's pfnLineDrawH function.

- "vline" tests vertical line drawing. White vertical lines are drawn at the left and right and a right-angled triangle is constructed in the center of the display. If any line is missing or the triangle is incorrect, this points to a problem in the driver's pfnLineDrawV function.

- "mult" tests the driver's pfnPixelDrawMultiple function. This is the most complex driver function and the one most prone to errors. The tool fills the display with each of the included test images in turn. These cover all the pixel formats (1-, 4- and 8-bpp) that the driver is required to handle and the image clipping and x positions are set to ensure that all alignment cases are handled for each format. In each case, the image is drawn inside a single pixel red rectangle. If the driver is handling each case correctly, the image should look correct and no part of the red rectangle should be overwritten when the image is drawn. In the displayed grid of images, the x alignment increases from 1 to 8 across the display and each line increases the left-side image clipping by one pixel from 0 in the top row to 7 in the bottom row. An error in any image indicates that one of the cases handled by the driver's pfnPixelDrawMultiple function is not handled correctly.

2.26 Hello World (hello)

A very simple "hello world" example. It simply displays "Hello World!" on the display and is a starting point for more complicated applications. This example uses calls to the TivaWare Graphics Library graphics primitives functions to update the display. For a similar example using widgets, please see "hello_widget".

2.27 Hello using Widgets (hello_widget)

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.

2.28 Hibernate Example (hibernate)

An example to demonstrate the use of the Hibernation module. The user can put the microcontroller in hibernation by touching the display. The microcontroller will then wake on its own after 5 seconds, or immediately if the user presses the RESET button. External WAKE pin and GPIO (PK5) wake sources can also be used to wake immediately from hibernation. The following wiring enables the use of these pins as wake sources. WAKE on J27 to SEL on J37 PK5 on J28 to UP on J37

The program keeps a count of the number of times it has entered hibernation. The value of the counter is stored in the battery backed memory of the Hibernation module so that it can be retrieved when the microcontroller wakes. The program displays the wall time and date by making use of the calendar function of the Hibernate module. User can modify the date and time if so desired.

2.29 Interrupts (interrupts)

This example application demonstrates the interrupt preemption and tail-chaining capabilities of Cortex-M4 microprocessor and NVIC. Nested interrupts are synthesized when the interrupts have the same priority, increasing priorities, and decreasing priorities. With increasing priorities, preemption will occur; in the other two cases tail-chaining will occur. The currently pending interrupts and the currently executing interrupt will be displayed on the display; GPIO pins B3, L1 and L0 (the GPIO on jumper J27 on the left edge of the board) will be asserted upon interrupt handler entry and de-asserted before interrupt handler exit so that the off-to-on time can be observed with a scope or logic analyzer to see the speed of tail-chaining (for the two cases where tail-chaining is occurring).

2.30 Graphics Library String Table Demonstration (lang_demo)

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 first panel provides a large string with introductory text and basic instructions for operation of the application.

The second panel shows the available languages and allows them to be switched between English, German, Spanish and Italian.

The string table and custom fonts used by this application can be found under /third_party/fonts/lang_demo. The original strings that the application intends displaying are found 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 which has been remapped to a custom codepage allowing the table to be reduced 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 third_party/fonts/lang_demo.

By default, the application builds to use the custom codepage version of the string table and its matching custom font. To build using the UTF8 string table and Unicode-indexed custom font, ensure that the definition of **USE_REMAPPED_STRINGS** at the top of the lang_demo.c source file is commented out.

2.31 MPU (mpu_fault)

This example application demonstrates the use of the MPU to protect a region of memory from access, and to generate a memory management fault when there is an access violation.

2.32 Project Zero (project0)

This example demonstrates the use of TivaWare to setup the clocks and toggle GPIO pin to make the LED blink. This is a good place to start understanding your launchpad and the tools that can be used to program it.

2.33 Quickstart Weather Application (qs_weather)

This example application demonstrates the operation of the Tiva C series evaluation kit as a weather reporting application.

The application supports updating weather information from Open Weather Map weather provider(`http://openweathermap.org/`). The application uses the lwIP stack to obtain an address through DNS, resolve the address of the Open Weather Map site and then build and handle all of the requests necessary to access the weather information. The application can also use a web proxy, allows for a custom city to be added to the list of cities and toggles temperature units from Celsius to Fahrenheit. The application uses gestures to navigate between various screens. To open the settings screen just press and drag down on any city screen. To exit the setting screen press and drag up and you are returned to the main city display. To navigate between cities, press and drag left or right and the new city information is displayed.

For additional details about Open Weather Map refer to their web page at: `http://openweathermap.org/`

For additional details on lwIP, refer to the lwIP web page at: `http://savannah.nongnu.org/projects/lwip/`

2.34 Scribble Pad (scribble)

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.

2.35 SD card using FAT file system (sd_card)

This example application demonstrates reading a file system from an SD card. It makes use of FatFs, a FAT file system driver. It provides a simple widget-based console on the display and also a UART-based command line for viewing and navigating the file system on the SD card.

For additional details about FatFs, see the following site: `http://elm-chan.org/fsw/ff/00index_e.html`

The application may also be operated via a serial terminal attached to UART0. The RS232 communication parameters should be set to 115,200 bits per second, and 8-n-1 mode. When the program is started a message will be printed to the terminal. Type “help” for command help.

2.36 SHA1 Hash Demo (sha1_hash)

Simple example showing SHA1 hash generation using a block of random data.

Please note that the use of uDMA is not required for the operation of the SHA module. It is for demostration purposes only.

2.37 SHA1 HMAC Demo (sha1_hmac)

Simple example showing SHA1 HMAC generation using a block of random data.

2.38 Synthesizer (synth)

This application provides a single-octave synthesizer utilizing the touch screen as a virtual piano keyboard. The notes played on the virtual piano are played out via the on-board speaker.

2.39 Tamper (tamper)

An example to demonstrate the use of tamper function in Hibernate module. The user can ground any of these four GPIO pins(PM4, PM5, PM6, PM7 on J28 and J30 headers on the development kit) to manually triggle tamper events(s). The red indicators on the top of display should reflect on which pin has triggered a tamper event. The event along with the time stamp will be printed on the display. The user can put the system in hibernation by pressing the HIB button. The system should wake when the user either press RESET button, or ground any of the four pins to trigger tamper event(s). When the system boots up, the display should show whether the system woke from hibernation or booted up from POR in which case a description of howto instruction is printed on the display.

The RTC clock is displayed on the bottom of the display, the clock starts from August 1st, 2013 at midnight when the app starts. The date and time can be changed by pressing the CLOCK button. The clock is update every second using hibernate calendar match interrupt. When the system is in hibernation, the clock update on the display is paused, it resumes once the system wakes up from hibernation.

WARNING: XOSC failure is implemented in this example code, care must be taken to ensure that the XOSCn pin(Y3) is properly grounded in order to safely generate the external oscillator failure without damaging the external oscillator. XOSCFAIL can be triggered as a tamper event, as well as wakeup event from hibernation.

2.40 TDES CBC Decryption Demo (tdes_cbc_decrypt)

Simple demo showing an decryption operation using the DES module with triple DES in CBC mode. A single block of data is decrypted at a time. The module is also capable of performing in DES mode, but this has been proven to be cryptographically insecure. CBC mode is also not recommended because it will always produce the same ciphertext for a block of plaintext. CBC and CFB modes are recommended instead.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.41 TDES CBC Encryption Demo (tdes_cbc_encrypt)

Simple demo showing an encryption operation using the DES module with triple DES in CBC mode. A single block of data is encrypted at a time. The module is also capable of performing in DES mode, but this has been proven to be cryptographically insecure. CBC mode is also not recommended because it will always produce the same ciphertext for a block of plaintext. CBC and CFB modes are recommended instead.

Please note that the use of interrupts and uDMA is not required for the operation of the module. It is only done for demonstration purposes.

2.42 Timer (timers)

This example application demonstrates the use of the timers to generate periodic interrupts. One timer is set up to interrupt once per second and the other to interrupt twice per second; each interrupt handler will toggle its own indicator on the display.

2.43 UART Echo (uart_echo)

This example application utilizes the UART to echo text. The first UART (connected to the FTDI virtual serial port on the evaluation board) will be configured in 115,200 baud, 8-n-1 mode. All characters received on the UART are transmitted back to the UART.

2.44 uDMA (udma_demo)

This example application demonstrates the use of the uDMA controller to transfer data between memory buffers, and to transfer data to and from a UART. The test runs for 10 seconds before exiting.

2.45 USB Generic Bulk Device (usb_dev_bulk)

This example provides a generic USB device offering simple bulk data transfer to and from the host. The device uses a vendor-specific class ID and supports a single bulk IN endpoint and a single bulk OUT endpoint. Data received from the host is assumed to be ASCII text and it is echoed back with the case of all alphabetic characters swapped.

Assuming you installed TivaWare in the default directory, a driver information (INF) file for use with Windows XP, Windows Vista, Windows 7, and Windows 10 can be found in C:/TivaWare_C_Series-x.x/windows_drivers. For Windows 2000, the required INF file is in C:/TivaWare_C_Series-x.x/windows_drivers/win2K.

A sample Windows command-line application, usb_bulk_example, illustrating how to connect to and communicate with the bulk device is also provided. The application binary is installed as part of the "Windows-side examples for USB kits" package (SW-USB-win) on the installation CD or via download from `http://www.ti.com/tivaware` . Project files are included to allow the examples to be built using Microsoft Visual Studio 2008. Source code for this application can be found in directory ti/TivaWare-for-C-Series/tools/usb_bulk_example.

2.46 USB Composite HID Keyboard Mouse Device (usb_dev_chid)

This example application turns the evaluation board into a composite USB keyboard and mouse example using the Human Interface Device class. The color LCD displays a blank area which acts as a mouse touchpad. The button on the bottom of the screen acts as a toggle between keyboard and mouse mode. Pressing it toggles the screen to keyboard mode and allows keys to be sent to the USB host. The board status LED is used to indicate the current Caps Lock state and is updated in response to pressing the "Caps" key on the virtual keyboard or any other keyboard attached to the same USB host system.

2.47 USB HID Gamepad Device (usb_dev_gamepad)

This example application turns the evaluation board into USB game pad device using the Human Interface Device gamepad class. The buttons on the board are reported as buttons 1, 2, and 3. The X and Y coordinates are reported using the touch screen input. This example also demonstrates how to use a custom HID report descriptor which is specified in the usb_game_structs.c in the g_pui8GameReportDescriptor structure.

2.48 USB HID Keyboard Device (usb_dev_keyboard)

This example application turns the evaluation board into a USB keyboard supporting the Human Interface Device class. The color LCD display shows a virtual keyboard and taps on the touchscreen will send appropriate key usage codes back to the USB host. Modifier keys (Shift, Ctrl and Alt) are "sticky" and tapping them toggles their state. The board status LED is used to indicate the current Caps Lock state and is updated in response to pressing the "Caps" key on the virtual keyboard or any other keyboard attached to the same USB host system.

The device implemented by this application also supports USB remote wakeup allowing it to request the host to reactivate a suspended bus. If the bus is suspended (as indicated on the application display), touching the display will request a remote wakeup assuming the host has not specifically disabled such requests.

2.49 USB MSC Device (usb_dev_msc)

This example application turns the evaluation board into a USB mass storage class device. The application will use the onboard flash memory for the storage media for the mass storage device. The screen will display the current action occurring on the device ranging from disconnected, reading, writing and idle.

2.50 USB Serial Device (usb_dev_serial)

This example application turns the development kit into a virtual serial port when connected to the USB host system. The application supports the USB Communication Device Class, Abstract Control Model to redirect UART0 traffic to and from the USB host system.

The application can be recompiled to run using and external USB phy to implement a high speed device using an external USB phy. To use the external phy the application must be built with **USE_ULPI** defined. This disables the internal phy and the connector on the DK-TM4C129X board and enables the connections to the external ULPI phy pins on the DK-TM4C129X board.

Assuming you installed TivaWare in the default directory, a driver information (INF) file for use with Windows XP, Windows Vista, Windows 7, and Windows 10 can be found in C:/TivaWare_C_Series-x.x/windows_drivers. For Windows 2000, the required INF file is in C:/TivaWare_C_Series-x.x/windows_drivers/win2K.

2.51 USB host audio example application using SD Card FAT file system (usb_host_audio)

This example application demonstrates playing .wav files from an SD card that is formatted with a FAT file system using USB host audio class. The application can browse the file system on the SD card and displays all files that are found. Files can be selected to show their format and then played if the application determines that they are a valid .wav file. Only PCM format (uncompressed) files may be played.

For additional details about FatFs, see the following site: `http://elm-chan.org/fsw/ff/00index_e.html`

The application can be recompiled to run using and external USB phy to implement a high speed host using an external USB phy. To use the external phy the application must be built with **USE_ULPI** defined. This disables the internal phy and the connector on the DK-TM4C129X board and enables the connections to the external ULPI phy pins on the DK-TM4C129X board.

2.52 USB host audio example application using a USB audio device for input and PWM audio to the on board speaker for output. (usb_host_audio_in)

This example application demonstrates streaming audio from a USB audio device that supports recording an audio source at 48000 16 bit stereo. The application starts recording audio from the USB audio device when the "Record" button is pressed and stream it to the speaker on the board. Because some audio devices require more power, you may need to use an external 5 volt supply to provide enough power to the USB audio device.

The application can be recompiled to run using and external USB phy to implement a high speed host using an external USB phy. To use the external phy the application must be built with **USE_ULPI** defined. This disables the internal phy and the connector on the DK-TM4C129X board and enables the connections to the external ULPI phy pins on the DK-TM4C129X board.

2.53 USB HUB Host example(usb_host_hub)

This example application demonstrates how to support a USB keyboard and a USB mass storage with a USB Hub. The application emulates a very simple console with the USB keyboard used for input. The application requires that the mass storage device is also inserted or the console will generate errors when accessing the file system. The console supports the following commands: "ls", "cat", "pwd", "cd" and "help". The "ls" command will provide a listing of the files in the current directory. The "cat" command can be used to print the contents of a file to the screen. The "pwd" command displays the current working directory. The "cd" command allows the application to move to a new directory. The cd command is simplified and only supports "cd .." but not directory changes like "cd ../somedir". The "help" command has other aliases that are displayed when the "help" command is issued.

Any keyboard that supports the USB HID BIOS protocol should work with this demo application.

The application can be recompiled to run using and external USB phy to implement a high speed host using an external USB phy. To use the external phy the application must be built with **USE_ULPI** defined. This disables the internal phy and the connector on the DK-TM4C129X board and enables the connections to the external ULPI phy pins on the DK-TM4C129X board.

2.54 USB Host keyboard example(usb_host_keyboard)

This example application demonstrates how to support a USB keyboard using the DK-TM4C129X development board. This application supports only a standard keyboard HID device, but can report on the types of other devices that are connected without having the ability to access them. Key presses are shown on the display as well as the caps-lock, scroll-lock, and num-lock states of the keyboard. The bottom left status bar reports the type of device attached. The user interface for the application is handled in the keyboard_ui.c file while the usb_host_keyboard.c file handles start up and the USB interface.

The application can be recompiled to run using and external USB phy to implement a high speed host using an external USB phy. To use the external phy the application must be built with **USE_ULPI** defined. This disables the internal phy and the connector on the DK-TM4C129X board and enables the connections to the external ULPI phy pins on the DK-TM4C129X board.

2.55 USB Host mouse example(usb_host_mouse)

This example application demonstrates how to support a USB mouse using the DK-TM4C129X development board. This application supports only a standard mouse HID device, but can report on the types of other devices that are connected without having the ability to access them. The bottom left status bar reports the type of device attached. The user interface for the application is handled in the mouse_ui.c file while the usb_host_mouse.c file handles start up and the USB interface.

The application can be recompiled to run using and external USB phy to implement a high speed host using an external USB phy. To use the external phy the application must be built with **USE_ULPI** defined. This disables the internal phy and the connector on the DK-TM4C129X board and enables the connections to the external ULPI phy pins on the DK-TM4C129X board.

2.56 USB Mass Storage Class Host Example (usb_host_msc)

This example application demonstrates reading a file system from a USB flash disk. It makes use of FatFs, a FAT file system driver. It provides a simple widget-based interface on the display for viewing and navigating the file system on the flash disk.

For additional details about FatFs, see the following site: `http://elm-chan.org/fsw/ff/00index_e.html`

The application can be recompiled to run using and external USB phy to implement a high speed host using an external USB phy. To use the external phy the application must be built with **USE_ULPI** defined. This disables the internal phy and the connector on the DK-TM4C129X board and enables the connections to the external ULPI phy pins on the DK-TM4C129X board.

2.57 USB OTG HID Mouse Example (usb_otg_mouse)

This example application demonstrates the use of USB On-The-Go (OTG) to offer both USB host and device operation. When the DK board is connected to a USB host, it acts as a BIOS-compatible USB mouse. The select button on the board (on the bottom right corner) acts as mouse button 1 and the mouse pointer may be moved by dragging your finger or a stylus across the touchscreen in the desired direction.

If a USB mouse is connected to the USB OTG port, the board operates as a USB host and draws dots on the display to track the mouse movement. The states of up to three mouse buttons are shown at the bottom right of the display.

2.58 USB Stick Update Demo (usb_stick_demo)

An example to demonstrate the use of the flash-based USB stick update program. This example is meant to be loaded into flash memory from a USB memory stick, using the USB stick update program (usb_stick_update), running on the microcontroller.

After this program is built, the binary file (usb_stick_demo.bin), should be renamed to the filename expected by usb_stick_update ("FIRMWARE.BIN" by default) and copied to the root directory of a USB memory stick. Then, when the memory stick is plugged into the eval board that is running the usb_stick_update program, this example program will be loaded into flash and then run on the microcontroller.

This program simply displays a message on the screen and prompts the user to press the select button. Once the button is pressed, control is passed back to the usb_stick_update program which is still is flash, and it will attempt to load another program from the memory stick. This shows how a user application can force a new firmware update from the memory stick.

2.59 USB Memory Stick Updater (usb_stick_update)

This example application behaves the same way as a boot loader. It resides at the beginning of flash, and will read a binary file from a USB memory stick and program it into another location in flash. Once the user application has been programmed into flash, this program will always start the user application until requested to load a new application.

When this application starts, if there is a user application already in flash (at **APP_START_ADDRESS**), then it will just run the user application. It will attempt to load a new application from a USB memory stick under the following conditions:

- no user application is present at **APP_START_ADDRESS**
- the user application has requested an update by transferring control to the updater
- the user holds down the eval board push button when the board is reset

When this application is attempting to perform an update, it will wait forever for a USB memory stick to be plugged in. Once a USB memory stick is found, it will search the root directory for a specific file name, which is *FIRMWARE.BIN* by default. This file must be a binary image of the program you want to load (the .bin file), linked to run from the correct address, at **APP_START_ADDRESS**.

The USB memory stick must be formatted as a FAT16 or FAT32 file system (the normal case), and the binary file must be located in the root directory. Other files can exist on the memory stick but they will be ignored.

2.60 Watchdog (watchdog)

This example application demonstrates the use of the watchdog as a simple heartbeat for the system. If a watchdog is not periodically fed, it will reset the system. The GREEN LED will blink once every second to show every time watchdog 0 is being fed, the Amber LED will blink once every second to indicate watchdog 1 is being fed. To stop the watchdog being fed and, hence, cause a system reset, tap the left screen to starve the watchdog 0, and right screen to starve the watchdog 1.

The counters on the screen show the number of interrupts that each watchdog served; the count wraps at 255. Since the two watchdogs run in different clock domains, the counters will get out of sync over time.

3 Buttons Driver

3.1 Introduction

The buttons driver provides functions to make it easy to use the push buttons on this evaluation board. The driver provides a function to initialize all the hardware required for the buttons, and features for debouncing and querying the button state.

This driver is located in `examples/boards/dk-tm4c1294x/drivers`, with `buttons.c` containing the source code and `buttons.h` containing the API declarations for use by applications.

3.2 API Functions

Functions

- void ButtonsInit (uint8_t ui8Buttons)
- uint8_t ButtonsPoll (uint8_t *pui8Delta, uint8_t *pui8RawState)

3.2.1 Function Documentation

3.2.1.1 ButtonsInit

Initializes the GPIO pins used by the board pushbuttons.

Prototype:

```
void
ButtonsInit(uint8_t ui8Buttons)
```

Parameters:

ui8Buttons is the logical OR of the buttons to initialize.

Description:

This function must be called during application initialization to configure the GPIO pins to which the pushbuttons are attached. It enables the port used by the buttons and configures each button GPIO as an input with a weak pull-up. The *ui8Buttons* value must be a logical OR combination of the following three buttons on the board: **UP_BUTTON**, **DOWN_BUTTON**, or **SELECT_BUTTON**.

Returns:

None.

3.2.1.2 ButtonsPoll

Polls the current state of the buttons and determines which have changed.

Prototype:

```
uint8_t
ButtonsPoll(uint8_t *pui8Delta,
            uint8_t *pui8RawState)
```

Parameters:

pui8Delta points to a character that will be written to indicate which button states changed since the last time this function was called. This value is derived from the debounced state of the buttons.

pui8RawState points to a location where the raw button state will be stored.

Description:

This function should be called periodically by the application to poll the pushbuttons. It determines both the current debounced state of the buttons and also which buttons have changed state since the last time the function was called.

In order for button debouncing to work properly, this function should be caled at a regular interval, even if the state of the buttons is not needed that often.

If button debouncing is not required, the the caller can pass a pointer for the *pui8RawState* parameter in order to get the raw state of the buttons. The value returned in *pui8RawState* will be a bit mask where a 1 indicates the buttons is pressed.

Returns:

Returns the current debounced state of the buttons where a 1 in the button ID's position indicates that the button is pressed and a 0 indicates that it is released.

3.3 Programming Example

The following example shows how to use the buttons driver to initialize the buttons, debounce and read the buttons state.

```
//
// Map Up button to the GPIO Pin 3 of the button port.
//
#define UP_BUTTON             GPIO_PIN_3

//
// The button example
//
void
ButtonExample(void)
{
    uint8_t ui8ButtonChange, ui8ButtonState;

    //
    // Initialize the Up button.
    //
    ButtonsInit(UP_BUTTON);

    //
```

```
    // From timed processing loop (for example every 10 ms)
    //
    {
        //
        // Poll the buttons.  When called periodically this function will
        // run the button debouncing algorithm.
        //
                ButtonsPoll(&ui8ButtonChange, &ui8ButtonState);

        //
        // Test to see if the Up button was pressed and do something
        //
        if((ui8ButtonChange & UP_BUTTON) && (ui8ButtonState & UP_BUTTON))
        {
            //
            // TODO: Up button action code
            //
        }
    }
}
```

4 Frame Module

4.1 Introduction

The frame module is a common function for drawing an application frame on the display. This is used by the example applications to provide a uniform appearance.

This driver is located in `examples/boards/dk-tm4c129x/drivers`, with `frame.c` containing the source code and `frame.h` containing the API declarations for use by applications.

4.2 API Functions

Functions

- void FrameDraw (tContext *psContext, const char *pcAppName)

4.2.1 Function Documentation

4.2.1.1 FrameDraw

Draws a frame on the LCD with the application name in the title bar.

Prototype:

```
void
FrameDraw(tContext *psContext,
          const char *pcAppName)
```

Parameters:

psContext is a pointer to the graphics library context used to draw the application frame.

pcAppName is a pointer to a string that contains the name of the application.

Description:

This function draws an application frame onto the LCD, using the supplied graphics library context to access the LCD and the given name in the title bar of the application frame.

Returns:

None.

4.3 Programming Example

The following example shows how to draw the application frame.

```
//
// The frame example.
//
void
FrameExample(void)
{
    tContext sContext;

    //
    // Draw the application frame.  This code assumes the the graphics library
    // context has already been initialized.
    //
    FrameDraw(&sContext, "example");
}
```

5 Kentec 320x240x16 Display Driver

5.1 Introduction

The display driver offers a standard interface to access display functions on the Kentec K350QVG-V2-F 320x240 16-bit color TFT display and is used by the TivaWare Graphics Library and widget manager. The display is controlled by the embedded SSD2119 display controller, which provides the frame buffer for the display. In addition to providing the `tDisplay` structure required by the graphics library, the display driver also provides an API for initializing the display.

The display driver can be built to operate in one of four orientations:

- **LANDSCAPE** - In this orientation, the screen is wider than it is tall; this is the normal orientation for a television or a computer monitor, and is the normal orientation for photographs of the outdoors (hence the name). For the K350QVG-V2-F, the flex connector is on the bottom side of the screen when viewed in **LANDSCAPE** orientation.
- **PORTRAIT** - In this orientation, the screen is taller than it is wide; this is the normal orientation of photographs of people (hence the name). For the K350QVG-V2-F, the flex connector is on the left side of the screen when viewed in **PORTRAIT** orientation.
- **LANDSCAPE_FLIP** - **LANDSCAPE** mode rotated 180 degrees (in other words, the flex connector is on the top side of the screen).
- **PORTRAIT_FLIP** - **PORTRAIT** mode rotated 180 degrees (in other words, the flex connector is on the right side of the screen).

One of the above highlighed defines selects the orientation that the display driver will use. If none is defined, the default orientation is **LANDSCAPE_FLIP** (which corresponds to how the display is mounted to the DK-TM4C129X development board).

This driver is located in `examples/boards/dk-tm4c129x/drivers`, with `kentec320x240x16_ssd2119.c` containing the source code and `kentec320x240x16_ssd2119.h` containing the API declarations for use by applications.

5.2 API Functions

Functions

- void Kentec320x240x16_SSD2119Init (uint32_t ui32SysClock)

Variables

- const tDisplay g_sKentec320x240x16_SSD2119

5.2.1 Function Documentation

5.2.1.1 Kentec320x240x16_SSD2119Init

Initializes the display driver.

Prototype:

```
void
Kentec320x240x16_SSD2119Init(uint32_t ui32SysClock)
```

Parameters:

ui32SysClock is the frequency of the system clock.

Description:

This function initializes the LCD controller and the SSD2119 display controller on the panel, preparing it to display data.

Returns:

None.

5.2.2 Variable Documentation

5.2.2.1 g_sKentec320x240x16_SSD2119

Definition:

```
const tDisplay g_sKentec320x240x16_SSD2119
```

Description:

The display structure that describes the driver for the Kentec K350QVG-V2-F TFT panel with an SSD2119 controller.

5.3 Programming Example

The following example shows how to initialize the display and prepare to draw on it using the graphics library.

```
//
// The Kentec 320x240x16 SSD2119 example.
//
void
Kentec320x240x16_SSD2119Example(void)
{
    uint32_t ui32SysClock;
    tContext sContext;
```

```
    //
    // Initialize the display.  This code assumes that ui32SysClock has been
    // set to the clock frequency of the device (for example, the value
    // returned by SysCtlClockFreqSet).
    //
    Kentec320x240x16_SSD2119Init(ui32SysClock);

    //
    // Initialize a graphics library drawing context.
    //
    GrContextInit(&sContext, &g_sKentec320x240x16_SSD2119);
}
```

6 MX66L51235F Driver

6.1 Introduction

The MX66l51235F driver provides functions to make it easy to use the MX66L51235F SPI flash on the DK-TM4C129X development board. The driver provides a function to read, erase, and program the SPI flash.

On the DK-TM4C129X development board, the SPI flash shares a SPI port with the SD card socket. If not properly initialized into SPI mode, the SD card will occasionally drive data onto the SPI bus despite the fact that it is “not selected” (which is in fact valid since there is not chip select for an SD card in SD card mode). Therefore, the SD card must be properly initialized (via a call to the disk_initialize() function in the SD card driver) prior to using this driver to access the SPI flash.

This driver is located in `examples/boards/dk-tm4c129x/drivers`, with `mx66l51235f.c` containing the source code and `mx66l51235f.h` containing the API declarations for use by applications.

6.2 API Functions

Functions

- void MX66L51235FBlockErase32 (uint32_t ui32Addr)
- void MX66L51235FBlockErase64 (uint32_t ui32Addr)
- void MX66L51235FChipErase (void)
- void MX66L51235FInit (uint32_t ui32SysClock)
- void MX66L51235FPageProgram (uint32_t ui32Addr, const uint8_t *pui8Data, uint32_t ui32Count)
- void MX66L51235FRead (uint32_t ui32Addr, uint8_t *pui8Data, uint32_t ui32Count)
- void MX66L51235FSectorErase (uint32_t ui32Addr)

6.2.1 Function Documentation

6.2.1.1 MX66L51235FBlockErase32

Erases a 32 KB block of the MX66L51235F.

Prototype:

```
void
MX66L51235FBlockErase32(uint32_t ui32Addr)
```

Parameters:
ui32Addr is the address of the block to erase.

Description:
This function erases a 32 KB block of the MX66L51235F. Each 32 KB block has a 32 KB alignment; the MX66L51235F will ignore the lower 15 bits of the address provided. This function will not return until the data has be erased.

Returns:
None.

6.2.1.2 MX66L51235FBlockErase64

Erases a 64 KB block of the MX66L51235F.

Prototype:
```
void
MX66L51235FBlockErase64(uint32_t ui32Addr)
```

Parameters:
ui32Addr is the address of the block to erase.

Description:
This function erases a 64 KB block of the MX66L51235F. Each 64 KB block has a 64 KB alignment; the MX66L51235F will ignore the lower 16 bits of the address provided. This function will not return until the data has be erased.

Returns:
None.

6.2.1.3 MX66L51235FChipErase

Erases the entire MX66L51235F.

Prototype:
```
void
MX66L51235FChipErase(void)
```

Description:
This command erase the entire contents of the MX66L51235F. This takes two minutes, nominally, to complete. This function will not return until the data has be erased.

Returns:
None.

6.2.1.4 MX66L51235FInit

Initializes the MX66L51235F driver.

Prototype:

```
void
MX66L51235FInit(uint32_t ui32SysClock)
```

Parameters:

ui32SysClock is the frequency of the system clock.

Description:

This function initializes the MX66L51235F driver and SSI interface, preparing for accesses to the SPI flash device. Since the SSI interface on the DK-TM4C129X board is shared with the SD card, this must be called prior to any SPI flash access the immediately follows an SD card access.

Returns:

None.

6.2.1.5 MX66L51235FPageProgram

Programs the MX66L51235F.

Prototype:

```
void
MX66L51235FPageProgram(uint32_t ui32Addr,
                       const uint8_t *pui8Data,
                       uint32_t ui32Count)
```

Parameters:

ui32Addr is the address to be programmed.

pui8Data is a pointer to the data to be programmed.

ui32Count is the number of bytes to be programmed.

Description:

This function programs data into the MX66L51235F. This function will not return until the data has be programmed. The addresses to be programmed must not span a 256-byte boundary (in other words, "*ui32Addr* & ~255" must be the same as "(*ui32Addr* + *ui32Count*) & ~255").

Returns:

None.

6.2.1.6 MX66L51235FRead

Reads data from the MX66L51235F.

Prototype:

```
void
MX66L51235FRead(uint32_t ui32Addr,
                uint8_t *pui8Data,
                uint32_t ui32Count)
```

Parameters:

ui32Addr is the address to read.

pui8Data is a pointer to the data buffer to into which to read the data.

ui32Count is the number of bytes to read.

Description:

This function reads data from the MX66L51235F.

Returns:

None.

6.2.1.7 MX66L51235FSectorErase

Erases a 4 KB sector of the MX66L51235F.

Prototype:

```
void
MX66L51235FSectorErase(uint32_t ui32Addr)
```

Parameters:

ui32Addr is the address of the sector to erase.

Description:

This function erases a sector of the MX66L51235F. Each sector is 4 KB with a 4 KB alignment; the MX66L51235F will ignore the lower ten bits of the address provided. This function will not return until the data has be erased.

Returns:

None.

6.3 Programming Example

The following example shows how to use the MX66L51235F driver to read and write data in the SPI flash.

```
//
// A buffer to hold the data read from and written to the SPI flash.
//
uint8_t g_pui8MX66L51235FData[32];

//
// The MX66L51235F example.
//
void
MX66L51235FExample(void)
{
    uint32_t ui32SysClock;

    //
    // Initialize the SPI flash driver.  This code assumes that ui32SysClock
    // has been set to the clock frequency of the device (for example, the
    // value returned by SysCtlClockFreqSet).
    //
    MX66L51235FInit(ui32SysClock);

    //
```

```
    // Erase the first sector (4 KB) of the SPI flash.
    //
    MX66L51235FSectorErase(0);

    //
    // Program some data into the first page of the SPI flash.  This assumes
    // that the data buffer has been filled with the data to be programmed.
    //
    MX66L51235FPageProgram(0, g_pui8MX66L51235FData,
                           sizeof(g_pui8MX66L51235FData));

    //
    // Read some data from the second page of the SPI flash.
    //
    MX66L51235FRead(0x100, g_pui8MX66L51235FData,
                    sizeof(g_pui8MX66L51235FData));
}
```

7 Pinout Module

7.1 Introduction

The pinout module is a common function for configuring the device pins for use by example applications. The pins are configured into the most common usage; it is possible that some of the pins might need to be reconfigured in order to support more specialized usage.

This driver is located in `examples/boards/dk-tm4c129x/drivers`, with `pinout.c` containing the source code and `pinout.h` containing the API declarations for use by applications.

7.2 API Functions

Functions

- void PinoutSet (void)

7.2.1 Function Documentation

7.2.1.1 PinoutSet

Configures the device pins for the standard usages on the DK-TM4C129X.

Prototype:

```
void
PinoutSet(void)
```

Description:

This function enables the GPIO modules and configures the device pins for the default, standard usages on the DK-TM4C129X. Applications that require alternate configurations of the device pins can either not call this function and take full responsibility for configuring all the device pins, or can reconfigure the required device pins after calling this function.

Returns:

None.

7.3 Programming Example

The following example shows how to configure the device pins.

```
//
// The pinout example.
//
void
PinoutExample(void)
{
    //
    // Configure the device pins.
    //
    PinoutSet();
}
```

8 Sound Driver

8.1 Introduction

The sound driver provides a set of functions to stream 16-bit PCM audio data to the speaker on the DK-TM4C129X development board. The audio can be played at 8 kHz, 16 kHz, 32 kHz, or 64 kHz; in each case the data is output to the speaker at 64 kHz, and at lower playback rates the intervening samples are computed via linear interpolation (which is fast but introduces high-frequency artifacts).

The audio data is supplied via a ping-pong buffer. This is a buffer that is logically split into two halves; the “ping” half and the “pong” half. While the sound driver is playing data from one half, the application is responsible for filling the other half with new audio data. A callback from the sound driver indicates when it transitions from one half to the other, which provides the indication that one of the halves has been consumed and must be filled with new data.

The sound driver utilizes timer 5 subtimer A. The interrupt from the timer 5 subtimer A is used to process the audio stream; the SoundIntHandler() function should be called when this interrupt occurs (which is typically accomplished by placing it in the vector table in the startup code for the application).

This driver is located in `examples/boards/dk-tm4c129x/drivers`, with `sound.c` containing the source code and `sound.h` containing the API declarations for use by applications.

8.2 API Functions

Functions

- bool SoundBusy (void)
- void SoundInit (uint32_t ui32SysClock)
- void SoundIntHandler (void)
- void SoundPeriodAdjust (int32_t i32RateAdjust)
- bool SoundStart (int16_t *pi16Buffer, uint32_t ui32Length, uint32_t ui32Rate, void (*pfnCallback)(uint32_t ui32Half))
- void SoundStop (void)
- void SoundVolumeDown (int32_t i32Volume)
- void SoundVolumeSet (int32_t i32Volume)
- void SoundVolumeUp (int32_t i32Volume)

8.2.1 Function Documentation

8.2.1.1 SoundBusy

Determines if the sound driver is busy.

Prototype:

```
bool
SoundBusy(void)
```

Description:

This function determines if the sound driver is busy, either performing the startup or shutdown ramp for the speaker or playing a sound stream.

Returns:

Returns **true** if the sound driver is busy and **false** otherwise.

8.2.1.2 SoundInit

Initializes the sound driver.

Prototype:

```
void
SoundInit(uint32_t ui32SysClock)
```

Parameters:

ui32SysClock is the frequency of the system clock.

Description:

This function initializes the sound driver, preparing it to output sound data to the speaker.

The system clock should be as high as possible; lower clock rates reduces the quality of the produced sound. For the best quality sound, the system should be clocked at 120 MHz.

Note:

In order for the sound driver to function properly, the sound driver interrupt handler (SoundIntHandler()) must be installed into the vector table for the timer 5 subtimer A interrupt.

Returns:

None.

8.2.1.3 SoundIntHandler

Handles the TIMER5A interrupt.

Prototype:

```
void
SoundIntHandler(void)
```

Description:
This function responds to the TIMER5A interrupt, updating the duty cycle of the output waveform in order to produce sound. It is the application's responsibility to ensure that this function is called in response to the TIMER5A interrupt, typically by installing it in the vector table as the handler for the TIMER5A interrupt.

Returns:
None.

8.2.1.4 SoundPeriodAdjust

Make adjustments to the sample period of the PWM audio.

Prototype:
```
void
SoundPeriodAdjust(int32_t i32RateAdjust)
```

Parameters:
i32RateAdjust is a signed value of the adjustment to make to the current sample period.

Description:
This function allows the sample period to be adjusted if the application needs to make small adjustments to the playback rate of the audio. This should only be used to makke smaller adjustments to the sample rate since large changes cause distortion in the output.

Returns:
None.

8.2.1.5 SoundStart

Starts playback of a sound stream.

Prototype:
```
bool
SoundStart(int16_t *pi16Buffer,
           uint32_t ui32Length,
           uint32_t ui32Rate,
           void (*ui32Half))(uint32_t pfnCallback)
```

Parameters:
pi16Buffer is a pointer to the buffer that contains the sound to play.
ui32Length is the length of the buffer in samples. This should be a multiple of two.
ui32Rate is the sound playback rate; valid values are 8000, 16000, 32000, and 64000.
pfnCallback is the callback function that is called when either half of the sound buffer has been played.

Description:
This function starts the playback of a sound stream contained in an audio ping-pong buffer. The buffer is played repeatedly until SoundStop() is called. Playback of the sound stream begins immediately, so the buffer should be pre-filled with the initial sound data prior to calling this function.

Returns:
Returns **true** if playback was started and **false** if it could not be started (because something is already playing).

8.2.1.6 SoundStop

Stops playback of the current sound stream.

Prototype:
```
void
SoundStop(void)
```

Description:
This function immediately stops playback of the current sound stream. As a result, the output is changed directly to the mid-point, possibly resulting in a pop or click. It is then ramped down to no output, eliminating the current draw through the amplifier and speaker.

Returns:
None.

8.2.1.7 SoundVolumeDown

Decreases the volume of the sound playback.

Prototype:
```
void
SoundVolumeDown(int32_t i32Volume)
```

Parameters:
i32Volume is the amount by which to decrease the volume of the sound playback, specified as a value between 0 (for no adjustment) and 255 maximum adjustment).

Description:
This function decreases the volume of the sound playback relative to the current volume.

Returns:
None.

8.2.1.8 SoundVolumeSet

Sets the volume of the sound playback.

Prototype:
```
void
SoundVolumeSet(int32_t i32Volume)
```

Parameters:
i32Volume is the volume of the sound playback, specified as a value between 0 (for silence) and 255 (for full volume).

Description:
This function sets the volume of the sound playback. Setting the volume to 0 mutes the output, while setting the volume to 256 plays the sound stream without any volume adjustment (that is, full volume).

Returns:
None.

8.2.1.9 SoundVolumeUp

Increases the volume of the sound playback.

Prototype:
```
void
SoundVolumeUp(int32_t i32Volume)
```

Parameters:
i32Volume is the amount by which to increase the volume of the sound playback, specified as a value between 0 (for no adjustment) and 255 maximum adjustment).

Description:
This function increases the volume of the sound playback relative to the current volume.

Returns:
None.

8.3 Programming Example

The following example shows how to use the sound driver to playback a stream of 8 kHz audio data.

```
//
// The buffer used as the audio ping-pong buffer.
//
#define SOUND_NUM_SAMPLES       256
int16_t g_pi16SoundExampleBuffer[SOUND_NUM_SAMPLES];

//
// The callback function for when half of the buffer has been consumed.
//
void
SoundExampleCallback(uint32_t ui32Half)
{
    //
    // Generate new audio in the half the has just been consumed.  As this is
    // in the context of the timer interrupt, it is possible/likely that this
    // should just set a flag to trigger something else (outside of interrupt
    // context) to do the actual work.  In either case, this needs to return
    // prior to the next timer interrupt (in other words, within ~15 us).
    //
}

//
// The sound example.
```

```
//
void
SoundExample(void)
{
    uint32_t ui32SysClock;

    //
    // Initialize the sound driver.  This code assumes that ui32SysClock has
    // been set to the clock frequency of the device (for example, the value
    // returned by SysCtlClockFreqSet).
    //
    SoundInit(ui32SysClock);

    //
    // Prefill the audio buffer with the first segment of the audio to be
    // played.
    //

    //
    // Start the playback of audio.
    //
    SoundStart(g_pi16SoundExampleBuffer, SOUND_NUM_SAMPLES, 8000,
               SoundExampleCallback);
}
```

9 Touch Screen Driver

9.1 Introduction

The touch screen is a pair of resistive layers on the surface of the display. One layer has connection points at the top and bottom of the screen, and the other layer has connection points at the left and right of the screen. When the screen is touched, the two layers make contact and electricity can flow between them.

The horizontal position of a touch can be found by applying positive voltage to the right side of the horizontal layer and negative voltage to to the left side. When not driving the top and bottom of the vertical layer, the voltage potential on that layer will be proportional to the horizontal distance across the screen of the press, which can be measured with an ADC channel. By reversing these connections, the vertical position can also be measured. When the screen is not being touched, there will be no voltage on the non-powered layer.

By monitoring the voltage on each layer when the other layer is appropriately driven, touches and releases on the screen, as well as movements of the touch, can be detected and reported.

In order to read the current voltage on the two layers and also drive the appropriate voltages onto the layers, each side of each layer is connected to both a GPIO and an ADC channel. The GPIO is used to drive the node to a particular voltage, and when the GPIO is configured as an input, the corresponding ADC channel can be used to read the layer's voltage.

The touch screen is sampled every 2.5 ms, with four samples required to properly read both the X and Y position. Therefore, 100 X/Y sample pairs are captured every second.

Like the display driver, the touch screen driver operates in the same four orientations (selected in the same manner). Default calibrations are provided for using the touch screen in each orientation; the calibrate application can be used to determine new calibration values if necessary.

The touch screen driver utilizes sample sequence 3 of ADC0 and timer 5 subtimer B. The interrupt from the ADC0 sample sequence 3 is used to process the touch screen readings; the TouchScreenIntHandler() function should be called when this interrupt occurs (which is typically accomplished by placing it in the vector table in the startup code for the application).

The touch screen driver makes use of calibration parameters determined using the "calibrate" example application. The theory behind these parameters is explained by Carlos E. Videles in the June 2002 issue of Embedded Systems Design. It can be found online at `http://www.embedded.com/story/OEG20020529S0046`.

This driver is located in `examples/boards/dk-tm4c129x/drivers`, with `touch.c` containing the source code and `touch.h` containing the API declarations for use by applications.

9.2 API Functions

Functions

- void TouchScreenCallbackSet (int32_t (*pfnCallback)(uint32_t ui32Message, int32_t i32X, int32_t i32Y))
- void TouchScreenInit (uint32_t ui32SysClock)
- void TouchScreenIntHandler (void)

9.2.1 Function Documentation

9.2.1.1 TouchScreenCallbackSet

Sets the callback function for touch screen events.

Prototype:

```
void
TouchScreenCallbackSet(int32_t (*ui32Message,)(uint32_t int32_t i32X,
int32_t i32Y) pfnCallback)
```

Parameters:

pfnCallback is a pointer to the function to be called when touch screen events occur.

Description:

This function sets the address of the function to be called when touch screen events occur. The events that are recognized are the screen being touched (“pen down”), the touch position moving while the screen is touched (“pen move”), and the screen no longer being touched (“pen up”).

Returns:

None.

9.2.1.2 TouchScreenInit

Initializes the touch screen driver.

Prototype:

```
void
TouchScreenInit(uint32_t ui32SysClock)
```

Parameters:

ui32SysClock is the frequency of the system clock.

Description:

This function initializes the touch screen driver, beginning the process of reading from the touch screen. This driver uses the following hardware resources:

- ADC 0 sample sequence 3
- Timer 5 subtimer B

Returns:
None.

9.2.1.3 TouchScreenIntHandler

Handles the ADC interrupt for the touch screen.

Prototype:

```
void
TouchScreenIntHandler(void)
```

Description:
This function is called when the ADC sequence that samples the touch screen has completed its acquisition. The touch screen state machine is advanced and the acquired ADC sample is processed appropriately.

It is the responsibility of the application using the touch screen driver to ensure that this function is installed in the interrupt vector table for the ADC0 samples sequencer 3 interrupt.

Returns:
None.

9.3 Programming Example

The following example shows how to initialize the touchscreen driver and the callback function which receives notifications of touch and release events in cases where the StellarisWare Graphics Library widget manager is not being used by the application.

```
//
// The touch screen driver calls this function to report all state changes.
//
static long
TouchTestCallback(uint32_t ui32Message, int32_t i32X, int32_t i32Y)
{
    //
    // Check the message to determine what to do.
    //
    switch(ui32Message)
    {
        //
        // The screen is no longer being touched (in other words, pen/pointer
        // up).
        //
        case WIDGET_MSG_PTR_UP:
        {
            //
            // Handle the pointer up message if required.
            //
            break;
        }

        //
        // The screen has just been touched (in other words, pen/pointer down).
        //
        case WIDGET_MSG_PTR_DOWN:
```

```
            {
                //
                // Handle the pointer down message if required.
                //
                break;
            }

            //
            // The location of the touch on the screen has moved (in other words,
            // the pen/pointer has moved).
            //
            case WIDGET_MSG_PTR_MOVE:
            {
                //
                // Handle the pointer move message if required.
                //
                break;
            }

            //
            // An unknown message was received.
            //
            default:
            {
                //
                // Ignore all unknown messages.
                //
                break;
            }
        }

        //
        // Success.
        //
        return(0);
    }

    //
    // The first touch screen example.
    //
    void
    TouchScreenExample1(void)
    {
        uint32_t ui32SysClock;

        //
        // Initialize the touch screen driver.  This code assumes that ui32SysClock
        // has been set to the clock frequency of the device (for example, the
        // value returned by SysCtlClockFreqSet).
        //
        TouchScreenInit(ui32SysClock);

        //
        // Register the application callback function that is to receive touch
        // screen messages.
        //
        TouchScreenCallbackSet(TouchTestCallback);
    }
```

If using the StellarisWare Graphics Library widget manager, touchscreen initialization code is as follows. In this case, the touchscreen callback is provided within the widget manager so no additional function is required in the application code.

```
    //
```

```
// The second touch screen example.
//
void
TouchScreenExample2(void)
{
    uint32_t ui32SysClock;

    //
    // Initialize the touch screen driver.  This code assumes that ui32SysClock
    // has been set to the clock frequency of the device (for example, the
    // value returned by SysCtlClockFreqSet).
    //
    TouchScreenInit(ui32SysClock);

    //
    // Register the graphics library pointer message callback function so that
    // it receives touch screen messages.
    //
    TouchScreenCallbackSet(WidgetPointerMessage);
}
```

10 USB Sound Driver

10.1 Introduction

The USB sound driver provides a set of API's that can be used for USB host audio applications. It also provides the structure to store information about any connected audio devices.

The USB sound driver provides the required USB Callback API, an initialization API, and functions to control the sending and receiving of audio data over USB. Further API's can set the sound output and control the volume.

The file uses Pulse-Code Modulation (PCM) audio samples for sending and receiving audio data via the buffers.

This driver is located in `examples/boards/dk-tm4c129x/drivers`, with `usb_sound.c` containing the source code and `usb_sound.h` containing the API declarations for use by applications.

10.2 API Functions

Functions

- void USBHCDEvents (void *pvData)
- uint32_t USBSoundBufferIn (const void *pvBuffer, uint32_t ui32Size, tUSBBufferCallback pfnCallback)
- uint32_t USBSoundBufferOut (const void *pvBuffer, uint32_t ui32Size, tUSBBufferCallback pfnCallback)
- void USBSoundInit (uint32_t ui32Flags, tEventCallback pfnCallback)
- uint32_t USBSoundInputFormatGet (uint32_t ui32SampleRate, uint32_t ui32Bits, uint32_t ui32Channels)
- uint32_t USBSoundInputFormatSet (uint32_t ui32SampleRate, uint32_t ui32BitsPerSample, uint32_t ui32Channels)
- void USBSoundMain (void)
- uint32_t USBSoundOutputFormatGet (uint32_t ui32SampleRate, uint32_t ui32Bits, uint32_t ui32Channels)
- uint32_t USBSoundOutputFormatSet (uint32_t ui32SampleRate, uint32_t ui32BitsPerSample, uint32_t ui32Channels)
- uint32_t USBSoundVolumeGet (uint32_t ui32Channel)
- void USBSoundVolumeSet (uint32_t ui32Percent)

10.2.1 Function Documentation

10.2.1.1 USBHCDEvents

This is the generic callback from host stack.

Prototype:

```
void
USBHCDEvents(void *pvData)
```

Parameters:

pvData is actually a pointer to a tEventInfo structure.

Description:

This function is called to inform the application when a USB event has occurred that is outside those related to the audio device. At this point this is used to detect unsupported devices being inserted and removed. It is also used to inform the application when a power fault has occurred. This function is required when the g_USBGenerii8EventDriver is included in the host controller driver array that is passed in to the USBHCDRegisterDrivers() function.

Returns:

None.

10.2.1.2 USBSoundBufferIn

Requests a new block of PCM audio samples from a USB audio device.

Prototype:

```
uint32_t
USBSoundBufferIn(const void *pvBuffer,
                 uint32_t ui32Size,
                 tUSBBufferCallback pfnCallback)
```

Parameters:

pvBuffer is a pointer to a location to store the audio data.

ui32Size is the size of the pvData buffer in bytes.

pfnCallback is a function to call when this buffer has new data.

Description:

This function request a new block of PCM audio samples from a USB audio device.

Returns:

This function returns a non-zero value if the buffer was accepted, and returns zero if the buffer was not accepted.

10.2.1.3 USBSoundBufferOut

Starts output of a block of PCM audio samples.

Prototype:

```
uint32_t
USBSoundBufferOut(const void *pvBuffer,
                  uint32_t ui32Size,
                  tUSBBufferCallback pfnCallback)
```

Parameters:

pvBuffer is a pointer to the audio data to play.

ui32Size is the length of the data in bytes.

pfnCallback is a function to call when this buffer has be played.

Description:

This function starts the output of a block of PCM audio samples.

Returns:

This function returns a non-zero value if the buffer was accepted, and returns zero if the buffer was not accepted.

10.2.1.4 USBSoundInit

Initializes the sound output.

Prototype:

```
void
USBSoundInit(uint32_t ui32Flags,
             tEventCallback pfnCallback)
```

Parameters:

ui32Flags is unused as this point but is included for future functionality.

pfnCallback is the event callback function for audio devices.

Description:

This function prepares the sound driver to enumerate an audio device and prepares to play audio once a valid audio device is detected. The *pfnCallback* function can be used to receive callbacks when there are changes related to the audio device. The ui32Event parameter to the callback is one of the SOUND_EVENT_* values.

Returns:

None

10.2.1.5 USBSoundInputFormatGet

Returns the current sample rate.

Prototype:

```
uint32_t
USBSoundInputFormatGet(uint32_t ui32SampleRate,
                       uint32_t ui32Bits,
                       uint32_t ui32Channels)
```

Parameters:

ui32SampleRate is the sample rate.

ui32Bits is the number of bits per sample in the audio stream.

ui32Channels is the number of channels.

Description:

This function returns the sample rate that was set by a call to USBSoundSetFormat(). This is needed to retrieve the exact sample rate that is in use in case the requested rate could not be matched exactly.

Returns:

The current sample rate in samples/second.

10.2.1.6 USBSoundInputFormatSet

This sets the current input audio format of the USB audio device

Prototype:

```
uint32_t
USBSoundInputFormatSet(uint32_t ui32SampleRate,
                       uint32_t ui32BitsPerSample,
                       uint32_t ui32Channels)
```

Parameters:

ui32SampleRate is the sample rate.

ui32BitsPerSample is the number of bits per sample.

ui32Channels is the number of channels.

Description:

This sets the current format for the USB device that is currently connect. If there is no USB device connected or the format is not supported then the function returns 0. The function returns 1 if the USB audio device was successfully configured to the requested format.

Returns:

Returns 1 if the format was successfully set or returns 0 if the format was not changed.

10.2.1.7 USBSoundMain

The main routine for USB audio applications to call.

Prototype:

```
void
USBSoundMain(void)
```

Description:

This is the main routine for handling USB audio, and it should be called periodically by the main application.

Returns:

none

10.2.1.8 USBSoundOutputFormatGet

Returns the current sample rate.

Prototype:

```
uint32_t
USBSoundOutputFormatGet(uint32_t ui32SampleRate,
                        uint32_t ui32Bits,
                        uint32_t ui32Channels)
```

Parameters:

ui32SampleRate is the sample rate.

ui32Bits is the number of bits per sample in the audio stream.

ui32Channels is the number of channels.

Description:

This function returns the sample rate that was set by a call to USBSoundSetFormat(). This is needed to retrieve the exact sample rate that is in use in case the requested rate could not be matched exactly.

Returns:

The current sample rate in samples/second.

10.2.1.9 USBSoundOutputFormatSet

This sets the current output audio format of the USB audio device.

Prototype:

```
uint32_t
USBSoundOutputFormatSet(uint32_t ui32SampleRate,
                        uint32_t ui32BitsPerSample,
                        uint32_t ui32Channels)
```

Parameters:

ui32SampleRate is the sample rate.

ui32BitsPerSample is the number of bits per sample.

ui32Channels is the number of channels.

Description:

This sets the current audio format for the USB device that is currently connected. If there is no USB device connected or the format is not supported then the function returns a non-zero value. The function returns zero if the USB audio device was successfully configured to the requested audio format.

Returns:

Returns zero if the format was successfully set or returns an non-zero value if the format was not able to be set.

10.2.1.10 USBSoundVolumeGet

Returns the current volume level.

Prototype:

```
uint32_t
USBSoundVolumeGet(uint32_t ui32Channel)
```

Parameters:

ui32Channel is the 0 based channel number to query.

Description:

This function returns the current volume, specified as a percentage between 0% (silence) and 100% (full volume), inclusive. The *ui32Channel* value starts with 0 which is the master audio volume control interface. The remaining *ui32Channel* values provide access to various other audio channels, with 1 and 2 being left and right audio channels.

Returns:

Returns the current volume.

10.2.1.11 USBSoundVolumeSet

Sets the volume of the audio device.

Prototype:

```
void
USBSoundVolumeSet(uint32_t ui32Percent)
```

Parameters:

ui32Percent is the volume percentage, which must be between 0% (silence) and 100% (full volume), inclusive.

Description:

This function sets the volume of the sound output to a value between silence (0%) and full volume (100%).

Returns:

None.

10.3 Programming Example

See the usb_host_audio example in the `examples/boards/dk-tm4c129x` folder.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as "components") are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI's terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

TI assumes no liability for applications assistance or the design of Buyers' products. Buyers are responsible for their products and applications using TI components. To minimize the risks associated with Buyers' products and applications, Buyers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI components or services with statements different from or beyond the parameters stated by TI for that component or service voids all express and any implied warranties for the associated TI component or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI's goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or "enhanced plastic" are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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