Contents

Articles

References

Article Licenses

Quick Start Guide StarterWare 02.00.XX.XX (supports AM335x)

StarterWare Overview

StarterWare 02.00.XX.XX provides no-OS platform support for AM335x. The StarterWare package contains Device Abstraction Layer libraries and peripheral/board level sample/demo examples that demonstrate the capabilities of the peripherals on AM335x.

Running The Demo Application

For those who want a quick look on the StarterWare deliverable for AM335x, described below, are the steps to load and run the StarterWare bootloader and a system level demo application from an SD card.

- **Set Up Requirements**
  - For beaglebone [1] board
    - The mini USB port (Connector P3) has to be connected to the host. Please make sure that the Virtual COM Port driver is installed. The driver is available here [2] for downloading. This mini USB-to-serial connection is used for displaying messages on the serial console on the host, if the port is properly selected.
    - A serial terminal application (like Tera Term / HyperTerminal / Minicom) should be running on the host.
    - The host serial port is configured at 115200 baud, no parity, 1 stop bit and no flow control.
    - Ethernet port on board connected to a port on the LAN.
    - A microSD card is required. MMC/SD booting on beaglebone board essentially means booting via a microSD card.
  - For TI AM335X (General Purpose) EVM [3]
    - The serial port (Connector J12) on the baseboard of the EVM is to be connected to the host serial port via a NULL modem cable.
    - A serial terminal application (like Tera Term / HyperTerminal / Minicom) should be running on the host.
    - The host serial port is configured at 115200 baud, no parity, 1 stop bit and no flow control.
    - Ethernet port on the base board connected to a port on the LAN.
    - Audio LINE OUT of the EVM connected to headphone/speakers with 3.5mm audio jack.
    - An SD card is required for MMC/SD booting. Booting via MMC cards are not supported.
  - For EVM Starter-Kit (EVM-SK) board
    - The micro USB port (Jumper P3) has to be connected to the host. Please make sure that the Virtual COM Port driver is installed. The driver is available here [2] for downloading. This micro USB-to-serial connection is used for displaying messages on the serial console of the host, provided the port is properly selected.
    - A serial terminal application (like Tera Term / HyperTerminal / Minicom) should be running on the host.
    - The host serial port is configured at 115200 baud, no parity, 1 stop bit and no flow control.
    - Ethernet port(s) on board connected to port(s) on the LAN.
    - A microSD card is required. MMC/SD booting on EVM-SK board essentially means booting via a microSD card.

- **Locating The Bootloader And Demo Application Binary Images**
  - For beaglebone board
    - The bootloader binary image (boot.bin) appended with the TI Image Header at the beginning is named as MLO. This is available at "\binary\armv7a\cgt_ccs\am335x\beaglebone\bootloader\Release_MMCSD\".
    - The demo application image (demo.bin) appended with the TI Image Header at the beginning is named as 'app'. This is available at "\binary\armv7a\cgt_ccs\am335x\beaglebone\demo\Release\".
  - For TI AM335x EVM
    - The bootloader binary image (boot.bin) appended with the TI Image Header at the beginning is named as MLO. This is available at "\binary\armv7a\cgt_ccs\am335x\evmAM335x\bootloader\Release_MMCSD\".
    - The demo application image (demo.bin) appended with the TI Image Header at the beginning is named as 'app'. This is available at "\binary\armv7a\cgt_ccs\am335x\evmAM335x\demo\Release\".
  - For EVM-SK board
    - The bootloader binary image (boot.bin) appended with the TI Image Header at the beginning is named as MLO. This is available at "\binary\armv7a\cgt_ccs\am335x\evmskAM335x\bootloader\Release_MMCSD\".
    - The demo application image (demo.bin) appended with the TI Image Header at the beginning is named as 'app'. This is available at "\binary\armv7a\cgt_ccs\am335x\evmskAM335x\demo\Release\".
- **Loading and running**
  - For beaglebone board
    - Format the microSD card and load the binaries as explained here.
    - Insert the microSD card into the proper slot and reboot the board. A URL will be displayed on the UART Console. On accessing the URL, a webpage will be displayed which will allow the user to explore the features presented in the Demo application.
  - For TI AM335x EVM
    - Format the SD card and load the binaries as explained here.
    - Set the EVM in profile 0 (SW8[1:4] = OFF). For more details refer to EVM reference manual [3].
    - Configure the BOOT pins for MMC/SD Boot mode
    - Insert the SD card into the proper slot and reboot the EVM. Observe the messages on the UART console/LCD display
  - For EVM-SK board
    - Format the microSD card and load the binaries as explained here.
    - Insert the microSD card into the proper slot and reboot the board. Observe the messages on the UART console/LCD display.

NOTE

The application start time/delay may depend on the size of the application itself (copy time from storage device increases), peripheral initialization time and others.

The pre-converted demo application image "app" is not available in releases prior to StarterWare_02.00.00.05. For releases prior to StarterWare_02.00.00.05, MMCSD booting does not expect the binary image to have the TI Image header. In this case, rename the "demo.bin" to "app" and use the latter.

For more information on StarterWare AM335X please refer to getting started guide [4].

References

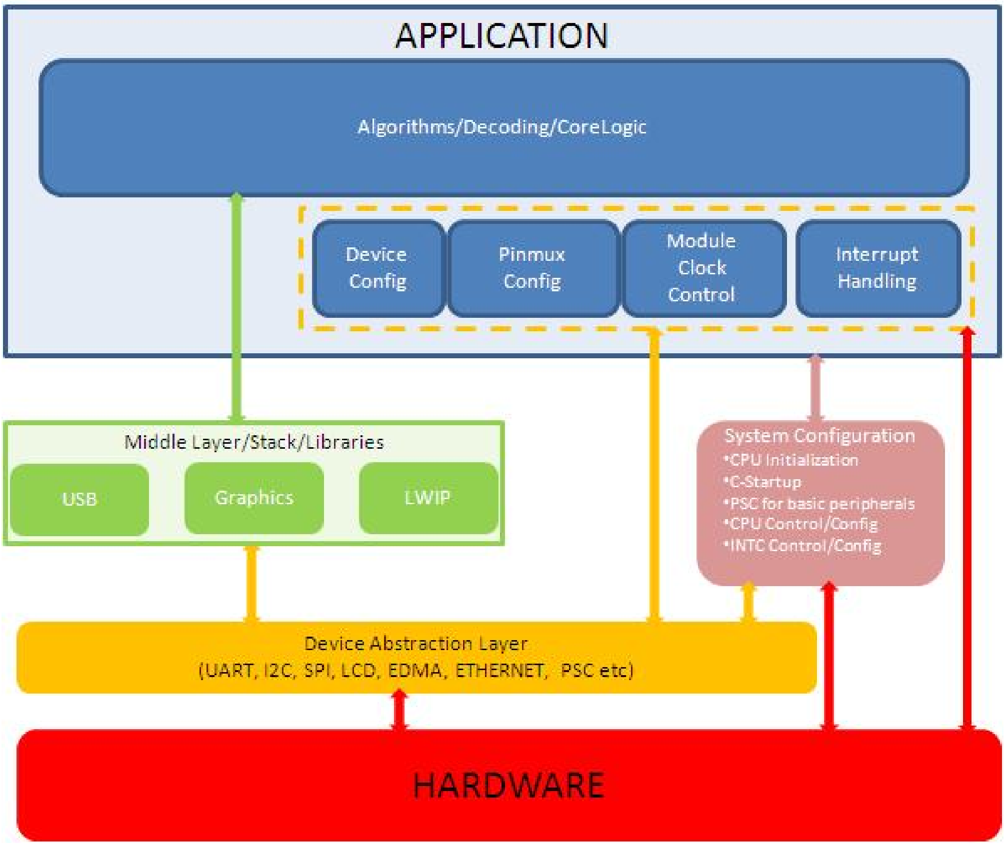
[1] http://beagleboard.org/static/beaglebone/latest/README.htm
[2] http://www.ftdichip.com/Drivers/VCP.html
[3] http://www.ti.com/tool/tmdxevm3358
[4] http://processors.wiki.ti.com/index.php/StarterWare_Getting_Started_02.00.XX.XX

StarterWare Getting Started 02.00.XX.XX

Document License

StarterWare Overview

StarterWare provides a no-OS platform support for ARM/DSP based SoCs. StarterWare provides Device Abstraction Layer (DAL) libraries and peripheral/board level sample/demo examples that demonstrate the capabilities of the peripherals. This package includes the following components:



- **Device Abstraction Layer Library.** This contains device abstraction layer APIs for supported peripherals.

- **Peripheral Examples.** This contains the sample applications which shall demonstrate some of the capabilities of the respective peripheral by using the Device Abstraction Layer APIs.
- **System Configuration code.** This contains the primary code that sets up the processor core and prepares the system for execution of the sample application.
- **Platform Code.** This contains the EVM specific initialization code. This code sets up the EVM specifics like Pin Multiplexing, communication with I/O expander and similar board specific tasks that shall enable proper execution of the applications.

Directory Structure

```
|----StarterWare_#.#.#.#
|-- Software-manifest.pdf
|-- docs
|    |-- ReleaseNotes-#.#.#.#.pdf
|    |-- UserGuide-#.#.#.#.pdf
|-- drivers
|-- examples
|    |--evmAM335x
|       |-uart
|-- grlib
|-- usblib
|-- mmcsdlib
|-- nandlib
|-- host_apps
|-- build
|    |--armv7a
|       |--gcc
|          |--am335x
|              |--drivers
|              |--system_config
|              |--evmAM335x
|                  |--uart
|                  |--platform
|                  |--bootloader
|-- binary
|    |--armv7a
|       |--gcc
|          |--am335x
|              |--drivers
|              |--system_config
|              |--evmAM335x
|                  |--uart
|                  |--platform
|                  |--bootloader
|-- include
|    |-- hw
|    |-- armv7a
|        |--am335x
```

```
|-- platform
|    |-- evmAM335x
|    |-- beaglebone
|-- system_config
|    |--armv7a
|       |-- am335x
|       |-- gcc
|-- bootloader
|    |--include
|    |--src
|-- third_party
|-- tools
|-- utils
|-- test_bench
```

- **drivers** - This directory contains the source files for the driver library APIs.
- **examples** - Examples provided as part of the package showcase some (not all) functionality of the peripheral. This depends on the availability of a peripheral instance and its association, Pin Multiplex settings etc on the EVM. Thus examples differ between EVMs and thus are classified at the top level based on the EVMs that are intended to be run on. Each example's source files are placed under examples/evmAM335x/<example_name>/ directory.
- **grlib** - This folder contains source files for the Graphics library.
- **mmcsdlib** - This folder contains source files for the MMCSD library.
- **nandlib** - This folder contains source files for the NAND library.
- **usblib** - This folder contains source files for USB library.
- **host_apps** - This folder contains files which are used to execute Ethernet examples.
- **build** - This folder will contain the makefile required to build libraries (drivers, system_config, platform, utils, usblib, grlib etc.) and example applications (uart, mcspi, hsi2c etc.). The build files are grouped based on the tools/compiler used to build them. For example, cgt_ccs/ shall contain the Code Composer Studio (CCS) project files of applications and gcc/ shall only contain GCC makefile of applications. The makefile for GCC to build driver library is under build/armv7a/gcc/am335x/drivers directory. Once built, the library/archive is placed in a tool specific directory under binary/armv7a/gcc/am335x/drivers/. For example, when make is invoked from build/armv7a/gcc/am335x/drivers, the libdriver.a is placed at binary/armv7a/gcc/am335x/drivers/.
- **binary** - All the executable are placed in this directory. The generated library or executable/binary is placed in an appropriate path under tool specific directory. For example, uartEcho.out built from CGT TI arm compiler will be placed in the following path - binary/armv7a/cgt_ccs/am335x/evmAM335x/uart/ .
- **include** - The header files for inclusion are all placed under include/. The include files are classified as,
  1. User interface driver headers : For example *include/mcspi.h*, which contain macro definitions used by peripheral APIs and prototypes of the peripheral APIs.
  2. Processor Family and SoC specific files: Certain processor family specific files like *include/armv7a/cpu.h*, *include/armv7a/mmu.h* and SoC specific files like *include/am335x/interrupt.h* are present.
  3. Register Layer files: Files like *include/hw/hw_mcspi.h* which contain the peripheral register offset macros and register field token macros.
- **platform** - Every supported EVM is called as a (hardware) platform inside the package. This exports functions specific to an EVM that usually do (as required) Peripheral Pin Multiplexing settings, Peripheral Clock settings, EVM Profile settings, I/O expander settings etc to enable a peripheral operation on the platform. Peripheral Pin

Multiplexing though depends only on the SoC Silicon Package, the availability of the external ports for communication is decided by the EVM. The code in platform is maintained as an entity distinct from the applications/examples to provide a simpler look at the first level.

- **system_config** - The system configuration and initialization code like the start-up code, interrupt vector initialization, low level CPU specific code etc are provided here. Since this may involve assembly level coding, such code is placed under tools specific directory.
- **bootloader** - Contains the bootloader specific files.
- **third_party** - This shall contain software from third party, provided they are used.
- **tools** - This shall contain software tools used, if any.
- **utils** - This shall contain user utilizable code. As an example, *utils/uartStdio.c* contains wrapper functions which uses UART APIs to interface with the user through the serial console.
- **test_bench** - This folder will contain StarterWare automation test framework and test scripts.

NOTE
Though StarterWare is intended to support multiple processors of the AM/DM family, support may be in a phased manner. The supported list of processors and the associated device abstraction layer of the peripherals as part of the library is mentioned in the release notes for every release.

Host platform Requirements

Building and running StarterWare applications can be from a Windows machine or a Linux machine

A host machine is required only for:

1. Compiling the StarterWare applications
2. Launching CCS v5 and loading and running the applications on the target

Development Host Requirements

For Windows host

TI ARM tools with CCSv5

- Code Composer Studio version CCS 5.4.0.00091 for **Windows** [1](needed for (re)building and/or loading and running applications on the target/EVM)
- Serial console terminal application (like Tera Term, minicom, HyperTerminal)

IAR Embedded work bench for ARM

- IAR embedded work bench for ARM version 6.50 for **Windows** [2](needed for (re)building and/or loading and running applications on the target/EVM)
- Serial console terminal application (like Tera Term, minicom, HyperTerminal)

GCC through Cygwin command line

- Linaro tool chain for ARM for **Windows** [3]
- **Cygwin** [4] to facilitate GCC build
- Serial console terminal application (like Tera Term, minicom, HyperTerminal)

Note on Cygwin Installation

Cygwin **FAQ** [5]is avaialable here.
Cygwin recommended installation procedure is **here** [6]
Cygwin installation should be ensured to include GCC, MAKE, BASH, libc, libgcc and other build tools. If

unsure install all the packages.

For Linux host

GCC through command line

- Linaro tool chain for ARM for **Linux** [7]
- Serial console terminal application (like minicom)

Building application

The StarterWare package contains pre-compiled executable (ELF and binary image) of the peripheral level examples. These were obtained by building the applications using TI ARM compiler. These executable can be executed on the AM335x. The binaries are delivered in release profile build, with optimization level set to O2. StarterWare package also contains the build files and CCS project files to enable the user to rebuild any modifications. StarterWare supports the following compilers/Tools. The Steps to (re)build the examples are detailed here in:

For Windows

- **GCC from Command line in Cygwin:** Refer here for instructions on building the project.
- **TI arm tools through CCSv5 project:** Refer here for instructions on importing and building the project.
- **IAR Embedded Workbench IDE project:** Refer here for instructions on building the project.

For Linux

- **GCC from Command line in Linux:** Refer here for instructions on building the project.

For CCSv5 getting started guide click here [8].

Refer the FAQ's [9] on CCSv5 for more information.

Setting Up The Tool Chain

Windows/Cygwin Linaro arm-none-eabi (command line):

1. If a Windows host is used, then Cygwin needs to be installed for gcc build.
2. Application makefiles invoke library makefiles internally.
3. PATH environment variable should contain the path of the compiler/tool chain.
4. LIB_PATH DOS shell environment variable points to the Linaro installation: Ex: LIB_PATH=C:\tools\Linaro.
5. "set CYGPATH=cygpath" to convert Windows native style filenames to POSIX style file names. Not doing this might lead to linker errors while trying to find libc path.
6. Ensure that the Linaro installation path does not contain any white space

Linux Linaro arm-none-eabi:

1. Application makefiles invoke library makefiles internally.
2. PATH environment variable should contain the path of the compiler/tool chain.
3. LIB_PATH shell environment variable points to the Linaro installation: Ex: LIB_PATH=/opt/linaro/eabi-4_7-2012q4.

Windows TI arm tool chain CCSv5:

1. Ensure that for converting the elf binaries to raw binary format, PATH environment variable is properly updated. Ex: C:\Program Files\Texas Instruments\ccsv5\utils\tiobj2bin.

Flashing binary images and standalone booting

Refer here for instruction.

Loading Executable and Debugging using CCSv5

CCSv5 can be used to load an executable on the Target, run the executable and debug the source code.
The steps for debugging on EVM AM335x can be found here.
The steps for debugging on BeagleBone board can be found here. Similar steps apply for EVM-SK also.

Creating a New CCSv5 Project

The information required to create a new project in CCSv5 is given here.

References

[1] http://processors.wiki.ti.com/index.php/Download_CCS
[2] http://supp.iar.com/Download/SW/?item=EWARM-EVAL
[3] https://launchpad.net/gcc-arm-embedded/4.7/4.7-2012-q4-major/+download/gcc-arm-none-eabi-4_7-2012q4-20121208-win32.exe
[4] http://www.cygwin.com/
[5] http://cygwin.com/faq.html
[6] http://cygwin.com/faq-nochunks.html#faq.setup.setup
[7] https://launchpad.net/gcc-arm-embedded/4.7/4.7-2012-q4-major/+download/gcc-arm-none-eabi-4_7-2012q4-20121208-linux.tar.bz2
[8] http://processors.wiki.ti.com/index.php/CCSv5_Getting_Started_Guide
[9] http://processors.wiki.ti.com/index.php/FAQ_-_CCSv5

AM335X StarterWare Booting And Flashing

AM335X Flashing And Booting

This section describes the flashing and booting of TI AM335x EVM, EVM-SK and Beagle Bone from different media. The below table describes, whether a boot mode is supported for the given board or not.

	MMCSD	UART	SPI	NAND
TI AM335x EVM	Yes	Yes	Yes	Yes
TI AM335x EVM-SK	Yes	Yes	No	No
Beagle Bone	Yes	Yes	No	No

The bootloader initializes the DDR and required peripherals. Once the bootloader and StarterWare application images are ready, they can be flashed/copied onto the media for standalone media booting. For information on the binaries supplied with the package and the steps to be followed to load them, please refer quick start guide [1]. For information on host platform requirements and steps to build, please refer getting started guide [4].

Two types of images has to be flashed on to the media. A bootloader image and an application binary image. Both application and the bootstrap image are required to be in a special format containing a header including the size and load address of the binary image.

- When a bootloader/application is built from any toolchain, bootloader/application binary image ***binary/armv7a/gcc/am335x/<EVM>/<bootloader/app_folder/<build-configuration>/<app_name>.bin*** is automatically converted to ***binary/armv7a/gcc/am335x/<EVM>/<bootloader/app_folder>/<build-configuration>/<app_name>_ti.bin***.
- Any binary image can be converted to this special format using the image converter application located at */tools/ti_image/*. The Usage of the tool is described at usage of tiimage tool

Once both the bootloader and the StarterWare application images are ready in the special format, the images can be flashed/copied to the media as below.

Booting Via SD Card

Booting from SD Card involves two steps.

1. Preparing the SD card.
2. Booting the target.

Preparing the SD card

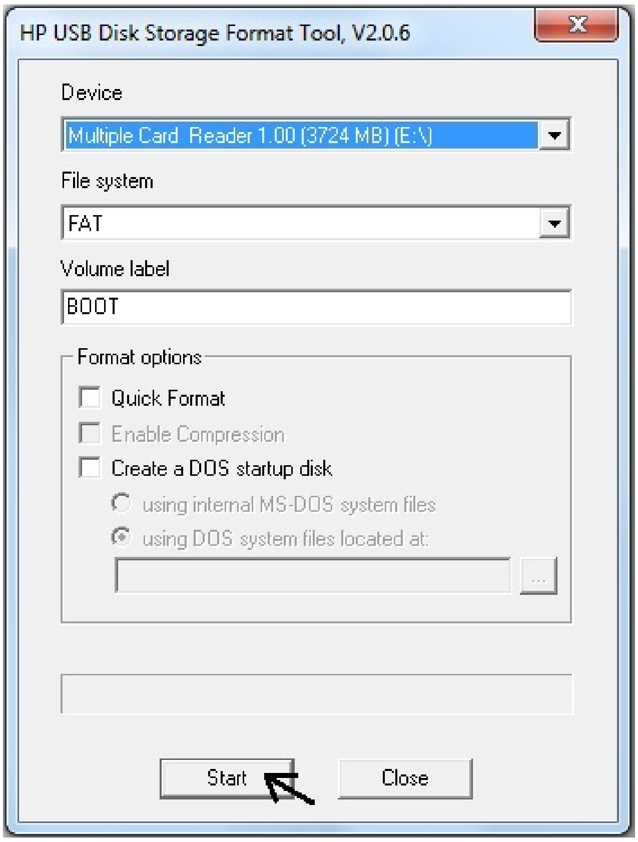
The SD card needs to be prepared, by FAT formatting it as follows.

1. ***HP USB Disk Storage Format Tool v2.0.6 Portable*** has to be used to format the SD card. This tool can be downloaded from this link [2].
2. Choose a SD card and a USB based or similar SD card reader/writer. Plug it to a Windows host system.
3. Run the ***HP USB Disk Storage Format Tool v2.0.6 Portable*** executable. The executable should automatically detect the SD card plugged via reader. Else point it to the new disk.
4. Choose FAT32 if the SD card size is greater that 4GB. Else FAT should be good to go.
5. Click on 'Start'.
6. After the formatting is complete, the card is ready to be populated with the files required.
7. The converted bootloader binary image(Bootloader binary image(boot.bin) appended with the TI Image Header) has to be renamed to "MLO" from "boot_ti.bin". The boot_ti.bin is created by building the bootloader in MMC/SD boot mode.
8. Similarly the converted application binary image has to be renamed to "app" from "<app_name>_ti.bin"
9. Copy both the boot loader image "MLO" and the application image "app" to the SD card.
10. Safely eject/remove the card from the host, unplug the card reader, remove the SD card. The SD card is ready for use on AM335x target.

NOTE:

1. The above steps are applicable when formatting a microSD card also.

The below image gives the snap shot of SD card format tool:



Booting the target

1. Insert the SD card into the SD slot (For GP EVM, the SD Card slot on baseboard). Connect a UART cable to a host running a serial terminal application (teraterm/hyperterminal) with 115200 baud, 8bit, No parity and 1 STOP bit configuration.
2. Configure the board (Applicable for AM335X EVM only, there is no boot mode selection for Beaglebone) for SD Boot mode
    1. SD instance 0 (on base board) is available in all profiles (Applicable for AM335X EVM only). For more details refer to EVM reference manual [3].
    2. SD instance 0 boot mode needs to appropriately set (Applicable for AM335X EVM only). For booting out of SD card, SD boot should appear first in the boot device list. If any other boot mode is selected, even if a SD boot card is inserted, and does not appear first in the list, the first available valid boot image (from NAND or SPI etc) is booted and SD is not selected. Only if no valid boot image is found in the first devices, SD boot image will be selected.
3. On power on, the MLO is first detected and copied and executed from the OCMC0 RAM. The MLO then copies the application image (app) from the card to the DDR and passes the control to the application. On Successful boot, following messages appear on the serial console.

```
   StarterWareAM335x Boot Loader
Copying application image from MMC/SD card to RAM
Jumping to StarterWare Application...
```

After this the application should take control and execute.

Booting Via UART

The ROM code and StarterWare Bootloader supports XMODEM protocol, and the images shall be the plain binary images, without header.

1. Configure the board (Applicable for AM335X EVM only, there is no boot mode selection for Beaglebone) for UART Boot mode
2. On power up, the ROM code will send "CC.." on UART console expecting StarterWare Bootloader via XMODEM
3. On sending the bootloader image, boot.bin via XMODEM, following message appear on the serial console.

```
   StarterWareAM335x Boot Loader
CC
```

4. Now any application binary (without header) can be sent via XMODEM. The application will start executing.

Booting Via SPI

Booting from SPI involves two steps.

1. Flashing bootloader and application to SPI Flash
2. Booting the target.

Flashing bootloader and application to SPI Flash

- Configure the EVM in profile 2 (SW8[1] = OFF, SW8[2] = ON, SW8[3:4] = OFF).For more details refer to EVM reference manual [3].
- Configure the BOOT pins for SPI Boot mode
- Connect the target.
- Load the GEL file */tools/gel/AM335X.gel*.
- Load the *spi_flash_writer_AM335X.out* onto the EVM.
- It will prompt for binary file name. Update the file with the path.
- It will prompt again for load address in flash. If bootloader is to be flashed, provide 0x00000. For StarterWare application provide 0x20000.
- Once SPI flash writing completes, disconnect from CCS.

Booting the target

- Connect a UART cable to a host running a serial terminal application (teraterm/hyperterminal) with 115200 baud, 8bit, No parity and 1 STOP bit configuration.
- Configure the board for SPI boot mode
- SPI is available in profile 2 (SW8[1] = OFF, SW8[2] = ON, SW8[3:4] = OFF). For more details refer to EVM reference manual [3].
- Once the SPI boot mode is chosen, the bootloader is first detected and copied and executed from the OCMC0 RAM. The bootloader then copies the application image from the SPI to the SDRAM and passed the control to the application. If the process is succesful, following messages appear on the serial console.

```
   StarterWareAM335x Boot Loader
Copying application image to RAM
Jumping to StarterWare Application...
```

After this the application should take control and execute.

Booting Via NAND

Booting from NAND involves two steps.

1. Flashing bootloader and application to NAND Flash
2. Booting the target.

Flashing bootloader and application to NAND Flash

- Configure the EVM in profile 0 (SW8[1:4] = OFF). For more details refer to EVM reference manual [3].
- Configure the BOOT pins for NAND Boot mode
- Connect the target.
- Load the GEL file */tools/gel/AM335X.gel*.
- Load the *nand_flash_writer_AM335X.out* onto the EVM and Run. The flash writer will output messages on the CCS console. When it prompts for inputs, proper inputs shall be given via CCS console.
- When prompted for binary file name, update the file with the proper path.
- Select option for flashing.

```
Choose your operation
Enter 1 ---> To Flash an Image
Enter 2 ---> To ERASE the whole NAND
Enter 3 ---> To EXIT
```

Select option 1 when prompted. Select option 2 in case if you want to erase the whole NAND.

- Enter the image path to flash when prompted as shown below.

```
Enter image file path
```

Provide the complete path (e.g.c:\images\boot_ti.bin)

- Enter the offset when prompted as shown below.

```
Enter offset (in hex):
```

This offset is the start location from where the image should be flashed.
NOTE:

1. Use hex format
2. If bootloader is to be flashed, provide 0x00000. For StarterWare application, provide 0x80000.

- Select ECC for flashing.

```
Choose the ECC scheme from given options
Enter 1 ---> BCH 8 bit
Enter 2 ---> HAM
Enter 3 ---> T0 EXIT
Please enter ECC scheme type:
```

Always select BCH8 for bootloader and application as the ROM code and bootloader uses the BCH8 ECC scheme. **Enter 1** for u-boot.min.nand.

- Ensure that the flash info displayed by the tool matches the NAND flash in the EVM.
- After this the tool should first erase the required region in flash and then start flashing the new image.
- Finally you should see the following message.

```
Application is successfully flashed
NAND boot preparation was successful!
```

- Once NAND flash writing completes, disconnect from CCS.

Booting the target

- Connect a UART cable to a host running a serial terminal application (teraterm/hyperterminal) with 115200 baud, 8bit, No parity and 1 STOP bit configuration.
- Configure the board for NAND boot mode
- NAND is available in profile 0 (SW8[1:4] = OFF). For more details refer to EVM reference manual [3].
- Once the NAND boot mode is chosen, the bootloader is first detected and copied and executed from the OCMC0 RAM. The bootloader then copies the application image from the NAND to the SDRAM and passed the control to the application. If the process is succesful, following messages appear on the serial console.

```
   StarterWareAM335x Boot Loader
Copying application image from NAND to RAM
Jumping to StarterWare Application...
```

After this the application should take control and execute.

Boot Modes

This section is applicable for AM335X EVM only.

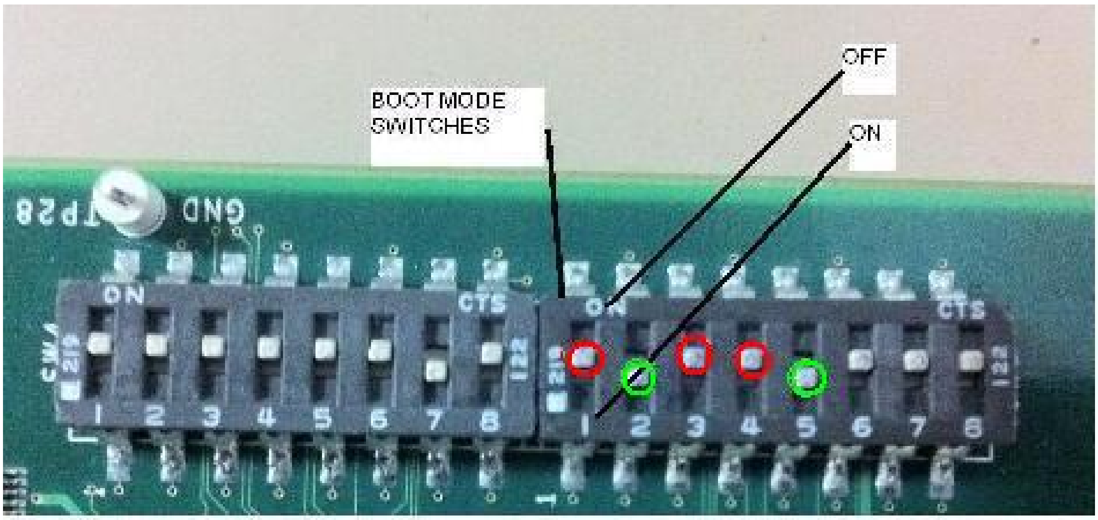
The device supports variety of boot modes through the ROM Code. Boot mode selection is determined by the state of the SYSBOOT configuration pins.

Switch SW3 is for selecting the boot modes and device selection depends on the DIP switches intended for profile selection on EVMs.

1. Make sure that the EVM boot switch settings are set to required boot mode and then power on the board. The picture below shows the boot mode configuration switch SW3 on the AM335X EVM.

Note

- The bootmode setting in this picture is for NAND boot.Nand boot corresponds to(SW3 5:1) 10010



RED: circle shows **OFF** and **GREEN** circles shows **ON** switches.

Note: ON is labeled on the wrong side of SW3 boot mode switch.

2. Due to heavy pin-muxing, boot device is selectively available on selected AM335x EVMs & profiles. Ensure that boot device is available in EVM/profile before setting the boot mode.

3. Refer the following table which explains the boot mode settings for different media.

Boot Modes				SYSBOOT[4:0]
1st	2nd	3rd	4th	
reserved	reserved	reserved	reserved	00000
UART	XIP w/ WAIT (MUX2)	MMC	SPI	00001
UART	SPI	NAND	NANDI2C	00010
UART	SPI	XIP (MUX2)	MMC	00011
UART	XIP w/ WAIT (MUX1)	MMC	NAND	00100
UART	XIP (MUX1)	SPI	NANDI2C	00101
EMAC	SPI	NAND	NANDI2C	00110
EMAC	MMC	XIP WAIT (MUX2)	NAND	00111
EMAC	MMC	XIP(MUX2)	NANDI2C	01000
EMAC	XIP WAIT (MUX1)	NAND	MMC	01001
EMAC	XIP (MUX1)	SPI	NANDI2C	01010
USB	NAND	SPI	MMC	01011
USB	NAND	XIP(MUX2)	NANDI2C	01100
USB	NAND	XIP(MUX1)	SPI	01101
reserved	reserved	reserved	reserved	01110
GP Fast External Boot	UART	EMAC	reserved	01111
XIP (MUX1)	UART	EMAC	MMC	10000
XIP w/WAIT (MUX1)	UART	EMAC	MMC	10001
NAND	NANDI2C	USB	UART	10010
NAND	NANDI2C	MMC	UART	10011
NAND	NAND12C	SPI	EMAC	10100
NANDI2C	MMC	EMAC	UART	10101
SPI	MMC	UART	EMAC	10110
MMC	SPI	UART	USB	10111
SPI	MMC	USB	UART	11000
SPI	MMC	EMAC	UART	11001
XIP (MUX2)	UART	SPI	MMC	11010
XIP w/ WAIT (MUX2)	UART	SPI	MMC	11011
MMC1	MMC	UART	USB	11100
Reserved	reserved	reserved	reserved	11101
Reserved	reserved	reserved	reserved	11110
GP Fast external Boot	EMAC	UART	reserved	11111

Bootloader

StarterWare AM335X provides a simple bootloader, which can be flashed to the media, which after a power-on-reset can bootstrap the board. Additionally it can load an application from the media to DDR and transfer the control to the application. This can be used for out-of-box experience.

Upon board power on, the ROM Code, residing in the ROM of the AM335X kickstarts. ROM Code checks the boot mode setting (which should be SPI mode in case of StarterWare bootloading) and depending upon the boot mode setting it copies boot loader from the respective flash device. ROM Code requires boot loader to be in a special format with a header appended to the binary image. The header shall contain the load address of the bootloader and the size of the bootloader image. The ROM Code copies the boot loader to onchip OCMC RAM located at 0x402F0400 and gives control to it.

The boot loader initializes the PLLs and enables peripheral clocks and then it initializes the DDR. The configurations will set the processor core operating frequency to 720MHz. Once all the initialization is done, it copies the application from flash to the DDR and transfers the control to the application. The application starts to execute from DDR.

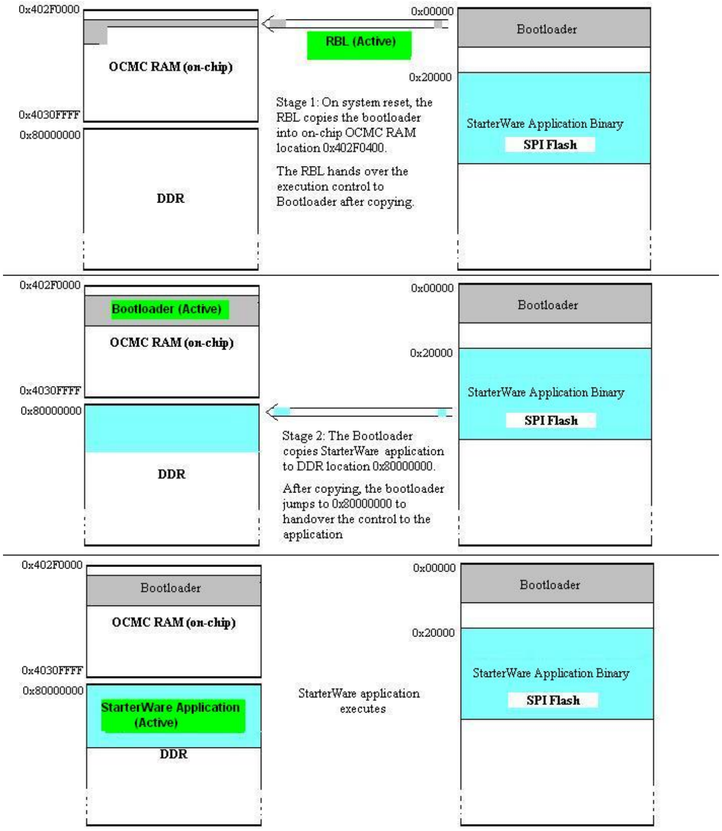
NOTE: The application binary has to be converted to a special format for NAND, SPI and MMCSD boot modes. This is required by the bootloader to read the size of the application and the address where it has to be loaded in DDR. The special format will contain the load address and the size of the application binary.

NOTE: This is not required for booting via UART.

Please following the steps mentioned above to convert a binary to special format.

Stages in SPI Booting

The different stages in standalone booting of StarterWare applications from the SPI flash is described in the below diagram



Stages in NAND Booting

NAND boot follows same steps as SPI Boot but application offset for NAND boot is at 0x80000.

Stages in SD Booting

- Stage 1 : On System reset, the ROM Code copies the bootloader by reading the MLO file from the SD to the address embedded on its header, Then ROM Code handsover the execution control to Bootloader.
- Stage 2 : The Bootloader copies the StarterWare application by reading the file named app to the addresss embedded on its header. After copying, the bootloader jumps to start address of the application to handover the control to the application.
- Stage 3 : StarterWare application executes.

Stages in UART Booting

- Stage 1: On System reset, the ROM Code expects the StarterWare Bootloader via XMODEM from the UART console. (prints CCC.. on the UART console)
- Stage 2: The StarterWare Bootloader image compiled for UART boot mode, boot.bin (without header) can be sent via XMODEM protocol. the ROM code copies the bootloader image onto OCMC RAM, hands over the execution control to StarterWare Bootloader.
- Stage 3: The StarterWare Bootloader will expect an application image to be sent through UART console via XMODEM. (prints CCC.. on the UART console)
- Stage 4: The StarterWare application image (<image>.bin, without header), can be sent via XMODEM through UART console. The StarterWare Bootloader copies this application image to DDR and the application starts execution.

Bootloader Execution Flow

1. Entry point to the bootloader is the function *Entry()* which is inside the file bl_init.s (src/gcc). This function initializes stack and bss.
2. Invokes *bl_start()*, which is in the file bl_main.c.
    1. This invokes *DeviceConfig()*, an EVM dependent function, for AM335x, function will be present in the file bl_am335x.c.This calls PLL and DDR initialization functions.
    2. *PLLInit()* Initializes PLL0 and PLL1.
    3. *DDRInit()* function initializes the DDR.
3. Initializes the UART by calling *UartConfigure()*.
4. Executes Imagecopy() function which is present in the device independent file bl_copy.c .Depending upon the defined boot mode it calls one of the *SPIBootcopy()/ NANDBootcopy()* function. This function copies the application to the DDR.
5. Transfer execution control to the application. The application now starts executing.

Usage of ti_image Tool

TI image tool, available in "tools" directory of the package can be used to add the header information to the boot loader or application binary images. It has separate executable for Windows and Linux.

Usage of the same is explained below.

For Windows

1. Open command prompt
2. Go to /tools/ti_image
3. If the tiimage.exe is not available, generate one with Cygwin or MingW [3]environvment using the command "gcc tiimage.c -o tiimage"
4. Execute the image converter by giving proper inputs in the format, tiimage.exe <load address> <boot mode> <input image path/name> <output image path/name>

- Examples
    - tiimage.exe 0x402F0400 MMCSD boot.bin MLO
    - tiimage.exe 0x80000000 NONE uartEcho.bin app

For Linux

1. Go to /tools/ti_image
2. If the tiimage binary is not available, generate one with GCC using the command "gcc tiimage.c -o tiimage".
3. Execute the image converter by giving proper inputs in the format, ./tiimage <load address> <boot mode> <input image path/name> <output image path/name>

- Examples
    - ./tiimage 0x402F0400 MMCSD boot.bin MLO
    - ./tiimage 0x80000000 NONE uartEcho.bin app

NOTE

- For bootloader, value of <*boot mode*> should be corresponding bootmedia name i,e *NAND/SPI/MMCSD/UART.*
- For application, value of <*boot mode*> should be *NONE.*
- While building bootloader the binary image is automatically converted to special format and placed in ***binary/armv7a/gcc/am335x/<EVM>/bootloader/boot_ti.bin***. And While building application the binary image is automatically converted to special format and placed in **binary/armv7a/gcc/am335x/<EVM>/<application>/<application>_ti.bin**.

References

[1] http://processors.wiki.ti.com/index.php/Quick_Start_Guide_StarterWare_02.00.XX.XX_%28supports_AM335x%29
[2] http://www.mediafire.com/download/4d2jzmqfifi/HP+USB+Disk+Storage+Format+Tool+v2.0.6+Portable.exe
[3] http://www.mingw.org/

StarterWare Project Creation

Introduction

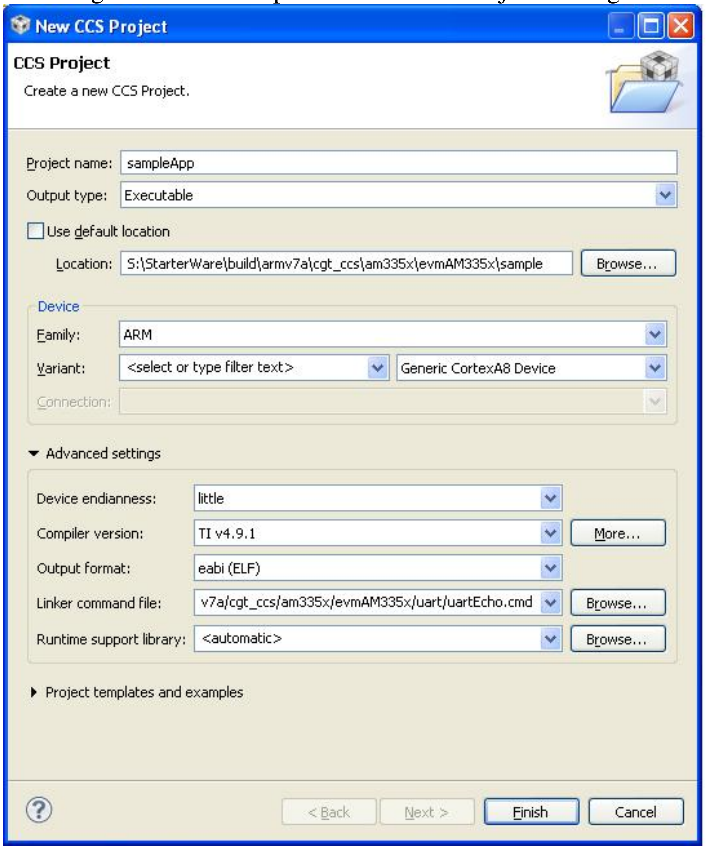
This page details the steps to create projects in IDEs that are supported by StarterWare.

CCS Projects

Creating a CCSv5 Project

- In the File Menu, go to *New* and select *CCS Project*. A window shall appear where the details of the new CCS Project have to be specified.
- Enter the *Project Name*.
- The Project type can be an Executable(Application) or a Static Library. Choose the appropriate project type under *Output type* parameter. However if the project is neither an Executable nor a Static Library, select the 'Other' option.
- Select the location where the project has to be created.
- Options for *Device*
  - Select the Processor Family type under *Family* parameter. For AM335x, the family is ARM.
  - Select the Processor Variant under *Variant* parameter. For AM335x, the Variant is 'Generic CortexA8 Device'.
- Options for *Advanced Settings*
  - Select the appropriate option for *Device Endianness* parameter.
  - Select the *Compiler Version* used for compilation. The Compiler libraries are part of the CCS Installation Package and CCS automatically selects the compiler version upon selection of the Family. However, compilers of other versions can also be used for compiling the CCS Project.
  - Select the format of the output file to be generated under *Output Format* option. The Output file can conform to either ELF or COFF standard file formats.
  - If the project is an Executable one, then the path to the Linker Command File should be specified under *Linker Command File* parameter.
  - Additionally, the *Runtime Support Library* parameter should be appropriately selected.
- Options for *Project templates and examples*
  - Select the *Empty Project* option under *Empty Projects* parameter. This creates an empty project fully initialized for the selected device.

The image below is a snapshot of the New Project Settings for AM335x CCS Projects for applications.



Linking/Adding Source Files to Project

The source files that are expected to be built should be associated with the project. These files can either be linked to the project or added to it. In the former case, the project associates itself with the specified source file but does not copy it to the project directory. Thus any changes to the source file in its parent directory gets reflected in the project. In the latter case, a copy of the source file is added to the project directory.

- To associate source files with the project, travel to 'Project' and select 'Add Files'.

Populating the Project Properties

The newly created CCS Project has to be populated with additional properties. Some of the important Project Properties to be specified are given below.

- **Include Options**

The directories containing header files to be included in the Project can be specified. To select, right-click on the Project Name in Project Explorer window and select *Properties*. Under the *CCS Build* option, select *Include Options*. Now specify the path(s) of Include Directory(ies) which have to be searched for header files during compilation.

- **Predefined Symbols**

There is a provision to define Pre-defined symbols before compilation. To do this, under *Properties*, travel to *CCS Build*, then to *Advanced Options* and then select *Predefined Symbols* option.

- **Runtime Environment**

The memory type where the executable has to be executed has to be chosen. The available options are RAM and ROM. To do this, under *Properties*, travel to *CCS Build*, then to *ARM Linker*, then to *Advanced Options* and then select *Runtime Environment*. Select the appropriate option for the *Initialization Model*.

- **Build Artifact**

The path and name of the output executable has to be specified. To configure this, under *Properties*, travel to *C/C++ Build*, then to *Settings* and select *Build Artifact*. Here specify the path and name of the output executable under *Artifact name*.

- **Build Steps**

There tool chain performs Pre-build and Post-build operations as specified in the project. To specify these operations, under *Properties*, travel to *C/C++ Build*, then to *Settings* and select *Build Steps*. Here specify the Commands and Descriptions for the Pre-build and Post-build steps, if required.

Note

- In the CCS Projects of AM335x applications, there is a Post-build step command which does the following two tasks:
  - Generates a Binary file from the ELF executable file (.out) that is already obtained from the Project's build.
  - Adds a TI Image Header to the Binary file.
- The above mentioned information regarding CCSv5 Project Creation and Properties are discussed with reference to the CCS version 5.1.1.00031 and above.
- From CCSv5.3 to generate binary file from ELF executable file armofd.exe and armhex.exe, present in tools\compiler\arm_5.0.X\bin, to be used. ofd470.exe and hex470.exe, present in tools\compiler\tms470\bin, used in earlier versions are not available from CCSv5.3.

StarterWare 02.00.01.01 User Guide

NOTE :

1. For updated document please refer online wiki page: http://processors.wiki.ti.com/index.php/StarterWare_02.00.01.01_User_Guide
2. Before starting to use the drivers and applications please read information on how to build and use StarterWare package.
3. StarterWare examples print messages on the UART Serial Console running on the host. Hence, a serial terminal application (like Tera Term/HyperTerminal/minicom) should be running on the host. The host serial port is configured at 115200 baud, no parity, 1 stop bit and no flow control. Please ensure that the local echo setting for the terminal is turned off.

System Configuration

An AM335x SoC integrates a Cortex-A8 core which can act as the overall system controller, a Cortex-M3 core to manage entry/exit of various power down modes, associated memories and peripherals.

This section describes the guidelines for programming on the AM335x SoC System.

The ARM Subsystem

StarterWare exports APIs for configuring Cortex-A8 to operate in privileged mode or non privileged mode and APIs to configure MMU and Cache. The APIs for configuration of the CPU can be found in /include/armv7a/cpu.h, for MMU can be found in /include/armv7a/mmu.h and the APIs for configuration and maintanance operations of cache can be found in /include/armv7a/cache.h

- **Features Not Supported**
  - Security extension features

Programming Guidelines

Applications can execute in privileged or non-privileged (user) mode of ARM. On entry to the *main()* function of application, the system will be in privileged mode. However, the application can switch to nonprivileged mode (user mode) using *CPUSwitchToUserMode()* and back to privileged mode using *CPUSwitchToPrivilegedMode()* at any point of time. While executing in user mode, the application shall not access system resources which needs privileged access. The privileged mode used in StarterWare is system mode of Cortex-A8 core. Note that all ISRs will be executing in privileged mode.

Branch prediction is enabled by default during initialization. Separate APIs are provided for enabling/disabling instruction and data cache. Also, APIs are given for invalidation and cleaning of caches. Invalidating a cache will clear it of any stored data. Cleaning a cache will force it to write the dirty values to main memory. Note that MMU (Memory Management Unit) shall be enabled before enabling the data cache.

StarterWare supports one level of paging with one-to-one mapping. That is, virtual address will be equal to physical address. However, we can define separate regions with its own desired attributes like cache policies, access permissions etc.

The application shall define the master page table. *MMUInit()* will initialize the master page table with fault entries. mmu.h file exports a structure type named REGION. With the help of this, we can specify each memory region with intended attributes such as section, start address, number of pages, memory type, number of pages in the region, security type and access permissions. *MMUMemRegionMap()* will update the page table for a memory region. After the desired regions are mapped, MMU can be enabled using *MMUEnable()*. The APIs for configuring MMU are exported in /include/armv7a/mmu.h. The file also describes the parameters passed to the APIs.

StarterWare exports APIs for Cache maintanance operations. *CacheEnable/Disable()* APIs can be used to enable/disable Caches. For maintaining cache coherency for non-shareable memory, cache maintanance operations are provided. For example, *CacheDataInvalidateAll()* can be used to invalidate the contents of Data Cache and *CacheDataCleanAll()* can be used to write all the contents of Data Cache to memory to make the Data Cache and memory coherent. The APIs for cache operations and the parameter description are exported in /include/armv7a/cache.h

Please refer to ARM Architecture Reference Manual [1] for more information on ARMv7a architecture. The document discusses concepts of Memory Management Unit, Memory Attributes, Caches and their effect on the performance in detail.

Interrupt Controller

AM335x uses Cortex A8 interrupt controller as an interface between different peripherals of the system and the Cortex A8 core interrupt lines. The Host Cortex A8 Interrupt Controller is responsible for prioritizing all service requests from the system peripherals and generating either nIRQ or nFIQ to the host. It has the capability to handle up to 128 requests which can be steered/prioritized as A8 nFIQ or nIRQ interrupt requests, priority 0 being the highest. Note that the SoC doesnt support routing of interrupts to FIQ. The API functions exported are listed in /include/armv7a/am335x/interrupt.h

Note : *StarterWare implements only Prioritized IRQ Handler. However, if prioritization is not desired, all interrupts can be assigned the same priority level. For example, if all interrupts in an application are assigned priority level 0, no IRQ preemption will occur.*

- **Prioritized IRQ Handler Execution Sequence**

1. Save ARM Core Register Context (IRQ is disabled in CPSR by the ARM core on jumping to the IRQ vector).

2. Save the current IRQ threshold value. We need to change this for preventing any same or lower priority interrupt happening.
3. Get the active IRQ priority and set it as the new threshold value.
4. Enable new IRQ Generation at INTC. Here we acknowledge the IRQ and INTC can generate a new IRQ anytime.
5. Enable IRQ generation in CPSR of ARM core and switch to System mode of ARM core (All ISR will be executed in System mode).
6. Get the vector table and jump to the ISR of active IRQ. The link register is programmed to return here after executing ISR.
7. Return from the ISR.
8. Disable IRQ generation in CPSR of ARM core and switch back to IRQ mode (We have the context saved in IRQ stack).
9. Restore the threshold value.
10. Restore ARM Core Register Context and return from IRQ handler.

- **Features Not Supported**
  - Routing of interrupts to FIQ
  - Security extension features in the interrupt controller.

Programming Guidelines

Interrupt Service Routines are part of the application.The application shall decide the priority level to be assigned for each interrupt. Also, there should be a registered interrupt service routine for each system interrupts enabled for processing.

- The following sequence can be used to set up the Cortex A8 interrupt controller for a system interrupt.
  - Initialize the Cortex A8 interrupt controller using *IntAINTCInit()*. This will reset the interrupt controller.
  - Register the ISR using *IntRegister()*. After this point, when an interrupt is generated, the control will reach the ISR if the interrupt processing is enabled at the peripheral and interrupt controller.
  - Set the system interrupt priority and the required host interrupt generation controller using *IntPrioritySet()*. The interrupt shall be routed to IRQ.
  - Enable the system interrupt at AINTC using *IntSystemEnable()*.
  - Enable IRQ in CPSR using IntMasterIRQEnable()

The API *IntRawStatusGet* can be used to read the raw status of a system interrupt and *IntPendingIrqMaskedStatusGet()* API can be used to read the masked status of interrupts routed to IRQ.

Example Configurations

- **Example Configurations For Interrupt Controller**
  - The uartEcho (examples/evmAM335x/uart/) application demonstrates the interrupt handling for UART interrupts. The sample application uses UART0 peripheral to demonstrate interrupt processing. The UART0 system interrupt is mapped to host interrupt line IRQ. UART0Isr() is the Interrupt Service Routine registered for this system interrupt.
  - The irqPreemption (examples/evmAM335x/irq_preemption/) application demonstrates IRQ preemption by assigning different priority levels to UART, RTC and Timer interrupts. UART interrupt is given the lowest priority and Timer is given the highest priority. The application demonstrates RTC ISR preempting UART ISR and Timer ISR preempting RTC ISR. Messages will be printed on the UART console at the entry and exit of each ISR.
- **Example Configuration For Cache and MMU**
  - The example application uartEdma_Cache (examples/evmAM335x/cache_mmu) demonstrates the usage of MMU and cache by direct mapping of virtual memory to physical memory. The pages are divided into 1MB

sections with only one level of translation. A page can be either cacheable or non-cacheable. The memory mapped peripherals regions are configured as device memory which cannot be cached. The OCMC/DDR memory are marked as cacheable with the following attributes.

  - Memory Type - Normal Memory
  - Inner Cache(L1 Data cache) - Write through, No Allocate on Write
  - Outer Cache(L2 Unified cache)- Write Back, Write Allocate
- When cache example application is compiled two executable are generated. They are,

uartEdma_Cache - This executable demonstrates the effects of not cleaning the cache before a third party (EDMA) tries to access the buffer from the main memory.

Execution sequence

1. Populate a buffer with lower case alphabets, a..z.

2. Send the buffer via DMA onto the serial console. *a..z gets printed on the console*

3. Enable Cache and Populate the buffer with upper case alphabets, A..Z.

4. Send the buffer via DMA onto the serial console. *a..z gets printed on the console*

uartEdma_CacheFlush - This executable demonstrates the cleaning of the cache before a third party (EDMA) tries to access the buffer from the main memory.

Execution sequence

1. Populate a buffer with lower case alphabets, a..z.

2. Send the buffer via DMA onto the serial console. *a..z gets printed on the console*

3. Enable Cache and Populate the buffer with upper case alphabets, A..Z.

4. Clean the Data CacheSend the buffer via DMA onto the serial console.*A..Z gets printed on the console*

Serial Peripherals

UART

Introduction

The UART controller present in AM335x is compatible with the 16C750 standard. There is a 64-byte FIFO each for Transmitter and Receiver which holds the outgoing and incoming data respectively. There are programmable and selectable transmit and receive FIFO trigger levels for DMA and Interrupt request generation. The UART can transit to Sleep mode with complete status reporting capabilities in both normal and sleep modes. There are provisions for Hardware Flow Control (RTS/CTS) and Software Flow Control (XON/XOFF). The UART can generate two DMA requests and 1 interrupt request to the system.

- **Operational modes that are not supported in Software**
  - IrDA/CIR modes of operation

The programming sequence for UART can be found here.

Executing the Example application

- For TI AM335x EVM
  - Connect the serial port on the baseboard to the host serial port via a NULL modem cable.
- For Beaglebone
  - The board has a USB-to-serial port. Connect this mini USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- For EVM-SK
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.

A serial terminal application should be running on the host. When the example application is loaded on the target and executed, a string is printed on the serial terminal showcasing transmission. After this, it indefinitely expects for characters input from the user on the serial terminal and echoes the same back on the terminal. Whereas in DMA application, it expects the user to input specified number of characters and later echoes them back.

- Modules used in Interrupt application
  - UART-0
  - Interrupt Controller
- Modules used in DMA application
  - UART-0
  - EDMA
  - Interrupt Controller

Note

- The UART EDMA application can be executed with or without UART FIFO being used. To use the UART FIFOs, define the macro named *UART_ENABLE_FIFO*. The application by default does not use UART FIFOs.

HSI2C

Introduction

The HSI2C component is in complaint with the Philips Semiconductors Inter-IC bus (I2C-bus) specification version 2.1. The HSI2C module supports only Fast mode (up to 400 kbps) of operation.HSI2C can be configured to multiple master-transmitters and slave-receivers mode and multiple slave-transmitters and master-receivers mode.HSI2C also could be configured to generate DMA events to the DMA controller for transfer of data. The HSI2C driver library exports a set of APIs to configure and use HSI2C module for data transfers. The APIs are exported in /include/hsi2c.h

- Features Not Supported
  - High speed data transfer

Clocking Constraint

This module input clock is derived from the Peripheral PLL, which is 48MHz. HSI2C module imposes a constraint that the module input frequency is limited between 12MHz and 24MHz for proper operation, which is achieved by a first level divisor, which is called the pre-scaler. The actual output clock or the operating clock frequency obtained by calculating the clock divisor as per the formula provided in the HSI2C peripheral user's guide

The programming sequence for HSI2C can be found here.

EEPROM Application

Executing the Example application

- For TI AM335x EVM
  - Connect the serial port on the baseboard to the host serial port via a NULL modem cable.
- For Beaglebone
  - The board has a USB-to-serial port. Connect this mini USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- For EVM-SK
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.

A serial terminal application should be running on the host.

When the example application is loaded and executed on the target, the data flashed to the EEPROM is read over I2C bus and printed on the serial console.This functionality is demonstrated both in interrupt and DMA mode

- Modules used in the Interrupt Mode of example
  - I2C-0
  - Interrupt Controller
  - UART-0
- Modules used in the DMA Mode of example
  - I2C-0
  - EDMA
  - Interrupt Controller
  - UART-0

Temperature Sensor Application

Temperature sensor is a device which is used to measure the surrounding temperature. On Revision 1.1A EVM (generally called Beta EVM) a Digital Temperature sensor with part number TMP275 is present and can be accessed through I2C.

Executing the Example application

- Modules used in this example
  - I2C-1
  - Interrupt
  - UART-0

When the temperature sensor example application is loaded and executed on the target,the temperature measured by the sensor is displayed on serial console. If there is any change in temperature, then the temperature measured will be updated and displayed on the serial console.

Accelerometer application

An accelerometer is an electromechanical device that will measure acceleration forces. On Revision 1.1A EVM (Beta EVM) an accelerometer with part number LIS331DLH is present. This can be accessed through I2C/SPI. The StarterWare accelerometer example application measures the angle of tilt of the EVM with respect to earth. This is done by measuring the amount of static acceleration due to gravity.

Steps to measure the acceleration along x, y and z axis.

- Acceleration experienced by the device at different axes is measured from OUT_L and OUT_H register of accelerometer device. For example acceleration along Z axis is measured by reading the data present in OUT_Z_L and OUT_Z_H. Similarly for Y and X axis.
- Measured data(OUT_Z_H + OUT_Z_L) is shifted by 4. This is because of the 12bit representation of the device.
- One bit of the measured data corresponds to (1/1024)g i.e 0.9mg.
- So by multiplying the ((OUT_Z_H + OUT_Z_L) >> 4) by 0.9mg gives the acceleration along the Z axis. Similarly for y and x axis.
- Cosine(acceleration measured along z axis / g) gives the angle by which device is tilted with respect to earth.
- The direction to which device is tilted is interpreted based on the sign of the data measured along a axis.For example when device tilted by 30 degree right,then data measured along z axis is positive value. Similarly when device tilted by 30 degree left, then data measured along z axis is negative value.

Note: 'g' is acceleration due to gravity. 1g corresponds to 1024 digital value.

Executing the Example application

- For TI AM335x EVM
  - Connect the serial port on the baseboard to the host serial port via a NULL modem cable.
- For EVM-SK
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.

A serial terminal application should be running on the host.

- Modules used in this example
  - I2C-1
  - Interrupt
  - UART-0

When the accelerometer example application is loaded and executed on the target, the angle of tilt of EVM is displayed on serial console.

McSPI

Introduction

McSPI is a general-purpose receive/transmit, master/slave controller that can interface with up to four slave external devices or one single external master. It allows a duplex, synchronous, serial communication between CPU and SPI compliant external devices (Slaves and Masters). It supports maximum frequency of 48MHz. McSPI could be configured to generate DMA event to EDMA controller for transfer of data.

The programming sequence for McSPI can be found here.

Executing the Example application

Set the EVM in profile 2 (SW8[1] = OFF, SW8[2] = ON, SW8[3:4] = OFF). For more details refer to EVM reference manual [3]. Connect the serial port on the baseboard to the host serial port via a NULL modem cable. A serial terminal application should be running on the host. Certain data is written to SPI flash using SPI bus. Then, the written data is read back. The read data is compared with the data that was written. If they match, then an appropriate message gets displayed on the serial communication console. The same functionality is demonstrated in both DMA and interrupt mode of operation.

- Modules used in McSPI-Interrupt application
  - McSPI-0
  - UART-0
  - Interrupt Controller
- Modules used in McSPI-EDMA application
  - McSPI-0
  - UART-0
  - EDMA-0
  - Interrupt Controller

Note

- Slave mode of McSPI controller is not supported in StarterWare.

DCAN

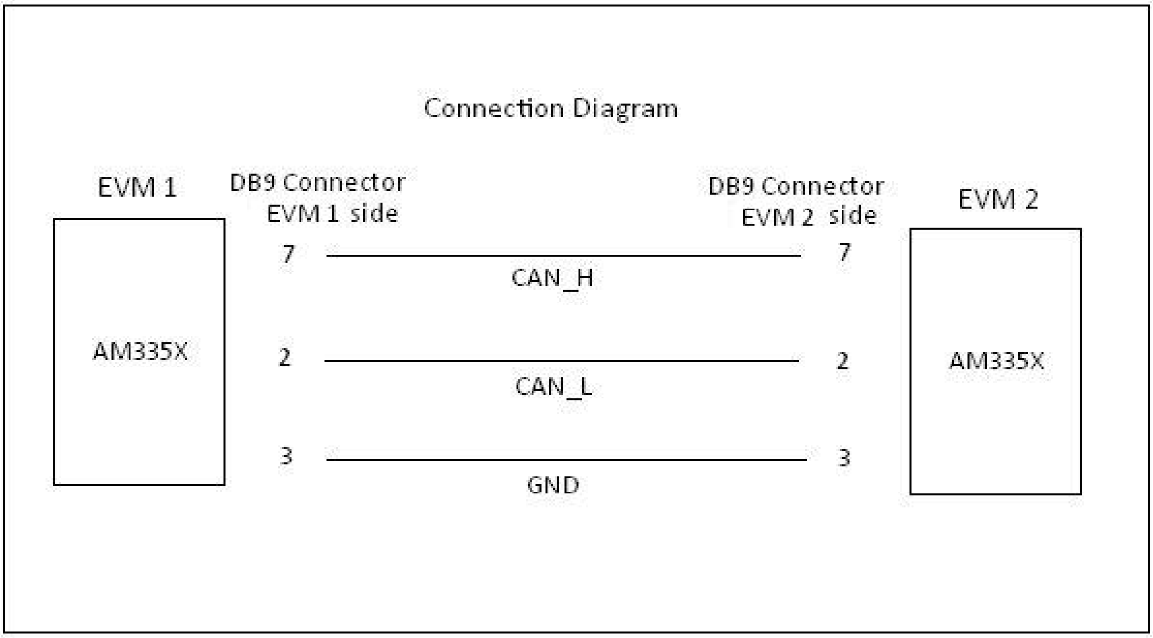
Introduction

The Controller Area Network is a serial communications protocol which efficiently supports distributed real time control with high level of security. The DCAN module present in the AM335x supports bitrates up to 1 Mbit/s and is compliant to the CAN 2.0B protocol specification. The core IP within DCAN is provided by Bosch.

The programming sequence for DCAN can be found here.

Executing the Board-To-Board example application

The example application demonstrates board-to-board CAN communication. The setup requirements are,



- Two TI AM335x EVMs in profile 1 (SW8[1] = ON, SW8[2:4] = OFF). (Lets call them as EVM1 and EVM2)
- Connect the two EVMs using DB9F-DB9F straight cable and details can be found from digikey [2] website and corresponding datasheet [3]
- On AM335x Beta EVM CAN1 is available on J11(DB9 connector) port. The CANH and CANL pins of the CAN transceiver are connected to pins 7 and 2 of J11 respectively. Pin 3 of J11 is ground. Connect the respective pins(7-7, 2-2, 3-3) on both boards as shown in the Connection diagram.
- Connect the serial port on the baseboard to the host serial port via a NULL modem cable for both the boards.
- A serial terminal application should be running on the host. Let us call the console where EVM1 and EVM2 display messages as Console1 and Console2 respectively.

Execute DCANTxRx application on EVM1 and DCANLpBk application on EVM2.

The below message will be printed on Console1
Please Input
1)Transmit a single data frame
2)Transmit 'n' data frames

And, the below message will be printed on Console2
**** Waiting for data ****

On Console1 if option 1 is selected by entering '1' then it will prompt to input bytes as
Please input data not exceeding 8 characters

On entering 1-8 data bytes the data frame will be transmitted from EVM1 -> EVM2 -> EVM1.

On reception the data will be printed on the serial terminals for verification.

If option 2 is selected on Console1 then ‘n’ data frames/messages can be transmitted, where data(8 bytes) will be populated by the application itself.

- Modules used in DCAN board-board application
  - DCAN-1
  - UART-0
  - Interrupt Controller

Note

- The remote enable feature of DCAN for a transmit message object is not supported in StarterWare.
- The receive message objects configured in DCANTxRx and DCANLpBk applications are provided with acceptance filtering. Hence these receive message objects can receive CAN frames with any valid IDs.
- The applications DCANTxRx and DCANLpBk by default, will transmit and receive CAN frames with standard IDs. On undefining the macro DCAN_STD_ID in DCANTxRx application, both the applications will transmit and receive CAN frames with extended Ids.

DMTimer

Introduction

DMTimer is a 32 bit timer and the module contains a free running upward counter with auto reload capability on overflow. The timer counter can be read and written in real-time (while counting). The timer module includes compare logic to allow an interrupt event on a programmable counter matching value. A dedicated output signal can be pulsed or toggled on overflow and match event.

The programming sequence for DMTimer can be found here.

Executing The Example Application

- For TI AM335x EVM
  - Connect the serial port on the baseboard to the host serial port via a NULL modem cable.
- For Beaglebone
  - The board has a USB-to-serial port. Connect this mini USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- For EVM-SK
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.

A serial terminal application should be running on the host.

- Modules used in this example
  - DMTimer-2
  - UART-0
  - Interrupt Controller

The example application demonstrates the use of timer as a countdown timer, counting down from 9 to 0. When the example application is executed, a string "Tencounter:" is displayed on the serial console. After this it starts to count down from 9 to 0.

Note

- DMTimer 1ms is not supported in StarterWare
- PWM signal generation as well as Capture mode of DMTimer is not supported in StarterWare

WatchDog Timer

Introduction

The watchdog timer is an upward counter capable of generating a pulse on the reset pin and an interrupt to the device system modules following an overflow condition. The watchdog timer serves resets to the PRCM module and serves watchdog interrupts to the host ARM. The reset of the PRCM module causes warm reset of the device. The default state of the watchdog timer is enabled and not running.

The programming sequence for Watchdog Timer can be found here.

Executing The Example Application

- For TI AM335x EVM
  - Connect the serial port on the baseboard to the host serial port via a NULL modem cable.
- For Beaglebone
  - The board has a USB-to-serial port. Connect this mini USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- For EVM-SK
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- A serial terminal application should be running on the host.

The example application wdtReset demonstrates the use of WatchDog Timer. When the example application is executed, a string "Program Reset! Input any key at least once in every 4 seconds to avoid a further reset." will appear on the screen. If no key is input, program will restart within few seconds.

- Modules Used
  - Watchdog timer-1
  - UART-0

Raster LCD

Introduction

Raster LCD Controller is used to display image on LCD panel. Raster LCD Controller is a synchronous LCD interface. It supports 1/2/4/8/16/24 bits per pixel (bpp) configuration in packed or unpacked mode.

The programming sequence for Raster LCD can be found here.

Executing the Example Application

The example application configures the raster to display image having 24bpp and stored in unpacked mode.

Before executing the raster LCD program, make sure that raster LCD is hooked on to the board. When the program is executed, the image will be displayed on LCD.

- Modules used in this example
  - LCD-0
  - Interrupt Controller

The example application works in double frame buffer mode. The ISR handles only End-Of-frame interrupt. The frame buffer registers are updated in each End-of-frame interrupt.

Note: While using dual frame buffers in Raster, before toggling to a new updated frame buffer, a cache clean has to be performed. The Raster example displays a single image. Though two frame buffers are updated with the same data, a cache clean is not performed between the updations. However the Cache Clean operation is demonstrated in OOB demo example in connection with the usage of dual frame buffers. Refer to the function toggleFrameBuffer() in demoRaster.c present in demo example.

Note
The resolution of the LCD panels used on the EVM and EVM-SK are given below:

- TI AM335x EVM : 800 x 480
- EVM-SK : 480 x 272

EHRPWM

Introduction

The enhanced High Resolution PWM module in AM335x is able to generate complex pulse width waveforms with minimal CPU overhead or intervention. The ePWM module represents one complete PWM channel composed of two PWM outputs: EPWMxA and EPWMxB. Each ePWM instance is identical with one exception. Some instances include a hardware extension that allows more precise control of the PWM outputs. This extension is the high-resolution pulse width modulator. The programming sequence for EHRPWM can be found here.

- Modules used in this example
  - EHRPWM

Example Application

The example application is provided for only TI AM335x EVM. The example application demonstrates the rotation of Haptics Motor present on the Daughter Card.

- Notes
  - The EVM should have the CPLD programmed for controlling Haptics Motor via eHRPWM.
  - The EVM has to be kept in profile 4.

Touchscreen

Introduction

The touchscreen module is an 8 channel general purpose ADC,with optional support for interleaving Touch screen conversation for 4-wire,5-wire, or 8-wire resistive panel. It also has a programmable FSM (Finite State Machine) Sequencer that supports 16 steps. A step is a general term for describing which input values to send to the AFE (Analog Front End)and how, when and which channel to sample. For more information on steps, please refer to Touchscreen chapter in AM335x TRM.

The programming sequence for Touchscreen can be found here.

Executing the Example Application

- For TI AM335x EVM
  - Connect the serial port on the baseboard to the host serial port via a NULL modem cable.
- For EVM-SK
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- A serial terminal application should be running on the host.
- Modules used in this example.
  - ADC/Touchscreen Controller.
  - UART-0
  - Interrupt Controller

When the example application is loaded on the target and executed, in order to calibrate, it will ask the user to touch at three different corners as prompted on the serial console. Once calibration is done, then whenever there is a touch on the panel, coordinates are displayed on the serial console until the touch is released.

ADC

Introduction

The touchscreen/ADC module is an 8 channel general purpose ADC,with optional support for interleaving Touchscreen conversion for a 4-wire, 5-wire, or 8-wire resistive panel. It also has a programmable FSM sequencer that supports 16 steps. A step is a general term for describing which input values to send to the AFE and how, when and which channel to sample. For more information on step, please refer to Touchscreen/ADC chapter in AM335x Technical Reference Manual(TRM).

Programing sequence for ADC can be found here.

Example Application

- For TI AM335x EVM
  - Connect the serial port on the baseboard to the host serial port via a NULL modem cable.
- For EVM-SK
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- A serial terminal application should be running on the host.
- Modules used in this example
  - ADC/Touchscreen controller
  - UART-0
  - Interrupt Controller

When the example application is loaded on the target and executed, the voltages measured across the AN0 and AN1 lines are displayed on the serial console. The example application pulls up the AN0 line and pulls down the AN1 line by configuring the internal AFE transistors. Thus voltage across the AN0 is 1.8 and voltage across AN1 is 0.

GPIO

Introduction

Each GPIO module provides 32 dedicated general-purpose pins with input and output capabilities. These pins can be configured for the following applications:

- Data input (capture)/output (drive)
- Keyboard interface with a debounce cell
- Interrupt generation in active mode upon the detection of external events. Detected events are processed by two parallel independent interrupt-generation submodules to support biprocessor operations.

The APIs are exported in include/gpio_v2.h

The programming sequence for GPIO can be found here.

LCD Back-light

- The example is demonstrated for TI AM335x EVM.
- The example application switches the LCD back-light ON/OFF periodically.
- Module Used
  - GPIO-0

LED Blink

- The example is demonstrated for Beagle Bone.
- When the example application is executed, a user LED on the EVM blinks periodically.
- Module Used
  - GPIO-0

MMCSD Card Detection

- The example is demonstrated for EVM-SK.
- The application notifies over UART Console on the event of an MMCSD card insertion/ejection.

Executing the application

- The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- A serial terminal application should be running on the host.
- Modules used
  - GPIO0 and GPIO1
  - UART0
  - Interrupt

Audio Buzzer

- The example is demonstrated for TI AM335X EVM.
- The Audio buzzer present on the General Purpose Daughter board is 'AI_1027' from PUI Audio Inc. A GPIO pin controls the ON and OFF of a transistor switch that is present in series with the audio buzzer. The audio buzzer shall rotate when the transistor in ON (circuit is closed) and stops when the transistor is OFF (circuit is open).

Executing the application

This application turns the Haptics Motor ON and OFF alternately for periodic intervals. It uses Timer peripheral to interrupt the processor after a certain period has elapsed. The modules used in this example are:

- GPIO-1
- DMTimer-7
- Interrupt

RTC

Introduction

The RTC provides a time reference to an application running on the device. The current date and time is tracked in a set of counter registers that update once per second. The time can be represented in 12-hour or 24-hour mode. The calendar and time registers are buffered during reads and writes so that updates do not interfere with the accuracy of the time and date. Alarms are available to interrupt the CPU at a particular time, or at periodic time intervals, such as once per minute or once per day. In addition, the RTC can interrupt the CPU every time the calendar and time registers are updated, or at programmable periodic intervals.

The APIs are exported in include/rtc.h

The programming sequence for RTC can be found here.

Example Application

- For TI AM335x EVM
  - Connect the serial port on the baseboard to the host serial port via a NULL modem cable.
- For Beaglebone
  - The board has a USB-to-serial port. Connect this mini USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- For EVM-SK
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.

A serial terminal application should be running on the host.

- Modules used in this example
  - RTC
  - UART-0
  - Interrupt Controller
- On running the application, the user sees a request on the terminal to enter time and calendar information. On entering the information, the application programs these time and calendar information in the respective registers of RTC. The time and calendar information currently held by the RTC registers are displayed on the terminal.

Memory Devices

AM335x is integrated with the GPMC(General Purpose Memory Controller) to which NAND is interfaced and MMC controller through which MMC/SD is accessed. The StarterWare package contains the device abstraction layers (DAL) for these peripherals and example applications to demonstrate the same.

MMC/SD

AM335X devices have multimedia card high-speed/secure data/secure digital I/O (MMC/SD/SDIO) host controller, which provides an interface between microprocessor and either MMC, SD memory cards, or SDIO cards.

The programming sequence for MMC controller can be found here.

The StarterWare MMC/SD driver design overview can be found here.

The example application provided as part of the package demonstrates the use of MMC/SD card with FAT filesystem. The application only supports reading from the card and basic shell like interface is provided for user experience. ***Only SD cards are supported***

Preparing the SD card

The SD card needs to be prepared, by FAT formatting it as follows.

1. Choose a SD card and a USB based or similar SD card reader/writer.Plug it to a Windows host system.
2. Run the ***TI_SDCard_boot_utility_v1_0.exe*** executable (which is in release package tools/sd_format/ directory). Select the SD Card drive name, location of MLO file and application image path.
3. Click on 'Proceed'.
4. After the formatting is complete, the card is ready to be populated with the files required.
5. Copy any files into the newly formed file system.
6. Safely eject/remove the card from the host, unplug the card reader, remove the SD card. The SD card is ready for use.

Executing the example application

- For TI AM335X EVM
  - Set the EVM in profile 0 (SW8[1:4] = OFF). For more details refer to EVM reference manual [3].
  - Connect the serial port on the baseboard to the host serial port via a NULL modem cable.
- For Beaglebone
  - The board has a USB-to-serial port. Connect this mini USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- For EVM-SK
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
- A serial terminal application should be running on the host.

1. Using CCS with appropriate target configuration and GEL files, connect to the target AM335x.
2. Insert the card into the base board MMC/SD slot.
3. Load the application ELF binary (.out) on the target via CCS.
4. Click "Go" or "Run".
5. A shell like interface comes up on the serial console.
6. Following commands are support in this interface
   1. help - Displays the help contents, available commands

2. ls - lists the current directory
3. chdir - Change the current/present working directory
4. mkdir - Create a new directory
5. pwd - shows the current/present working directory
6. cd - Change the current/present working directory
7. cat - Write or dump, the content of a file or input from UART (through xmodem), to a file or on console respectively (A text file is preferred, since binary/non-ascii files may corrupt the serial console display with garbage)
8. rm - Remove a file or empty directory

- Modules used in this example
  - MMCHS-0
  - MMC/SD Library
  - FAT File System
  - UART-0

NAND

AM335x have GPMC(General Purpose Memory Controller) to which NAND is interfaced . GPMC an unified memory controller to interface external memory devices like NAND, NOR, Asynchronous SRAM etc. By configuring the bit fields in the GPMC registers, the application can be able to access the mentioned type of device through GPMC.

The programming sequence for GPMC can be found here.

Along with GPMC, ELM module is used to support error calculation and correction capability.

The programming sequence for ELM for error calculation can be found here.

The StarterWare NAND driver design overview can be found here.

The example application provided as part of the package demonstrates the use of NAND. The application writes default data pattern (to the user specified block, page for number of pages) and read the data and checks for the data integrity. If the macro NAND_DATAINTEGRITY_TEST_WITH_FIXED_ADDR is defined, application does the erase, write and read for default block and pages. Also, By default application uses BCH 8-bit as the ECC type and Polled mode as operating mode. To change the ECC type, operating mode(polled or DMA)etc change the corresponding field in the nandCtrlInfo, nandDevInfo data object(s) before controller initialization.

Executing the example application

1. Set the EVM in profile 0 (SW8[1:4] = OFF). For more details refer to EVM reference manual [3].
2. Connect the serial port on the baseboard to the host serial port via a NULL modem cable. A serial terminal application should be running on the host.
3. Using CCS with appropriate target configuration and GEL files, connect to the target AM335x.
4. Load the application ELF binary (.out) on the target via CCS.
5. Click "Go" or "Run".
6. NAND Device Info is printed on the serial console and asks for the user to enter block, page number and number of pages information.
7. Then application does the following ---
   1. Checks whether block is bad or not.
   2. If not erases the block.
   3. Writes the data with ECC enabled.
   4. Write the ECC data to the spare area.

5. Reads the data with ECC enabled and checks for the ECC errors.
6. If any ECC errors, and these are correctable corrects the data else prints the error message.
7. Checks for the data integrity.
8. Repeat the above steps for user entered number of pages.

- Modules used in this example
  - GPMC-0
  - EDMA
  - Interrupt Controller
  - UART-0

Ethernet

Introduction

The AM335x uses CPSW (Common Platform Ethernet Switch) for ethernet interface. The peripheral is compliant to IEEE 802.3 standard, describing the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer specifications. CPSW has one host port and two slave ports, each of which are capable of 10/100/1000 Mbps with MII/GMII/RGMII interfaces. The CPSW also has an Address Lookup Engine which processes all received packets to determine which port(s) if any that the packet should the forwarded to. The ALE uses the incoming packet received port number, destination address, source address, length/type, and VLAN information to determine how the packet should be forwarded. The CPSW incorporates an 8kB internal RAM to hold CPDMA buffer descriptors (also known as CPPI RAM). The MDIO module implements the 802.3 serial management interface to interrogate and control up to 32 Ethernet PHYs connected to the device by using a shared two-wire bus. The application shall use the MDIO module to configure the auto negotiation parameters of each PHY attached to the CPSW slave ports, retrieve the negotiation results, and configure required parameters in the CPSW module for correct operation.

The programming sequence for CPSW can be found here.

The overview on StarterWare CPSW port for lwIP can be found here.

- StarterWare supports both Switch Mode and Dual MAC mode for CPSW. In Switch Mode, any one of the two slave ports can be connected to a network and the host connected to the other port will be able to access the network. In Dual MAC mode, both ports can be connected to same/different networks and each will acquire different IP addresses. Both modes can be demonstrated in EVM-SK
- StarterWare enet_lwip example provides interface for CPSW switch configurations. Refer here for the supported CPSW Switch configurations.

Example applications

The Ethernet examples for Beagle Bone demonstrates MII interface and TI AM335x EVM / EVM-SK demonstrates RGMII interface operating at 100Mbps.

- For TI AM335X EVM
  - Set the EVM in profile 0 (SW8[1:4] = OFF). For more details refer to EVM reference manual [3].
  - Connect the serial port on the baseboard to the host serial port via a NULL modem cable.
  - Connect the Ethernet port on the baseboard of the EVM to a LAN port.
- For Beaglebone
  - The board has a USB-to-serial port. Connect this mini USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.

  - Connect the Ethernet port on the board to a port on the LAN.
- For EVM-SK
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
  - By default, applications configure CPSW to operate in Dual MAC mode where both ports can be connected to same/different networks and each will acquire different IP addresses.
  - If the CPSW configuration is to be changed to switch mode, the macro CPSW_DUAL_MAC_MODE shall be undefined in lwipopts.h file, which is part of the application. By configuring CPSW to switch mode, any one of the two slave ports can be connected to a network and the host connected to the other port will be able to access the network.
- A serial terminal application should be running on the host.

The StarterWare AM335X ethernet application over lwIP stack is demonstrated using two applications.

1. An embedded web server application, which hosts a default page when requested
2. An echo server application, which demonstrates a simple data transfer between a client and server.

- Modules used in this example
  - CPSW
  - MDIO
  - PHY
  - Interrupt Controller
  - UART-0
  - lwIP Stack

In the following examples, the IP address can be configured in *enet_lwip/include/lwipopts.h*. The macro STATIC_IP_ADDRESS shall specify the static IP address to be used. If a dynamic IP address to be used, STATIC_IP_ADDRESS shall be defined as 0.

Embedded Web Server application

A sample http server application is demonstrated, using lwIP stack. This is located at */examples/evmAM335x/enet_lwip/*.

A serial terminal application should be running on the host. The example uses a serial console to display the dynamic IP address assigned for the EVM by the DHCP server. Flash the Ethernet example application or load the executable ELF (.out) file on to the EVM.

- Testing by connecting peer to peer with a host machine:
  - Connect the Ethernet port on board to the host Ethernet port via an Ethernet cable
  - Assign a static IP address to the host machine.
  - Run a DHCP server application on the host.
  - Execute the example application
  - Note the dynamic IP address assigned which displayed on the serial console.
  - Access the http server application default page using `http://<ip_address>/index.html` via a web browser on the host.
  - Ensure that proxy server is not used for the dynamic IP address assigned for the board.
- Testing by connecting to a corporate network:
  - Connect the Ethernet port on board to a port on the corporate network.
  - Execute the example application.
  - Note the dynamic IP address assigned which displayed on the serial console.

- Access the http server application default page using `http://<ip_address>/index.html` via a web browser on the host.
- Ensure that proxy server is not used for the dynamic IP address assigned for the board.

Echo server application

A sample echo server-client set up is demonstrated, using lwIP stack. The echo server, which runs on the target just echos back the received data to the sender - typically the client running on a Linux host in this case. The client then compares the sent and received data for integrity and the result is printed on the client console. The client application is also delivered as part of package and is located at *StarterWare_xx_yy_mm_pp/host_apps/enet_client*.

A serial terminal application should be running on the host. The example uses a serial console to display the dynamic IP address assigned for the EVM by the DHCP server. Flash the Ethernet example application or load the executable ELF (.out) file on to the EVM.

- Testing by connecting peer to peer with a host machine:
  - Connect the Ethernet port on board to the host Ethernet port via an Ethernet cable
  - Assign a static IP address to the host machine.
  - Run a DHCP server application on the host.
  - Execute the example application on the target
- Testing by connecting to a corporate network:
  - Connect the Ethernet port on board to a port on the corporate network.
  - Execute the example application on the target.
  - Note the dynamic IP address assigned which displayed on the serial console.

Now execute the client application on the host.

```
$ ./client <ip-address-announced-by-the-echo-server>
```

The client prints the status of the application on the console

makefsfile utility

'makefsfile' may be used to create file system images to embed in ethernet applications offering web server interfaces. It can be used to generate an ASCII C file containing initialized data structures, which represents the html pages which need to be embedded in the application. 'makefsfile' is at "tools/makefsfile/", provided in source and binary form. Executing the binary without any input provides a detailed help menu which guides the user.

Executing makefsfile utility

$ ./makefsfile
This prints a detailed help menu

$./makefsfile -i <directory-path>
This takes an input directory path which contains the saved html pages which need to be converted to a C file which can be embedded in an application. The file fsdata.c will be generated inside directory from where makefsfile is executed.

McASP

Introduction

Audio in StarterWare is played via the Multichannel Audio Serial Port (McASP). The McASP functions as a general-purpose audio serial port optimized for the needs of multichannel audio applications. The McASP can be used for time division multiplexed (TDM) stream, Inter-IC Sound (I2S) protocols, and inter component digital audio interface transmission (DIT). The McASP has separate transmit and receive sections that can operate synchronously or independently. The McASP can be configured for external or internal clock and frame sync signals. It has 16 serialisers each of which can be configured as transmitter or as a receiver.

The APIs of McASP are exported to include/mcasp.h

The programming sequence for McASP can be found here.

Example application

The McASP example demonstrates audio data transmission and reception in I2S mode using DMA.

- For TI AM335x EVM
  - The audio input on the LINE IN is looped back to the audio ouput on LINE OUT of the EVM.
  - Set the EVM to profile 0 (SW8[1:4] = OFF). For more details refer to EVM reference manual [3].
  - Plug in a 3.5mm audio jack which takes analog audio signals into the audio LINE IN of the EVM. Also, plug a 3.5mm audio jack which is connected to a headphone or speakers into the audio LINE OUT of the EVM.
- For EVM-SK
  - A stored audio tone will be ouput on LINE OUT of the EVM.
  - Plug a 3.5mm audio jack which is connected to a headphone or speakers into the audio LINE OUT of the EVM.
- Modules used in this example
  - McASP-1
  - EDMA
  - I2C-1
  - Interrupt Controller

More information about the Audio application can be found here.

USB

AM335x has two integrated Mentor Graphics USB controllers(USB0 and USB1) with external PHY. The MUSB controller supports both host and device functionalities with OTG capability. StarterWare USB package provides all the necessary software support for the MUSB controller which includes, Device Abstraction Layer (DAL), the USB Stack for CDC,MSC and custom Bulk Device class and the sample application. More about starterware USB can be found here.

Power Management

StarterWare demonstrates different power management modes in OOB Demo example application.For more details please refer here.

Graphics

The graphics library(grlib) is reused from StellarisWare. StarterWare delivers graphics examples based on raster device abstraction layer (DAL) APIs which can be used as reference and/or adapted for user's requirements. For more information on graphics library refer here [4].

Neon VFP

Starterware provides support for Neon and VFP coprocessors included in the ARM Cortex A8. For more information on the support provided refer here [5].

Semihosting

For details on what is semihosting and how to enable the same in StarterWare please refer StarterWare Semihosting.

Out-Of-Box Demo Application

Introduction

The out-of-box demo application demonstrates the capabilities of the device abstraction layer of StarterWare. The application executable can be found in binary/armv7a/gcc/am335x/<evmAM335x-evmAM335x-beablebone>/demo/. The demo application can be navigated through Touchscreen and/or Ethernet and/or Console.

- The application demonstrates the below listed peripherals.
  - Ethernet
  - Audio (NA for Beaglebone)
  - Real Time Clock
  - Timer
  - UART
  - Raster (NA for Beaglebone)
  - Touchscreen (NA for Beaglebone)
  - Accelerometer (NA for Beaglebone)
  - Audio Buzzer (NA for EVM-SK & Beaglebone)
  - Temperature Sensor (NA for EVM-SK & Beaglebone)
- The demo application also supports Dynamic Voltage Frequency Scaling(DVFS). The desired OPP can be selected and the demo application executes further in the selected OPP. Different OPPs supported and the associated Voltage and ARM Core Operating Frequencies are listed in the below table.

	PG1.0 Ver		PG2.x Ver	
	Voltage	Frequency	Voltage	Frequency
OPP50	1.1V +/- 4%	275 MHz	0.95V +/- 4%	300 MHz
OPP100	1.1V +/- 4%	500 MHz	1.1V +/- 4%	600 MHz
OPP120	1.2V +/- 4%	600 MHz	1.2V +/- 4%	720 MHz
SR Turbo	1.26V +/- 4%	720 MHz	1.26V +/- 4%	800 MHz
SR Turbo	NA	NA	1.325V +/- 4%	1000 MHz

- The application also demonstrates different power saving modes. The supported power saving modes are

1. Deep Sleep 0,
2. Deep Sleep 1,
3. RTC Only

Different wake sources can be selected for waking up from the power down mode. The wake sources supported are Touchscreen (not supported on Beaglebone), Timer1, UART0, GPIO0_2 (Push Button SW9 on EVM, SW3 on EVM-SK, Not Supported on Beaglebone), RTC Alarm. For more information on power consumption refer here [6].

Modules Used

The modules used in the out-of-box demo application are listed below.

- LCD-0
- ADC/Touchscreen Controller
- CPSW
- MDIO
- PHY
- lwIP Stack
- McASP-1
- EDMA
- UART-0
- I2C-0,1
- DMTimer-2
- ECAP-0
- RTC
- Interrupt Controller
- Graphics Library

Design overview

The out-of-box demo application combines functionality of multiple peripherals to demonstrate StarterWare capabilities to be used for various use case scenarios. The application is designed to be driven by both Touch and Ethernet. The programming sequence is given below.

- Enable the module clock and pin multiplexing for the peripherals used.
- Initialize the AINTC, register all the interrupt handlers, enable the interrupts at AINTC
- Initialize the required peripherals and enable peripheral level interrupts.
- Display the banner image
- Start playing the audio tone. This tone will be looped forever.
- Detect a touch on the LCD or a click on the embedded page accessed via ethernet.

- If a touch is detected, validate the coordinates. If the coordinates are verified, display the proper image and demonstrate the peripheral.
- If a click is detected, display the proper image and demonstrate the peripheral.

The application maintains a list of contexts which include

- The image to display
- Number of Icons in the image
- Specification for each Icon in the image. The specification includes
  - Valid coordinates of an Icon
  - The action to be taken when the valid coordinates are touched.

If a touch is detected in the current context, the touch coordinates are validated based on the specification of each Icon in the image, and the corresponding action will be taken.

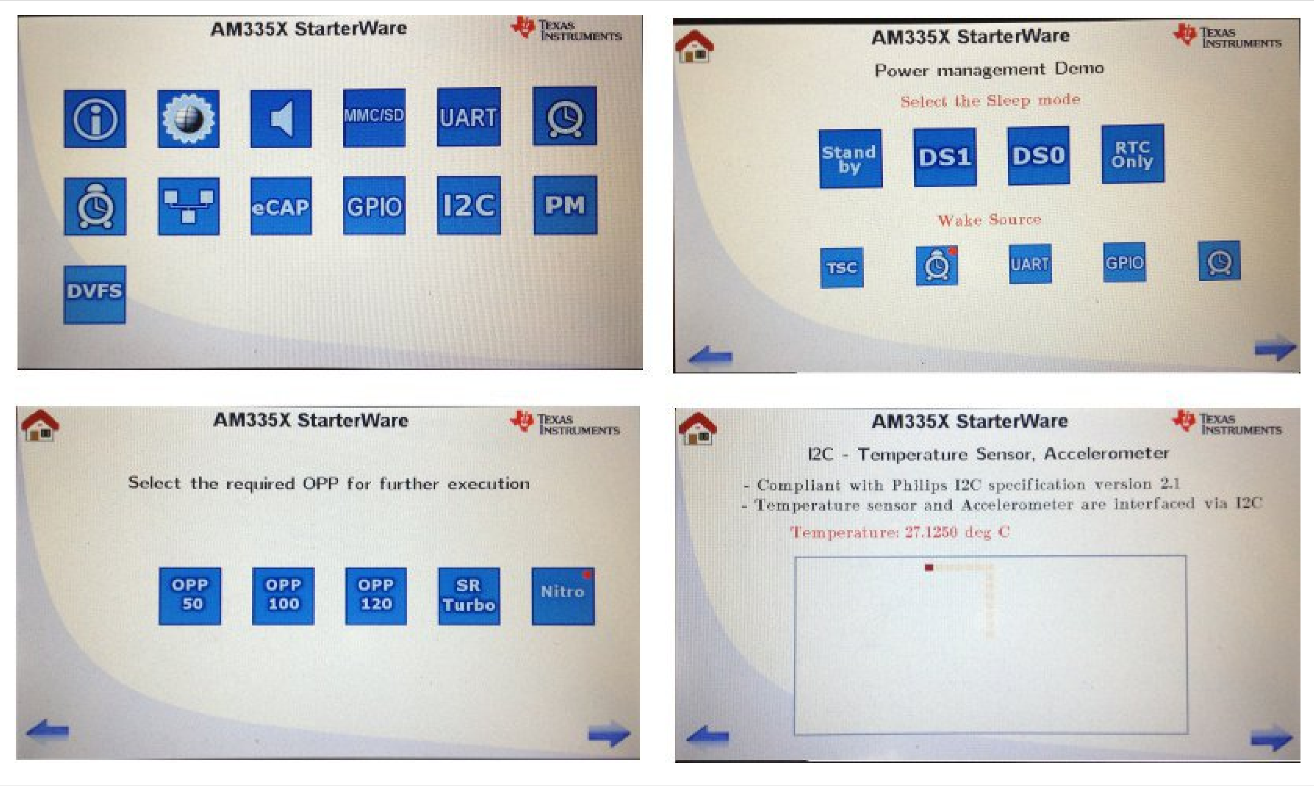
When a button on the embedded page is clicked, the click information will be updated by the CGI handler which is registered during the initialization of the http server. Based on this information, the current context will be updated to display the proper image and demonstrate the peripheral.

Executing The Application

Set Up Requirments

- A serial terminal application should be running on the host.
- *For Beagle Bone*
  - The mini USB port to be connected to the host. This mini USB connection is used for displaying messages on the serial console on the host, if the port is properly selected.
  - Ethernet port on board connected to a port on the LAN.
  - The Demo Application for Beagle Bone can be driven via Console/Ethernet. On booting the demo application the dynamic IP address assigned to Beagle Bone will be displayed on serial console. The embedded web page can be accessed anytime using http://<ip address>/index.html via a web browser on the host. Ensure that proxy server is not used for the dynamic IP address assigned for the board.
- *For EVM-SK*
  - The board has a USB-to-serial port. Connect this micro USB port to the host via a USB cable. This connection will help to display messages on the serial console on the host. Ensure that the USB-to-serial driver is installed on the host machine and proper port is selected in the host serial terminal application.
  - Plug a LCD(Raster) panel to the EVM.
  - Ethernet port on the base board connected to a port on the LAN.
  - Audio LINE OUT of the EVM connected to headphone/speakers with 3.5mm audio jack.
- *For TI AM335X EVM*
  - The serial port on the baseboard of the EVM to be connected to the host serial port via a NULL modem cable.
  - LCD(Raster) module to be plugged into the EVM
  - Ethernet port on the base board connected to a port on the LAN.
  - Audio LINE OUT of the EVM connected to headphone/speakers with 3.5mm audio jack.
  - Some snaps from EVM OOB demo.

```
Note: SPI boot mode is not supported for EVM. SPI boot mode requires
Profile2 configuration but peripherals used in EVM OOB need Profile0
configuration.
```



- The Demo Application for TI AM335X can be driven via Touch and/or Ethernet. On booting the demo application on TI AM335X EVM, a banner will be displayed on the LCD, followed by an introductory slide. If the demo is to be driven via Ethernet, please note the dynamic IP address displayed on the serial console. The embedded web page can be accessed anytime using http://<ip address>/index.html via a web browser on the host. Ensure that proxy server is not used for the dynamic IP address assigned for the board.

Memory Usage Statistics

This section provides the code and data section for every device abstraction layer library object and the executables of TI AM335x EVM, generated using TMS470 Code Generation Tools. Note that the code is compiled for ARM instruction set.

- Device Abstraction Layer

DAL object	text size (bytes)	data size (bytes)
watchdog	544	0
usbphyGS70	72	0
usb	3,484	0
uart_irda_cir	3,724	0
tsc_adc	1,956	0
rtc	2,760	0
raster	1,744	0
phy	848	0
mdio	220	0
mcspi	1,384	0
mcasp	1,648	0

mailbox	372	0
hsi2c	1,224	0
hs_mmcsd	1,408	0
gpmc	3,260	0
gpio_v2	1,112	0
elm	888	0
ehrpwm	1396	0
edma	2,784	4
ecap	684	0
dmtimer	1100	0
dcan	2,068	0
cpsw	2,300	0
cppi41dma	4,460	353,568

- Libraries

Library	text size (bytes)	data size (bytes)
graphics	48043	543075
mmcsd	2728	128
nand	5536	0
usb	20916	60941
utils	4993	1072
system_config	5316	12
platform	11140	968

- Executables

Binary	Code (B)	Data (B)	BSS (B)	Const(B)	Notes
uartEcho	16,324	56	513	56	
i2cAccelerometer	19,720	364	539	56	
adcVoltMeasure	18,636	4	529	56	
buzzerBeep	11,084	4	512	56	
boot	42,508	136	2044	8	
uartEdma_Cache_Cache	15,140	4	17,188	140	
uartEdma_Cache_CacheFlush	15,164	4	17,188	140	
dcanLpBk	20,892	48	537	56	
dcanTxRx	21,700	64	537	96	
demo	148,084	20,460	5,151,876	1,163,164	Includes buffers for peripherals context save/restore, audio raw tone and image data
dmtimerCounter	18,116	8	513	56	
edma	20,500	0	913	56	

enetEcho	63,976	24	483,264	416	
enetLwip	65,900	24	480,384	4,150	
gpioLCDBacklight	7,804	0	512	56	
ehrpwm_haptics	7,428	0	512	56	
hsMmcSdRw	42,508	1308	85,912	1340	
hsi2cEeprom	16,580	4	585	56	
hsi2cEeprom_edma	19,940	10	569	56	
irqPreemption	21,416	4	512	56	
mcaspPlayBk	24,248	68	48,537	144	
mcspiFlash	19,244	12	1,297	56	
mcspiFlash_edma	23,328	2	1,549	56	
nandReadWrite	33,940	4	21,061	168	
neonVFPBenchmark	19,564	24	19,297	166	
rasterDisplay	12,096	4	532	1,536,088	Includes image data
rtcClock	20,752	0	513	56	
hsi2cTempSensor	16,936	4	533	56	
tscCalibrate	22,920	8	565	80	
uartEcho_edma	20,220	176	824	56	
usb_dev_bulk	47,408	188	1,556,480	5,865	Includes graphics frame buffer memory
usb_dev_comp_mouse_ser	57,156	688	1,556,480	8,321	
usb_dev_comp_ser_ser	43,252	488	1,556,480	5,997	
usb_dev_mouse	40,828	268	1,556,480	5,529	Includes graphics frame buffer memory
usb_dev_msc	50,760	200	2.7E+08	3,270	Includes 256 MB ram space allocated for usage as USB device memory. Also Includes graphics frame buffer memory.
usb_dev_serial	45,048	220	1,556,480	5,945	Includes graphics releated buffers
usb_device_host_msc	62,452	557	17,154,539	1160	
usb_host_mouse	50,712	188	1,909,788	2,784	Includes graphics frame buffer memory
usb_host_mouse_msc	65,560	529	1,911,214	3268	
usb_host_msc	53,072	16,881	376,832	1268	
wdtReset	15,832	0	513	56	
game	43,616	296	3,124,900	160,910	Includes raw audio file
grlib_demo	49,272	4428	1,580,480	90,381	Includes raw audio file

API Reference Guide

Driver library API Reference Guide provides information on APIs and is included in the release package (\docs).

References

[1] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0406b/index.html

[2] http://www.digikey.com/product-detail/en/AK152-2-R/AE9872-ND/821627?cur=USD

[3] http://www.assmann.us/specs/AK152-2-R.pdf

[4] http://processors.wiki.ti.com/index.php/StarterWare_Graphics

[5] http://processors.wiki.ti.com/index.php/StarterWare_NeonVFP

[6] http://processors.wiki.ti.com/index.php/AM335x_StarterWare_Power_management#Power_Consumption

StarterWare CPSW

The peripheral is compliant to IEEE 802.3 standard, describing the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer specifications. CPSW has one host port and two slave ports, each of which are capable of 10/100/1000 Mbps with MII/GMII/RGMII interfaces. The CPSW also has an Address Lookup Engine which processes all received packets to determine which port(s) if any that the packet should the forwarded to. The ALE uses the incoming packet received port number, destination address, source address, length/type, and VLAN information to determine how the packet should be forwarded. The CPSW incorporates an 8kB internal RAM to hold CPDMA buffer descriptors (also known as CPPI RAM). The MDIO module implements the 802.3 serial management interface to interrogate and control up to 32 Ethernet PHYs connected to the device by using a shared two-wire bus. The application shall use the MDIO module to configure the auto negotiation parameters of each PHY attached to the CPSW slave ports, retrieve the negotiation results, and configure required parameters in the CPSW module for correct operation.

The APIs for configuring and using CPSW, MDIO and generic PHYs are exported in /include/cpsw.h, /include/mdio.h and /include/phy.h respectively.

- Give proper settings for enabling CPSW interface at the chip configuration module.
  - Enable the pin multiplexing for CPSW by invoking the API ‘CPSWPinMuxSetup()’
  - According to the EVM settings, determine the MII transfer mode and enable the interface. In case of RGMII, ‘EVMPortRGMIIModeSelect()’ can be used to enable RGMII mode in the chip configuration
- Reset the CPSW hardware. Each submodule can be reset by calling the APIs ‘CPSWSSReset()’, ‘CPSWCPDMAReset()’ and ‘ CPSWWrReset()’.
- Initialize the MDIO Module using ‘MDIOInit()’. Enough delay shall be given to ensure the successful completion of the MDIO module initialization before any further access to MDIO.
- Initialize the ALE logic and clear the entries in the ALE table using ’CPSWALEInit()’
- Set the port states for each port used inside the ALE by invoking ‘CPSWALEPortStateSet()’
- Set the ALE table entries for unicast/multicast with the required Ethernet address (MAC address). The MAC address can be read by invoking the API ‘EVMMACAddrGet()’
- If port statistics need to be enabled, enable using ‘CPSWStatisticsEnable()’
- Initialize and configure the slave ports
  - Reset the sliver logic (which correspond to slave port), by invoking the API ‘CPSWSlReset()’
  - Set the MAC address for the slave ports using ‘CPSWPortSrcAddrSet()’
  - Auto negotiate with the PHY device connected through the MDIO, if the phy is alive. Respective PHY Auto negotiation API can be used for this.
  - According to the Auto negotiation results from the PHY, set the GMII/MII/RGMII mode and the duplex of transmission for CPSW using ‘CPSWSlTransferModeSet()’.

- Initialize the TX and RX buffer descriptors in the CPPI RAM, which is local to the CPSW.
- Enable the transmission and reception at CPDMA using ‘CPSWCPDMATxEnable()’ and CPSWCPDMARxEnable()’
- Enable the GMII/RGMII interface at CPSW by invoking ‘CPSWSlRGMIIEnable()’ for all the slave ports to be used
- Enable interrupt generation.
  - Enable the transmission/reception interrupt generation at CPDMA using ’CPSWCPDMATxIntEnable()’ and ‘ CPSWCPDMARxIntEnable()’
  - Enable the transmission/reception interrupt generation at CPSW wrapper module for the required core using ‘CPSWWrCoreIntEnable()’
- In the receive interrupt handler, the buffer descriptors associated with a received packet can be found using the SOP(Start Of Packet)/EOP(End Of Packet) fields of the buffer descriptor. After the packet is processed, the completion pointer shall be written using ‘CPSWCPDMARxCPWrite()’ to acknowledge the CPDMA. After the receive interrupt handler processes all the received packets, the CPSW shall acknowledge the end of interrupt processing with ‘CPSWCPDMAEndOfIntVectorWrite()’.
- In the transmit interrupt handler, the buffer descriptors associated with a transmitted packets shall be freed by writing, the completion pointer shall be written using ‘CPSWCPDMATxCPWrite()’ . After the transmit interrupt handler processes all the transmitted packets, the CPSW shall acknowledge the end of interrupt processing with ‘CPSWCPDMAEndOfIntVectorWrite()’.

Device Abstraction Layer (DAL) provides necessary register level API's required for configuration of Switch. CPSW port for lwIP give interface API to configure Switch. It encapsulates the implementation of the functionality. enet_lwip application provides Console interface to configure the CPSW Switch with available configurations.

Following ALE configurations are implemented in latest release of StarterWare

- Add a Multicast or VLAN/Multicast Entry to ALE Table
- Add a Unicast or VLAN/Unicast Entry to ALE Table
- Add OUI Entry to ALE Table
- Add a VLAN Entry to ALE Table
- Delete a Multicast or VLAN/Multicast Entry in ALE Table
- Delete Unicast or VLAN/Unicast Entry in ALE Table
- Delete a VLAN Entry in ALE Table
- Search for Address of VLAN in ALE Table
- Search for VLAN in ALE Table
- Configure Rate Limit for Broadcast/Unicast transmit (RX/TX)*
- Enable VLAN ID Ingression check*
- Configure Port State*
- Configure MAC Authentication Mode*
- Set Unknown VLAN Info*
- Clear Untouched ageable ALE table entries
- Print an ALE Entry
- Print Valid ALE entries in given range
- Configure ALEAware

```
Note: (*)Features not fully tested
```

Following Port configurations are implemented in latest release of StarterWare

- Configure Port VLAN Parameters
- Configure ALEAware

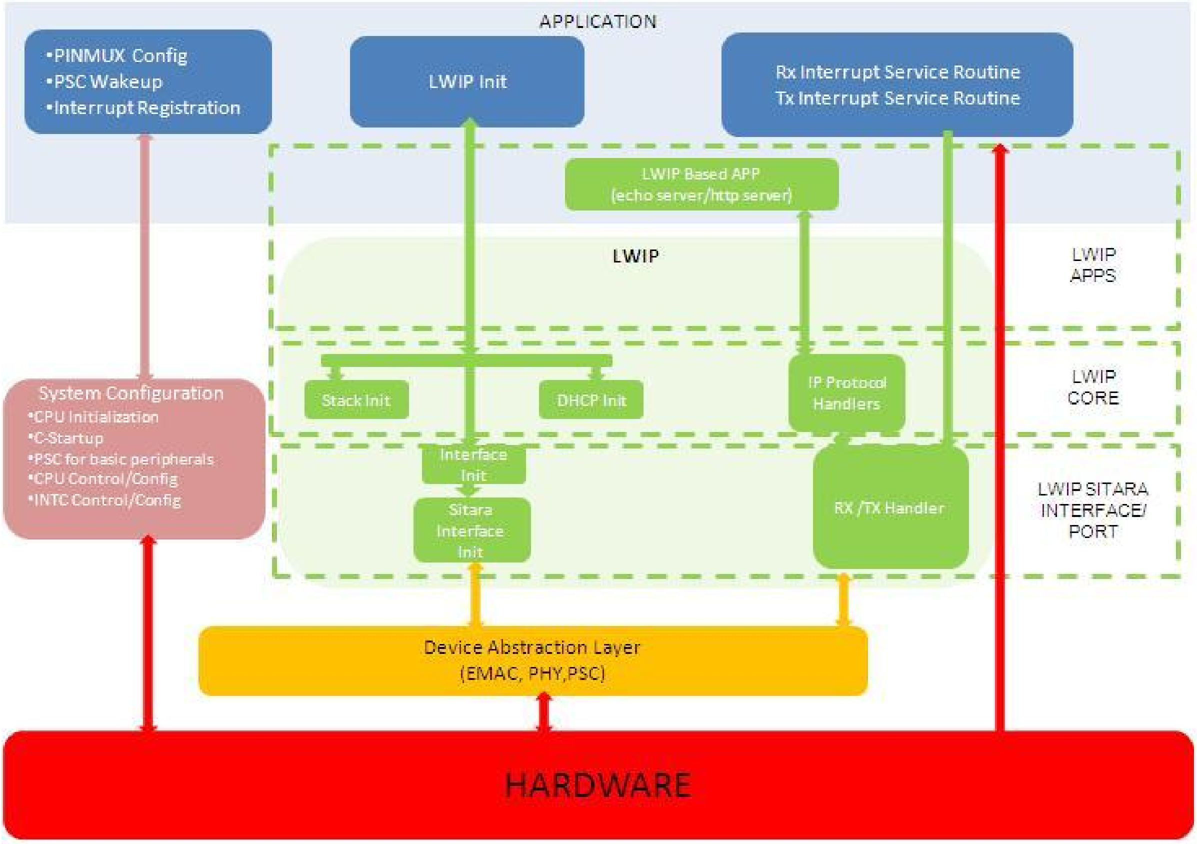
<a _fcknotitle="true" href="Category:StarterWare">StarterWare

StarterWare CPSW Port lwIP

Design overview

The StarterWare Ethernet software deliverable consists of four main layers:

1. **Device Abstraction Layer** - CPSW DAL, MDIO DAL, PHY DAL
2. **LWIP Network Interface Layer** - StarterWare NetIF port for LWIP
3. **LWIP Application Layer** - An IP stack based application based on LWIP. Examples provided with StarterWare include an HTTP server, a UDP based client, an echo server, etc. The packets start and end at this layer.
4. **System Application Layer** - The application layer that is used for system initialization and can be used for any other algorithm of the system



Device Abstraction Layer

This layer implements the lowest level hardware abstraction APIs that can be used for control and configuration of the EMAC device. There are three files that form this layer and are located at `StarterWare_xx_yy_mm_pp/drivers`:

- `cpsw.c` - CPSW configurations
- `mdio.c` - MDIO interface between the PHY and MAC
- `lan8710a.c` - PHY device

The device abstraction layer is implemented as a very light layer just for hardware/device access and conforms to the generic framework of StarterWare. It does not maintain any state variables.

lwIP Interface Layer

To interface with the rest of the network, the device abstraction layer needs to be glued with a network stack that can form and interpret network packets. StarterWare uses lwIP for this purpose because it has no OS dependency and supports many standard network protocols. The device abstraction hooks into the interface layer of lwIP. This is also referred to as the device-specific "port" or the StarterWare-interface for lwIP. This lwIP interface layer forms a major part of the StarterWare network driver. It defines standard interface entry points and state variables. A network device is represented by `struct netif`, generically referred to as netif. The netif contains all the information about the interface, including the IP/MAC address(s), TCP/IP options, protocol handlers, link information, and (most importantly) the network device driver entry point callbacks. Every network interface must implement the `linkoutput` and `init` callbacks, and all state information is maintained in this structure. The interface layer also implements the core interrupt handling and DMA handling. All the required function calls for initializing the lwIP stack and registering the StarterWare network interface are performed in `lwiplib.c`. Refer to the lwIP documentation [1] for more information about the lwIP stack implementation. Further sections below explain the network interface initialization and registration. This is located at *third_party/lwip-1.x.x/ports/cpsw/*

StarterWare Network Interface Layer

The main tasks of the StarterWare network interface layer are:

- Network device initialization

The first step towards bringing up the interface is done as part of the `cpswif_init`. This function is called when the network device is registered with the lwIP stack using `netif_add`. As part of the initialization, the netif output callbacks are registered and hardware initialization, including PHY and DMA initialization, is performed. DMA buffer descriptor (BD) pools are maintained in the CPPI RAM for both TX and RX channels. The descriptor chains are maintained by the "free_head", which points to the next unused/free descriptor in the BD pool, and "send_tail/recv_tail", which points to the last BD in the active queue that has been en-queued to the hardware. The packet buffers (pbuf) are pre-allocated for maximum length and queued in the receive buffer descriptors before the reception begins. Please refer to the lwIP documentation [1] for details on pbuf handling by lwIP.

- Packet data transmission

Packet data transmission takes place inside the `linkoutput` callback registered with the lwIP stack. This callback is invoked whenever the lwIP stack receives a packet for transmission from the application layer. The pbuf can contain a chain of packet buffers and hence the DMA descriptors are properly updated (chained if necessary), with SOP, EOP, TO_PORT and length fields. The first DMA descriptor is marked with the EOP and OWNER flags, while only the last is set with the EOP flag. After filling the BD's with the pbuf information, the BD, which corresponds to the SOP is written to the HEAD descriptor pointer register to start the transmission. Once a packet is transmitted, the CPSW Control Core generates a transmit interrupt. This interrupt is cleared only if the completion pointer is written with the last BD processed. In the interrupt handler, the next BD to process is taken and traversed to reach the BD that corresponds to the end of the packet. This BD, which corresponds to the end of the packet, is written to the completion pointer. After this, the pbuf that corresponds to this packet is freed. Thus it is made sure that the freeing of pbuf is done only after the packet transmission is complete.

- Packet data reception

Packet reception takes place in the context of the interrupt handler for receive. As described earlier, the receive buffer descriptors are en-queued to the DMA before the reception can actually begin. The pbuf allocated for maximum length, may actually contain a chain of packet buffers. The descriptors are updated for OWNER flag only. The EOP, SOP, FROM_PORT and EOQ are updated by the DMA upon completion of the reception. One BD is expected to receive one entire packet since the pbuf allocation from PBUF_POOL allows it to be contigous. Each BD is checked for OWNER bit having been cleared by the DMA. Thus the packet is passed to the upper layer for processing. After processing the packet by the upper layer, the processed BDs can be allocated and used for further reception.

lwIP Application Layer

This layer contains the ethernet application (HTTP server, echo server, etc.). This is located at StarterWare_xx_yy_mm_pp/third_party/lwip-1.x.x/apps/<application>. This is the layer at which all the imcoming packets terminate and all outgoing packets originate.

System Application Layer

This layer implements system level initialization and provides options for lwIP stack. This layer can contain any other algorithms, decoding, etc. The main IP stack based application is part of the lwip directory as mentioned above.

References

[1] http://www.sics.se/~adam/lwip/doc/lwip.pdf

StarterWare USB

Introduction

The StarterWare USB stack is an OS-independent USB solution that is migrated from the StellarisWare. For more information on the original StellarisWare USB library, please refer to the StellarisWare user guide [1].

The API reference guide for StarterWare USB is available as part of release package. Look for the [Device]_StarterWare_x_xx_xx_xx.chm file in docs for the for detailed API documentation.

The StarterWare USB stack currently supports the following features:

Release	Device Mode					Host Mode			Host + Device Mode		Host + Host Mode	
	CDC Serial (PIO Mode)	Custom Bulk (PIO Mode)	HID Mouse (PIO Mode)	MSC (PIO Mode)	MSC (DMA Mode)	HID Mouse (PIO Mode)	MSC (PIO Mode)	MSC (DMA Mode)	MSC (PIO Mode)	MSC (DMA Mode)	MSC (PIO Mode)/Mouse (PIO Mode)	MSC (DMA Mode)/Mouse (PIO Mode)
1.00.01.01	YES	YES	YES	YES	YES	YES	YES	YES	NO	NO	NO	NO
1.10.01.01	YES	YES	YES	YES	YES	YES	YES	YES	NO	NO	NO	NO
1.20.01.01	YES	YES	YES	YES	YES	YES	YES	YES	NO	NO	NO	NO
2.00.00.05(AM335x EVM)	YES	YES	YES	YES	YES	YES	YES	YES	NO	NO	NO	NO
2.00.00.05(AM335x Beaglebone)	YES	YES	NO	YES	YES	NO	NO	NO	NO	NO	NO	NO
2.00.00.06(AM335x EVM)	YES	YES	YES	YES	YES	YES	YES	YES	NO	YES	NO	NO
2.00.00.06(AM335x Beaglebone)	YES	YES	NO	YES	YES	NO	NO	NO	NO	NO	NO	NO
2.00.00.07(AM335x EVM)	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES

2.00.00.07(AM335x Beaglebone)	YES	YES	NO	YES	YES	NO	YES	YES	YES	YES	NO	NO
2.00.00.07(AM335x EVMSK)	YES	YES	YES	YES	YES	YES	YES	YES	YES	YES	NO	NO

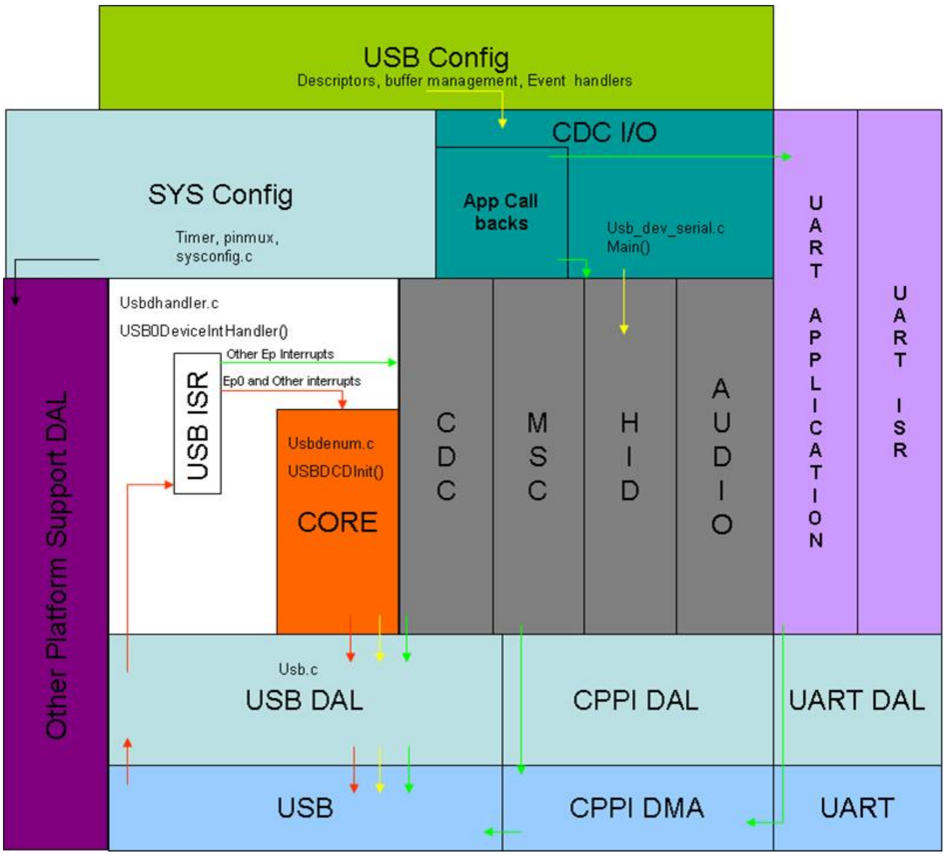
Comparison to StellarisWare USB

The following table compares the StarterWare and StellarisWare USB stacks.

Feature	**AM1808/OMAPL138/C6748**	**AM335x**	**Stellaris USB Subsystem**
Core IP	MUSB	MUSB	MUSB
Number Of Endpoints	4 + 1	15 + 1	15 + 1
DMA	CPPI DMA 4.1	CPPI DMA 4.1	Micro DMA
Mapped to Wrapper Registers	Yes, Only the Interrupt registers	Yes, Only the Interrupt registers	No
Endpoint 0 FIFO size	64 Bytes	64 Bytes	64 Bytes
Other Endpoint FIFO	8 to 8192 bytes configurable (Verified for 64/512 only)	8 to 8192 bytes configurable (Verified for 64/512 only)	8 to 8192 bytes configurable
Speed	High Speed/Full Speed (Refer to Class Description for details)	High Speed/Full Speed (Refer to Class Description for details)	High Speed Not Supported

Core Design

The following diagram shows the architecture of the StarterWare USB stack.



Mentor Controller Device Abstraction Layer (MUSB DAL)

The USBOTG Subsystem is based on Mentor Graphics USB Controller. IP core supports 4Tx/4Rx endpoints and an integrated CPPI41 DMA engine. The Device Abstraction Layer (DAL) is the only module in StarterWare USB which directly communicates with the Mentor controller. The MUSB DAL layers abstracts all the functionalities or services used by the layer above to configure and control the USB OTG Controller. The DAL APIs are used by the other components of device or host stack.

The MUSB DAL is implemented in single file (`usb.c`), and important APIs are summarized below.

- `USBDevConnect` - Connects the USB controller to the bus in device mode. This API is used by the USB Core module during initialization.
- `USBDevDisconnect` - Removes the USB controller from the bus in device mode. This API is used by the USB core while unplugging or resetting the device.
- `USBReset` - Resets the USB controller. This API is used by the core during enumeration or whenever the reset is necessary.
- `USBEndpointDataPut` - Puts data into the given endpoint's FIFO. This API is used by the individual device stack.
- `USBEndpointDataSend` - Starts the transfer of data from an endpoint's FIFO. This API is used by the individual device stack in order to send the data.
- `USBEndpointDataGet` - Retrieves data from the given endpoint's FIFO. This API is used by the individual device stack.

Device Core Layer

The Device Core Layer is responsible for device enumeration and handling all of the control transfers. The interrupt service routine (ISR) is a part of the core, so interrupt handling is also done by the core. The typical tasks handled by the device core layer are as follows:

- Initialize the device controller driver
- Enable clocking to the USB controller
- Switch on the USB PHY
- USB enumeration handling
- Interrupt handling
- Standard request handling
- Terminate the device controller driver

The standard requests that are handled by the device core are as follows:

- `USBDGetStatus`
- `USBDClearFeature`
- `USBDSetFeature`
- `USBDSetAddress`
- `USBDGetDescriptor`
- `USBDSetDescriptor`
- `USBDGetConfiguration`
- `USBDSetConfiguration`
- `USBDGetInterface`
- `USBDSetInterface`
- `USBDSyncFrame`

In `USBDeviceIntHandlerInternal`, the device core layer handles the following interrupts, invoking their corresponding callbacks:

- `Reset`
- `Suspend`
- `Resume`
- `Disconnect`
- `Start of Frame` (SOF)
- `Endpoint Interrupt`

The initialization API is called by the class driver in order to start enumeration. Once enumeration is started, all of the standard request routines are called from the ISR in response to the appropriate interrupt event. For any class-specific request, the application or class driver must register the callback handler with the core so that the ISR can branch there when necessary.

The device core layer is implemented by a single source file, `usbdenum.c`. The major APIs provided by this layer are summarized below:

- `USBDCDInit` - Initializes the USB library device control driver for a given hardware controller. This API is used by the individual class stack.
- `USBDCDTerm` - Frees the USB library device control driver for a given hardware controller.
- `USBDCDSendDataEP0` - Requests transfer of data to the host on endpoint zero.
- `USBDCDStallEP0` - Generates a stall condition on endpoint zero.
- `USBDeviceEnumHandler` - Interrupt handler for endpoint zero.
- `USBDeviceIntHandlerInternal` - Internal USB device interrupt handler.

USB Configuration Layer

This layer contains all of the parameters required to configure the USB device. These parameters include VID, PID, Descriptors, event handlers, buffers, and more. These parameters are specified by the application writer. Once all the information is fed in to this layer, the configuration layer can give a device instance structure to the other layers within the USB stack. This device instance contains all of the information regarding the device, and the required information is extracted when necessary.

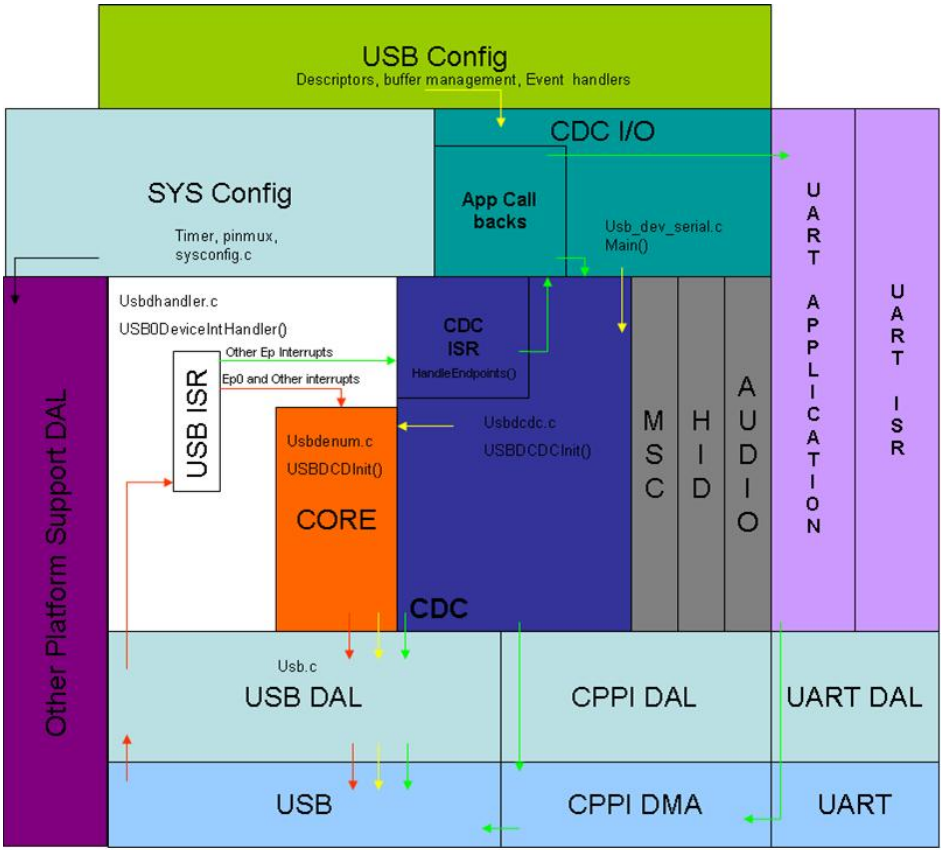
This layer is considered to be part of application, and most of the data is specified by the application programmer. This layer is implemented by the source file `usb_serial_structs.c`. The following parameters are configurable by the application programmer:

- Global instance for the USB device structure
- Rx and Tx Buffers
- Vendor ID
- Product ID
- Power configuration parameters
- Control event call backs
- Application event call backs
- String descriptors
- Language descriptor

System Configuration Layer

This layer provides APIs which can be used to configure platform-related parameters such as the clock speed, timers, pinmux, and more. These parameters are specified by the application developers as well as the underlying peripheral driver (i.e. DAL). Sometimes the entire system behavior depends on these parameters (ex. clock speed), so it is very important to select appropriate values for these parameters.

CDC Device Class



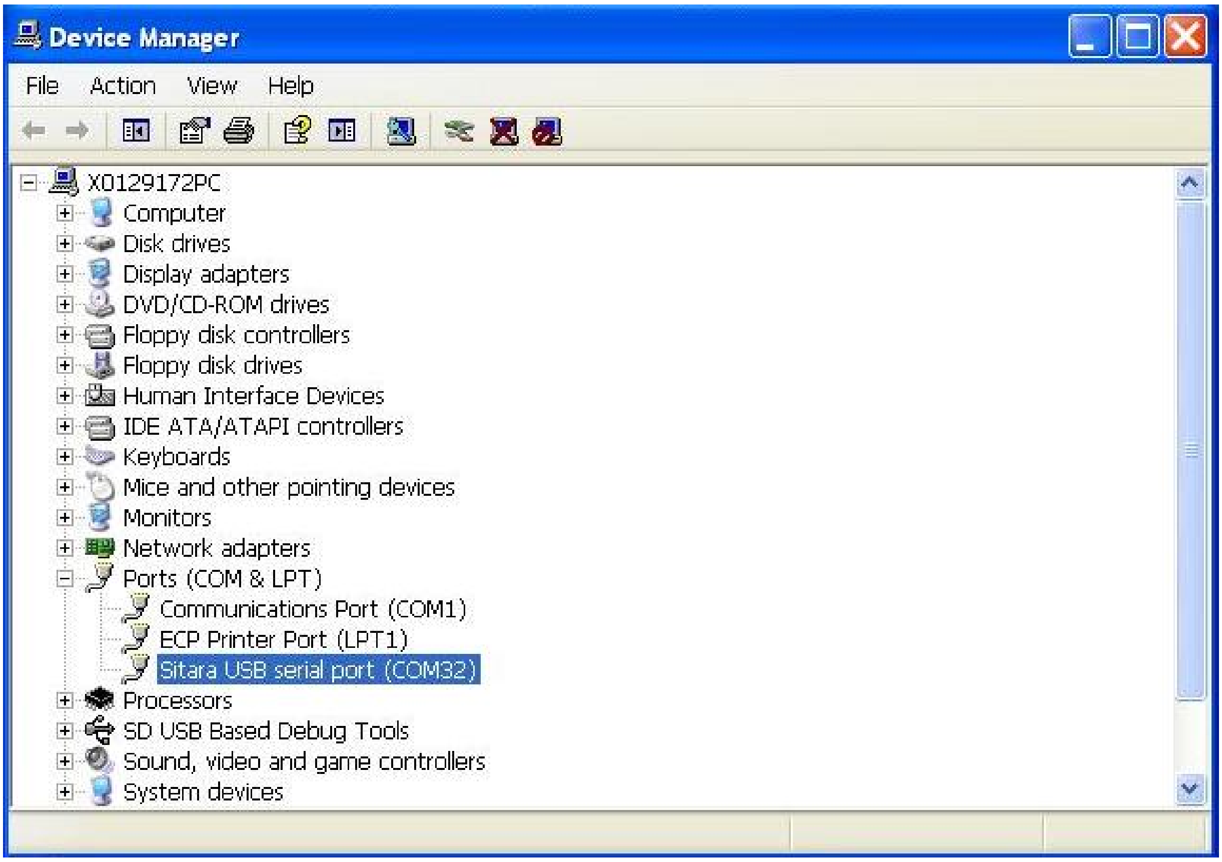
The USB Communication Device Class (CDC) class driver supports the CDC Abstract Control Model variant and allows a client application to be seen as a virtual serial port to the USB host system. The driver provides two channels: one transmit and one receive. The channels may be used in conjunction with USB buffers to provide a simple read/write interface for data transfer to and from the host. Additional APIs and events are used to support serial link-specific operations such as notification of UART errors, sending break conditions, and setting communication line parameters. The data transmission capabilities of this device class driver are very similar to the generic bulk class, but (because this is a standard device class) the host operating system should be able to access the device without the need for any host-side drivers. On Windows, a simple INF file is all that is required to make the USB device appear as a COM port that can be accessed by any serial terminal application. This device class uses three endpoints in addition to endpoint zero. Two bulk endpoints carry data to and from the host and an interrupt IN endpoint is used to signal any serial errors such as break, framing error, or parity error detected by the device. Endpoint zero carries standard USB requests and also CDC-specific requests which translate to events passed to the application via the control channel callback . This layer is implemented in the source file `usbdcdc.c`.

Example Application

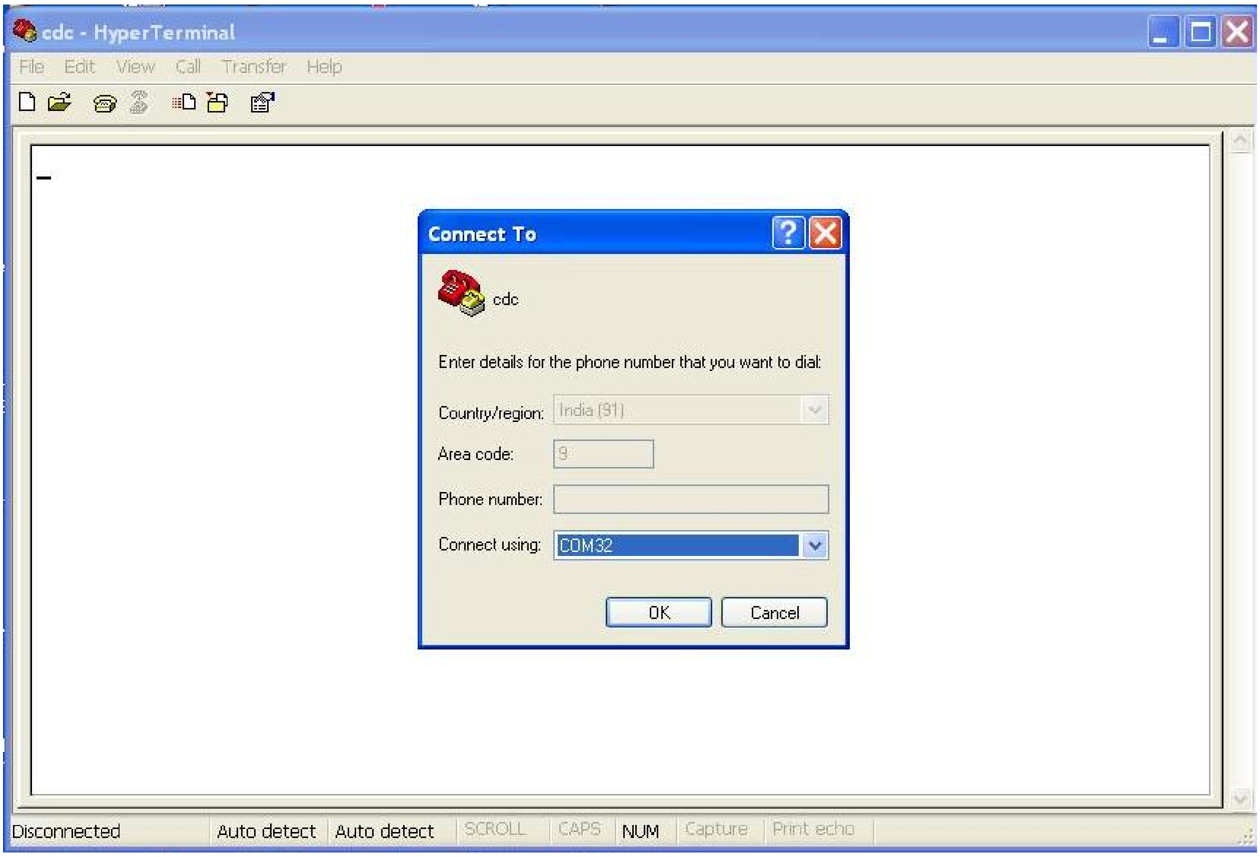
Running the Application

- **Since this is a USB device mode application, the target (EVM) must be connected to host (Windows PC) by a USB cable.**
- Connect your host PC's serial port to the UART connector on the EVM using a null modem cable.
- The CDC device requires an INF file at the host side in order to enable as virtual COM port. Please use the INF provided in the `tools/usb_inf` folder.
  - The INF file is required only with a windows based host.
  - To use different VID and PID, the INF file must be updated with the new VID and PID.
- Once the device is connected to the host, the host will ask for INF file. The user must browse to the INF file to proceed.

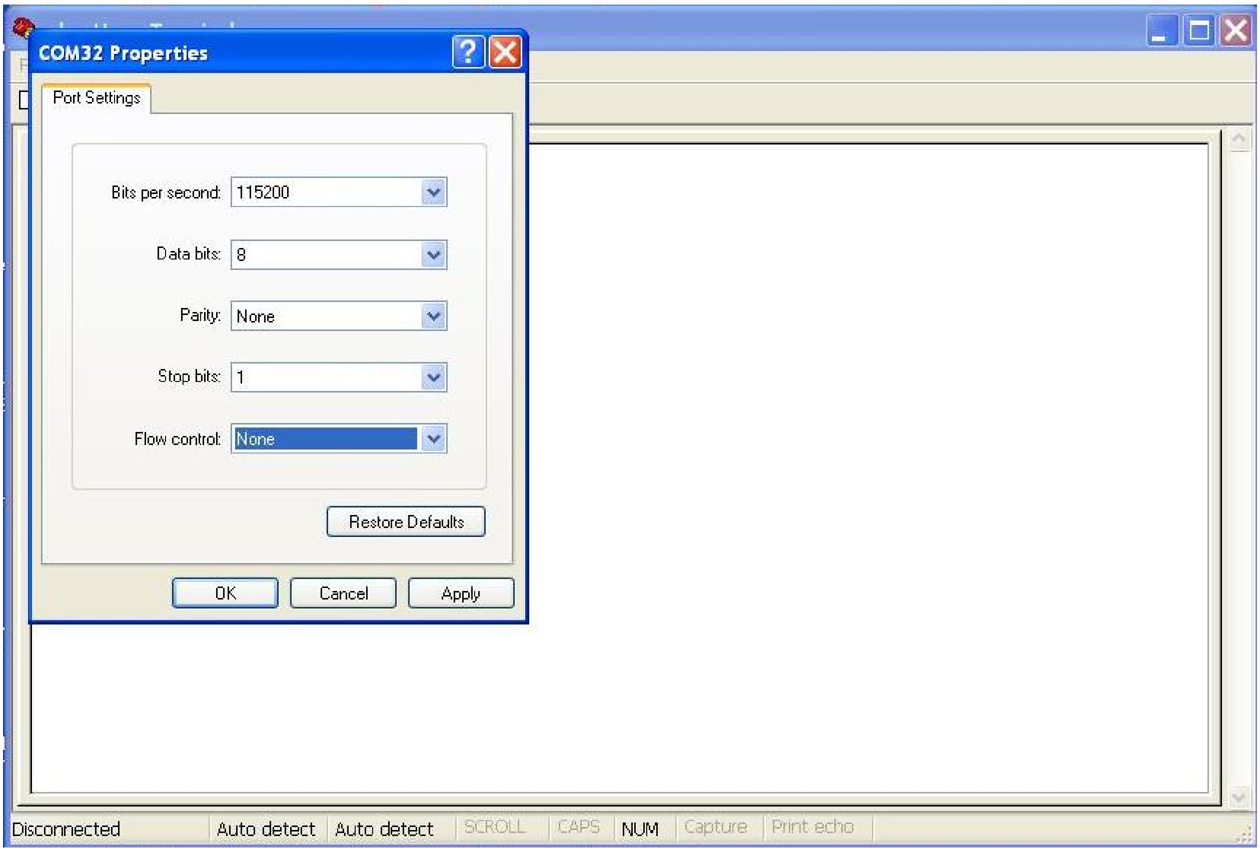
- Once the proper INF file is pointed out by the user, then virtual COM port will appear in the device manager



- Open a serial terminal application (ex. Teraterm, Hyperterminal) and choose the virtual COM port.



- Set the parameters as shown.



- Open another terminal window and choose COM1 (or whichever serial port you previously connected to the EVM's UART connector).
- Type in either terminal window on the host PC. The text should appear on the other terminal screen.

Note: The CDC class is configured and tested for Hi-Speed PIO mode only

Modules used by this application:

- USB
- Interrupt

CDC Device Enumeration

The Application main must perform the following steps:

- Configure the system interrupts
- Register the interrupt handler (i.e. ISR)
- Initialize the buffers
- Call `USBDCDCInit()` with the device instance structure as a parameter

With this call, control is given to the CDC device class layer. The device class layer must perform the following steps:

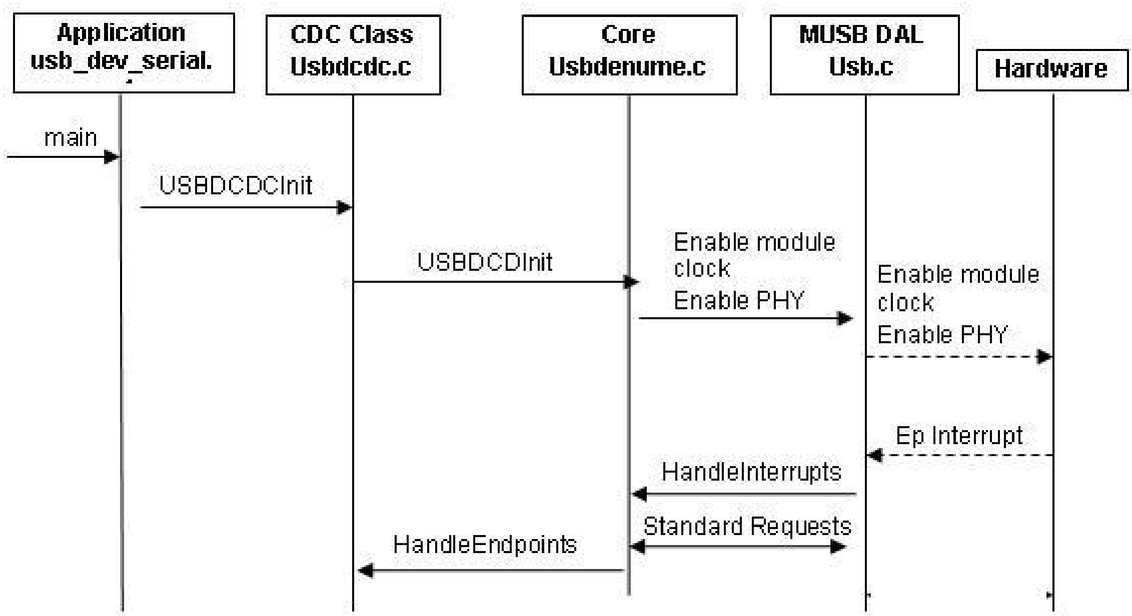
- Assign endpoints to the device instance structure
- Apply the configuration descriptor to the device instance structure
- Call `USBDCDInit()` and pass the device instance structure to the core

With this call, control is passed to the core layer. The core layer must perform the following steps in order to complete enumeration:

- Enable the PSC clock
- Reset the USB module
- Switch on the USB PHY

- Initialize the USB tick module
- Clear all pending interrupts
- Enable the required interrupts
- Set the configuration parameters
- Disconnect the device
- Reconnect the device

Now the device will start receiving interrupts from the host. The interrupt handler in the core layer will identify all of the Ep0 interrupts and call the appropriate handler. If all the standard requests are serviced by the device, then enumeration is complete and the device is ready for communication. Now control return to the application and wait for data interrupts.



CONNECT / DISCONNECT

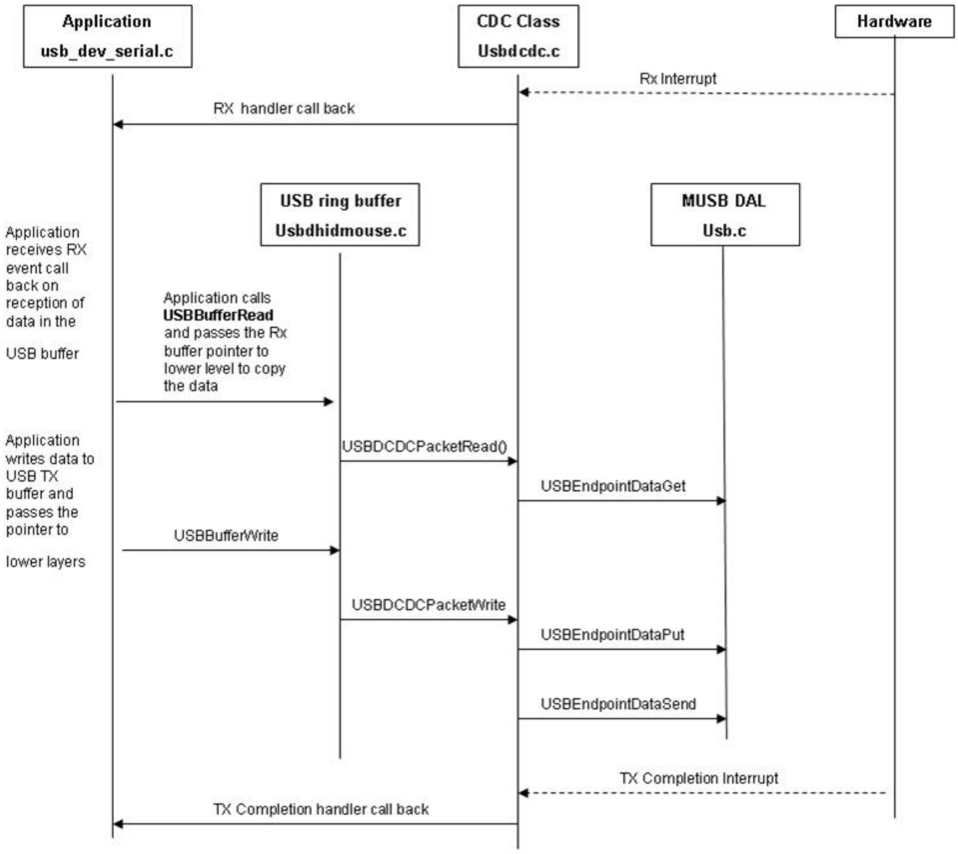
If required, the MUSB can allow software to control its connection to the USB bus. When the Soft Connect/Disconnect option is selected with the MUSB operating in peripheral mode, the UTMI+-compliant PHY (used alongside the MUSB) can be switched between normal mode and non-driving mode by setting or clearing bit 6 of the Power register. (This bit is identified as the Soft Conn bit.) When this Soft Conn bit is set to 1, the PHY is placed in its normal mode and the D+/D- lines of the USB bus are enabled. At the same time, the MUSB is placed in "Powered" state, where it will not respond to any USB signaling except a USB reset. When this feature is enabled and the Soft Conn bit is zero, the PHY is put into non-driving mode, D+ and D- are tri-stated, and the MUSB appears to other devices on the USB bus as if it has been disconnected.

After a hardware reset (NRST = 0), Soft Conn is cleared to 0. The MUSB will therefore appear disconnected until the software sets Soft Conn to 1. The application software can then choose when to set the PHY into its normal mode. Systems with a lengthy initialization procedure may use this to ensure that initialization is complete and the system is ready to perform enumeration before connecting to the USB. Once the Soft Conn bit has been set to 1, the software can also simulate a disconnect by clearing this bit to 0.

USB Tick module

The USB tick module contains the functions related to USB stack tick timer handling. It has functions to initialize the variables used in processing timer ticks, function handles registering OTG, Host, or Device SOF timer handler functions. This module is implemented in the source file `usbtick.c`.

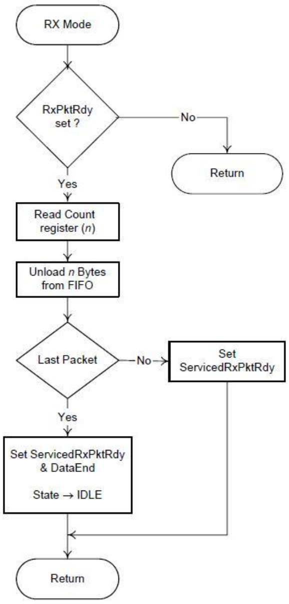
CDC Application Data Flow



Data out Flow (From Host to Device)

When data arrives from the host, the device will receive an interrupt from the host. The interrupt handler must perform the following steps on reception:

- Read the interrupt status register
- Identify the interrupt
- Call the appropriate handler (i.e. Read Data handler)
- Read handler checks whether the RX Packet Ready bit is set
- Check the number of byte available in the buffer
- Read the data to buffer
- Clear the RX Packet Ready bit
- Give the read data to the application



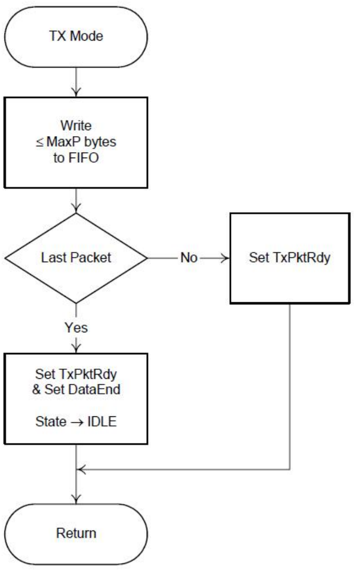
Count Register

RxCount is a 13-bit read-only register that holds the number of received data bytes in the packet currently in line to be read from the Rx FIFO. If the packet was transmitted as multiple bulk packets, the number given will be for the combined packet. The value returned changes as the FIFO is unloaded and is only valid while RxPktRdy is set.

Data in Flow (From Device to Host)

When the device side application needs to send data to the host, the device must follow the below steps.

- Application calls the write data API and passes the data buffer to the CDC layer
- Write data API checks the amount of data to be sent
- Copy data the TX buffer
- Set the TX Packet Ready bit
- After the transmit operation completes, the host sends a TX completion interrupt
- TX Packet Ready bit will be cleared automatically
- After receiving the TX Complete Interrupt, the ISR sends an event to the application indicating the completion of data transfer



How To Write a New CDC Device Application

A new application must take the following steps to add CDC transmit and receive capability.

- Define the 6-entry string descriptor table, which is used to describe various features of your new device (i.e. application) to the host system.
- Define a `tCDCSerInstance` structure, which the USB CDC serial device class driver uses to store its internal state information. This should never be accessed directly by the application.

```
tCDCSerInstance g_sSerialInstance;
```

- Define a `tUSBDCDCDevice` structure and initialize all fields as required by your application.
- Add a receive event handler function to your application. This function must handle all messages that require a particular response. For the CDC device class, `USB_EVENT_RX_AVAILABLE` and `USB_EVENT_DATA_REMAINING` **must** be handled by the receive event handler.
- Add a transmit event handler function to your application. This function must handle all messages that require a particular response. For the CDC device class, there are no events sent to the transmit callback that **must** be handled, but applications typically handle `SB_EVENT_TX_COMPLETE` since this is an interlock message indicating that the previous packet sent has been acknowledged by the host and a new packet can now be sent.
- Add a control event handler function to your application and ensure that it handles `USBD_CDC_EVENT_GET_LINE_CODING`, returning a valid line coding configuration even if the device is not actually driving a UART. Handle any other control events as required by your application.
- From your main initialization function call the CDC device class driver initialization function to configure the USB controller and place the device on the bus.

```
pDevice = USBDCDCInit(0, &g_sCDCDevice);
```

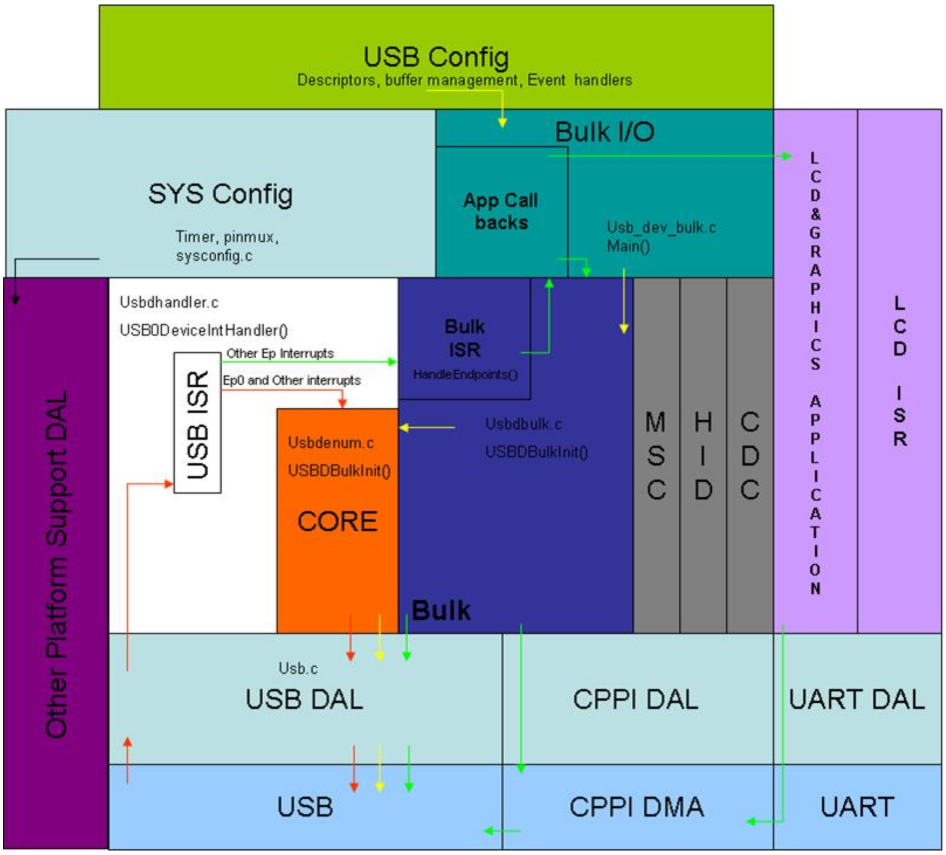
- Assuming `pDevice` returned is not NULL, the device is now ready to communicate with a USB host.
- Once the host connects, your control event handler will be receive a `USB_EVENT_CONNECTED` event, and the first packet of data may be sent to the host as soon as `USB_EVENT_TX_COMPLETE` is received via the transmit event handler.

For connector information please check the connector information table

Power Configuration

The power configuration for the CDC class device is held in two of the `tUSBDCDCDevice` members variables, `usMaxPowermA` and `ucPwrAttributes`. The `usMaxPowermA` variable holds the maximum power consumption for the device and is expressed in milliamps. The power configuration is held in the `ucPwrAttributes` member variable and indicates whether the device is self- or bus-powered. Valid values are `USB_CONF_ATTR_SELF_PWR` or `USB_CONF_ATTR_BUS_PWR`.

Bulk Device Class



The generic bulk device class driver offers a very simple method for an application to set up USB communication with a paired application running on the USB host system. The class driver offers a single bulk receive channel and a single bulk transmit channel and, when coupled with USB buffers on each channel, provides a straightforward read/write interface to the application. The device supports a single interface containing bulk IN and bulk OUT endpoints. The configuration and interface descriptors published by the device contain vendor-specific class identifiers, so an application on the host will have to communicate with the device using either a custom driver or a subsystem such as WinUSB or libusb-win32 on Windows to allow the device to be accessed. An example of this is provided in the usb_dev_bulk application. This class driver is particularly useful for applications that need to pass high volumes of data via USB and where host-side application code is being developed in partnership with the device.

Bulk Device Enumeration

An application using the USB bulk device class must perform the following initialization steps:

- Configure the system interrupts
- Register the interrupt handler (i.e. ISR)
- Initialize the buffers
- Call the `USBDBulkInit()` API with the device instance structure as a parameter

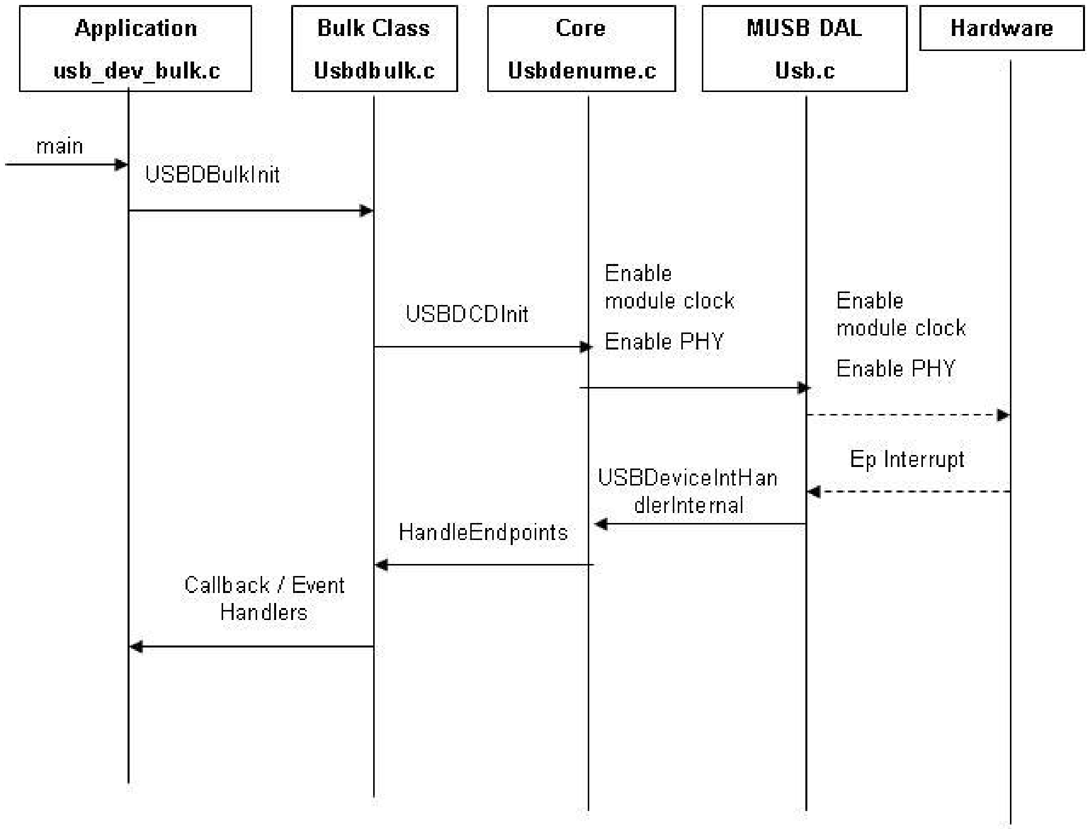
This function call gives contol to the bulk device class layer. The device class layer then performs the following steps:

- Assign the endpoints to the device instance structure
- Assign the configuration descriptor to the device instance structure.
- Call the `USBDCDInit()` API and pass the device instance structure to the core

This function call gives contol to the core layer. The core layer then performs the following steps in order to complete the bulk device enumeration:

- Enable the PSC clock
- Reset the USB module
- Switch on the USB PHY
- Initialize the USB tick module
- Clear all pending interrupts
- Enable the required interrupts
- Set the configuration parameters
- Disconnect the device
- Reconnect the device

Now the device will start receiving interrupts from the host. The interrupt handler in the core layer will identify all the Ep0 interrupts and call the appropriate handler. Once all the standard requests are serviced by the device, enumeration is complete and the device is ready for communication. Now control will return to the application to wait for data interrupts.



How to Write a New Bulk Application

To add USB bulk data transmit and receive capability to an application, take the following steps:

- Add the following header files to the source file(s) that will support USB:

```
#include "usb.h"
```

```
#include "usblib.h"
```

```
#include "usbdevice.h"
```

```
#include "usbdbulk.h"
```

- Define the 5-entry string table that is used to describe various features of your new device to the host system.
- Define an area of RAM of for the private data for the bulk device class driver. This structure should never be accessed by the application.
- Define a `tUSBDBulkDevice` structure and initialize all fields as required for your application. The following example illustrates a simple case where no USB buffers are in use. For an example using USB buffers, see the source file `usb_bulk_structs.c` in the usb_dev_bulk example application.
- Add a receive event handler function (ex. `YourUSBReceiveEventCallback()` in the previous example) to your application, taking care to handle all messages that require a particular response. For the generic bulk device class, only the `USB_EVENT_RX_AVAILABLE` **must** be handled by the receive event handler. In response to `SB_EVENT_RX_AVAILABLE`, your handler should check the amount of data received by calling the `USBDBulkRxPacketAvailable()` API then read it by calling `USBDBulkPacketRead()`. This causes the newly received data to be acknowledged to the host and instructs the host that it may now transmit another packet. If you are unable to read the data immediately, return 0 from the callback handler and it will be called again a few milliseconds later. Although no other events must be handled, `USB_EVENT_CONNECTED` and `USB_EVENT_DISCONNECTED` are typically required since these indicate when a host connects or disconnects and allow the application to flush any buffers or reset state as required. Attempts to send data when the host is disconnected will fail.
- Add a transmit event handler function (ex. `YourUSBTransmitEventCallback()` in the previous example) to your application, taking care to handle all messages that require a particular response. For the generic bulk device class, there are no events sent to the transmit callback that **must** be handled, but applications typically want to note `USB_EVENT_TX_COMPLETE` since this is an interlock message indicating that the previous packet sent has been acknowledged by the host and a new packet can now be sent
- From your main initialization function call the generic bulk device class driver initialization function to configure the USB controller and place the device on the bus.

```
pDevice = USBDBulkInit(0, &g_sBulkDevice);
```

- Assuming `pDevice` returned is not NULL, your device is now ready to communicate with a USB host. Once the host connects, your receive event handler receives `USB_EVENT_CONNECTED`, and the first packet of data may be sent to the host using the `USBDBulkPacketWrite()` API as soon as `USB_EVENT_TX_COMPLETE` is received.

Windows Drivers for Generic Bulk Devices

Since generic bulk devices appear to a host operating system as vendor-specific devices, no device drivers on the host system will be able to communicate with them without device-specific drivers. This require writing a Windows kernel driver for the device or, if that task is too daunting, steering Windows to use one of several generic kernel drivers that can manage the device with assistance from a Windows application. The second option does not require the developer to write any Windows driver code, but the developer instead needs to to write an application or DLL that interfaces with the device via user-mode APIs offered by generic USB drivers.

The developer is also responsible for producing a suitable INF file to allow Windows to associate the device (identified via its VID/PID combination) with a particular driver. A least two suitable USB subsystems are available for Windows:

1. **WinUSB** - from Microsoft
2. **libusb-win32** - an opensource project available from SourceForge

WinUSB supports only WindowsXP and Vista systems. Further information can be obtained from the MSDN website [2].

To develop applications using WinUSB, the Windows Driver Development Kit (DDK) must be installed on your host PC. These applications can be found in the package PDL-LM3S-win, which is available for downloaded from the Stellaris website [3].

libusb-win32 supports all versions of Windows from Windows98SE onward and can be downloaded from the SourceForge website [4].

Running the Example Application

- Download and install Windows host driver and application (SW-USB-WIN and SW-USB-WINDRIVERS) from the StellarisWare download page [3].
    - The INF file is located at `windows_drivers/usb_dev_bulk.inf`.
    - The host application is located at `usb_examples/usb_bulk_example.exe`.
    - The host application should only be run **after** the target application is already running (see below).
- Build and download the usb_dev_bulk application to the target board .
- Connect the target board to PC via a USB cable.
- When Windows detects your target device, browse to the INF file downloaded above.
- Open the host application on the host PC. (Note: the host application will fail if you run it before the target application is running and connected to the PC.)
- Type any text in the host application window to send it to the target device.
- Follow the instructions from the host application to stop execution.

Note: The bulk class is configured and tested for Full-Speed PIO mode only

For connector information please check the connector information table

Modules used by this application:

- USB
- Interrupt
- Timer (for 100 ms delay during USB enumeration; graphics too?)
- UART (non-interrupt; for debug output)
- LCDC (raster mode; display # bytes transferred)

HID Device Class

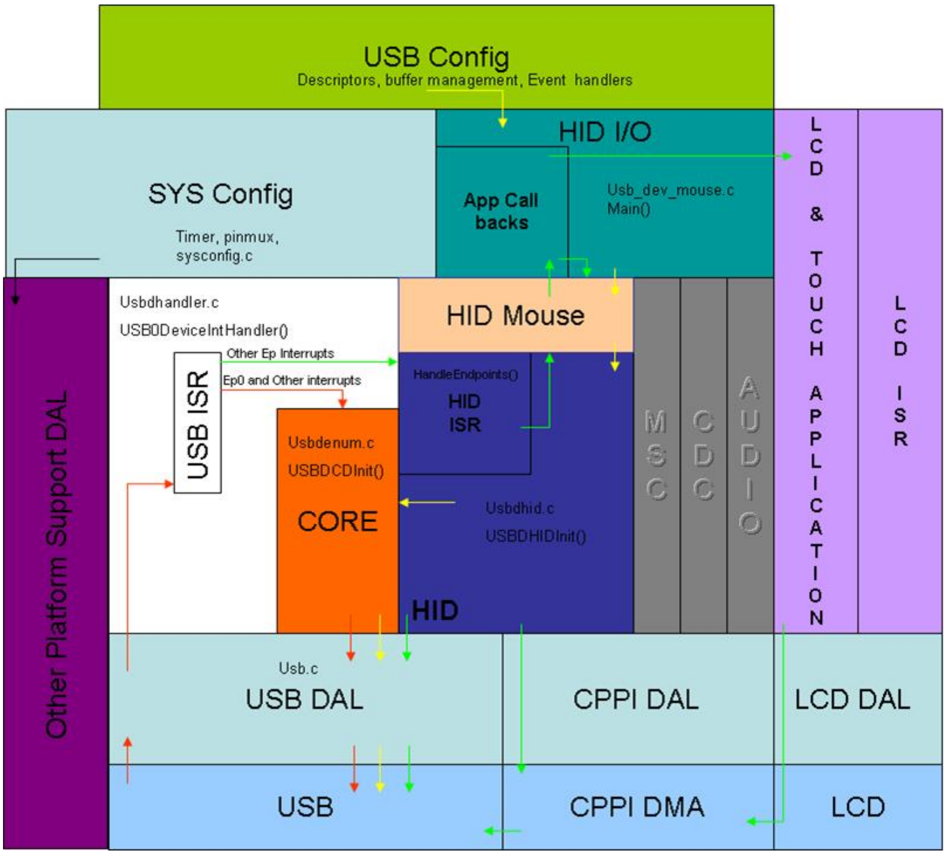
Please note that in the current EVM only uses touchscreen hardware as the pointer device. The below text uses HIDMouse **as a generic term to refer to this.**

The USB Human Interface Class device class supports a wide variety of input/output devices, not all of which actually pertain to "Human Interfaces." While the most commonly supported devices are keyboards, mice, and joysticks, the specification can cover practically any device offering user controls or data gathering functionality. Communication between the HID device and host is conducted according to a collection of "report" structures. The device populates HID report descriptors, which the host can query. Reports are defined both for communication of device input to the host and for output or feature selection from the host.

In addition to the flexibility offered by the basic architecture, HID devices also benefit from excellent operating system support for the class. This means that device-specific drivers are generally not necessary. For standard devices such as keyboards and joysticks, the device can connect to and operate with the host system without any new host software having to be written. Even in the case of a nonstandard or vendor-specific HID device, the operating system support makes writing the host-side software very straightforward compared to developing the device using a vendor-specific class.

Despite these advantages, there is one downside to using HID. The interface is limited in the amount of data that can be transferred, so HID is not suitable for devices that need to use a significant percentage of the USB bus bandwidth. Devices are limited to a maximum of 64KB per second for each report descriptor they support. Multiple reports can be used if necessary, but high bandwidth devices are better implemented using a class that supports bulk rather than interrupt endpoints, such as CDC or the generic bulk device class.

The HID device class uses one or two endpoints in addition to endpoint zero. One interrupt IN endpoint carries HID input reports from the device to the host. Output and Feature reports from the host to the device are typically carried via endpoint zero but devices with high host-to-device data rates can select to offer an independent interrupt OUT endpoint to carry these. Endpoint zero carries standard USB requests and also HID-specific descriptor requests. This layer is implemented in the source files `Usbdhid.c` and `Usbdhidmouse.c`.



HID Device Enumeration

The Application must perform the following steps to enumerate the USB in HID device mode:

- Configure the system interrupts
- Register the Interrupt handlers
- Initialize the buffers
- Call the `USBDHIDMouseInit()` API with the device instance structure as a parameter

With this call, control passes to the HID mouse layer, which performs the following additional steps:

- Initialize the various fields in the instance structure
- Initialize the HID device class instance structure based on input from the caller
- Initialize the lower layer HID driver by calling the `USBDHIDInit()` API

With this call, control passes to the HID device class layer, which performs the following additional steps:

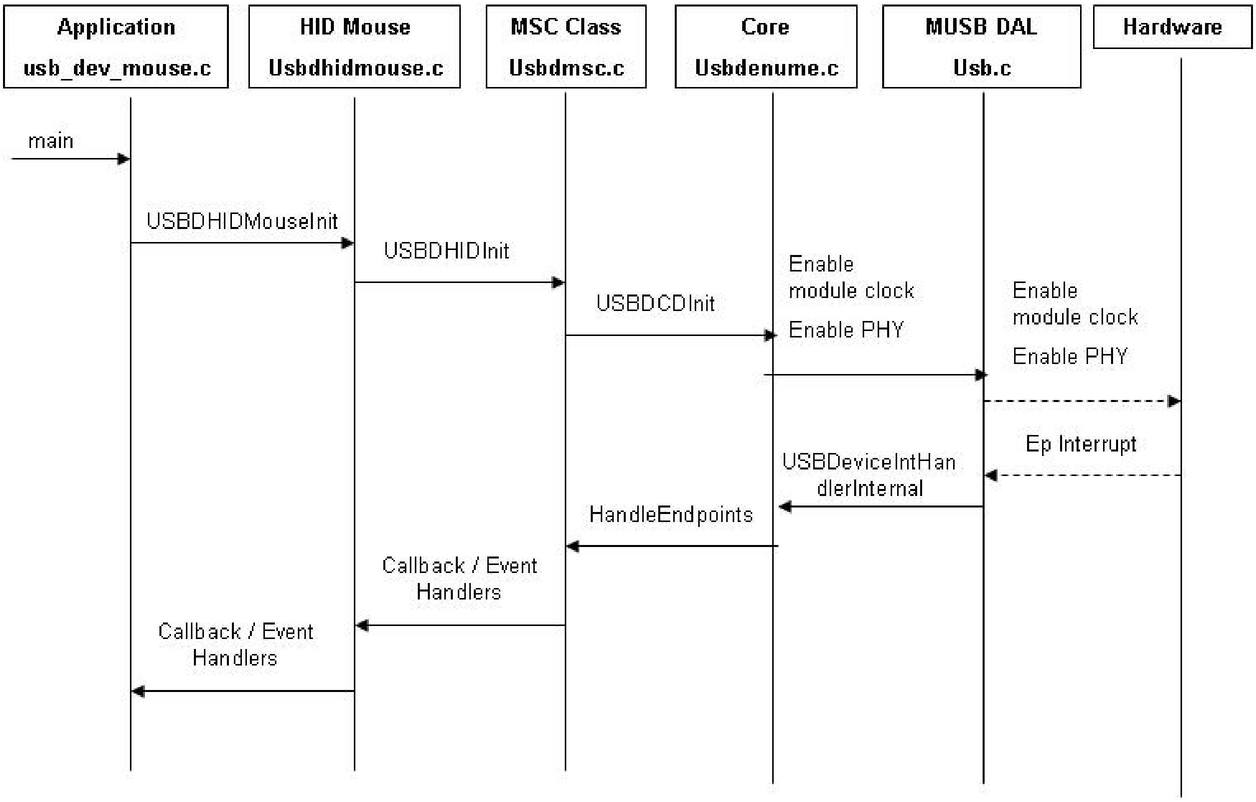
- Assign the endpoints to the device instance structure
- Assign the configuration descriptor to the device instance structure
- Call the `USBDCDInit()` API and pass the device instance structure to the core

With this call, control passes to the core layer, which performs the following additional steps to complete enumeration:

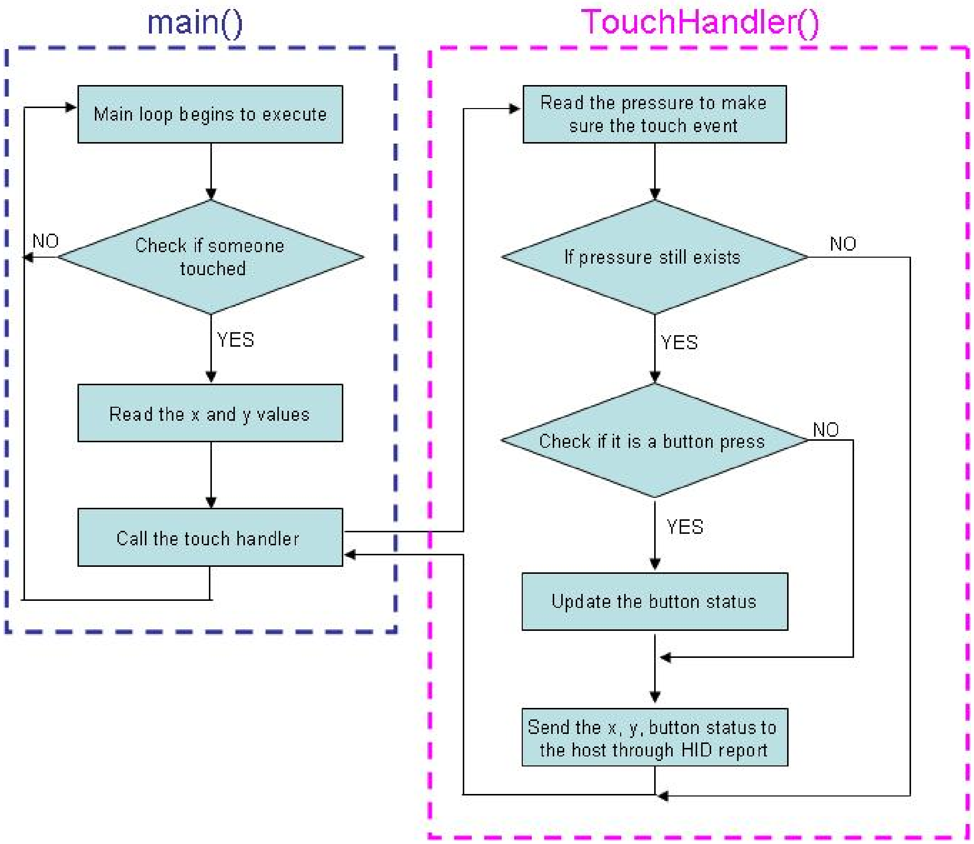
- Enable the PSC clock
- Reset the USB module
- Switch on the USB PHY
- Initialize the USB tick module
- Clear all the pending interrupts
- Enable the required interrupts
- Set the configuration parameters

- Disconnect the device
- Reconnect the device

Now the device starts receiving interrupts from the host. The interrupt handler in the core layer identifies all the Ep0 interrupts and calls the appropriate handler. If all the standard requests are serviced by the device, then enumeration is complete and the device is ready for communication. Now the control returns to the application, where it waits for touch screen inputs.



Example Application



The above figure shows how the HID device (mouse) application works. The application continuously polls for a touch screen event. Once an event is detected, the application reads the x and y values. The touch handler checks whether this is a button press or not (i.e. if the x and y coordinates belongs to the button area). If this is not a button press event, then read the x and y values are re-read multiple times. This helps to avoid the "bouncing" effects and confirm the touch event. Afterward, the current and previous x and y values are used to calculate how much we moved from the previous x/y location. The new location values are sent to the host through HID report.

Running the Application

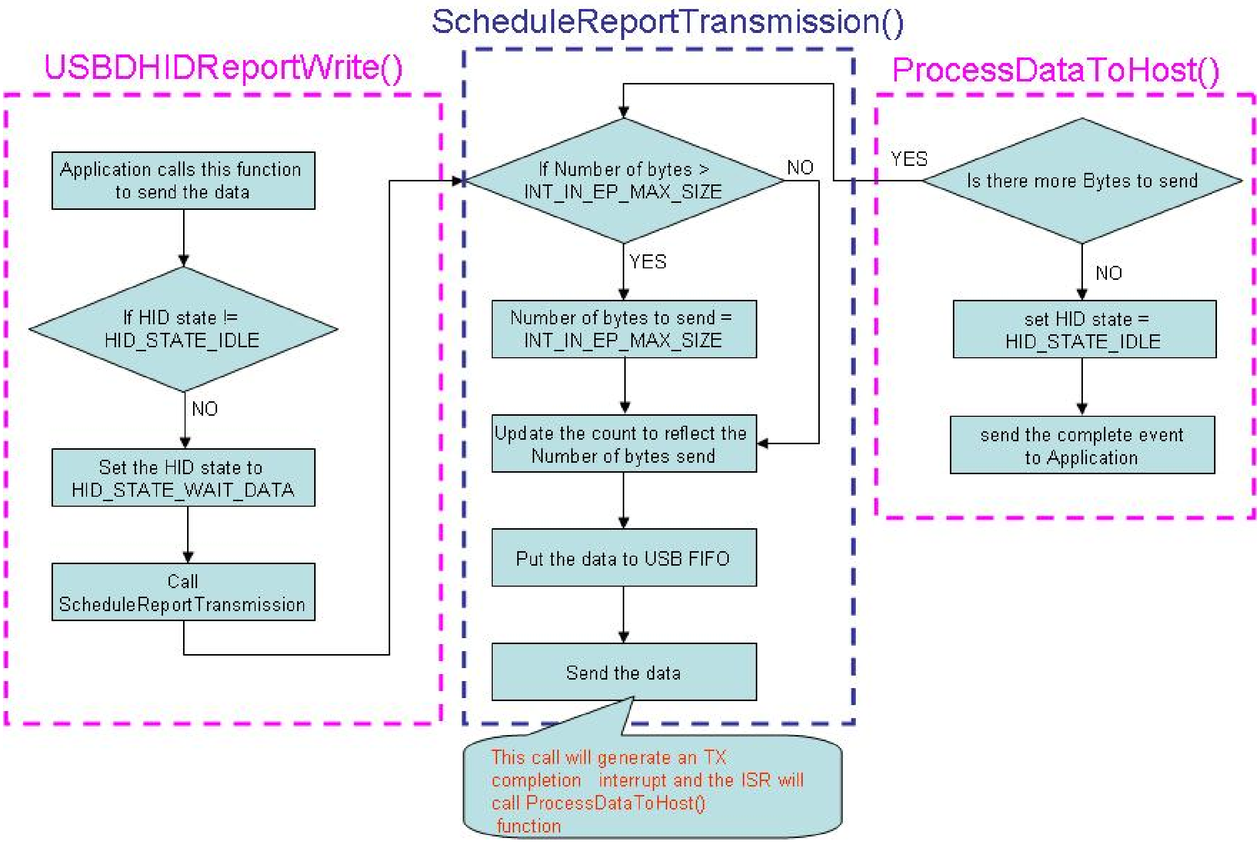
- Connect the target board to the host PC via a USB cable
- Load and run the target application
- Once the device is connected to the host, it enumerates as HID mouse
- User can check the device manager of the host to confirm the completion of the enumeration
- Touch screen is initialized and buttons are displayed
- User can use the touch screen to move the cursor on the host PC and press the mouse button to trigger mouse clicks

For connector information please check the connector information table

Modules used by this application:

- USB HID
- LCD raster
- Graphics library
- Touch screen (I2C)
- Interrupt

In Data Flow (Device to Host)



How to write a new HID Application

To add USB HID mouse support to an application using the USB stack, use the following procedure:

- Add the following header files to the source file(s) that will support USB:

```
#include "src/usb.h"
```

```
#include "usblib/usblib.h"
```

```
#include "usblib/device/usbdhidmouse.h"
```

- Define the string table that is used to describe various features of your new device to the host system. This table must include a minimum of 6 entries: string descriptor 0 defines the language(s) available, and 5 strings for each supported language.
- Define an area of RAM for the HID mouse class driver private data. This structure should never be accessed by the application.

```
static tHIDMouseInstance g_sMouseInstance;
```

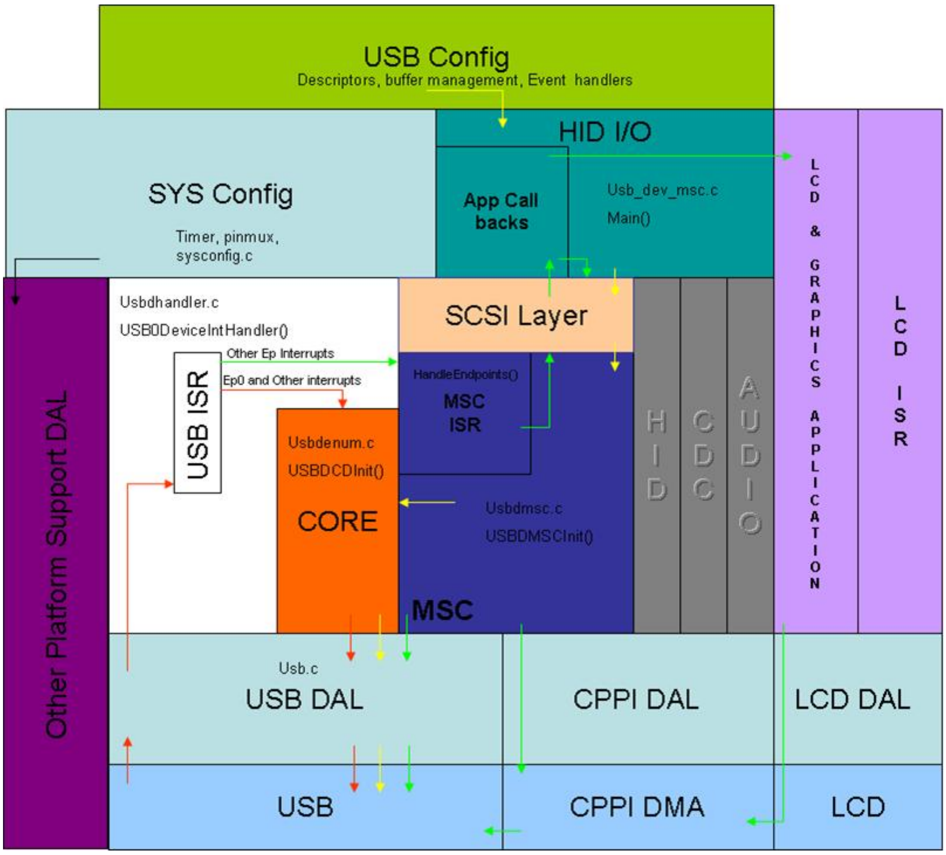
- Define a `tUSBDHIDMouseDevice` structure and initialize all fields as required by your application.
- Add a mouse event handler function to your application. A minimal implementation can ignore all events, but `USB_EVENT_TX_COMPLETE` is typically used to ensure that mouse messages are not sent when a previous report is still in transit to the host. Attempts to send a new mouse report when the previous report has not yet been acknowledged will result in return code `MOUSE_ERR_TX_ERROR` from `USBDHIDMouseStateChange()`.
- From your main initialization function call the HID mouse device initialization API to configure the USB controller and place the device on the bus.

```
pDevice = USBDHIDMouseInit(0, &g_sMouseDevice);
```

- Assuming `pDevice` is not returned as NULL, your mouse device is now ready to communicate with a USB host
- Once the host connects, your mouse event handler receives an `USB_EVENT_CONNECTED` event. Afterward, calls can be made to `USBDHIDMouseStateChange()` to inform the host of mouse position and button state changes.

Mass Storage Device Class

The USB mass storage device class allows an application to act as a physical storage device for use by another USB application or host PC. Because the type of storage can vary per application, the mass storage class abstracts the storage with a set of block-based APIs that are provided by the application to the USB library. These APIs allow the USB mass storage class to call an external set of functions that actually perform the operations on the physical storage media. The storage APIs are given to the USB library's mass storage device class initialization function and are called by the USB library whenever it needs to access the physical media.

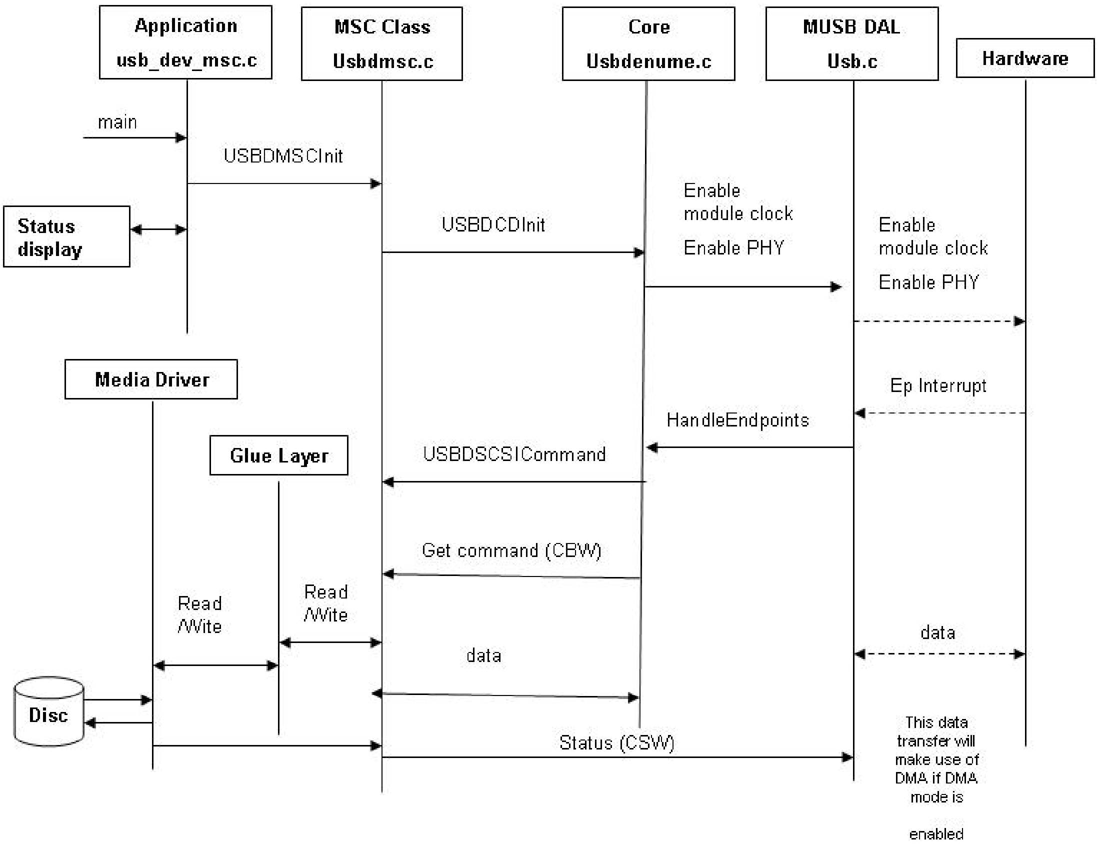


Initialization and Data Flow

The USB library's mass storage class provides a simple interface to initialize the mass storage class and pass it the needed functions to access a device without having any knowledge of the physical media. The `USBDMSCInit()` API is the only initialization required by the mass storage class, and it uses the structure `tUSBDMSCDevice` to hold all customizable values for the mass storage class.

The USB library's mass storage class provides the ability to customize how the device is reported to the USB host controller in the `tUSBDMSCDevice` structure. The members of this structure contain all of the customizable parameters.

The below diagram shows how data and commands flow to and from the MSC device. The diagram does not show using DMA to automate any data transfers.



Block Media Access

The media access functions are passed in to the USB mass storage device class in the `sMedia->Functions` member variable. This structure holds the access functions used by this instance of the mass storage class device. All of the functions in this structure are required to be populated with valid functions. These functions are called by the USB mass storage device class whenever it needs to read or write the physical media, and they always use a fixed block size of 512 bytes.

In some cases, the application may need to be informed when the state of the mass storage device has changed. The `pfnEventCallback` member of the `MSCDevice` structure provides event notification to applications for the following events:

- `USBD_MSC_EVENT_IDLE`
- `USBD_MSC_EVENT_READING`
- `USBD_MSC_WRITING`

When a function of type `tUSBCallback` is called, only the first two parameters (`pvCBData` and `ulEvent`) are valid. The `pvCBData` parameter is the value that was returned when the application called `USBDMSCInit()` and can be used with other APIs. These events are not used in the example application provided with StarterWare. Application writers can make use of these events as required.

RAM Disk Support

The MSC is provided with a RAM disk implementation as its block media. The current size supported by the RAM disk is 16 MB. Developers can change it by modifying the `RAMDISK_SIZE` macro in the `ramdisk.c` source file, depending on their board's RAM capacity.

DMA Support

The CPPI 4.1 DMA engine is interfaced with MUSB controller, and MSC device class supports DMA mode for data transfers by default. Currently, only transparent DMA mode is supported for RX and TX transactions.**Please note: to switch to PIO mode, the `DMA_MODE` macro must be removed from the compiler options**.

Example Application

- Application calls `USBDMSCInit()` with the device instance structure as a parameter
- Application also initializes the DMA, MUSB interrupts, MMU, etc.
- The MSC initialization API provided by the MSC stack populates the device info structure with Endpoint numbers, descriptors, device state, etc.
- MSC init calls `DCDInit()` with the device instance structure as parameter. The core layer then performs the following steps in order to complete the enumeration:
  1. Enable the PSC clock
  2. Reset the USB module
  3. Switch on the USB PHY
  4. Initialize the USB tick module
  5. Clear all the pending interrupts
  6. Enable the required interrupts
  7. Set the configuration parameters
  8. Disconnect the device
  9. Reconnect the device
- Once enumeration is complete, the device will appear on the host PC as a mass storage device
- When the user attempts to access the MSC device on the host PC, the host formats the RAM disk
- Once the format operation is complete, the device is ready for data transaction
- If any command comes from host, the device is notified through an interrupt
- The interrupt handler in the device core (`HandleEndpoint()`) reads the CBW and retrieves the command op code
- This op-code is passed to the command handler, and command handler calls the appropriate command
- The data transaction happens through DMA by default
- At the end of the transaction, The device will send a command status word (CSW) to indicate the end of the command

Running the Application

- Connect the target board to the host PC via a USB cable
- Load and run the target application
- Once the device is connected to the host PC, the device will enumerate as an MSC device
- The user can check the disk drives to see the mass storage device
- When the user attempts to access the MSC device on the host PC, the host formats the RAM disk
- Once the format operation is complete, the device is ready for data transaction
- Currently the device supports 16MB of RAM disk

For connector information please check the connector informationtable

Note: The MSC device class is configured and tested for Hi-Speed DMA mode only

Modules used in this example

- Interrupt module
- USB driver
- USB stack (including Cppi 4.1 DMA)
- LCD raster
- Graphics library

How to Integrate Alternate Block Media

The current package is provided with 16 MB of RAM Disk. If the user wants to use another block media, the following steps are required.

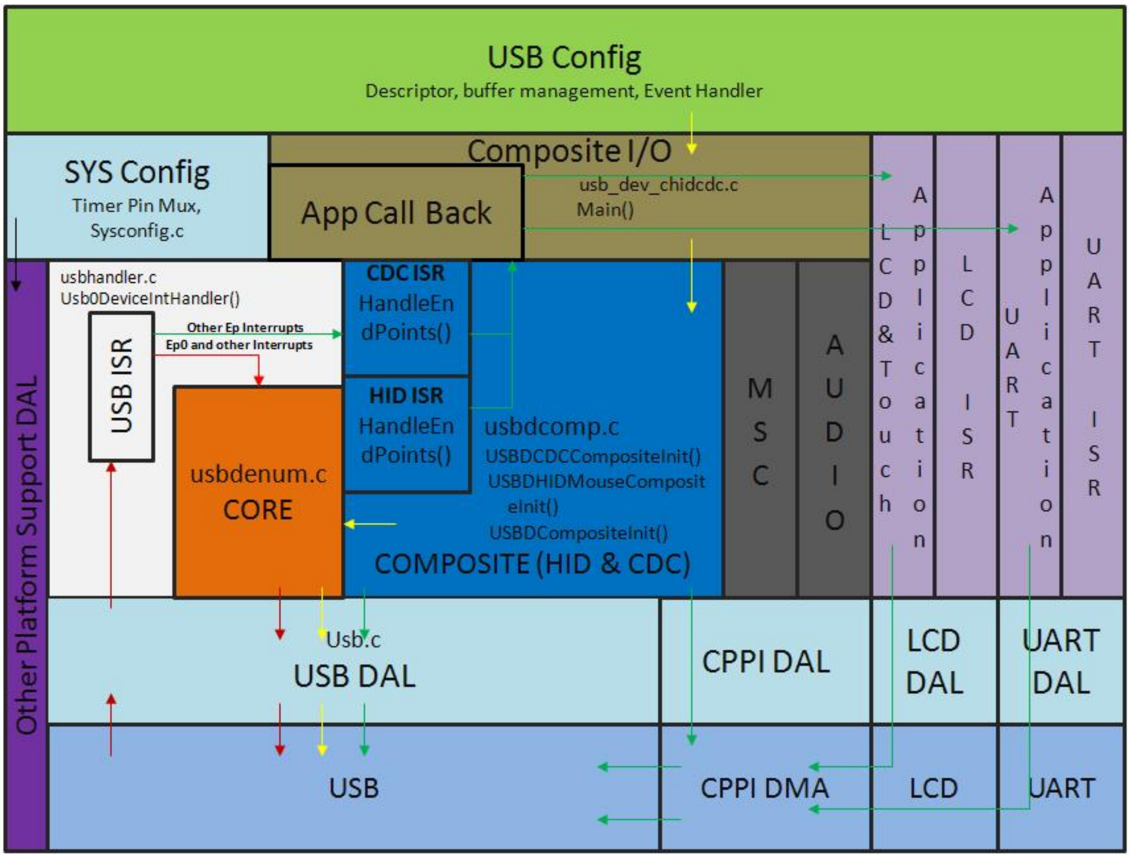
- Rename the source files `usbdmscglue.c` and `usbdmscglue.h` as appropriate
- The alternate block media must support the following APIs:
  1. `Disk_read()`
  2. `Disk_write()`
  3. `Disk_ioctl()`
- These APIs should be replaced in the new "glue" source file:
  1. `Disk_read()` should be replaced in the `USBDMSCStorageRead()` API
  2. `Disk_write()` should be replaced in the `USBDMSCStorageWrite()` API
  3. `Disk_ioctl()` should be replace in the `USBDMSCStorageNumBlocks()` API with appropriate parameters to return the total number of blocks.

The default block size is 512 bytes. If the alternate block media has a different block size, then this must be updated in the USB stack. This can be achieved by updating macros `DEVICE_BLOCK_SIZE` and `MAX_TRANSFER_SIZE` with the new block size.

Composite Device Class

The USB composite device class allows classes that are already defined in the USB library to be combined into a single composite device. The device configuration descriptors for the included device classes are merged at run time and returned to the USB host controller during device enumeration as a single composite USB device. Since each device class requires some unique initialization, the device classes provide a separate initialization API that does not touch the USB controller but does perform all other initialization. The initialization of the USB controller is deferred until the USB composite device is initialized and has merged the multiple device configuration descriptors into a single configuration descriptor so that it can properly initialize the USB controller. The endpoint numbers, interface numbers, and string indexes that are included in the device configuration descriptors are modified by the USB composite device class so that the values are valid in the composite device configuration descriptor.

Composite CDC Serial and HID-Mouse device Example Application



Device Enumeration

The Application must perform the following steps to enumerate the USB in Composite device mode:

- Configure the system interrupts
- Register the Interrupt handlers
- Initialize the buffers
- Call the `USBDHIDMouseCompositeInit()` API with the mouse device instance structure as a parameter.With this call, control passes to the HID mouse layer, which performs the following additional steps
  - Initialize the various fields in the instance structure
  - Initialize the HID device class instance structure based on input from the caller
  - Initialize the lower layer HID driver by calling the `USBDHIDCompositeInit()` API.With this call, control passes to the HID device class layer, which performs the following additional steps
    - Assign the endpoints to the device instance structure.
    - Assign the device, configuration descriptor to the device instance structure.
- Call the USBDCDCCompositeInit API with the CDC device instance structure as a parameter.With this call, control is given to the CDC device class layer. The device class layer must perform the following steps
  - Assign endpoints to the device instance structure
  - Apply the configuration descriptor to the device instance structure
- Call the USBDCompositeInit API to initialize the USB composite device class, with the composite device instance structure as a parameter.with this call, contol is given to the composite device class layer. The device class layer must perform the following steps
  - Initialize the various fields in the instance structure
  - Assign the device, configuration descriptor to the device instance structure

- Call the BuildCompositeDescriptor() API to merge the configuration descriptors into a single multiple instance device.
- Call the USBDCDInit() API and pass the device instance structure to the core

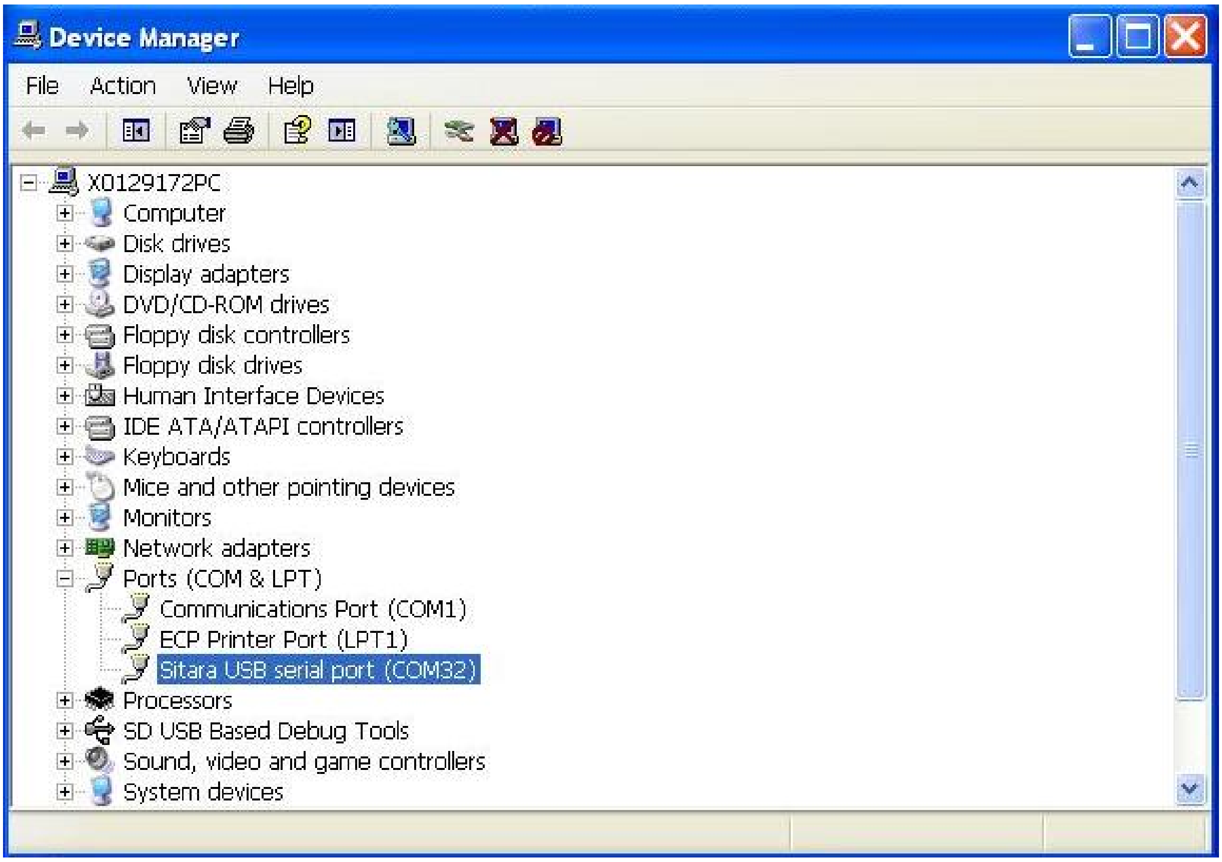
With this call, control passes to the core layer, which performs the following additional steps to complete enumeration:

- Enable the PSC clock
- Reset the USB module
- Switch on the USB PHY
- Initialize the USB tick module
- Clear all the pending interrupts
- Enable the required interrupts
- Set the configuration parameters
- Disconnect the device
- Reconnect the device
- Now the device starts receiving interrupts from the host. The interrupt handler in the core layer identifies all the Ep0 interrupts and calls the appropriate handler. If all the standard requests are serviced by the device, then enumeration is complete and the device is ready for communication. Now the control returns to the application, where it waits for touch screen inputs and data interrupts.

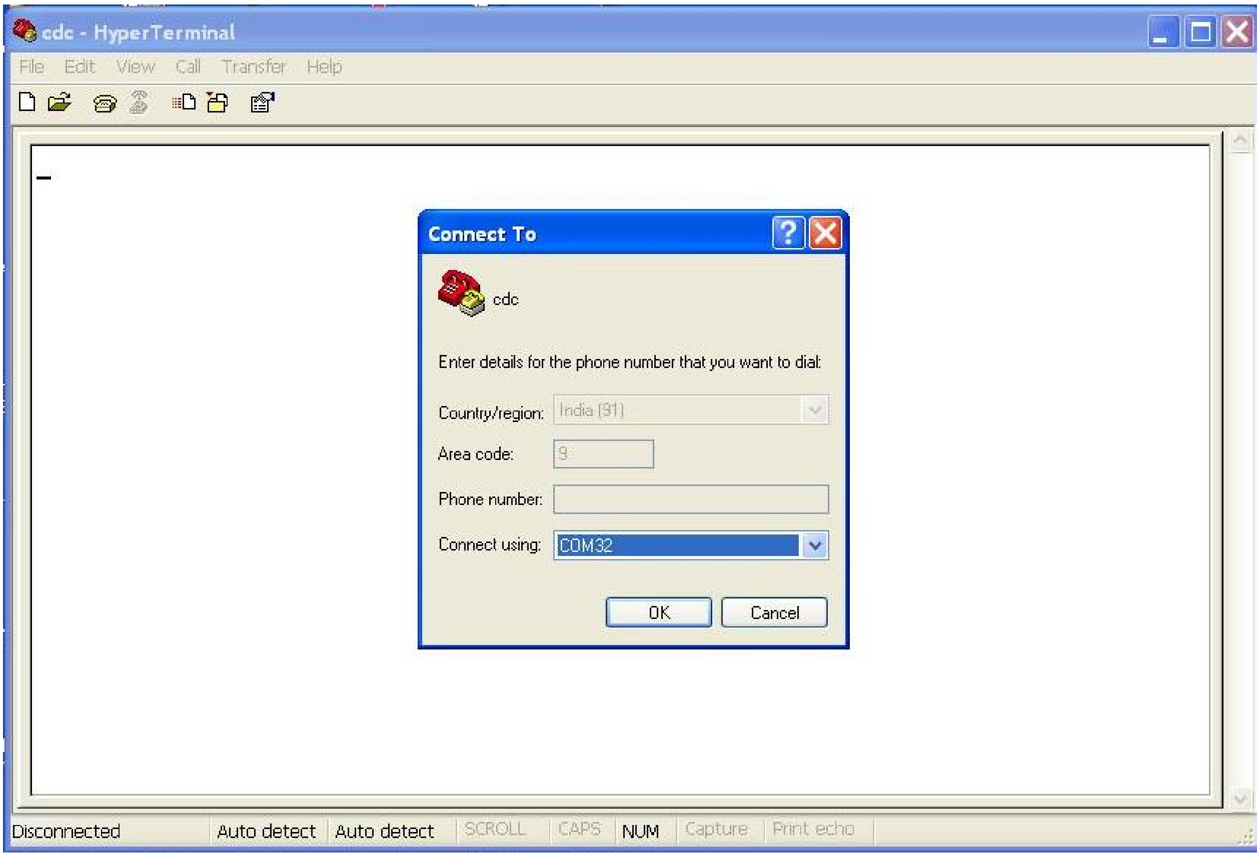
Running the Application

- **Since this is a USB device mode application, the target (EVM) must be connected to host (Windows PC) by a USB cable.**
- Load and run the target application
- Connect your host PC's serial port to the UART connector on the EVM using a null modem cable.
- The CDC device requires an INF file at the host side in order to enable as virtual COM port. Please use the INF provided in the `tools/usb_inf` folder.
  - The INF file is required only with a windows based host.
  - To use different VID and PID, the INF file must be updated with the new VID, PID and Interface ID.
- Once the device is connected to the host,it enumerates one device as HID mouse and for other CDC device host will ask for INF file. The user must browse to the INF file to proceed.

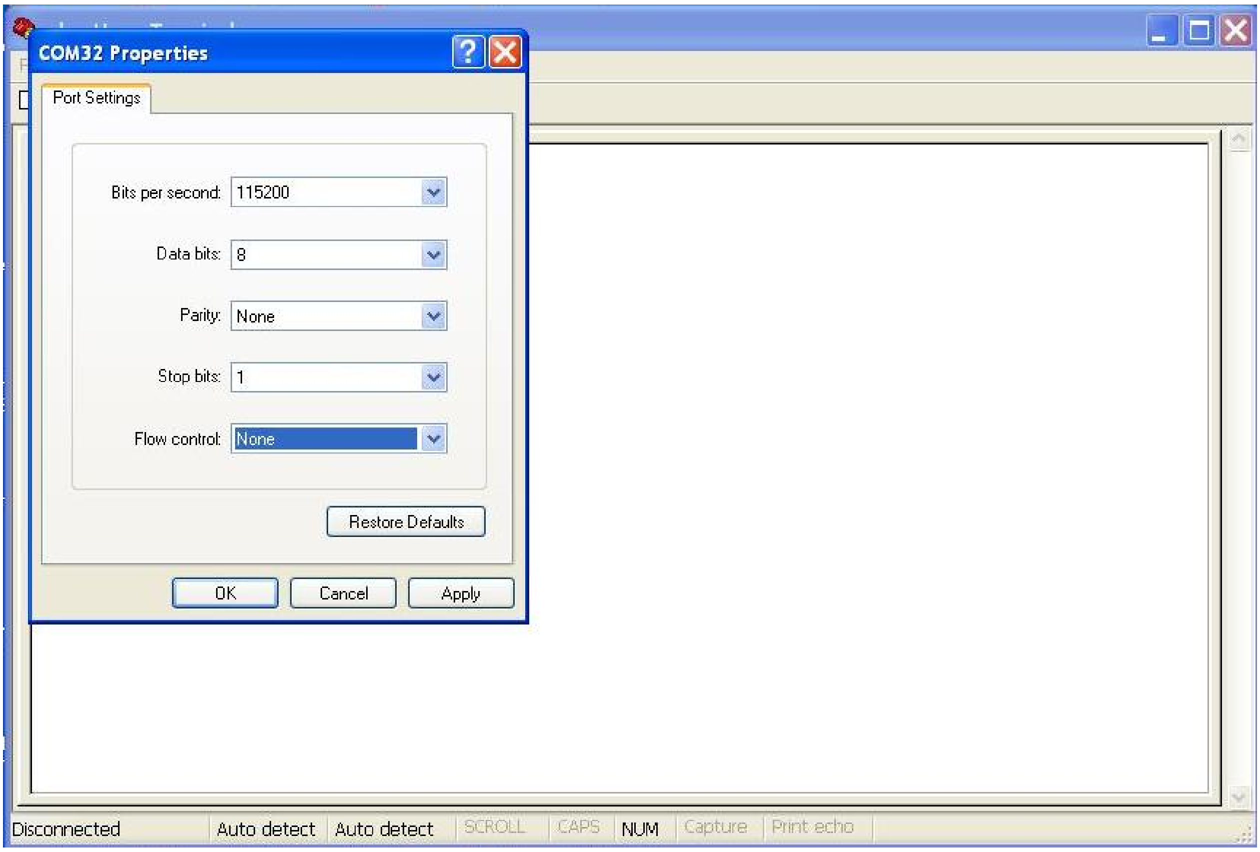
- Once the proper INF file is pointed out by the user, then virtual COM port will appear in the device manager



- Open a serial terminal application (ex. Teraterm, Hyperterminal) and choose the virtual COM port.



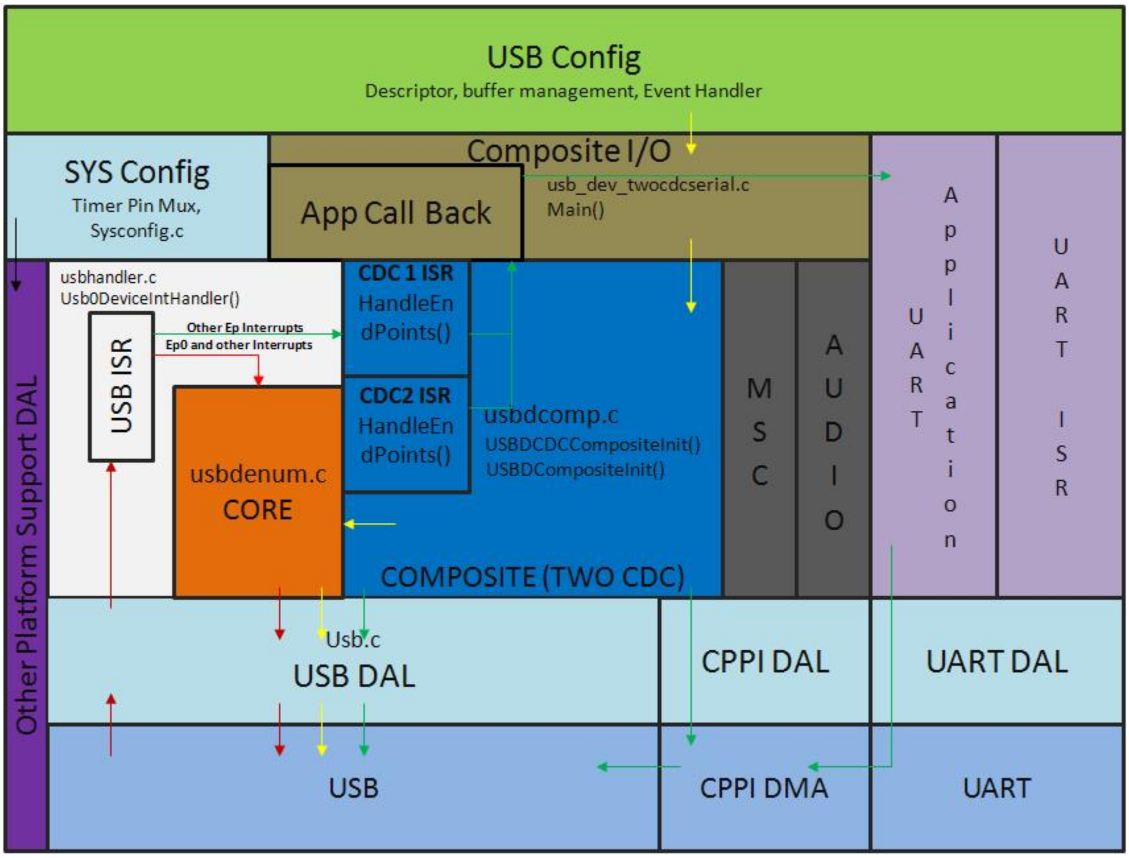
- Set the parameters as shown.



- Open another terminal window and choose COM1 (or whichever serial port you previously connected to the EVM's UART connector).
- Type in either terminal window on the host PC. The text should appear on the other terminal screen.

- For HID device, Touch screen is initialized and buttons are displayed
- User can use the touch screen to move the cursor on the host PC and press the mouse button to trigger mouse clicks.
- Both devices (HID mouse and CDC Serial) can be accessed at the same time.

Composite Dual CDC Serial Port device Example Application



Device Enumeration

The Application must perform the following steps to enumerate the USB in Composite device mode:

- Configure the system interrupts
- Register the Interrupt handlers
- Initialize the buffers
- Call the USBDCDCCompositeInit API twice with the CDC device instance structure as a parameter.With this call, control is given to the CDC device class layer. The device class layer must perform the following steps
  - Assign endpoints to the device instance structure
  - Apply the configuration descriptor to the device instance structure
- Call the USBDCompositeInit API to initialize the USB composite device class, with the composite device instance structure as a parameter.with this call, contol is given to the composite device class layer. The device class layer must perform the following steps
  - Initialize the various fields in the instance structure
  - Assign the device, configuration descriptor to the device instance structure
  - Call the BuildCompositeDescriptor() API to merge the configuration descriptors into a single multiple instance device.
  - Call the USBDCDInit() API and pass the device instance structure to the core

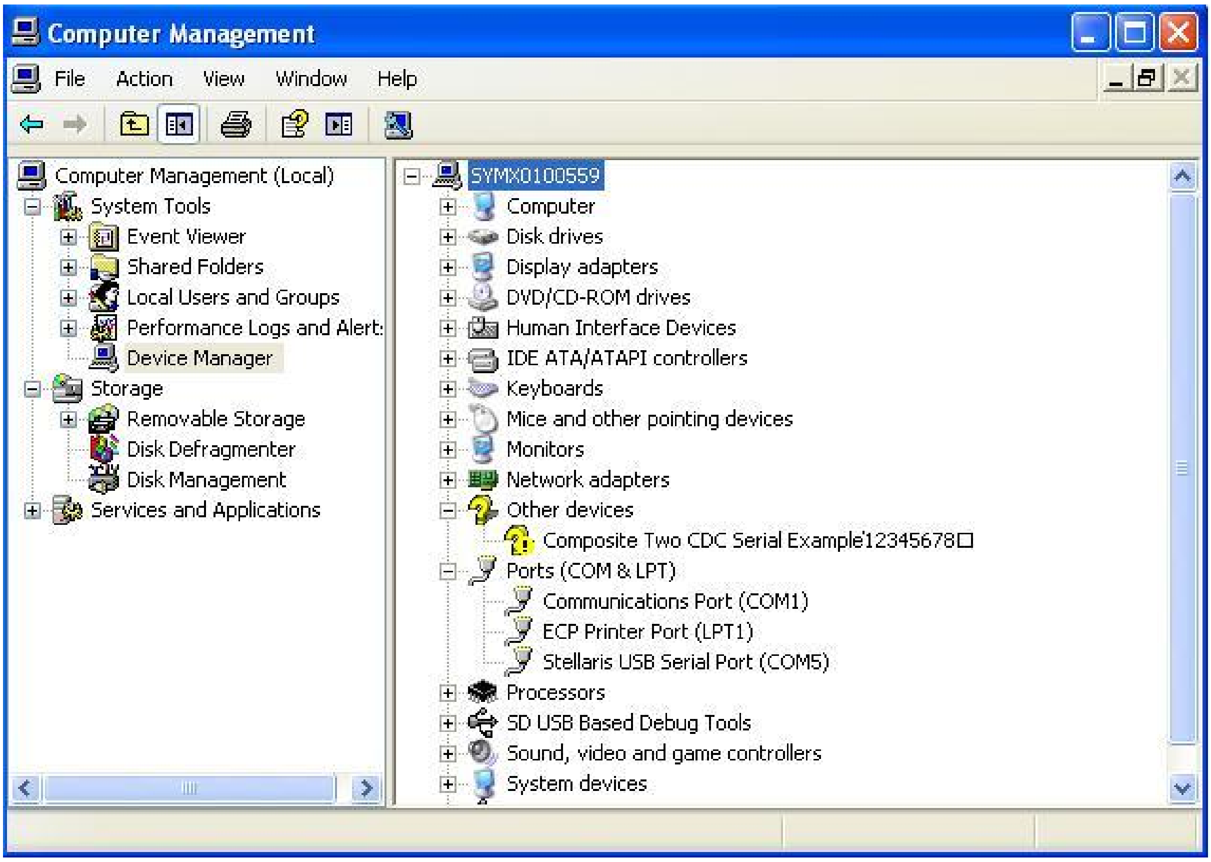
With this call, control passes to the core layer, which performs the following additional steps to complete enumeration:

- Enable the PSC clock
- Reset the USB module
- Switch on the USB PHY
- Initialize the USB tick module
- Clear all the pending interrupts
- Enable the required interrupts
- Set the configuration parameters
- Disconnect the device
- Reconnect the device
- Now the device starts receiving interrupts from the host. The interrupt handler in the core layer identifies all the Ep0 interrupts and calls the appropriate handler. If all the standard requests are serviced by the device, then enumeration is complete and the device is ready for communication. Now the control returns to the application, where it waits for data interrupts.

Running the Application

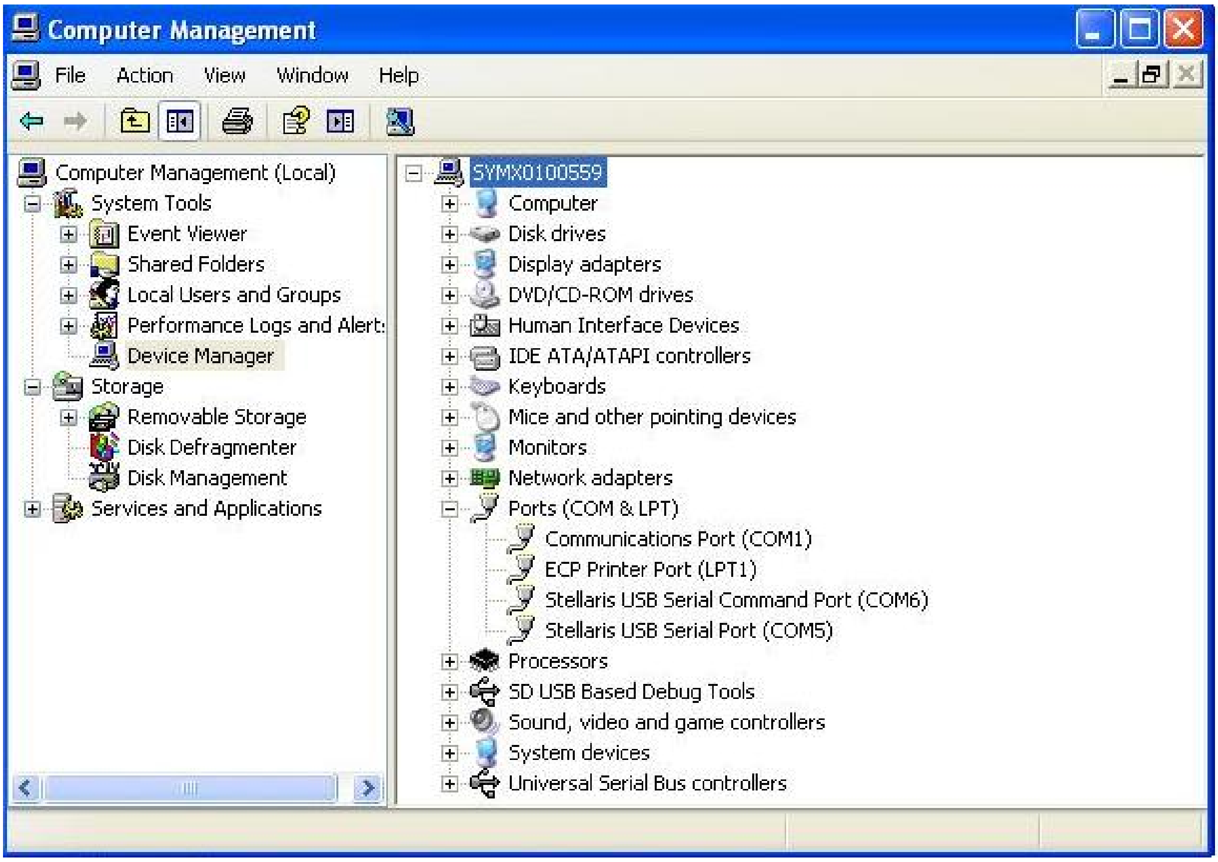
- **Since this is a USB device mode application, the target (EVM) must be connected to host (Windows PC) by a USB cable.**
- Load and run the target application
- The CDC device requires an INF file at the host side in order to enable as virtual COM port. Please use the INF provided in the `tools/usb_inf` folder.
  - The INF file is required only with a windows based host.
  - To use different VID and PID, the INF file must be updated with the new VID, PID and Interface IDs.
- Once the device is connected to the host,it enumerates as composite device, host will ask for INF file to enumerate one CDC device. The user must browse to the INF file to proceed.

- Once the proper INF file is pointed out by the user, then virtual COM port will appear in the device manager

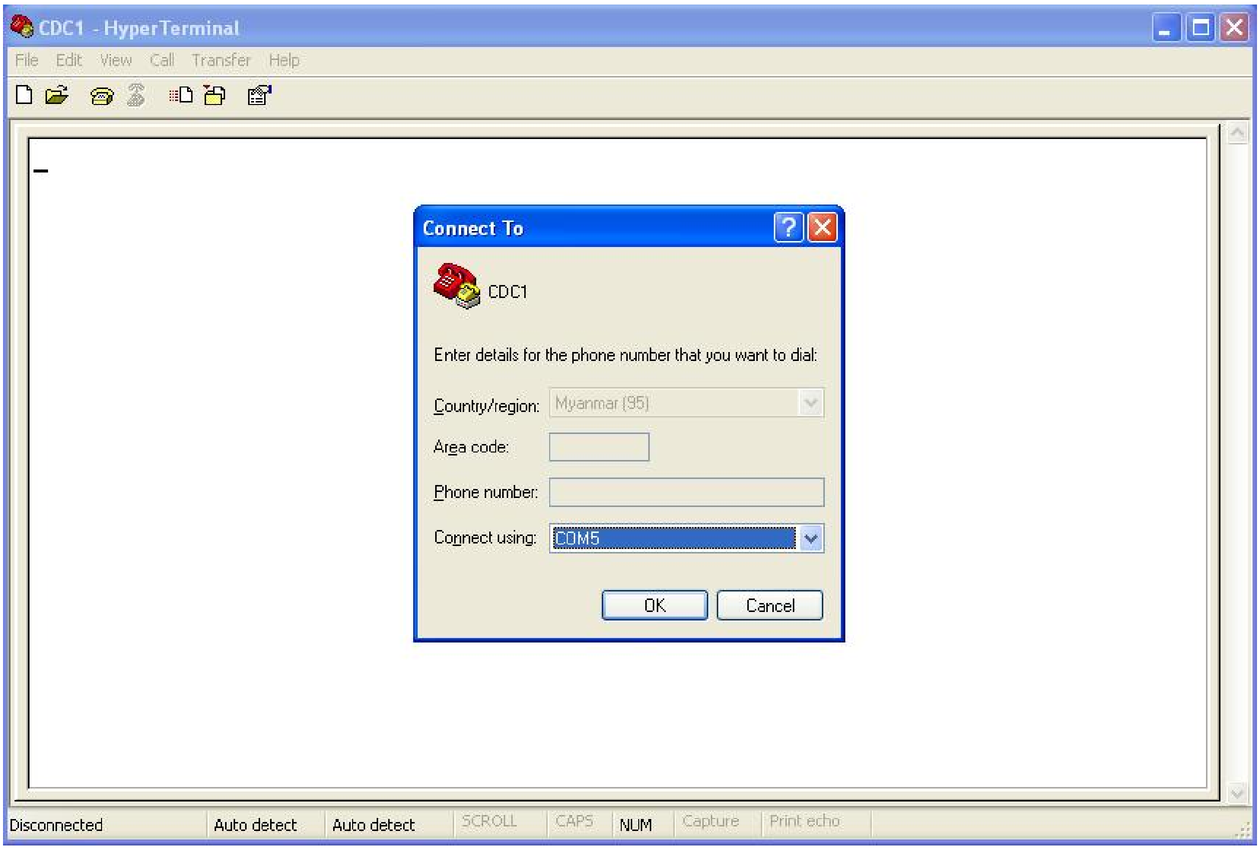


- Once one device is enumerated, host will ask for INF file to enumerate other CDC device. The user must browse to the INF file to proceed.

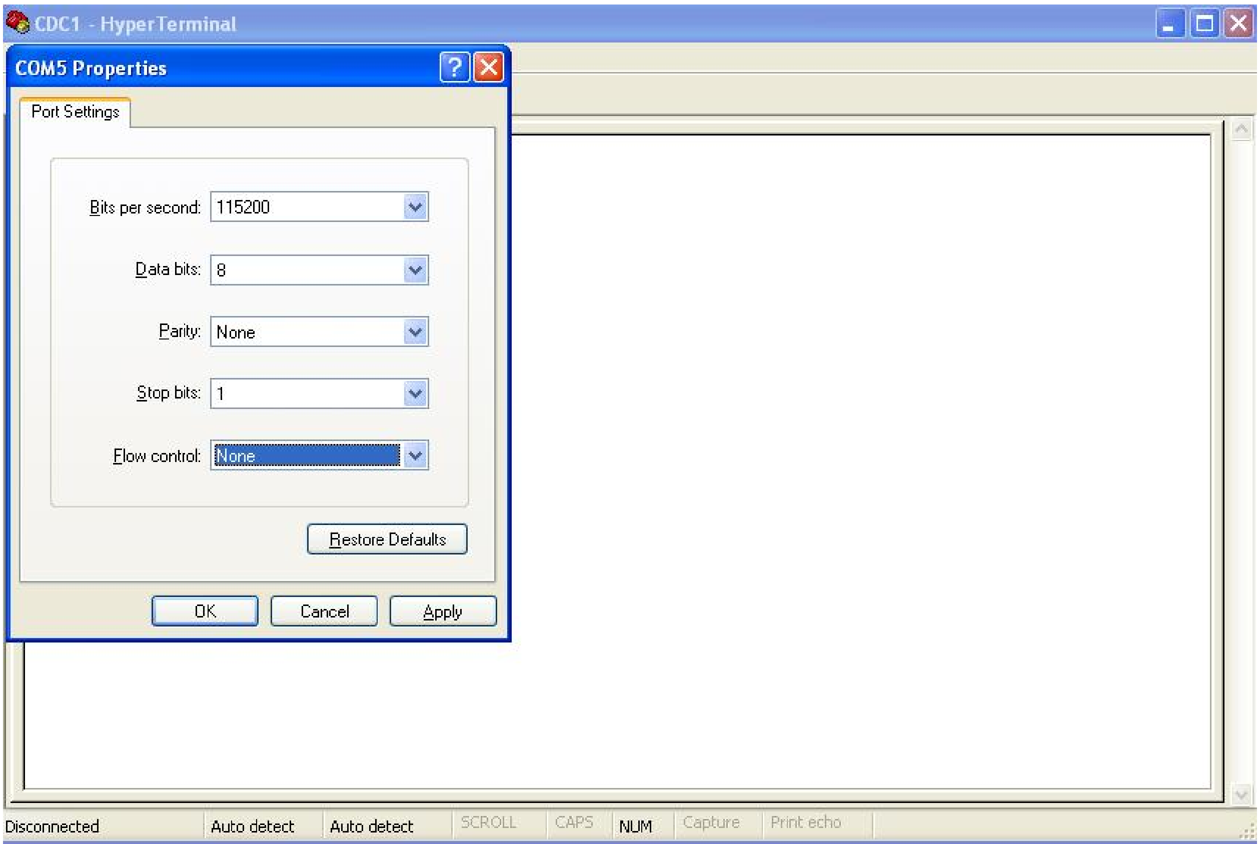
- Once the proper INF file is pointed out by the user, then two virtual COM ports will appear in the device manager



- Open a serial terminal application (ex. Teraterm, Hyperterminal) and choose the virtual COM port.



- Set the parameters as shown.



- Do the above two steps for other virtual COM port.
- Type in either terminal window on the host PC. The text should appear on the other terminal screen.

USB Host Class

The USB library host controller driver provides an interface to the host controller's hardware register interface. This is the lowest level of the driver interface, and it interacts directly with the driver library's USB APIs. The host controller driver provides all of the functionality necessary to perform enumeration of devices regardless of any type. This portion of the stack code only enumerates the device; higher level drivers handle further device operations. To allow the application to conserve code and data memory, the host controller driver only includes the host class drivers for USB device types used in the application. This allows an application to handle multiple classes of devices but only include the USB library code that the application needs for those devices that the application actually supports. While the host controller driver handles the enumeration of devices, it relies on USB pipes (allocated by the higher level class drivers) as the direct communications method with device end points.

Host Class Enumeration

The USB host controller driver handles all of the details necessary to discover and enumerate any USB device. The USB host controller driver only performs enumeration and relies on the host class drivers to perform any other communications with USB devices including the allocation of the endpoints for the device. Most of the code used to enumerate devices is run in interrupt context and is contained in the enumeration handler. In order to complete the enumeration process, the host controller driver also requires that the application periodically call the USBHCDMain() function. When a host class driver or an application needs access to endpoint 0 of a device, it uses the USBHCDControlTransfer() interface to send data to the device or receive data from the device. During the enumeration process the host controller driver searches a list of host class drivers provided by the application in the USBHCDRegisterDrivers() call. The details of this structure are covered in the host class drivers section of this document. If the host controller driver finds a host class driver that matches the class of the enumerated device, it will call the open function for that host class driver. If no host class driver is found the host controller driver will ignore the device and there will be no notification to the application. The host controller driver or the host class driver can provide callbacks up through the USB library to inform the application of enumeration events. The host class drivers are responsible for configuring the USB pipes based on the type of device that is discovered. The application will be notified that a new device has been discovered by a callback from the device interface that a device of that given type has been enumerated. When the device is removed the application will also get a callback that the device is no longer present. The events USB_EVENT_CONNECTED and USB_EVENT_DISCONNECTED are the only event notifications that will make it up to the application as a result of enumeration.

USB Pipes

The host controller driver layer uses interfaces called USB pipes as the primary method of communication with USB devices. These USB pipes are dynamically or statically allocated by the USB class drivers during enumeration. The USB pipes are usually only used within the USB library or by host class drivers and are usally not accessed directly by applications. The USB pipes are allocated and freed by calling `USBHCDPipeAlloc()` and `USBHCDPipeFree()` functions and are initially configured by calling `USBHCDPipeConfig()`. The `USBHCDPipeAlloc()` and `USBHCDPipeConfig()` APIs are used during USB device enumeration to allocate USB pipes for specific endpoints of the USB device. On disconnect, the `USBHCDPipeFree()` API is called to free up the USB pipe, which can then be used by a new USB device. While in use, the USB pipes can provide status and perform read and write operations. Calling `USBHCDPipeStatus()` allows a host class driver to check the status of a pipe. However, most access to the USB pipes occurs through the `USBHCDPipeWrite()` and `USBHCDPipeRead()` APIs and the callback function provided when the USB pipe was allocated. These are used to read or write to endpoints on USB devices on endpoints other than the control endpoint on endpoint 0. Since endpoint 0 is shared with all devices, the host controller interface does not use USB pipes for communications over endpoint 0 and instead uses the `USBHCDControlTransfer()` API.

Control Transactions

All USB control transactions are handled through the `USBHCDControlTransfer()` API. This function is primarily used inside the host controller driver itself during enumeration, but some devices may require using control transactions through endpoint 0. The HID class drivers are a good example of a USB class driver that uses control transactions to send data to a USB device. The `USBHCDControlTransfer()` API should not be called from within interrupt context because control transfers are blocking operations that rely on interrupts to proceed. Since most callbacks occur in interrupt context, any calls to `USBHCDControlTransfer()` should be deferred until running outside the callback event.

Interrupt Handling

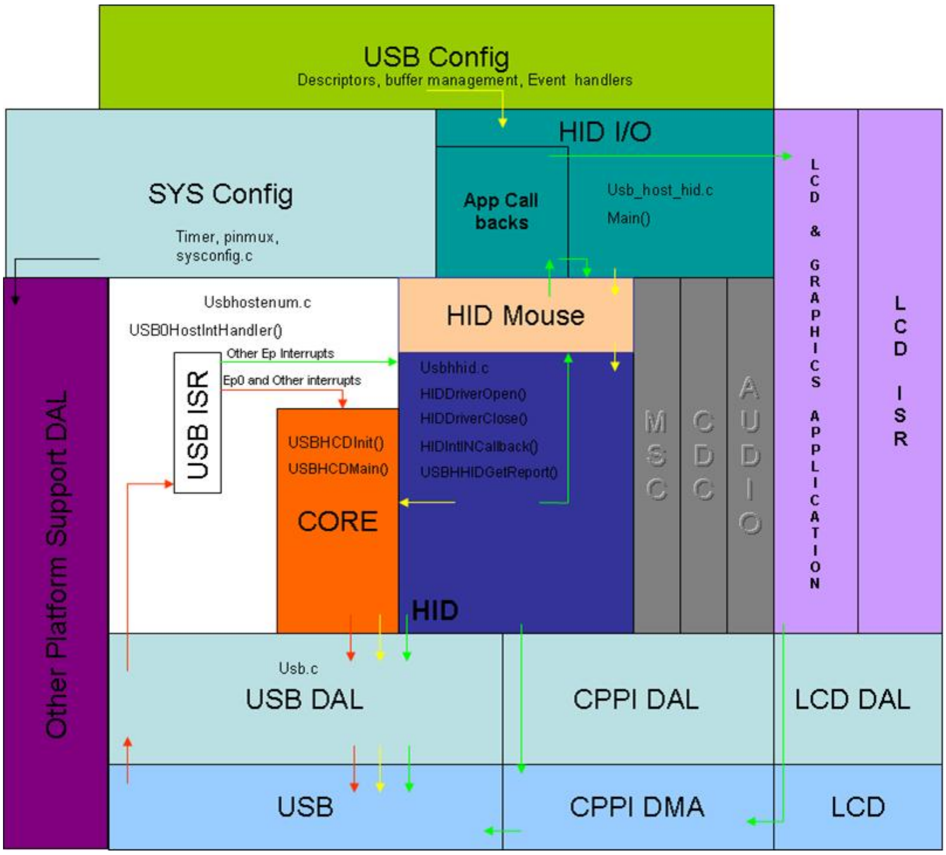
All interrupt handling is done by the USB library host controller driver. Most callbacks are called in interrupt context and, like interrupt handlers, should defer any real processing of events until execution returns from the interrupt context. Callbacks are used to notify the upper layers when events occur during enumeration or during normal operation. Because most of enumeration code is handled by interrupt handlers, the enumeration code requires the application to call `USBHCDMain()` in order to progress through the enumeration states without running all code in an interrupt context.

HID Host Class

The host class drivers provide access to devices that use a common USB class interface. In order to use the class drivers, the application must call `USBHCDRegisterDrivers()` to provide a list of the host class drivers that the application will use. The `g_USBHIDClassDriver` structure defines the interface for the Host HID class driver, and the host class driver provides the following interfaces:

- To the USB host controller driver (bottom layer)
- Device specific interfaces (top layer)

The lower layer interface to the USB host controller interface is the same for all USB host class drivers, while the device interface layer on top is common to all USB host device drivers of a given class. Aside from enumeration, all communication with the host class driver happens through its endpoint pipes. The host class driver parses and allocates any endpoints that it needs by calling the `USBHCDPipeAlloc()` and `USBHCDPipeConfig()` functions. These USB pipes provide methods to read/write and receive callback notification from the USB host controller driver layer.



The HID class driver provides access to any type of HID class by leaving the details of the HID device to the layer above the HID class driver. The top layer of the HID class driver provides common functions to open or close an instance of a HID device, read a device's report descriptor so that it can be parsed by the HID device code, and get/set reports on an HID device. The lower level interface that is connected to the host controller driver is specified in the `g_USBHIDClassDriver` structure. This structure is used to register the HID class driver with the host class driver so that it is called when an HID device is connected and enumerated. The functions in the `g_USBHIDClassDriver` structure should never be called directly by and application or a host class driver as they are reserved for access by the host controller driver. In the following example, the generic HID class driver is registered with the USB host controller driver and then a call is made to open an instance of a mouse class device. Typically, the call to `USBHHIDOpen()` is made from within a device class interface while the call to `SBHCDRegisterDrivers()` is made from the application directly. For instance, the `USBHHIDOpen()` API for the mouse device provided with the USB library is called from `USBHMouseOpen()`, which is part of the USB mouse interface.

Device Interface

At the top layer of the HID class driver, the driver has a device class interface for use with various HID devices. In order for the HID class driver to recognize a device, the device class must call `USBHHIDOpen()`. This call specifies the type of device and a callback for this device type so that any events related to this device type can be passed back to the device class driver. The defined classes are in the `tHIDSubClassProtocol` type and are passed into the `USBHHIDOpen()` call via the `eDeviceType` parameter. In order to release an instance of an HID class driver, the HID device class or application must call `USBHHIDClose()` to allow a new or different type of device to be connected. In the examples provided with the USB library, the report descriptors are retrieved but are not used as the examples rely on the "boot" mode of the USB keyboard and mouse to fix the format of the report descriptors. This is accomplished by using the `USBHHIDSetReport()` API to force the device into its boot protocol mode. As this could be limiting or not available in other types of applications or devices, the

`USBHHIDGetReportDescriptor()` API provides the ability for generic HID devices to query the device for its report descriptor(s). The last two remaining HID APIs, `USBHHIDSetReport()` and `USBHHIDGetReport()`, provide access to the HID reports.

Once a HID device has been opened, the application receives a `USB_EVENT_CONNECTED` callback event. This indicates that a HID device of the type passed into the `USBHHIDOpen()` has been connected and the USB library host controller driver has completed enumeration of the device. When the HID device is removed, a `SB_EVENT_DISCONNECTED` event occurs. When shutting down or to release a device, the application must call `USBHHIDClose()` to disable callbacks. This does not actually power down the device, but it stops the driver from calling the application. During normal operation, the host class driver receives `USB_EVENT_SCHEDULER` and `USB_EVENT_RX_AVAILABLE` events. The `USB_EVENT_SCHEDULER` event indicates that the HID class driver should schedule a new request if it is ready to do so. This done by calling `USBJHCDPipeSchedule()` to request that a new IN request be made on the given Interrupt IN pipe. When the `USB_EVENT_RX_AVAILABLE` occurs, this indicates that new data is available due to completion of the previous request for data on the Interrupt IN pipe. The `USB_EVENT_RX_AVAILABLE` is passed on the device class interface to allow it to request the data by calling `USBHHIDGetReport()`. It is up to the device class driver to interpret the data in the report structure that is returned. In some cases, like the keyboard example, the device class may also need to call the host class driver to issue a set report to send data to the device. This is done by calling `USBHHIDSetReport()` in the host class driver. This will send data to the device by using the correct USB OUT pipe.

Example Application (Mouse)

The USB library host stack initialization is handled in the `USBHCDInit()` API. This function must be called after registering class drivers using `USBHCDRegisterDrivers()` and (optionally) configuring power pins using `USBHCDPowerConfigInit()`. Both of these APIs are described later.

The `USBHCDInit()` API takes three parameters, the first of which specifies which USB controller to initialize. This is a zero-based index value. The next two parameters specify a memory pool for use by the host controller driver. The size of this buffer must be large enough to hold a typical configuration descriptor for devices that are going to be supported. This value is system-dependent, so it is left to the application to set the size. It should never be less than 32 bytes; in most cases it should be at least 64 bytes. If there is not enough memory to load a configuration descriptor from a device, the device will not be recognized by USB library’s host controller driver. The USB library also provides a method to shut down an instance of the host controller driver by calling the `USBHCDTerm()` function. The `USBHCDTerm()` function should be called any time the application wants to shut down the USB host controller in order to disable it, or possibly switch modes in the case of a dual role controller. The USB library assumes that the power pin configuration has an active high signal for controlling the external power. If this is not the case or if the application wants control over the power fault logic provided by the library, then the application should call `USBHCDPowerConfigInit()` before calling `USBHCDInit()` in order to properly configure the power control pins. The polarity of the power pin, the polarity of the the power fault pin, and any actions taken in response to a power fault are all controlled by passing a combination of values in the `ulPwrConfig` parameter. See the documentation for the `USBHCDPowerConfigInit()` API for more details on this function.

The USB library host stack requires that some portion of the code not run in the interrupt handler, so the `SBHCDMain()` function must be called periodically by the application during normal execution. This can be as a result of a timer tick or just once per main loop in a simple application. It should not be called by an interrupt handler. Calling the function too often is harmless as it will simply return if the USB host stack has no pending tasks. Calling `USBHCDMain()` too infrequently can cause enumeration to take longer than normal. It is up to the application to prioritize the importance of USB communications by calling `USBHCDMain()` at a rate that is reasonable to the application. All support devices will have a host class driver loaded in order to communicate with each type of device that is supported. The details of interacting with these host class drivers is explained in the host class driver sections that follow on this page.

When the application needs to shut down the host controller, it must shut down all host class drivers before shutting down the host controller itself. This gives the host class drivers a chance to close cleanly by calling each host class driver's close function. Finally, the `USBHCDTerm()` function should be called to shut down the host controller. This sequence will leave the USB controller and the USB library stack in a state such that it is ready to be re-initialized or switched to USB device mode (i.e. from host).

Running the Application

- Connect a USB mouse to the target board
  - **Note:** On the OMAP-L138/AM1808/C6748 EVM, use the J6 (mini) USB connector
- Load and run the target application
- The display at the bottom left of the screen will show "Connected" if the mouse is enumerated and connected to the host
- Move the mouse and see its motion tracked by the cursor on the screen
- Press the mouse buttons and observe the button indicator on the bottom right corner of the screen
- Press and hold the left mouse button and drag the cursor to draw lines or curves on the screen
- Unplug the mouse from the board and see the display at bottom left changes to "No device"

For connector information please check the connector information table

Modules used in this application:

- USB HID host
- LCD raster
- Graphics lib rary
- Timer
- Interrupt

Writing a New HID Host Application

The following shows the basic setup code needed for any application that is using the USB library in host mode.

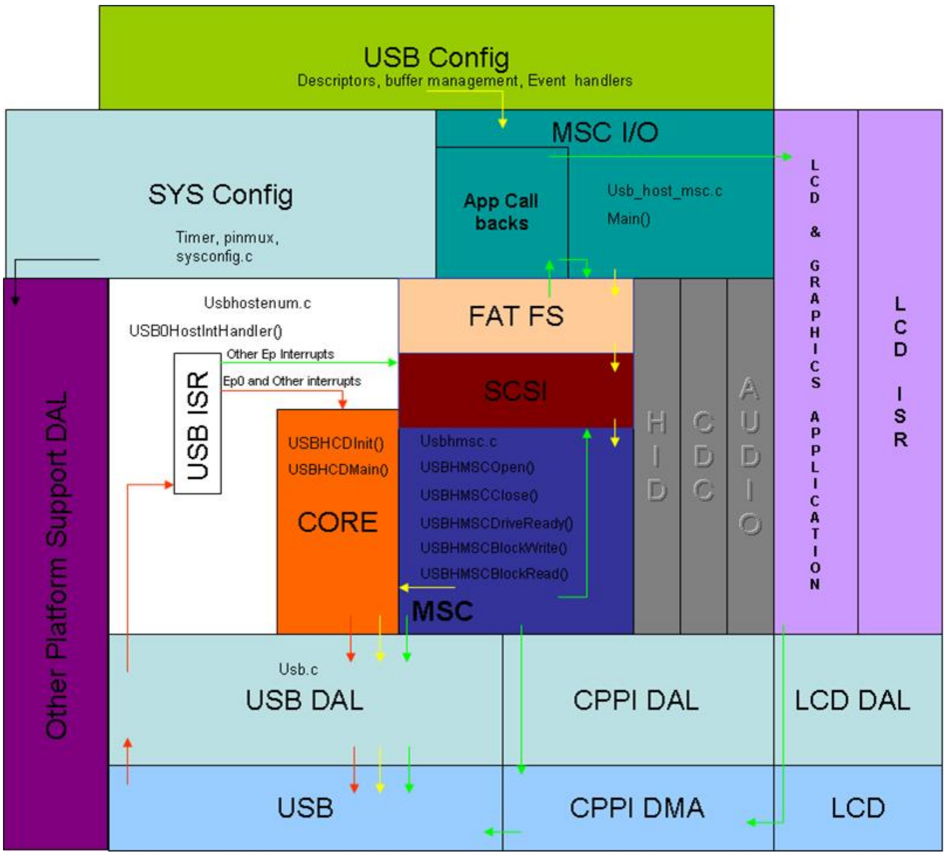
- The `g_pHCDPool` array that is passed to the `USBHCDInit()` API is used as heap memory by the USB library. This memory should not be used by the application. In the HID host mouse example, the `g_ppHostClassDrivers` array holds HID driver that makes it possible to support various devices.
- When calling `USBHCDRegisterDrivers()`, one argument specifies a static array of supported USB host class drivers that are be supported by the application.
- The application must initialize the interrupt controller and register the ISR for the MUSB controller.
- The application should always call the USB device interface open routines before calling `USBHCDInit()`. This enables the USB host controller and starts enumerating any connected device.
- The initial call to `USBHMouseOpen()` prepares the mouse device application interface to receive notifications from any USB mouse device that is connected.
- Since the mouse interface needs some basic configuration after being connected, the application should wait for the mouse to be connected and then call `USBHMouseInit()` to complete the mouse configuration.
- Once the mouse has been configured, the application's mouse callback routine will be notified any time there is a state change with the mouse. This includes the switching to the `MOUSE_INIT` state when a `USB_EVENT_CONNECTED` event occurs in order to trigger mouse device initialization.
- The `USB_EVENT_DISCONNECTED` event switches the state of the application to let it know that the mouse is no longer connected. The remaining events are mouse state changes that can be used by the application to move a

cursor or take action based on a mouse click.

```
//*****************************************************************************
// The size of the host controller's memory pool in bytes.
//*****************************************************************************
#define HCD_MEMORY_SIZE 128
//*****************************************************************************
// The memory pool to provide to the Host controller driver.
//*****************************************************************************
unsigned char g_pHCDPool[HCD_MEMORY_SIZE];
//*****************************************************************************
// The global that holds all of the host drivers in use in the application.
// In this case, only the Keyboard class is loaded.
//*****************************************************************************
static tUSBHostClassDriver const * const g_ppHostClassDrivers[] =
{
&g_USBHIDClassDriver,
&g_USBHostMSCClassDriver
};
//*****************************************************************************
// This global holds the number of class drivers in the g_ppHostClassDrivers
// list.
//*****************************************************************************
        static const unsigned long g_ulNumHostClassDrivers =
        sizeof(g_ppHostClassDrivers) / sizeof(tUSBHostClassDriver *);
        ...
        // Enable Clocking to the USB controller.

        // Enable the peripherals used by this example.

        // setup the interrupt controller
        SetupAINTCInt();

        //Configure and register the USB ISR
        ConfigureAINTCIntUSB();

        // Register the host class drivers.
        //
        USBHCDRegisterDrivers(0, g_ppUSBHostClassDrivers, g_ulNumHostClassDrivers);

        // Open an instance of the mouse driver. The mouse does not need
        // to be present at this time, this just saves a place for it and allows
        // the applications to be notified when a mouse is present.
        g_ulMouseInstance = USBHMouseOpen(MouseCallback, g_pucBuffer, 128);
        ...
        // Main loop of application.
        while(1)
        {
        switch(eMouseState)
        {
                // This state is entered when they mouse is first detected.
                case MOUSE_INIT:
                {
                        // Initialized the newly connected mouse.
                        USBHMouseInit(g_ulMouseInstance);
                        // Proceed to the mouse connected state.
                        eMouseState = MOUSE_CONNECTED;
                        break;
                }
                case MOUSE_CONNECTED:
                {
                        break;
                }
                case MOUSE_NOT_CONNECTED:
                default:
                {
                        break;
                }
        }

                // Periodic call the main loop for the Host controller driver.
                USBHCDMain();
}
```

Mass Storage Host Class

The mass storage host class driver allows applications to access external devices that support the mass storage class protocol. The driver provides a simple block based interface to the devices that can be matched up with an application's file system. A USB host class driver for MSC devices is included with the USB library. It provides direct access to mass storage devices based on logical block address.



The mass storage host class driver provides APIs to access USB mass storage drives. These APIs are meant to match with file systems that need block based read/write access to the drives. The `USBHMSCBlockRead()` and

USBHMSCBlockWrite() APIs provide block read and block write device access. These functions perform block operations at the size specified by the drive. Since some drives require setup time after enumeration before they are ready for drive access, the mass storage class driver provides the USBHMSCDriveReady() API to check if the drive is ready for normal operation. The mass storage host class driver also provides an interface to the USB library host controller driver to complete enumeration of mass storage class devices. The mass storage class driver information is held in the global structure g_USBMSCClassDriver. This structure should only be referenced by the application, and the function pointers in this structure should never called directly by anything other than the host controller driver. The USBHMSCOpen() and USBHMSCClose() APIs allow the host controller's enumeration code to signal when a mass storage class device is detected or removed. The mass storage host class driver is responsible for providing a callback to the file system or application for notification when the drive is connected or disconnected. To make the the mass storage class driver visible to the host controller driver, it must be added in the list of drivers provided when calling USBHCDRegisterDrivers(). The class enumeration constant is set to USB_CLASS_MASS_STORAGE so any devices enumerating with value will load this class driver.

Device Interface

This section describes how an application or file system interacts with the host mass storage class driver provided by the USB library. The application or file system must register the mass storage class driver with a call to USBHCDRegisterDrivers() with g_USBHostMSCClassDriver provided as a member of the array argument. Once the host mass storage class driver has been registered, the application must call USBHMSCDriveOpen() to allow the application or file system to be called when a new mass storage device is connected or disconnected or when any other mass storage class event occurs.

The first callback is the USB_EVENT_CONNECTED event, indicating that a mass storage class flash drive has been inserted and the USB library host stack has successfully enumerated the device. This does not indicate that the flash drive is ready for read/write operations; the device has only been detected. The USBHMSCDriveReady() API can be called to determine when the flash drive is ready for read/write operations. When the device is removed, a USB_EVENT_DISCONNECTED event occurs. When shutting down, the application must call USBHMSCDriveClose() to disable callbacks. This does not actually power down the mass storage device, but it does stop the driver from calling the application with further events. Once calling USBHMSCDriveReady() indicates that the flash drive is ready, the application can use the USBHMSCBlockRead() and USBHMSCBlockWrite() APIs to access the device. These are block-based functions that use the logical block address to indicate which block to access. It is important to note that the size passed to these functions is specified in blocks, not bytes. The most common block size is 512 bytes. These calls always read or write a full block, so space must be allocated and managed appropriately. The following example shows calls for both reading and writing blocks from a mass storage class device.

Host SCSI Layer

Since most mass storage class devices adhere to the SCSI protocol for block based calls, the USB library provides SCSI functions for the mass storage class driver to communicate with MSC drives. The commands and data pass over the USB pipes provided by the host controller driver. The only types of mass storage class device that are supported are devices that use the SCSI protocol. Since flash drives only support a limited subset of the SCSI protocol, only the SCSI functions needed by mass storage class to mount and access flash drives are implemented. The SCSIRead10() and SCSIWrite10() APIs are the two functions used for reading and writing to mass storage class devices. The remaining SCSI functions are used to get information about the mass storage class devices, including the device's block size and total number of blocks. Other APIs are used for error handling or to check whether or not the device is ready for a new command.

Example Application

The application must call `USBHMSCDriveOpen()` in order for the application to be ready for a new mass storage device. The application should also wait for the mass storage class device to be ready to receive commands by calling `USBHMSCDriveReady()` and waiting for a return value of 0 before attempting to read or write to the device. Typically, reading and writing to the device is handled by a file system layer.

The example application executes the following sequence:

1. Configure and enable the interrupts
2. Enable the USB clocking
3. Register the host class driver
4. Open an instance of the mass storage class driver
5. Initialize the power configuration
6. Initialize the host controller
7. Initialize the file system
8. Check for the device readiness

Running the Application

- Connect a USB flash drive to the target board using Host cable
  - **Note:** On the OMAP-L138/AM1808/C6748 EVM, use the J6 (mini) USB connector
- Connect the board to a PC through UART. A serial terminal application should be running on the host. Refer here for more details. The serial terminal displays a command line interface to the user at run time
- Load and run the example application on to target
- Connect a mass storage device to the board
- The PC terminal should display “Mass storage device connected”
- Type "help" to see the supported commands
- Use the necessary command to brows through the storage device

Note: The MSC Host class is by default configured for Hi-Speed DMA transfer mode. For Full-Speed mode, define 'USB_MODE_FULLSPEED' in 'Drivers', 'USBLib' and 'Application' make files.

For connector information please check the connector information table

Modules used by this example:

- USB MSC host
- UART
- Timer
- Interrupt

Integrating a Different File System

Using a different file system rather than the one provided with the package (i.e. FatFs) requires the following steps:

- Look for a source file containing the wrapper function to insert the stack APIs. In the current package, `fat_usbmsc.c` is the file that integrates the stack with the file system.
- Find the appropriate file system initialization APIs and use them in the application.
- Replace the file Open/Read/Write APIs as necessary in the application.

Multi-instance Support Information

StarterWare USB package provides the necessary software support for multi-instance capability. The application can select each of the USB instance and initialize, configure and start separately as per the corresponding class specific details detailed above. The multi-instance support is demonstrated/tested using below sample application(s).

- **Note:** EVM-AM335x/EVMSK-AM335x/BeagleBone has a Micro connector for USB0 and a Standard-A receptacle for USB1. Thus, USB0 can be exercised in host mode or in device mode, while USB1 can be exercised only in host mode.

MSC Device + MSC Host Example Application

The initialization, data flow, interfacing and other details remains same as that for MSC Device Class and MSC Host Class. In this example, USB0 is used for device mode and USB1 is used for host mode.

The example application executes in the following sequence:

1. Configure and enable the interrupts
2. Enable the USB clocking
3. Initialize the power configuration
4. Register the host class driver
5. Initialize the host controller with USB1
6. Initialize DMA for Host mode
7. Open an instance of the mass storage class driver
8. Initialize DMA for Device mode with endpoint details
9. Initialize the USB0 for MSC Device class
10. Initialize the file system
11. Check for the device readiness in Host mode

Running the Application

- Connect a USB flash drive to the target board on USB1
- Connect the target board to the host PC via a USB cable from USB0
- Connect the board to a PC through UART. A serial terminal application should be running on the host. Refer here for more details. The serial terminal displays a command line interface to the user at run time
- Load and run the example application on to target
- Host-mode: The PC terminal should display "Mass storage device connected"
- Host-mode: Type "help" to see the supported commands
- Host-mode: Use the necessary command to browse through the storage device
- Device-mode: On the host PC, the device will appear as a mass storage device
- Device-mode: On attempting to access the MSC device on the host PC, the host formats the RAM disk
- Device-mode: Once the format operation is complete, the device is ready for data transaction
  - **Note:** Device-mode: Currently the device supports 16MB of RAM disk
  - **Note:** User will have to wait till the current issued command completes to issue next command

For connector information please check the connector information table

Modules used by this example:

- USB MSC host
- USB MSC Device
- UART
- Timer

- Interrupt

MSC Host + MSC Mouse Example Application

The initialization, data flow, interfacing and other details remains same as that for Mouse Host Class and MSC Host Class. In this example, USB0 is used for MSC host mode and USB1 is used for Mouse host mode.

The example application executes in the following sequence:

1. Configure and enable the interrupts
2. Enable the USB clocking
3. Initialize the power configuration
4. Register the Mouse host class driver
5. Initialize the host controller and register mouse driver with USB1
6. Register the MSC host class driver
7. Initialize DMA for MSC Host mode
8. Initialize the host controller for MSC with USB0
9. Initialize the file system for MSC class
10. Check for the device readiness in Host mode

Running the Application

- Connect a USB flash drive to the target board on USB0
- Connect a USB mouse to the target board on USB1
- Connect the board to a PC through UART/virtual serial port. A serial terminal application should be running on the host. Refer here for more details. The serial terminal displays a command line interface to the user at run time
- Load and run the example application on to target
- MSC Host-mode: The PC terminal should display “Mass storage device connected”
- MSC Host-mode: Type "help" to see the supported commands
- MSC Host-mode: Use the necessary command to browse through the storage device
- Mouse Host-mode: The display at the bottom left of the screen will show “Connected” if the mouse is enumerated and connected to the host
- Mouse Host-mode: Move the mouse and see its motion tracked by the cursor on the screen
- Mouse Host-mode: Press the mouse buttons and observe the button indicator on the bottom right corner of the screen
- Mouse Host-mode: Press and hold the left mouse button and drag the cursor to draw lines or curves on the screen
- Mouse Host-mode: Unplug the mouse from the board and see the display at bottom left changes to “No device”

Note: User will have to wait till the current issued command completes

For connector information please check the connector information table

Modules used by this example:

- USB MSC host
- USB Mouse host
- UART
- Timer
- Interrupt

Performance

Steps to capture performance

- Open 'usblib\include\usblib.h'. Define 'PROFILE_USB_MSC_WRITE' OR 'PROFILE_USB_MSC_READ' based on what is being profiled. Build the example usb_dev_msc (Note: Below performance figures are captured using GCC build).
- Setup the CCS IDE and h/w environment as documented in USB user-guide to execute usb_dev_msc example.
- Add variable 'g_USBPerfInfo' to watch window.
- Execute the example.
- Format and copy two ~50MB files to USB device. Copy the two ~50MB files back to PC.
- Eject the device.
- "Pause" the example execution using CCS IDE.
- Open "memory browser" of CCS,
  - select "save memory" option,
  - provide <filename>.hex and click 'Next' button
  - Select 'Raw Data' option for 'Format'
  - Set 'Length' to 10000
  - Set 'Start Address' to address of 'g_USBPerfInfo'
  - 'Finish'
- Build '..\tools\hex2text.c' and execute it with command line argument being the <filename>.hex. E.g: hex2text write.hex
- A file namely '<filename>.hex.txt' will be created.
- Rename '<filename>.hex.txt' to '<filename>.hex.csv', open with Windows Excel to view the data used to profile and the summary at the end.

Warning: Timer 7 is currently used for capturing performance figures. One should not use the 'delay' function in between the code being profiled as it too uses Timer 7.

Performance Figures Table

Examples	**EVM-AM335x**	
	Read (DMA Mode - Raw read)	**Write (DMA Mode - Raw write)**
usb_host_msc	16.6 MB/Sec	17.0 MB/Sec
usb_dev_msc	14.2 MB/Sec	17.6 MB/Sec

Connector Information

Below table shows, which USB receptacle to use for each example in the package.

Examples	EVMs		
	AM1808	**AM335xEVM**	**Beaglebone**
usb_dev_serial	J6 - USB0 - mini AB receptacle	J14 - USB0 - micro AB receptacle	P3 - USB0 - mini B receptacle
usb_dev_bulk	J6 - USB0 - mini AB receptacle	J14 - USB0 - micro AB receptacle	P3 - USB0 - mini B receptacle
usb_dev_mouse	J6 - USB0 - mini AB receptacle	J14 - USB0 - micro AB receptacle	NA
usb_dev_msc	J6 - USB0 - mini AB receptacle	J14 - USB0 - micro AB receptacle	P3 - USB0 - mini B receptacle
usb_host_mouse	J6 - USB0 - mini AB receptacle	J18 - USB1 - USB A type receptacle	NA
usb_host_msc	J6 - USB0 - mini AB receptacle	J18 - USB1 - USB A type receptacle	NA

Limitations

1. SOF counter approach for timer can't be used in High Speed mode. Need to see other options for the functionalities where SOF counter/timer is used

References

[1] http://focus.ti.com/docs/toolsw/folders/print/sw-usbl.html#Technical%20Documents
[2] http://msdn.microsoft.com/en-us/library/aa476426.aspx
[3] http://www.luminarymicro.com/products/software_updates.html
[4] http://libusb-win32.sourceforge.net

StarterWare MMC

Introduction

Multimedia card high-speed/secure data/secure digital I/O (MMC/SD/SDIO) host controller, which provides an interface between microprocessor and either MMC, SD memory cards, or SDIO cards. The salient features of the aforementioned HS-MMC host controller are:

- Full Compliance with MMC4.3 and SD/SDIO 2.0 command/response sets as defined in the Specification.
- 1-bit, 4-bit and 8-bit MMC/SD/SDIO modes. ('Note:' 8-bit mode is only for MMC)
- Built-in 1024-byte buffer for read or write
- 32-bit-wide access bus to maximize bus throughput
- Single interrupt line for multiple interrupt source events
- Two slave DMA channels (1 for TX, 1 for RX)
- Designed for low power and Programmable clock generation
- Maximum operating frequency of 48MHz
- MMC/SD card hot insertion.

Programming

Following are the steps to follow to initialize the controller for device access ---

- Check the presence of the card using *HSMMCSDIsCardInserted().*, If card is present, follow the below steps for controller inittialization.
- Reset the controller using *HSMMCSDSoftReset().*
- Reset the controller lines using *HSMMCSDLinesReset.*
- Set the supported voltages using *HSMMCSDSupportedVoltSet().*
- Enable the AutoIdle mode using *HSMMCSDSystemConfig().*
- Set the bus width using *HSMMCSDBusWidthSet().*
- Set the bus voltage using *HSMMCSDBusVoltSet().*
- Power on the bus using *HSMMCSDBusPower().*
- Set the output bus frequency using *HSMMCSDBusFreqSet()*
- Send the INIT stream to the card using *HSMMCSDInitStreamSend().*
- Enable the command completion, command timeout, data timeout and transfer completion interrupt using *HSMMCSDIntrEnable()*

StarterWare MMCSD Driver

Features supported

- Support for SD v2.0 standard
- Support for Standard Capacity and High capacity cards
- Support for Standard Speed and High Speed cards
- DMA mode of operation
- FAT file system support based on FatFs project [1]

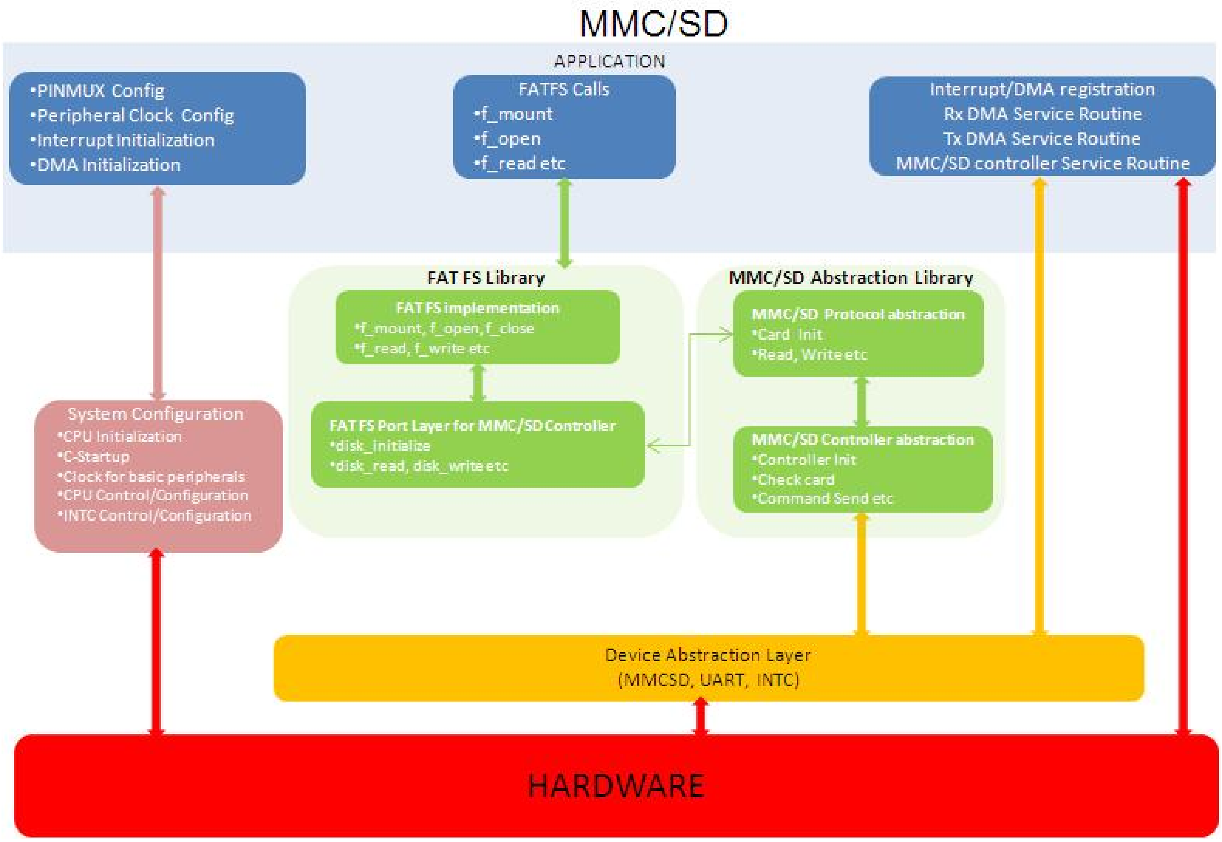
Features not supported

- No support for MMC standard
- No support for SDIO

StarterWare MMC/SD Support

MMC/SD controller, which supports SD and MMC bus protocols for data read/write form/to MMC/SD cards. StarterWare contains software support for MMC/SD controller by providing:

- MMC/SD controller Device Abstraction Layer
- MMC/SD protocol abstraction layer
- MMC/SD controller abstraction layer
- Application with filesystem enabled
- Bootloading support via SD cards



MMC/SD controller Device Abstraction Layer

The MMC/SD controller device abstration layer provides register layer access for the HS MMC/SD controller on the SoC. This layer only contains APIs for register level access.

MMC/SD Abstraction Layer

Overview

The MMC/SD Abstraction Layer contains two parts.

- **MMC/SD controller abstraction**

The MMC/SD controller abstraction layer provides a method for abstracting the controller specifics from the application and also increase reusability and quick start for the application. Thus, the application need not worry about every step to initialize the controller, need not worry about mapping the MMC/SD commands to be sent to the controller. The controller abstraction layer takes care of the controller specifics

- **MMC/SD protocol abstraction**

The MMC/SD protocol abstraction layer provides a method for abstracting the MMC/SD protocol specifics from the application and also increase reusability and quick start for the application. Thus, the application need not worry about each command to be set to the card. The MMC/SD protocol abstraction layer provides APIs that can be used to form and send commands, initialize cards and so on. The MMC/SD protocol abstraction layer then interacts with the MMC/SD controller abstraction layer which maps the MMC/SD commands to controller specifics and sends the command via the controller.

Design

The MMC/SD abstraction layer is designed two facilitate mainly

- Abstract the user/application from MMC/SD protocol, to quickstart application development
- Easy Integration with the lower level controller specifics and support multiple MMC/SD controllers
- A simple stack like approach for easy enhancements/improvements

Two files make up the abstraction layer

- mmcsd_proto.c - contains abstraction layer/APIs for the MMC/SD protocol. This layer implements the card identification and initalization sequence, commands for single/multi block read and write, card reset, and so on.
- hs_mmcsdlib.c - contains abstraction layer/APIs for the HSMMC/SD controller on the SoC. This layer implements a mapping from the MMC/SD protocol to suit the controller register layer. This involves converting command structures to controller register layer format, controller initialization sequence and so on

The MMC/SD abstraction layer contains the following main data objects.

- mmcsdCardInfo - This holds the card specific details derived from the card specific registers. These details are used to appropriately send commands to the card. For example, from the OCR register details information regading card capacity (standard/high) is derived. This helps in forming the offset address of the memory location to be read/written from/to the card. Bus widths and transfer speeds supported by the card can be used to setup the bus for transfer. This structure also, contains reference pointer to the controller object to which the card is plugged.
- mmcsdCtrlInfo - This holds the controller specific details that are populated by the user about the underlying controller. Details like the memory base address of the controller, input clock configured and the output clock desired, interrupt mask required to enable/disable default interrupts. It also contains the controller specific method hooks for various operations like, controller initialization, controller data transfer/DMA preparation, card insertion status, command transfer, data transfer, bus widths supported, voltage ranges supported and so on.
- mmcsdCmd - This holds the command details that are to be sent over the bus. This structure is always populated at the MMC/SD protocol abstraction phase and used by the controller abstraction phase for mapping it to the controller specific register layer details.

Though the abstraction layer is intended to increase reusability of code acrros platforms, owing to the principles of StarterWare, the application still have a major role to play. For example, Interrupt handling is still a part of the application. Also, neither the DAL nor the abstraction layers are concerned/impose any restriction of the mode of data transfer (DMA/Polled), type of DMA of used, interrupt enabling/disabling etc. Thus the application/user - the sole owner and cognizant of these details/methods, is required to provide these details/implemenent these methods. These are provided as part of the callback functions in the controller information.

File system support

Integration

A basic FAT file system support is provided based on the FatFs project [1]. The FatFs support library is placed in ***third_party/fatfs/***. *fatfs* support has two parts.

1. The core FAT filesystem intrinsics, that is implementation of filesystem calls and filesystem identification/initialization. (found in ***third_party/fatfs/src/ff.c***)
2. The storage media and SoC specific helper methods that are called by the core layer. (found in ***third_party/fatfs/port/***)

To integrate file system support one will have to provide the port or the media specific helper methods. These, helper methods are prototyped by the *fatfs* and needs to be defined for any new media support. MMC/SD support is integrated into *fatfs* as such. The methods currently supported are: disk_initialize - to support initialization of the media during auto-mount disk_read - to read from the media disk_write - to write to the media These methods further call the MMC/SD abstraction layer APIs for required operations.

Multiple media support

fatfs provides options for multiple media types to be integrated within the system. User perhaps requires USB and MMC media to support the filesystem based access. In such a scenario, the *disk_xxx* methods need to leverage the drive number (enumerated apriori), parameter. This helps in initializing, reading, writing and IOCTL operations etc specifically for each type of the media.

Programming Sequence

Following are the steps need to follow to use the MMCSDLIB services in the application.

1. Instantiate the *mmcsdCardInfo* and *mmcsdCtrlInfo* objects.
2. Initialize the members of *mmcsdCardInfo* and *mmcsdCtrlInfo*.
3. Check for the presence of the card using *MMCSDCardPresent().*
4. if card is preasent in the slot, then proceed for the next step, else return error.
5. Initialize the contrller using *MMCSDCtrlInit().*
6. Enable the interrupt(like command completion, command timeout, data timeout and transfer completion) using *HSMMCSDIntrEnable()*
7. Execute the shell commands like ls, chdir, cs, pwd, cat etc, which internally calls the MMCSDLIB functions *MMCSDCardInit(), MMCSDBusWidthSet(), MMCSDTranSpeedSet() (as part of disk_initialize())*, *MMCSDReadCmdSend() as part of disk_read()*, *MMCSDWriteCmdSend() as part of disk_write()*.

References

[1] http://elm-chan.org/fsw/ff/00index_e.html

StarterWare NAND Driver

Features supported

- Support for NAND ONFI standard
- DMA mode of operation

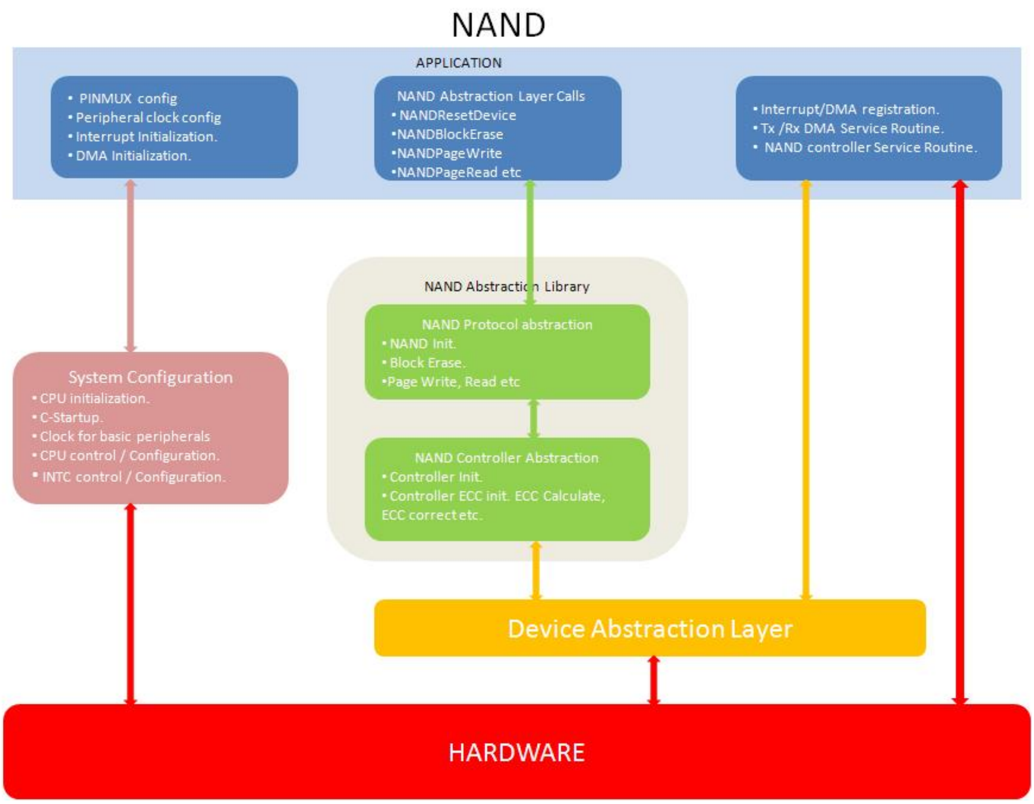
Features not supported

- No support for BCH 4 Bit 16 Bit ECC.
- No support for small page NAND.

StarterWare NAND Support

Memory controller like GPMC, EMIFA, which supports NAND protolcols for NAND data read/write. StarterWare contains software support for NAND by providing:

- Device Abstraction Layer
- NAND protocol abstraction layer
- NAND controller abstraction layer
- Application
- NAND Boot support



Device Abstraction Layer

The memory controller device abstration layer provides register layer access for the controller on the SoC. This layer only contains APIs for register level access.

NAND Abstraction Layer

Overview

The NAND Abstraction Layer contains two parts.

- **NAND controller abstraction**

The NAND controller abstraction layer provides a method for abstracting the controller specifics from the application and also increase reusability and quick start for the application. Thus, the application need not worry about every step to initialize the controller, need not worry about mapping the NAND commands to be sent to the controller. The controller abstraction layer takes care of the controller specifics

- **NAND protocol abstraction**

The NAND protocol abstraction layer provides a method for abstracting the NAND protocol specifics from the application and also increase reusability and quick start for the application. Thus, the application need not worry about each command to be sent to the NAND. The NAND protocol abstraction layer provides APIs that can be used to perform operations (like erase, write, read etc). The NAND protocol abstraction layer then maps the NAND commands to controller specifics and sends the command via the controller.

Design

The NAND abstraction layer is designed two facilitate mainly

- Abstract the user/application from NAND protocol, to quickstart application development
- Easy Integration with the lower level controller specifics and support multiple NAND controllers
- A simple stack like approach for easy enhancements/improvements
- Two files make up the abstraction layer

Two files make up the abstraction layer

- nandlib.c - contains abstraction layer/APIs for the NAND protocol. This layer implements the commands for reset, read id, block erase, page read, page write, and so on.
- nand_gpmc.c - contains abstraction layer/APIs for the GPMC controller on the SoC. This layer implements a mapping from the NAND protocol to suit the controller register layer. This involves controller initialization sequence, and so on

The NAND abstraction layer contains the following main data objects.

- nandDevInfo - This holds the NAND specific details like pagesize, blkSize, waitpin, chip select region address, chip select region size, data, command and address register etc. Some are populated by the user about the device connected like chip select on which device is interfaced, data/cmd/address register address etcand some are initialized as part read id command. These details are used to initialize the controller, to send commands etc. For example, using page size, ECC calculation iterations can be calculated as controller expects only 512 bytes for the ECC calculation. Also NAND protocal layer uses data, address and commmand register address to send command, address and data. Bus width can be used to setup the bus for transfer.
- nandCtrlInfo - This holds the controller specific details that are populated by the user about the underlying controller. Details like the memory base address of the controller, current active chip select and ecc supported by the controller.

- nandInfo - This holds the consolidated information(nand device and controller info). It also contains the controller specific method hooks for various operations like, controller initialization, controller ecc initialization, controller data transfer/DMA preparation, ECC enable/disable, ECC read/write set, ECC calculation and correction and so on.

Though the abstraction layer is intended to increase reusability of code acrros platforms, owing to the principles of StarterWare, the application still have a major role to play. Neither the DAL nor the abstraction layers are concerned/impose any restriction of the mode of data transfer (DMA/Polled), type of DMA of used etc. Thus the application/user - the sole owner and cognizant of these details/methods, is required to provide these details/implemenent these methods. These are provided as part of the callback functions in the nandInfo. Also application/user is responsible for initialization the fields in nandInfo structure.

Multiple NAND support

GPMC supports interfacing multiple NAND in the system. Current NAND driver design is such that application can communicate with any NAND device by just initializing/changing the curCS (current active chip select) in the controller info(nandCtrlInfo) data object.

Programming Sequence

Following are the steps need to follow to use the NANDLIB services in the application.

1. Instantiate the *NANDInfo* object.
2. Put the default value of chip select in *NANDDevInfo* object using *NANDDevInfoDefaultValSet().*
3. Initialize the members of *NANDDevInfo* and *NANDCtrlInfo* objects.
4. Assign/Initialize the proper controller abstraction layer functions to the hookfunctions of *NANDInfo* object.
5. Initialize the controller using *NANDCtrlInit()* which inturn calls register controller abstraction layer function.
6. Initialize the ECC using *NANDECCInit()* which inturn calls register controller abstraction layer function.
7. Reset the NAND using *NANDResetDevice().*
8. Read the device ID of the NAND using *NANDIdRead()* which initializes some members of *NANDDevInfo* like *devId, manId, busWidth, pageSize, blkSize, pagesPerBlk.*
9. For read/write, do the follwoing --
    1. Check whether block is bad or not using *NANDBadBlockCheck().*
    2. Erase the block using *NANDBlockErase().*
    3. If erase operation fails, then mark that block as BAD using *NANDMarkBlockAsBad().*
    4. Write the page data to NAND using *NANDPageWrite().*
    5. Read the page data of NAND using *NANDPageRead().*

NOTE : Mode of operation(DMA or Polled), ECC algorithm to use need to initialize (before calling *NANDCtrlInit()*) as part of object initialization.

StarterWare ADC

ADC

Introduction

The touchscreen/ADC module is an 8 channel general purpose ADC,with optional support for interleaving Touch screen onversation for 4-wire,5-wire, or 8-wire resistive panel. It also has a programable FSM sequencer that supports 16 steps.A step is a general term for describing which input values to send to the AFE,and how,when , and which channel to sample. For more information on steps please refer to touchscreen/ADC TRM.

Programing Sequence

The ADC can be programmed in the below sequence for desired operation.

- Enable module clocks for ADC is by invoking *TSCModuleClkConfig()* API.
- Multiplex AN0 - AN7 input pins invoking *TouchScreenPinMuxSetUp()*.
- Input ADC Clock is configured by invoking *TSCConfigureAFEClock()* API.
- Step Configuration register are write protected.Thus,before configuring any step register, write protection for step configuration register must be disabled by invoking *TSCStepConfigProtectionDisable()*
- Configuring Steps.
  - A step can be configured to select required input channel and refernce volatage by invoking *TSCTSStepConfig()*
  - A step can be configured to drive xpp,xnp and ypp pin to high,which in trun pull up the AN0-AN2 line by invoking *TSCTSStepAnalogSupplyConfig()*.
  - A step can be configured to drive the xnn,ypn,ynn and wpn pins to low,which in trun pull down AN1-AN4 line by invoking *TSCTSStepAnalogGroundConfig()* API.
  - A step can configured to store data,which is an outcome after a step is applied by fsm,in either FIFO0 or FIFO1 by invoking *TSCTSStepFIFOSelConfig()* API.
  - A step can configured in continous or oneshort mode for software enabled or HW event mapped step.
- ADC can be configured for differential or singled ended mode of operation by invoking *TSCTSStepOperationModeControl()* API.
- AFE can be configured for 4-wire or 5-wire or as general purpose inputs by invoking *TSCTSModeConfig()*
- Required steps can be enabled by invoking *TSCConfigureStepEnable()* API.
- ADC is enabled by invoking *TSCModuleStateSet()* API.

StarterWare DMTimer

DMTimer

Introduction

DMTimer is a 32 bit timer and the module contains a free running upward counter with auto reload capability on overflow. The timer counter can be read and written in real-time (while counting). The timer module includes compare logic to allow an interrupt event on a programmable counter matching value.

Programming

- DMTimer can be configured in two modes of operation. They are

1. Timer Mode
2. Compare Mode

- To enable the DMTimer to operate in Timer mode, the following sequence has to be followed.
  - Configure the timer for One-shot or Auto-reload operation by calling the API *DMTimerModeConfigure()*.
  - Load the count value for the timer by calling the API *DMTimerCounterSet()*.
  - If Auto-reload operation is enabled then use the API *DMTimerReloadSet()* to load the reload value.
  - Enable the timer interrupts by using the API *DMTimerIntEnable()*.
  - Start/Enable the timer by calling the API *DMTimerEnable()*. The timer will start counting.
  - Stop/Disable the timer by calling the API *DMTimerDisable()*. The timer will stop counting.
- To enable the DMTimer to operate in Compare mode, the following sequence has to be followed.
  - Load the compare value to the compare register by calling the API *DMTimerCompareSet*.
  - Configure the timer in one shot with compare enable mode or auto reload with compare enable mode by calling the *DMTimerModeConfigure* API.
  - Load the count value for the timer by calling the API *DMTimerCounterSet()*.
  - If Auto-reload operation is enabled then use the API *DMTimerReloadSet()* to load the reload value.
  - Enable the timer interrupts by using the API *DMTimerIntEnable()*.
  - Start/Enable the timer by calling the API *DMTimerEnable()*. The timer will start counting.
  - Stop/Disable the timer by calling the API *DMTimerDisable()*.

StarterWare EHRPWM

Introduction

The enhanced high resolution PWM (EHRPWM) is capable of generating complex pulse width waveforms with minimal CPU overhead. It is highly programmable and very flexible. Cross coupling or sharing of resources has been avoided; instead, the ePWM is built up from smaller single channel modules with separate resources and that can operate together as required to form a system.

Programming

To configure the EHRPWM, the following sequence can be used.

- EHRPWM needs to be first brought out of local reset by enabling the module inthe Power Sleep Controller by using PSCModuleControl().
- Pin multiplexing registers to enable EHRPWM pin and a standard configuration is provided as part of the function EHRPWMPinMuxSetup() in platform directory.
- Enable the AINTC to receive EHRPWM interrupts by using the AINTC API's given under /include/am1808/interrupt.h
- The EHRPWM has many sub-modules, which have to be configured,
  - Timebase
    - Scale the time-base clock (TBCLK) relative to the system clock (SYSCLKOUT) by using the API EHRPWMTimebaseClkConfig()
    - Configure the PWM time-base counter (TBCNT) frequency or period and set the mode for the time-base counter by using the API EHRPWMPWMOpFreqSet()
    - Configure the time-base phase relative to another ePWM module, synchronize the time-base counter between modules and configure the direction (up or down) of the time-base counter after a synchronization event by using the API EHRPWMTTimebaseSyncEnable()
    - Configure how the time-base counter will behave when the device is halted by an emulator by using the API EHRPWMTEmulationModeSet()
    - Specify the source for the synchronization output of the ePWM module by using the API EHRPWMTSyncOutModeSet()
  - Counter-Compare
    - Specify the PWM duty cycle for output EPWMxA and/or output EPWMxB and the time at which switching events occur by using the API's EHRPWMTLoadCMPA() & EHRPWMTLoadCMPB()
  - Action-qualifier
    - Specify the type of action taken when a time-base or counter-compare submodule event occurs by using the API EHRPWMTConfigureAQActionOnA() & EHRPWMTConfigureAQActionOnB()
  - Dead-band
    - Configure the output raising by using the API EHRPWMDBConfigureRED()
    - Configure the falling edge delay by using the API EHRPWMDBConfigureFED()
    - The source & polarity also can be configured
  - PWM-chopper
    - Configure the chopping (carrier) frequency by using the API EHRPWMConfigureChopperFreq()
    - Configure the carrier frequency by using the API EHRPWMConfigureChopperDuty()
    - The one shop pulse width can be configure by using the API EHRPWMConfigureChopperOSPW()

- HR
  - The High resolution can be configure using the API's EHRPWMTLoadTBPHSHR() and EHRPWMTLoadCMPAHR()

StarterWare ELM

Introduction

Non-managed NAND flash memories can be dense and nonvolatile in their own nature, but error-prone.When reading from NAND flash memories, some level of error-correction is required. GPMC uses ELM(error-location module) to extract the error from the syndrome polynomials which are calculated through GPMC as part of NAND read.Based on the syndrome polynomial value, the ELM can detect errors, compute the number of errors, and give the location of each error bit.

Features :

- 4, 8, and 16 bits per 512-byte block error-location based on BCH algorithms
- Eight simultaneous processing contexts
- Page-based and continuous modes
- Interrupt generation on error-location process completion.

Programming

Following sections explains the progrmming sequence to be followed to calculate the error location using ELM.

- Initialize the ELM module as part of GPMC controller initialization by following steps :
  - Reset the module using *ELMModuleReset()*.
  - Wait till ELM module is out of reset using *ELMModuleResetStatusGet()*.
  - Configure the internal OCP clock gating strategy to free running using *ELMCAutoGatingConfig()*.
  - Set the idle mode to no idle using *ELMCIdleModeSelect()*.
  - Set the OCP Clock activity when module is in IDLE mode to ON using *ELMOCPClkActivityConfig()*.
  - Clear the pending interrupts using *ELMIntStatusClear()*.
  - Enable the interrupt for syndrome polynomial 0 using *ELMIntConfig()*.
  - Sets the Error correction level for BCH alogorithm using *ELMErrCorrectionLevelSet()*.
  - Set the size of the buffers for which the error-location engine is used to 0x7FF using *ELMECCSizeSet()*.
  - Set the mode of module to PAGE MODE.
- Follow the follwing steps to calculate the number of errors and error location after syndrome polynomial read from GPMC --
  - Set the fragments of syndrome polynomial for error-location processing using *ELMSyndromeFrgmtSet()*.
  - Start the error-location processing for the polynomial set.
  - After the processing completion, interrupt is generated. Check the same using *ELMIntStatusGet()*.
  - Clear the interrupt using *ELMIntStatusClear()*.
  - Get the status of error-location processing using *ELMErrLocProcessingStatusGet()*.
  - Get the number of errors using *ELMNumOfErrsGet()*.

- Gets the Error-location bit address using *ELMErrLocBitAddrGet()*

StarterWare GPIO V2

GPIO

Introduction

Each GPIO module provides 32 dedicated general-purpose pins with input and output capabilities. These pins can be configured for the following applications:

- Data input (capture)/output (drive)
- Keyboard interface with a debounce cell
- Interrupt generation in active mode upon the detection of external events. Detected events are processed by two parallel independent interrupt-generation submodules to support biprocessor operations.

Programming

- Firstly, enable the functional clocks for the required GPIO instance.
- Perform a pin multiplexing for the required GPIO pin by calling the appropriate pin multiplexing function implemented in the respective Platform file.
- Enable the GPIO module using the API *GPIOModuleEnable()*. Doing this would confirm that the clocks to the GPIO module are not gated.
- Perform a module reset of the GPIO module using the API *GPIOModuleReset()*. This API also waits until the module reset is complete.
- A GPIO pin can be used either as an input or an output pin. This direction of the GPIO pin is configured using the API *GPIODirModeSet()*.
- When a GPIO pin is configured as an input pin, the following configurations become relevant:
  - Enable debouncing feature for the specified input GPIO pin if required, using the API *GPIODebounceFuncControl()*.
  - Program the debouncing time, if required using the API *GPIODebounceTimeConfig()*.
  - Interrupt trigger conditions need to be configured using the API *GPIOIntTypeSet()* using appropriate parameters.
  - Enable GPIO to generate interrupts on detection of the specified transitions on the decided GPIO pin using the API *GPIOPinIntEnable()*
- When a GPIO pin is configured as an output pin, the following configurations become relevant:
  - A logic HIGH or a logic LOW could be driven on the specified GPIO pin by invoking the API GPIOPinWrite() passing appropriate parameters.

Note: If GPIO peripheral interrupts will be used, then the system interrupt settings needs to be performed prior to enabling the GPIO peripheral interrupts. System interrupt settings involves configuring the ARM processor and the Interrupt Controller(INTC) to handle peripheral interrupts.

StarterWare GPMC

Introduction

GPMC(General Purpose Memory Controller)is an unified memory controller to interface external memory devices like NAND, NOR, Asynchronous SRAM etc. By configuring the bit fields in the GPMC registers, the application can be able to access the entioned type of device through GPMC.GPMC has the Error correction code(ECC) engine which can be used to calculate the ECC for writing and reading from the NAND.GPMC supports diffrent ECC algorithms like Hamming code, BCH 4, 8 and 16 Bit. To increase NAND read/write speed, GPMC has Prefetch and write posting engine which can be used read from or write to in a buffered manner. GPMC has on DMA event which can be used along with prefetch and write posting engine to increase the NAND read/write performance.

Programming

Following sections explains the progrmming sequence to be followed to access diffrent devicess from GPMC.

Programming the GPMC for NAND

GPMC For Read/Write Access

Following are the STPES to configure the GPMC for NAND access ---

- Pin multiplexing registers to enable the NAND pins and a standard configuration is provided as part of the function *NANDPinMuxSetup()* in platform directory.
- The GPMC is placed in local reset state using *GPMCModuleSoftReset()* and waited till the modele is out of reset using *GPMCModuleResetStatusGet()*.
- Enable/Disable the interrupt using *GPMCIntDisable()* and *GPMCIntEnable()*.
- Select the waitpin using *GPMCWaitPinSelect()*.
- Select the waitpin polarity using *GPMCWaitPinPolaritySelect()*.
- Select the write protect pin polarity using *GPMCWriteProtectPinLevelCtrl()*.
- Enable the limited address device support using *GPMCLimitedAddrDevSupportConfig()*.
- Disable the chip select (on which device is interface) before configuring using *GPMCCSConfig()*.
- Select the Signals timing latencies scalar factor using *GPMCTimeParaGranularitySelect()*.
- Select the attached device type to NAND flash like using *GPMCDevTypeSelect()*.
- Select the device size(i.e 8 Bit or 16 Bit) using *GPMCDevSizeSelect()*.
- Select the Address and data multiplexed protocol to non-multiplexed using *GPMCAddrDataMuxProtocolSelect()*.
- Select the write and read type to async using *GPMCWriteTypeSelect()*.
- Select the write and read access type to async using *GPMCAccessTypeSelect()*.
- Set the chip select base address using *GPMCBaseAddrSet()*.
- Set the Chip-select mask address or CS region size using *GPMCMaskAddrSet()*.
- Set the timing values for CS using *GPMC_CS_TIMING_CONFIG* macro and *GPMCCSTimingConfig()*.
- Set the timing values for ADV using *GPMC_ADV_TIMING_CONFIG* macro and *GPMCADVTimingConfig()*.
- Set the timing values for WE and OE using *GPMC_WE_OE_TIMING_CONFIG* macro and *GPMCWEAndOETimingConfig()*.

- Set the read access time and read/write cycle time using *GPMC_RDACCESS_CYCLETIME_TIMING_CONFIG* and *GPMCRdAccessAndCycleTimeTimingConfig().*
- Set the bus turn around time and cycle to cycle time using *GPMC_CYCLE2CYCLE_BUSTURNAROUND_TIMING_CONFIG* macro and *GPMCCycle2CycleAndTurnArndTimeTimingConfig().*
- Set the write access time and Data on ADMux timing using *GPMCWrAccessAndWrDataOnADMUXBusTimingConfig().*
- Enable the chip select (on which device is interface) using *GPMCCSConfig().*

With Above initialization application can access the NAND for read/write by sending the command, address and data using *GPMCSNANDCmdWrite()*, *GPMCNANDAddrWrite()*, *GPMCNANDDataWrite() and GPMCNANDDataRead().*

GPMC for NAND ECC

To use the ECC for read/write, need to initialize the GPMC for ECC. Following are the steps to initialize the GPMC for ECC ---

- Select the ECC algorithm using *GPMCECCAlgoSelect().*
- Depending on the ECC algorithm, GPMC initialization steps will vary.
- If algorithm is Hamming code, then
  - Select the columns(on which ECC has to calculate) as ECC columns using *GPMCECCColumnSelect().*
  - Select the chip select where ECC is computed using *GPMCECCCSSelect().*
  - Disable the ECC calculation using *GPMCECCDisable().*
  - Select the ECC result register using *GPMCECCResultRegSelect().*
  - Clear the ECC result register using *GPMCECCResultRegClear().*
  - Set the ECC size 0 and size 1 value to 0xFF using *GPMCECCSizeValSet().*
  - Select the ECC size for the ECC result register using *GPMCECCResultSizeSelect()*
- If algorithm is BCH, then
  - Set the ECC error correction capability using *GPMCECCBCHErrCorrectionCapSelect().*
  - Select the columns(on which ECC has to calculate) as ECC columns using *GPMCECCColumnSelect().*
  - Select the chip select where ECC is computed using *GPMCECCCSSelect().*
  - Select the number of sectors to process with the BCH algo as 512 bytes using *GPMCECCBCHNumOfSectorsSelect().*
  - Disable the ECC calculation using *GPMCECCDisable().*
  - Select the ECC result register using *GPMCECCResultRegSelect().*
  - Clear the ECC result register using *GPMCECCResultRegClear().*
  - Set the ECC size 0 and size 1 value to 0xFF using *GPMCECCSizeValSet().*
  - Select the ECC size for the ECC result register using *GPMCECCResultSizeSelect()*

GPMC for prefetch and write post engine for read/write access

- Ensure that engine is stopped using *GPMCPrefetchEngineStatusGet()*.
- Select the chip select associated with NAND using *GPMCPrefetchCSSelect()*.
- Select the access mode (read/write) using *GPMCPrefetchAccessModeSelect()*.
- Select the FIFO thershold value for DMA or interrupt using *GPMCPrefetchFifoThrldValSet()*.
- Select the transfer count value using *GPMCPrefetchTrnsCntValSet()*.
- Select the syncmode(when the engine starts the access to CS) using *GPMCPrefetchSyncModeConfig()*.
- Disable the cycle optimization for prefetcfh engine using *GPMCPrefetchAccessCycleOptConfig()*.
- Select the synchronization type(i.e DMA or interrupt) using *GPMCPrefetchSyncTypeSelect()*.
- If DMA sync type is select, Do the DMA related initializations and enble the DMA channel for transfer.
- Enable the engine using *GPMCPrefetchEngineEnable()*.
- Start the engine using *GPMCPrefetchEngineStart()*.
- After engine is started, If sync type is DMA, then DMA event is generated and data is tranffered using DMA. If sync type is interrupt, interupt is generated.
- After the transfer, stop the engine using *GPMCPrefetchEngineStop()*.
- Disable the engine using *GPMCPrefetchEngineDisable()*.

StarterWare HSI2C

HSI2C

Introduction

The I2C component is in compliance with the Philips Semiconductors Inter-IC bus (I2C-bus) specification version 2.1. The I2C module supports only Fast mode (upto 400 kbps) of operation. AM335X I2C can be configured to multiple master-transmitters and slave-receivers mode and multiple slave-transmitters and master-receivers mode. I2C also could be configured to generate DMA events to the DMA controller for transfer of data.

Programming sequence

Interrupt Mode

- Configuring the I2C in master transmitter/Receiver mode
  - The Pin multiplexing registers need to be configured for enabling the I2C_SDA and I2C_SCL pins.
  - The I2C is placed in local reset state using *I2CMasterDisable()*
  - The required operating clock is set using *I2CMasterInitExpClk()*
  - The address of the slave to be addressed is set using *I2CMasterSlaveAddrSet()*
  - The required I2C interrupts are enabled using *I2CMasterIntEnableEx()*
  - The mode of operation is set using *I2CMasterControl()*
    - In case of master transmitter mode of operation, the setting used is I2C_CFG_MST_TX. Optionally STOP mode also can be configured. Only after the required settings the module is brought out of reset.
    - In case of master Receiver mode of operation, the setting used is I2C_CFG_MST_RX. Optionally STOP mode also can be configured. Only after the required settings the module is brought out of reset.
- Before Configuring the I2C configuration and DataCount register make sure that I2C registers are ready for access by polling the Access ready bit of IRQ RAW status register.
- Finally the data transfer is started by commanding a START on the bus using *I2CMasterStart()*
- STOP condition generation

    - STOP can be configured to be automatically generated at the end of ICCNT number of bytes. In this case the I2C_CFG_STOP needs to be passed to *I2CMasterControl()* and also the ICCNT should be updated with the required number of bytes using *I2CSetDataCount()*
    - STOP can also be generated by manually. In this case I2C_CFG_STOP need not be supplied. But *I2CMasterControl()* can be used to set STOP manually.
    - Various combinations decide the STOP generation. Please refer to the I2C Peripheral User Guide for more details.
- Note: In interrupt handler Receive ready status should be cleared only after reading the received data from the I2C data register.

Similarly Transmit ready status sholud be cleared only after writing to I2C data register.

DMA Mode

- In DMA mode of operation, the data transfer happens via EDMA.
- Enable EDMA clocks.
- EDMA is initialized using *EDMA3Init()*, the DMA channels are mapped and enabled using *EDMA3RequestChannel()*.
- EDMA PaRAM set (options) for HSI2C transmit and receive are set using *EDMA3SetPaRAM()*.
- EDMA transfer is enabled using *EDMA3EnableTransfer()*.
- I2C DMA event generation for HSI2C transmit and receive is enabled using *I2CDMATxEventEnable()*.
- Configure the I2C in Master Transmitter/Receiver Mode as explained for interrupt mode (above).
- A transmit register empty/recieve byte condition generates a Tx/Rx EDMA event.
- The EDMA completion interrupt occurs after number of bytes configured in the PaRAM set are exhausted.
- The generation of I2C EDMA events is disabled using *I2CDMATxEventDisable()*
- Two interrupt handlers are registered for EDMA
- The completion interrupt handler *EDMA3ComplHandlerIsr()* to take action on the completion of transfer. Action usually is to disable the channel on completion of transfer.
- The error interrupt handler *EDMA3CCErrHandler()* to take action on the error conditions. Action usually is to disable the channel, clear error bits and terminating the transfer.

StarterWare LCDC

Raster LCD

Introduction

Raster LCD Controller is used to display image on LCD panel.Raster LCD Controller is a synchronous LCD interface. It provides timing and data for constant graphics refresh to a passive display. Graphics data is processed and stored in frame buffers. A frame buffer is a contiguous memory block in the system. A built-in DMA engine supplies the graphics data to the Raster engine which, in turn, outputs to the external LCD device.

Programing Sequence

To program the raster controller, the following sequence can be used.

- Enable clock for LCD module.
- Pin multiplexing registers to enable LCD raster pin and a standard configuration is provided as part of the function *LCDPinMuxSetup()* in platform directory
- **Enable Software Clock for DMA,LIDD submodule and for Core(which encompasses raster active and passive matrix logic) by invoking *RasterClocksEnable()* API.**
- Configure the rate at which pixel data should be output by configuring pixel clock frequency by invoking *RasterClkConfig()* API.
- Configuring the DMA for single or double frame buffer ,busrst size for DMA data transfer etc is done by invoking *RasterDMAConfig()* API.
- Configuring Panel type(TFT or STN) ,color display or monochrome, 1/2/4/8/16/**24** bit per pixel mode (**packed or unpacked(only for 24 bit)**) is done invoking *RasterModeConfig()*API.
- Configure the polarity of various timing parameters (for example frame clock , pixel clock, line clock etc.) used by raster by invoking *RasterTiming2Configure()*
- Configure the Horizontal timing parameters and pixel per line of the raster by invoking *RasterHparamConfig()* API.
- Configure the vertical timing parameters and Pixel per panel of the raster invoking *RasterVparamConfigure()* API.
- Configure the required amount of FIFO delay by invoking *RasterFIFODMADelayConfig()*
- Configure the base address register with base address of the array which contain pixels of the image to be displayed and ceiling address register with end address of the same array using API *RasterDMAFBConfig()*
- Enable End of frame 0 and 1 interrupt by invoking *RasterIntEnable()* API;
- Enable the Raster by inovking *RasterEnable()*.

Note: Text in blue refers to the configuration steps applicable to LCDC IP of the following SoCs.

- **AM335x**

StarterWare McASP

Introduction

StarterWare enables audio input and output via the Multichannel Audio Serial Port (McASP) peripheral. The McASP module is a serial port optimized for multi-channel audio applications. McASP supports time division multiplexed (TDM) streams, Inter-IC Sound (I2S) protocols, and inter component digital audio interface transmission (DIT). McASP has separate transmit and receive units that can operate synchronously or independently. McASP can be configured for external or internal clock and frame sync signals. StarterWare examples configure McASP as slave. McASP has 16 serialisers, each of which can be configured as a transmitter or a receiver. The StarterWare APIs to configure and operate McASP are listed in *include/mcasp.h*.

Programming

The McASP exchanges audio data with a codec that is connected to the McASP through data, clock, and frame sync lines. The codec is configured separately through the control bus (typically I2C or SPI). Prior to using the McASP for audio applications, the user must make the following design decisions:

- **Pin settings** - The McASP pins must be configured to operate in McASP mode. Also, if external clock/frame sync signals are to be used, respective pins must be configured as input pins. All the serializers that are transmitters must be configured as output pins, and all serializers that are receivers must be configured as input pins.
- **Sampling rate** - The clock and frame sync sources for the transmit and receive sections must be configured based on the desired sampling rate.
- **Data format** - Depends on the word size, slot size, and the protocol to be supported.
- **Data transfer mode** - Based on the application, the McASP must be configured for the mode of data transfer. Data can be handled using DMA, or interrupts/polling can be used to monitor the status bits.

The McASP can be initialized for separate transmit/receive sections. General programming guidelines for McASP are given below.

- Configure the McASP pins using *McASPPinMuxSetup()* and enable the PSC for McASP.
- Configure the audio codec which is connected to the McASP based on the clock/frame sync and data format settings via the control bus.
- Reset the McASP Transmit/Receive sections using *McASPTxReset()* or *McASPRxReset()*.
- If DMA mode of operation is intended, enable the Write FIFO using *McASPWriteFifoEnable()* for transmission and enable the Read FIFO using *McASPReadFifoEnable()* for reception.
- Set the data format using the *McASPTxFmtMaskSet()/McASPRxFmtMaskSet()* APIs and *McASPTxFmtSet()/McASPRxFmtSet()* APIs separately for transmit and receive sections, respectively. For I2S mode, *McASPTxFmtI2SSet()/McASPRxFmtI2SSet()* can be used instead.
- Configure the frame sync signal for transmit/receive sections using the *McASPTxFrameSyncCfg()/McASPRxFrameSyncCfg()* APIs. The frame sync source can be selected to be internal or external.
- Configure the bit clock for the transmit/receive sections using *McASPTxClkCfg()/McASPRxClkCfg()*. The clock source can be internal, external, or mixed.
- Select the polarity of the bit clock for transmission using *McASPTxClkPolaritySet()*. If the external receiver samples data on the rising edge, the McASP transmitter must shift the data out during the falling edge of the clock (or vice versa). The polarity of bit clock for reception must be selected using *McASPRxClkPolaritySet()*. If the external transmitter shifts the data out during the rising edge of the clock, the McASP receiver must sample the

data during the falling edge of the clock (or vice versa).
- Select the time slots during which transmission/reception must happen using the *McASPTxTimeSlotSet()/McASPRxTimeSlotSet()* APIs.
- Set the desired serializers as transmitters/receivers using *McASPSerializerTxSet()/McASPSerializerRxSet()*.
- Configure the McASP pins to be used for McASP using *McASPPinMcASPSet()*.
- Configure the pin directions of the output and input pins using *McASPPinDirOutputSet()/McASPPinDirInputSet()*.
- Start the transmitter/receiver clock by invoking the *McASPTxClkStart()/McASPRxClkStart()* APIs.
- If DMA mode of transfer is to be used, enable the DMA transfer at this step. If interrupt mode of transfer is to be used, enable the interrupts here using the *McASPTxIntEnable()/McASPRxIntEnable()* APIs.
- Activate the transmit/receive serializers by invoking *McASPTxSerActivate()/McASPRxSerActivate()*.
- If CPU polling mode/interrupt mode is used, write the TX buffer at this step using the *McASPTxBufWrite()* API.
- Enable the transmit/receive state machine and frame sync by invoking *McASPTxEnable()/McASPRxEnable()* APIs.
- If CPU polling method/interrupt method is used, before putting data into the transmit buffer and before reading data from the receive buffer, the data ready bit must be polled using the *McASPTxStatusGet()/McASPRxStatusGet()* APIs.

StarterWare Audio Application

DMA Buffer handling for Ping-Pong Operation

The StarterWare audio example application uses EDMA for audio data transmit and recieve operations. The audio data buffers associated with EDMA transmission are:

- **4 Transmit buffers** - TX buffer-0, TX buffer-1, TX buffer-2, and a loop buffer.
- **3 Receive buffers** - RX buffer-0, RX buffer-1, and RX buffer-2.

The EDMA paRAM sets are programmed to receive data in RX buffers and transmit data from TX buffers. When an RX buffer gets filled, the contents are be copied to the TX buffer, then it is transmitted. If no data is received, the EDMA paRAM sets for transmission are programmed to transmit from the loop buffer, which is a NULL buffer containing no valid audio data.

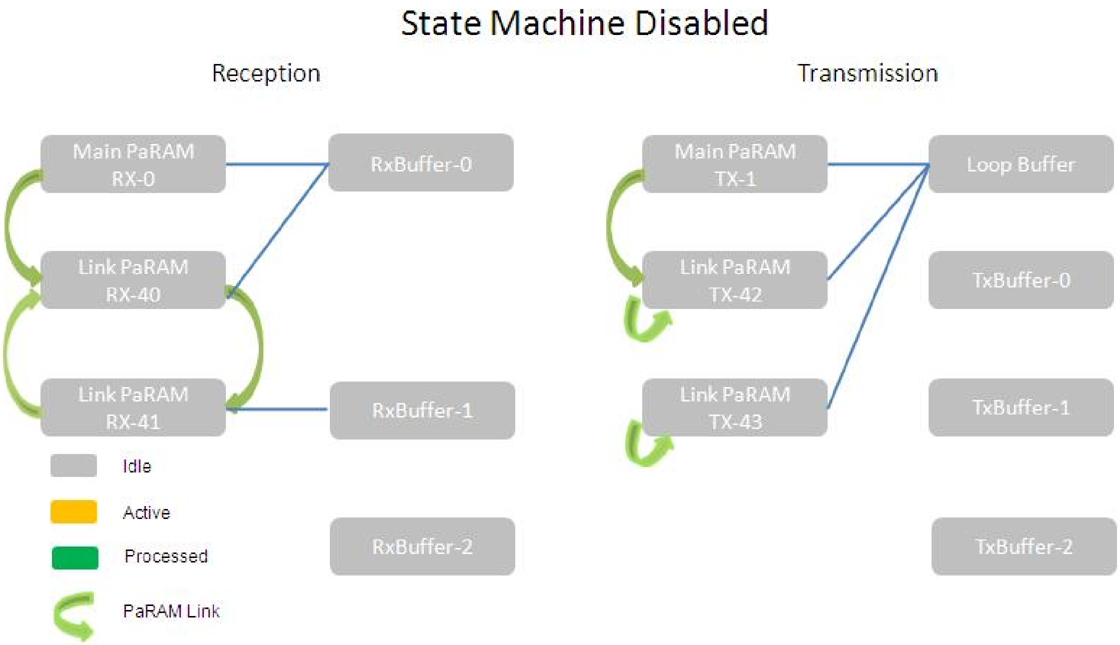
Initialization Of EDMA Parameters

Before the McASP transmit/receive state machines are brought out of reset, the EDMA paRAM sets are initialized. The main paRAM set for RX is paRAM set 0, and the main paRAM set for TX is paRAM set 1. After the main paRAM set expires, data transmission/reception continues on to linked paRAM sets. The linked paRAM sets do not expire since the EDMA copies the linked paRAM set to the main paRAM set and use it for data transfer. Hence, there is no need to update all the fields in a linked paRAM set after the associated transfer completion.

The RX paRAM set 0 is initialized to receive the first audio sample in the RX buffer-0. The transfer completion interrupt is not enabled for paRAM set 0. paRAM set 0 is linked to linked paRAM set 40. The paRAM set 40 resumes receiving data in RX Buffer 0. The paRAM set 40 is linked to paRAM set 41, which is initialized to receive data in RX Buffer 1. The paRAM set 41 is linked back to paRAM set 40. Hence the reception paRAM set is initialized as 0-->40-->41-->40. This linking does not change as the application executes.

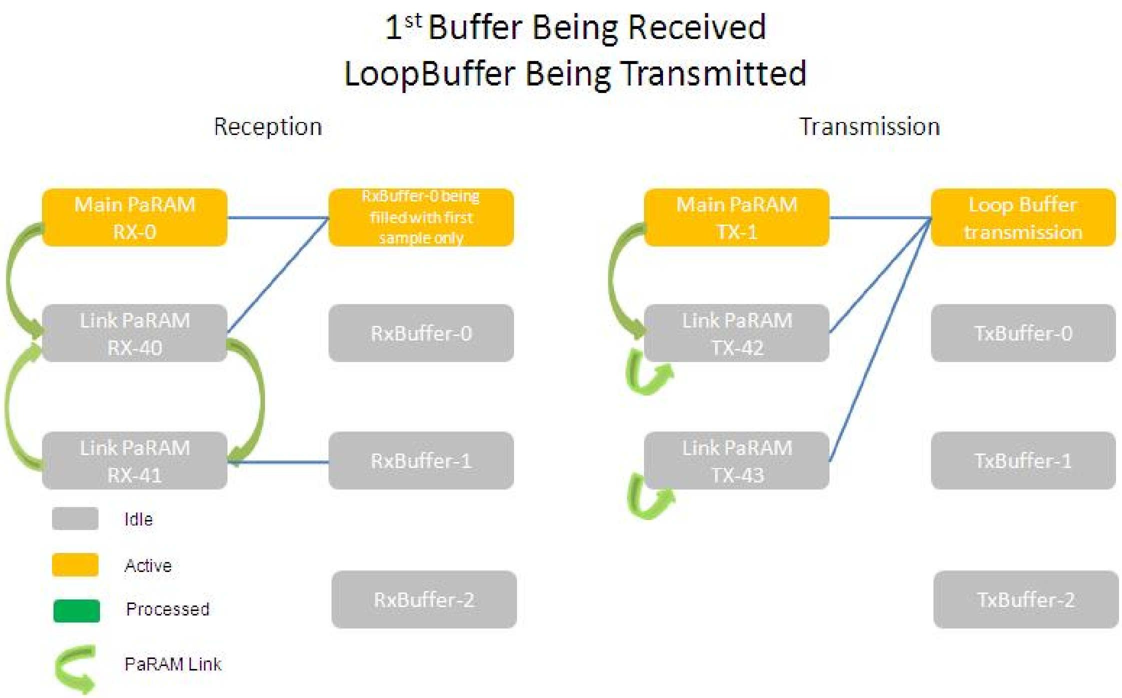
All the TX paRAM sets are initialized to transmit from the loop buffer. The transfer completion interrupt is not enabled for paRAM set 1. paRAM set 1 is linked to paRAM set 42. paRAM set 42 and 43 are linked to themselves.

Hence transmission paRAM set linking is initialized as 1-->42-->42, 43->43.

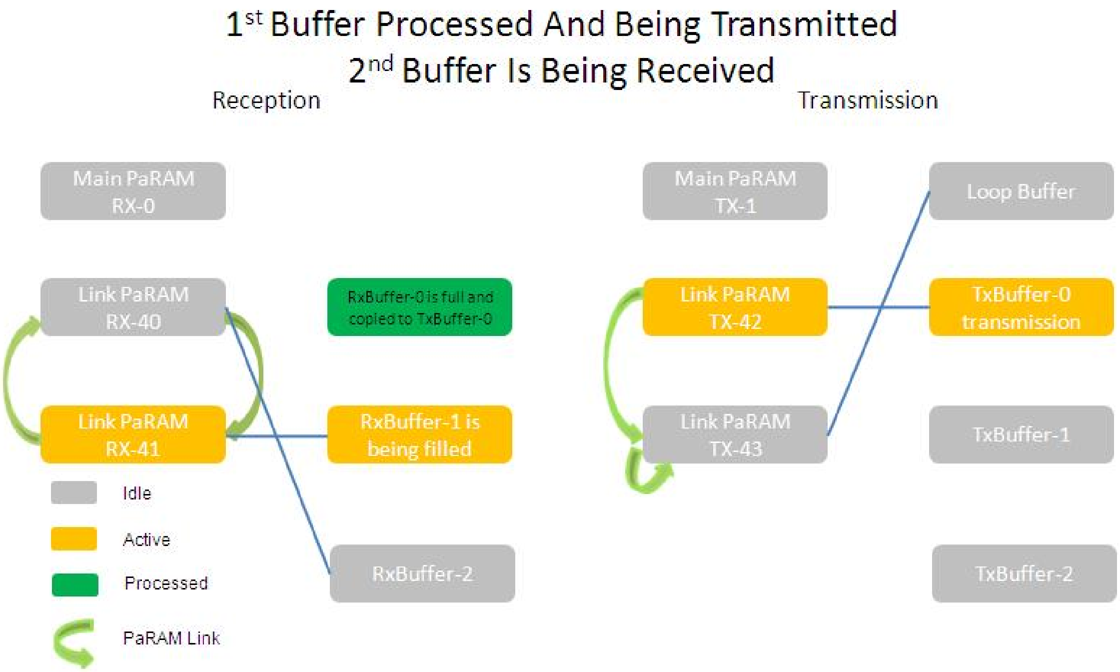


Releasing the McASP TX and RX state machines from reset

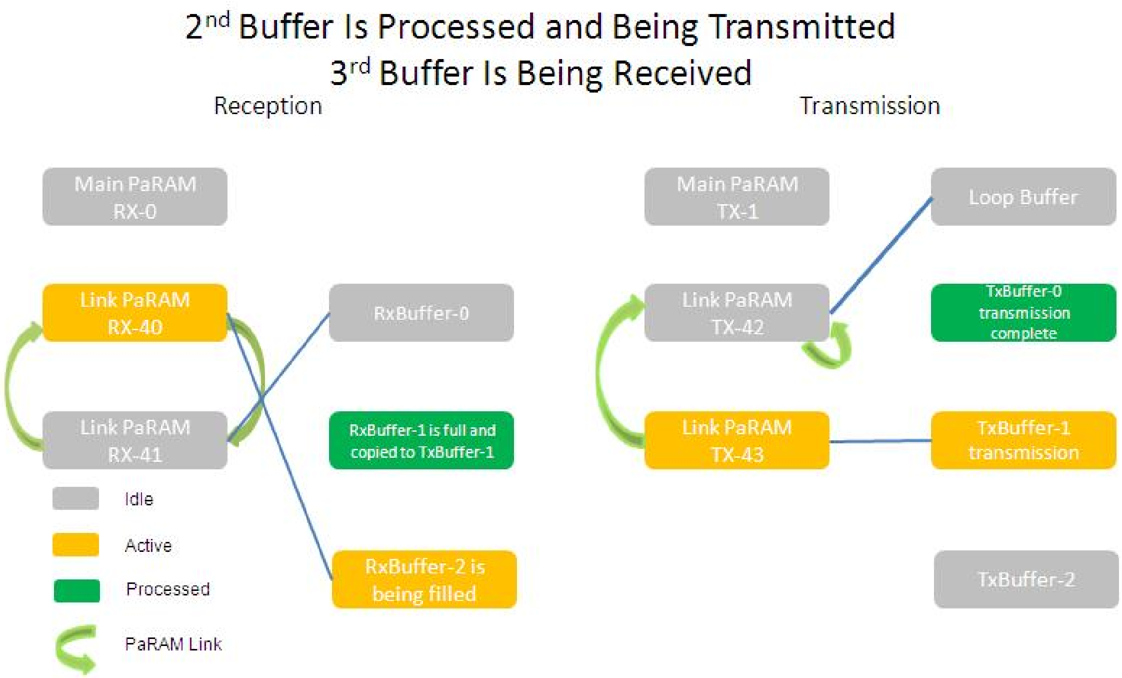
Once McASP TX and RX state machines are released from reset, the McASP triggers EDMA events for transmit and recieve operations. The first audio sample is received in RX buffer-0 via main paRAM set 0. Since it is linked to paRAM set 40, after receiving the first sample, EDMA resumes receiving data in RX buffer-0 via paRAM set 40. Similarly, the main paRAM set-1 enables transmssion from the loop buffer. When the main paRAM set expires, paRAM set 42 continuously transmits data from the loop buffer.



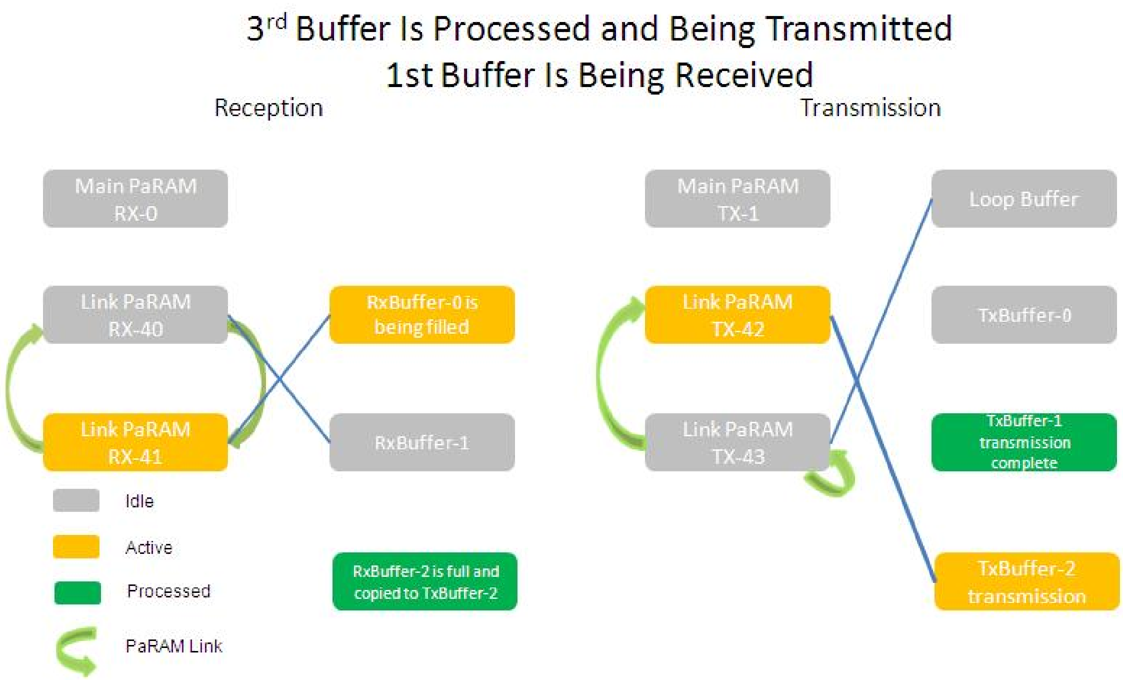
Once the EDMA reception is complete in RX Buffer-0, the application copies the RX Buffer-0 to TX Buffer-0 and updates the link paRAM set 42 to send data from TX Buffer-0. The paRAM set 42 is also linked to paRAM set 43. While the TX buffer-0 is being transmitted, the EDMA receives data in RX buffer-1 via paRAM set 41. Hence the paRAM set 42 is updated to receive data in RX buffer-2.



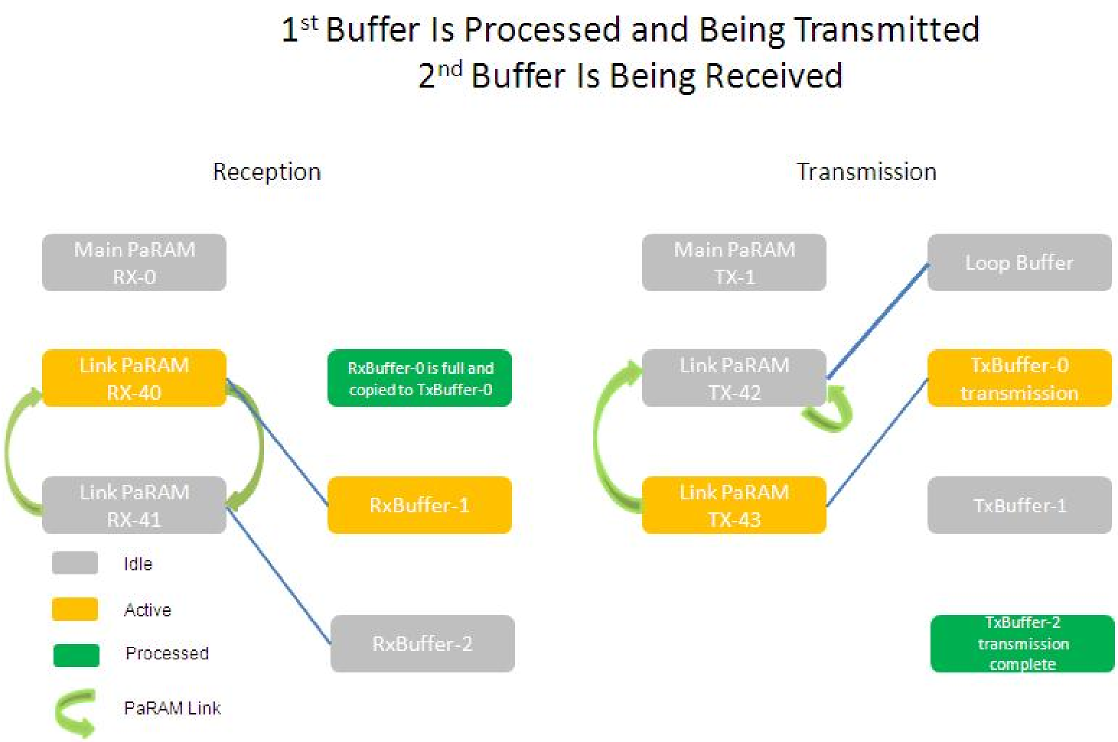
After the RX Buffer-1 is filled, it copies to TX Buffer-1 and the paRAM set 43 is updated to send from TX Buffer-1. After the EDMA transmission from TX Buffer-0 completes, the EDMA immediately starts transmitting from TX Buffer-1 since the paRAM set 42 is linked to paRAM set 43. During this time, the EDMA receives data in RX Buffer-2.



After the RX Buffer-2 is filled, it is copied to TX Buffer-2 and the paRAM set 42 is updated to send from TX Buffer-2. After the EDMA transmission from TX Buffer-1 completes, the EDMA immediately starts transmitting from TX Buffer-2 since the paRAM set 43 is linked to paRAM set 42. During this time, the EDMA receives data in RX Buffer-0.



After the RX Buffer-0 is filled, it is copied to TX Buffer-0 and the paRAM set 43 is updated to send from TX Buffer-0. After the EDMA transmission from TX Buffer-2 completes, the EDMA immediately starts transmitting from TX Buffer-0 since the paRAM set 42 is linked to paRAM set 43. During this time, the EDMA receives data in RX Buffer-1.



The EDMA data transfer resumes in the same sequence as explained in the above steps since the paRAM sets are programmed appropriately.

StarterWare McSPI

Introduction

McSPI is a general-purpose receive/transmit, master/slave controller that can interface with up to four slave external devices or one single external master. It allows a duplex, synchronous, serial communication between CPU and SPI compliant external devices (Slaves and Masters). It supports maximum frequency of 48MHz. McSPI could be configured to generate DMA event to EDMA controller for transfer of data.

Programming sequence

Interrupt Mode

Configuring McSPI in master mode with Chip Select

- Clocks for McSPI peripheral are enabled using the function *McSPI0ModuleClkConfig()* .
- Pin muxing for SPI_CLK, SPI_D0, SPI_D1 pins can be done by calling the function *McSPIPinMuxSetup()*.
- Pin muxing for chip select(CS) is done by using the function *McSPI0CSPinMuxSetup()*.
- McSPI is placed in local reset state by using *McSPIReset()* API.
- McSPI can be configured in 4-pin mode(CLK, D0, D1, CS) by using *McSPICSEnable()* API.
- McSPI is enabled in Master mode of operation using the *McSPIMasterModeEnable()* API.
- To configure Single/Multi channel mode, transmit/receive modes and settings for IS, DPE0, DPE1 can be done by using the *McSPIMasterModeConfig()* API. The settings for IS, DPE0 and DPE1 will configure the direction for SPID0 and SPID1 pins as input or output. Please refer to the schematics to verify the SPI data pin connections and do the setting accordingly. This API will return “FALSE” if an invalid configuration is done for IS,DPE0 and DPE1 pins which the McSPI controller cannot process.
- McSPI clock configuration is done using the *McSPIClkConfig()* API. Granularity settings of 1 clock cycle or 2^n clock cycles can be done in this API. Also phase and polarity of McSPI clock can be configured here.
- McSPI word length is configured using the *McSPIWordLengthSet()* API.
- Polarity of SPIEN(chip select) is configured using *McSPICSPolarityConfig()* API.
- To enable/disable the transmitter and receiver FIFOs the user can use *McSPITxFIFOConfig()* and *McSPIRxFIFOConfig()* APIs.
- The SPIEN line is forced to the required polarity level(active state) by using the *McSPICSAssert()* API.
- Enable the required interrupts by using the *McSPIIntEnable()* API.
- Enable the required McSPI channel by using the *McSPIChannelEnable()* API. Once this API is called McSPI can generate interrupts depending on the setting.
- Status on any interrupts can be got by using the *McSPIIntStatusGet()* API.
- Status of any interrupts can be cleared by using the *McSPIIntStatusClear()* API.
- Data to be transmitted from the McSPI peripheral is done using the *McSPITransmitData()* API.
- Data can be read or received by using the *McSPIReceiveData()* API.
- McSPI interrupts can be disabled by sending the necessary flags to the *McSPIIntDisable()* API.
- The SPIEN line is forced to the inactive state by using the *McSPICSDeAssert()* API.
- Disabling of McSPI channel can be done by using *McSPIChannelDisable()* API.

DMA Mode

Configuring McSPI in master mode with Chip Select

- In DMA mode of operation, the data transfer happens via EDMA.
- Clocks for McSPI peripheral are enabled using the function *McSPI0ModuleClkConfig()*.
- Pin muxing for SPI_CLK, SPI_D0, SPI_D1 pins can be done by calling the function *McSPIPinMuxSetup()*.
- Pin muxing for chip select(CS) is done by using the function *McSPI0CSPinMuxSetup()*.
- Clocks for EDMA peripheral are enabled using the *EDMAModuleClkConfig()* API.
- Initialize the EDMA by calling the *EDMA3Init()* API.
- Channels for McSPI Tx and Rx event can be requested to the EDMA3CC by calling the *EDMA3RequestChannel* API.
- McSPI is placed in local reset state by using *McSPIReset()* API.
- McSPI can be configured in 4-pin mode(CLK, D0, D1, CS) by using *McSPICSEnable()* API.
- McSPI is enabled in Master mode of operation using the *McSPIMasterModeEnable()* API.
- To configure Single/Multi channel mode, transmit/receive modes and settings for IS, DPE0, DPE1 can be done by using the *McSPIMasterModeConfig()* API. The settings for IS, DPE0 and DPE1 will configure the direction for SPID0 and SPID1 pins as input or output. Please refer to the schematics to verify the SPI data pin connections and do the setting accordingly. This API will return “FALSE” if an invalid configuration is done for IS,DPE0 and DPE1 pins which the McSPI controller cannot process.
- McSPI clock configuration is done using the *McSPIClkConfig()* API. Granularity settings of 1 clock cycle or 2^n clock cycles can be done in this API. Also phase and polarity of McSPI clock can be configured here.
- McSPI word length is configured using the *McSPIWordLengthSet()* API.
- Polarity of SPIEN(chip select) is configured using *McSPICSPolarityConfig()* API.
- To enable/disable the transmitter and receiver FIFOs the user can use *McSPITxFIFOConfig()* and *McSPIRxFIFOConfig()* APIs.
- Transfer parameters for any specific event of EDMA3 controller is set by using *EDMA3SetPaRAM()* API. Transfer parameters for both transmit as well as receive have to be set because McSPI is a transceiver device.
- Transfer from EDMA can be started by calling the *EDMA3EnableTransfer* API.
- By using the *McSPIWordCountSet()* API the user can keep a count of number of SPI words to be transmitted. Once full transfer is done the count in the McSPI Transfer level register should go to zero.
- The SPIEN line is forced to the required polarity level(active state) by using the *McSPICSAssert()* API.
- McSPI-EDMA events (transmit/receive) are enabled by calling the McSPIDMAEnable() API.
- Enable the required McSPI channel by using the *McSPIChannelEnable()* API. Once this API is called McSPI can generate EDMA events depending on the setting. A transmit empty register condition will generate a transmit EDMA event. A receive register full condition will generate a receive EDMA event.
- The EDMA completion interrupt occurs the no. of bytes configured in the Param Set are exhausted.
- Status of EDMA interrupts is got by using the *EDMA3GetIntrStatus()* API.
- When the transfer is completed the EDMA event generation by McSPI is disabled using the *McSPIDMADisable()* API.
- The SPIEN line is forced to the inactive state by using the *McSPICSDeAssert()* API.
- Disabling of McSPI channel can be done by using *McSPIChannelDisable()* API.

NOTE

It is advisable to use the dummy transfer concept while handling McSPI Transmit events. This is because if a transmit event occurs from the McSPI to the EDMA, the EDMA will start transferring the bytes to the transmit register/FIFO of McSPI and hence the EDMA param set for Tx event will get depleted. The data from the transmit register/FIFO is immediately sent to slave device. Once the McSPI register/FIFO is empty the McSPI will generate a transmit event to the EDMA and since the EDMA param set is 0 the EDMA will not be able to service this event and a missed event will be generated which will be handled by the EDMA error handler. Hence to avoid this missed event the concept of dummy transfer can be used. In this concept a dummy PaRAM-Set is linked to the PaRAM-Set of the previous transmit event. This is done by giving the address of a dummy PaRAM-Set as the link address of transmit PaRAM-Set.

StarterWare RTC

Introduction

The RTC peripheral provides a time reference to applications running on the device. The current date and time is tracked in a set of counter registers that update once per second. The time can be represented in 12- or 24-hour mode (i.e. 1:00 p.m. or 13:00). The calendar and time registers are buffered during reads and writes so that updates do not interfere with the accuracy of the time and date. Alarms are available to interrupt the CPU at a particular time or at periodic time intervals, such as once per minute or once per day. In addition, the RTC can generate CPU interrupts whenever the calendar or time registers are updated, or at programmable periodic intervals. The StarterWare APIs to configure and operate the RTC peripheral are listed in `include/rtc.h`.

Programming

The following guidelines and general procedure should be observed when programming the RTC peripheral.

- Configure the overall system clocks and the functional clocks for RTC. Apply appropriate Pin Multiplexing if required.
- Out of reset, RTC registers are write-protected. To disable this write-protection and to program the RTC registers, specific key values have to be programmed to the KICK registers (KICK0 and KICK1). Use the `RTCWriteProtectDisable()` API to do this.
- **Call the *RTC32KClkSourceSelect()* API passing appropriate parameters to select either internal clock source or external clock source for providing the 32KHz clock input to the RTC.**
- **Call the *RTC32KClkClockControl()* API passing appropriate arguments to enable RTC to receive clock inputs from the previously selected clock source.**
- Call the `RTCEnable()` API to enable the RTC. This ensures that the 32KHz clock input to the RTC is not gated.
- Use `RTCTimeSet()` to set the specified time in the relevant RTC registers. The time is passed as a parameter to this function. This function sets the second, minute, hour and the meridiem (AM or PM) values in the relevant registers. There are also APIs to set these values individually.
- Call the `RTCRun()` API immediately after calling `RTCTimeSet()` to start clock operation (i.e. "ticks").
- Use `RTCCalendarSet()` to set the specified calendar information in the relevant RTC registers. The calendar information is passed as a parameter to this function. This function sets the day (of the week and of the month), month, and year values in the relevant registers. There are also APIs to set these values individually.
- Call `RTCTimeGet()` and `RTCCalendarGet()` to read the current time and calendar information respectively from the relevant registers.
- The user can also enable interrupts to be generated when specific events occur. Use `RTCIntTimerEnable()` API to enable periodic generation of interrupts. The period between two periodic interrupts could be a second, a minute, an hour or a day.
- RTC peripheral can also generate interrupts when it reaches a certain time and calendar reading. These are called alarm interrupts, and they are enabled using the `RTCIntAlarmEnable()` API. The alarm time is written to the alarm registers of the RTC.

Note: Text in blue refers to configuration steps applicable to RTC IPs in the following SoCs.

- **AM335x.**

StarterWare Touchscreen

Touchscreen

Introduction

The touchscreen module is an 8 channel general purpose ADC,with optional support for interleaving Touchscreen conversation for 4-wire,5-wire, or 8-wire resistive panel. It also has a programmable FSM sequencer that supports 16 steps.A step is a general term for describing which input values to send to the AFE, and how, when , and which channel to sample.For more information on steps please refer to touchscreen TRM.

Programing Sequence

- Module clock for Touchscreen controller is enabled by invoking *TSCADCModuleClkConfig()* API.
- AN0 - AN7 input pins are multiplexed by invoking *TSCADCPinMuxSetUp()* API.
- Input ADC Clock is configured by invoking *TSCADCConfigureAFEClock()* API.
- Step Configuration register are write protected.Thus,before configuring any step register, write protection for step configuration register must be disabled by invoking *TSCStepConfigProtectionDisable()*;
- Configuring IdleStep.
  - A IdleStep can be configured to select required input channel and reference voltage by invoking *TSCADCIdleStepConfig()*;
  - A IdleStep can be configured to drive xpp,xnp and ypp pin to high,which in turn pull up the AN0-AN2 line by invoking *TSCADCIdleStepAnalogSupplyConfig()* API
  - A IdleStep can be configured to drive the xnn,ypn,ynn and wpn pin to low,which in turn pull down AN1-AN4 line by invoking *TSCADCIdleStepAnalogGroundConfig()* API.
  - ADC can be configured for differential or singled ended mode of operation by invoking *TSCADCIdleStepOperationModeControl()* API.
- Configuring ChargeStep.
  - A ChargeStep can be configured to select required input channel and reference voltage by invoking *TSCADCChargeStepConfig()*;
  - A ChargeStep can be configured to drive xpp,xnp and ypp pin to high,which in turn pull up the AN0-AN2 line by invoking *TSCADCChargeStepAnalogSupplyConfig()* API
  - A ChargeStep can be configured to drive the xnn,ypn,ynn and wpn pin to low,which in turn pull down AN1-AN4 line by invoking *TSCADCChargeStepAnalogGroundConfig()* API.
  - ADC can be configured for differential or singled ended mode of operation by invoking *TSCADCChargeStepOperationModeControl()* API.
  - Charge delay is configured by invoking *TSCADCTSChargeStepOpenDelayConfig()* API.
- Configuring Steps to measure x and y.
  - A step can be configured to select required input channel and reference voltage by invoking *TSCADCTSStepConfig()*;
  - A step can be configured to drive xpp,xnp and ypp pin to high,which in turn pull up the AN0-AN2 line by invoking *TSCADCTSStepAnalogSupplyConfig()* API

    - A step can be configured to drive the xnn,ypn,ynn and wpn pin to low,which in turn pull down AN1-AN4 line by invoking *TSCADCTSStepAnalogGroundConfig()* API.
    - A step can configured to store data,which is an outcome after a step is applied by FSM,in either FIFO 0 or 1 by invoking *TSCADCTSStepFIFOSelConfig()* API.
    - A step can be configured in continuous or one-shot mode for software enabled or HW event mapped step by invoking *TSCADCTSStepConfig()* API.
    - ADC can be configured for differential or singled ended mode of operation by invoking *TSCADCTSStepOperationModeControl()* API.
    - Open and Sample delay for step configured by invoking *TSCADCTSStepOpenDelayConfig()* and *TSCADCTSStepSampleDelayConfig()* .
- AFE can be configured for 4-wire or 5-wire or as general purpose inputs by invoking *TSCTSModeConfig()*;
- Touchscreen transistors are enabled by invoking *TSCADCTransistorConfig()* API.
- Map hardware event to pen touch IRQ by invoking *TSCADCHWEventMapSet()* API.
- Required steps can be enabled by invoking *TSCConfigureStepEnable()* API.
- TSC is enabled by invoking *TSCModuleStateSet()* API.

StarterWare UART/IrDA/CIR

UART

Introduction

A universal asynchronous receiver/transmitter, abbreviated UART, is a type of "asynchronous receiver/transmitter", a piece of computer hardware that translates data between parallel and serial forms. UARTs are commonly used in conjunction with communication standards such as EIA RS-232, RS-422 or RS-485. The UART takes bytes of data and transmits the individual bits in a sequential fashion. At the destination, a second UART re-assembles the bits into complete bytes. Each UART contains a shift register which is the fundamental method of conversion between serial and parallel forms. UART generally has FIFO buffers that assist in transmission and reception of data. Multiple bytes can be written to the transmit FIFO in a single turn by the processor and the UART transmits these bytes one bit at a time. Similarly, the UART can interrupt the processor once a sizeable number of bytes are received by the UART and stored in the receiver FIFO. Presence of FIFOs improve the software performance of the application. Usually the Transmit FIFOs and Receiver FIFOs have configurable threshold levels on attaining which they interrupt the processor.
The UART/IrDA/CIR IP also supports Infrared Data Association (IrDA) and Consumer Infrared (CIR) operational modes along with the general UART mode. IrDA defines physical specifications communications protocol standards for the short-range exchange of data over infrared light, for uses such as personal area networks (PANs). CIR, refers to devices employing the infrared electromagnetic spectrum for wireless communication. One of the popular application of CIR is in Television Remote Controls for communication with the Television set. IrDA and CIR transceivers shall be present in the UART/IrDA/CIR controller and shall use the same FIFO as that of UART mode to communicate with the external transmitter/receiver.

Programming sequence

Interrupt Mode

- Firstly, configure the system clocks for UART instance using the function provided in the platform directory.
- Perform Pin Multiplexing for the UART instance.
- Invoke the API *UARTModuleReset()* to perform a module reset of the UART instance.
- If the UART is to be configured in FIFO mode, use the API *UARTFIFOConfig()* to perform FIFO configurations.
- The specified baud rate of communication is achieved by appropriately programming the Divisor Latch registers. Specifically, the divisor value is a function of the Operating frequency and the desired baud rate. The computation formula for the divisor latch value also differs based on the operating mode specified. Use the API *UARTDivisorValCompute()* to compute the divisor value that is to be programmed to the Divisor Latch registers.
- Invoke the API *UARTDivisorLatchWrite()* to program the computed divisor value to the divisor latch registers.
- Switch to Register Configuration Mode B using the API *UARTRegConfigModeEnable()* passing appropriate parameters.
- Configure the Line Characteristics using the API *UARTLineCharacConfig()* passing appropriate parameters.
- Disable access to the divisor latch registers using the API *UARTDivisorLatchDisable()*.
- Ensure that the Break condition is disabled using the API *UARTBreakCtl()* passing appropriate parameters.
- Call the API *UARTOperatingModeSelect()* with appropriate parameters to switch the UART to 16x operating mode.
- Configure the ARM interrupt controller to generate UART interrupt by registering the UART ISR.
- Enable required UART interrupts using the API *UARTIntEnable()* passing appropriate parameters.

DMA Mode

- Configure the functional clocks of EDMA using the function *EDMAModuleClkConfig()*. Similarly, configure the functional clocks of UART0 instance using the function *UART0ModuleClkConfig()*.
- Perform Pin Multiplexing for the required UART instance using the function *UARTPinMuxSetup()* passing appropriate instance number.
- Initialize the EDMA3 instance using the API *EDMA3Init()* passing the appropriate event queue number to be used, as an argument.
- Perform the configurations to handle interrupts:
  - Enable IRQ bit in CPSR register of ARM processor using the API *IntMasterIRQEnable()*. This enables ARM to receive interrupt requests over IRQ line.
  - Initialize the Interrupt Controller (INTC) using the API *IntAINTCInit()*.
  - The ISRs (Interrupt Service Routine) of EDMA3 Completion interrupt and EDMA3 Error interrupt have to be registered in the INTC. This is done using the API *IntRegister()* passing appropriate arguments.
  - Set the priority for the above interrupts using the API *IntPrioritySet()* passing appropriate parameters.
  - Enable the INTC to receive the above interrupts using the API *IntSystemEnable()* passing appropriate arguments.
- Now the UART instance has to be initialized to appropriate settings for proper operation and communication. The initialization sequence used remains the same as that followed for operating UART in interrupt mode. As an addition, the API *UARTDMAEnable()* needs to be invoked to enable appropriate DMA mode of operation for the UART.
- EDMA3 communicates with other peripherals through logical channels. A logical channel each is required for UART TX Event and UART RX Event for these events to get serviced when they occur.
- Individually set the PaRAM set entries for UART Transmit and Receive DMA channels using the API *EDMA3SetPaRAM()* passing appropriate parameters.

- Start EDMA transfer on the required channels using the API *EDMA3EnableTransfer()* passing appropriate parameters.
- As mentioned above, two ISRs are used – EDMA3 Completion ISR and EDMA3 Error ISR. EDMA3 generates a completion interrupt when the count values (A, B and C) of the PaRAM set are depleted to zero. EDMA3 generates an error interrupt when EDMA could not service an event it received.
- These ISRs have code which usually clears certain bits in relevant registers of EDMA3 register set.
- Further, a callback function is usually written which is invoked from ISR. This function does operations specific to the channels. For example, disabling transfer over the specified EDMA channel.

Note: In the UART EDMA application present at examples/evmAM335x/uart_edma/src/, a Dummy transfer concept is used. The UART Transmit PaRAM set is linked to a Dummy PaRAM set. A Dummy PaRAM set is defined as one where at least one of the count fields (A, B or C) is and at least one of them is non-zero. The scenario of its use is explained below. UART generates a transmit event to the EDMA whenever its Transmit Holding Register(THR)/TX FIFO becomes empty. The EDMA then transfers the configured number of bytes to the THR/TX FIFO. On the last transaction, EDMA again transfers the requisite number of bytes and its count fields are depleted to zero. UART transfers these bytes and again generates an event to EDMA. If a Dummy PaRAM set is linked to the TX PaRAM set, the EDMA services the Dummy PaRAM set before raising a completion interrupt to the ARM processor. In the absence of this Dummy PaRAM set, EDMA registers a missed event and raises an error interrupt to the ARM.

StarterWare Watchdog Timer

Introduction

The watchdog timer is an upward counter capable of generating a pulse on the reset pin and an interrupt to the device system modules following an overflow condition.

Programming

- Enable the module clocks before accessing the module. This can be done by calling the API *WatchdogTimer1ModuleClkConfig()*.
- Watchdog timer software reset is done by calling the API *WatchdogTimerReset()*.
- Prescaler clock of the watchdog timer can be enabled and configured using the *WatchdogTimerPreScalerClkEnable()* API.
- The count value for the Watchdog timer can be set by calling the *WatchdogTimerCounterSet()* API.
- The reload value for the counter register of watchdog timer can be set by calling the API *WatchdogTimerReloadSet*.
- To Start/Enable the watchdog timer *WatchdogTimerEnable()* API can be used.
- To reload value from the load register into the counter register the *WatchdogTimerTriggerSet()* API has to be called. This API takes a parameter by name ‘trigVal’. If this API has to be called many times in an application then everytime this API is called ‘trigVal’ has to be different.

StarterWare DCAN

Introduction

The Controller Area network module is a high integrity serial communications protocol for distributed real time applications. The DCAN module present on AM335x is compliant to the CAN 2.0B protocol specification and supports bit-rates upto 1Mbits/s. The core IP for the DCAN controller is provided by Bosch.

Programming sequence

Interrupt Mode

Configuring DCAN for board-board communication

- Platform Related Configurations
  - Configure the system clocks for DCAN peripheral by using the function provided in the platform directory
  - Perform pin mux for DCAN peripherals Tx and Rx pins
  - DCAN message RAM has to be initialized before the peripheral can be enabled for communication. This can be done by calling the function *DCANMsgRAMInit*
- Initialization and Configuration
  - To place the DCAN in software reset *DCANReset* API can be used
  - DCAN module can be placed in initialization mode by calling the API *DCANInitModeSet*
  - Write access to the configuration registers can be enabled or disabled by calling the *DCANConfigRegWriteAccessControl* API
  - Configuration of CAN bit timing can be done by calling the *DCANBitTimingConfig* API
  - To enable the status change or error interrupts the *DCANIntEnable* API can be used
  - To enable the interrupt lines of DCAN peripheral the *DCANIntLineEnable* API can be used
  - To enable acceptance filtering for a receive message object *DCANUseAcceptanceMaskControl* and *DCANMsgObjectMskConfig* APIs can be used.
  - To invalidate a message object the *DCANMsgObjInvalidate* API can be used
  - To validate a message object the API *DCANMsgObjValidate* can be used
  - To configure the DCAN command register *DCANCommandRegSet* API can be used
  - To set the message ID of the CAN frame to be used for communication *DCANMsgIdSet* API can be used. Configure a receive message object with ID = 0, if CAN frames of any valid IDs(Standard/Extended) should be accepted.
  - To configure the direction of the message object *DCANMsgDirectionSet* API can be used
  - Data length code(DLC) can be configured using the *DCANDataLengthCodeSet* API
  - Data bytes to be transmitted as a part of a CAN frame can be written to the interface data registers by using the API *DCANDataWrite*
  - Message object interrupts can be enabled using the *DCANMsgObjIntEnable* API
  - Configuration for FIFO end of block control can be done by calling the *DCANFIFOEndOfBlockControl* API
  - To get the transmit request status the *DCANTxRqstStatGet* API can be used
  - To get the message valid status of a message object the *DCANMsgValidStatusGet* API can be used
  - To enable/disable the test modes of DCAN the *DCANTestModeControl* API can be used
  - To configure DCAN in various test modes the *DCANTestModesEnable* API can be used
  - Status of the interrupt register can be obtained by using the *DCANIntRegStatusGet* API
  - Status of DCAN error and status register can be obtained by using the *DCANErrAndStatusRegInfoGet* API

- To get CAN bus from bus off to bus on state once the CAN bus errors are within the specified limit the *DCANAutoBusOnControl* API can be used
- To clear the interrupt pending status of a message object the *DCANCommandRegSet* API with appropriate parameter can be used
- To check the message number and the word number which caused the parity error in that message *DCANParityErrCdRegStatusGet* API can be used
- To disable transmit or receive interrupts for a message object *DCANMsgObjIntDisable* API can be used
- To read message object data which is transferred from message RAM to the interface register the *DCANDataRead* API can be used

- Communication
  - To start communication on the CAN bus the *DCANNormalModeSet* API can be used

AM335x StarterWare Power management

Power Management

The power-management framework is built with three levels of resource management: clock, power, and voltage management.These management levels are enforced by defining the managed entities or building blocks of the power-management architecture, called the clock, power, and voltage domains. A domain is a group of modules or subsections of the device that share a common entity (for example, common clock source, common voltage source, or common power switch). The group forming the domain is managed by a policy manager. For example, a clock for a clock domain is managed by a dedicated clock manager within the power, reset, and clock management (PRCM) module. The clock manager considers the joint clocking constraints of all the modules belonging to that clock domain. For more details please refer AM335x TRM [1].

Power Management is handled by softwares running on A8 and CM3. This page explains about the software running on A8. CM3 software and prebuild binary can be downloaded from here [2]. The communication between A8 and CM3 is through IPC registers. The communication protocol is explained below.

Features supported

Features	EVM	EVM-SK	Beaglebone
Deep Sleep 0	Supported	Supported	Supported
Deep Sleep 1	Supported	Supported	Supported
RTC only *	Supported **	Not Supported	Not Supported
Standby	Supported	Supported	Supported
DVFS(MPU)	Supported	Supported	Not Supported
Smart Reflex	Not Supported	Not Supported	Not Supported
Dynamic power switching	Realized via deep sleep modes	Realized via deep sleep modes	Realized via deep sleep modes
Static Leakage Management	Realized via deep sleep modes	Realized via deep sleep modes	Realized via deep sleep modes

(*) - Supported from SoC version 2.0

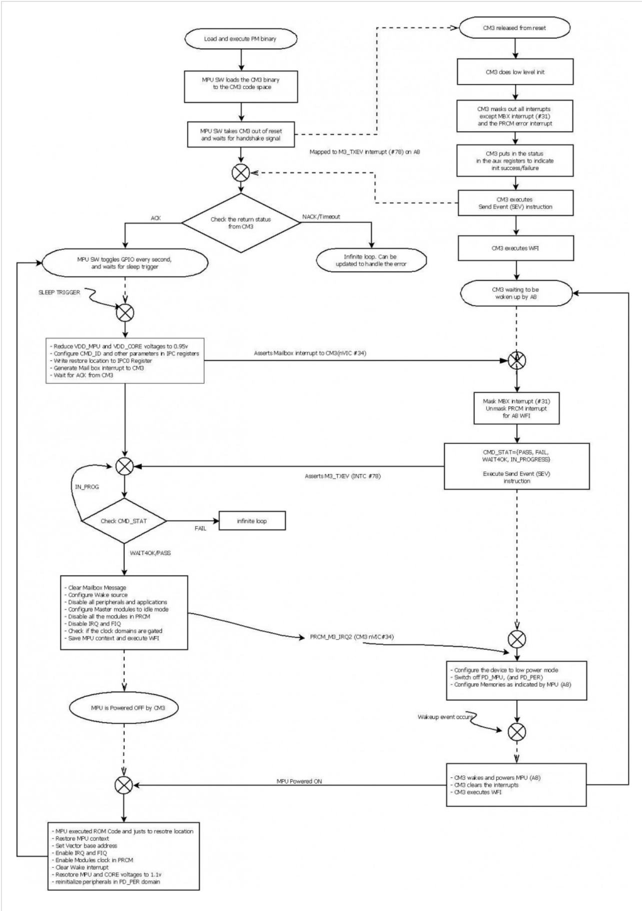
(**) - Supported from EVM Rev. 1.5x

Sleep modes

1. DS0 - Refer TRM section 8.1.4.3.4
2. DS1 - Refer TRM section 8.1.4.3.3
3. Standby - Refer TRM section 8.1.4.3.2
4. RTC Only - Refer TRM section 8.1.4.3.5

Programming Sequence

The sequence of steps to be followed to enter low power mode are listed below. The following state diagram depicts the same.



Initialization sequence

The following steps are to be followed to enable the device to enter low power mode.

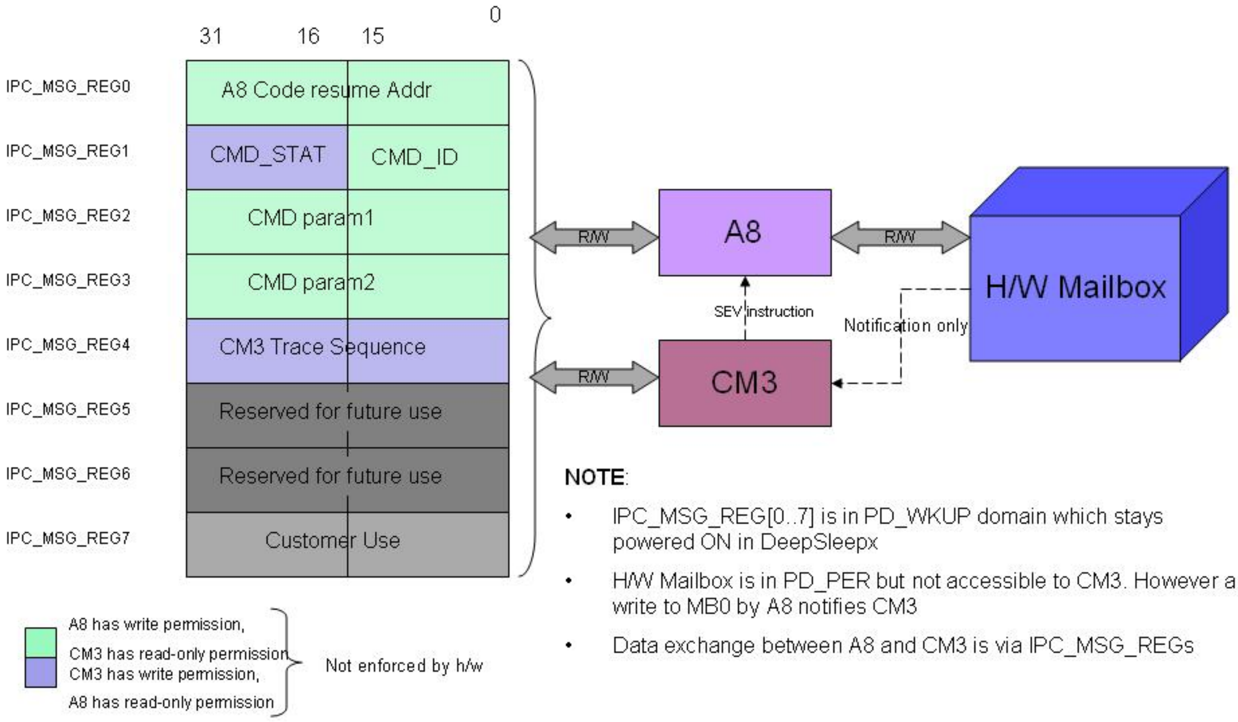
1. Initialize the interrupt controller and configure it to receive TXEV event from CM3
2. Load CM3 image (which is embedded (as header file) as part of A8 binary), to CM3 address space (0x44D00000u).
3. Release CM3 from reset and wait for ack (TXEV)
4. Initailize all the peripehrals used in the application (including mailbox)
5. Enter steady state execution and wait for command to enter deep sleep state.

Entering Deep sleep

The sequence listed below is applicable for Deep Sleep0 (a subset of these are applicable for Deep Sleep1). Once the command for entering Deep sleep mode is received the following steps are to be followed.

Configure the Deep sleep command

Configure the DS command ID and the relevant parameters in IPC registers. The format of the same is given below.



Command IDs

Following are the command IDs supported,

CMD_ID	Value	Description
CMD_DS0	0x3	1. Initiates force_sleep on interconnect clocks 2. Turns off the MPU and PER power domains 3. Configures the system for disabling MOSC when CM3 executes WFI
CMD_DS1	0x5	1. Initiates force_sleep on interconnect clocks 2. Turns off the MPU power domains 3. Configures the system for disabling MOSC when CM3 executes WFI
CMD_STANDBY	0xb	1. MOSC remains ON 2. Turn off the MPU power domain
CMD_RESET_STATE_MACHINE	0xe	1. Reset CM3 state machine

Command status

The possible status values are listed below.

CMD_STAT	Value	Description
PASS	0x0	1. In init phase this denotes that CM3 was initialized successfully. 2. When other commands are to be executed, this indicates completion of command.
FAIL	0x1	. In init phase this denotes that CM3 could not initialize properly. 2. When other commands are to be executed, this indicates some error in carrying out the task.
WAIT4OK	0x2	CM3 INTC will catch the next WFI of A8 and continue with the pre-defined sequence
IN_PROGRESS	0x3	Early indication of command being carried out

Command Parameters

The following structure shows the different parameters to be configured by A8. Only a subset of parameters are valid for a given command.

```
typedef struct
{
    /* Address to where the control should jump on wake up on A8 */
    unsigned int resumeAddr:32;

    /* MOSC to be kept on (1) or off (0) */
    unsigned int moscState :1;

    /* Count of how many OSC clocks needs to be seen before exiting deep sleep mode. Default = 0x6A75 */
    unsigned int deepSleepCount :16;

    /* If vdd_mpu is to be lowered, vdd_mpu in 0.01mV steps */
    unsigned int vddMpuVal :15;

    /* Powerstate of PD_MPU */
    unsigned int pdMpuState :2;

    /* State of Sabertooth RAM memory when power domain is in retention */
    unsigned int pdMpuRamRetState :1;

    /* State of L1 memory when power domain is in retention */
    unsigned int pdMpul1RetState :1;

    /* State of L2 memory when power domain is in retention */
    unsigned int pdMpul2RetState :1;

    /* State of Sabertooth RAM memory when power domain is ON */
    unsigned int pdMpuRamOnState :2;

    /* Powerstate of PD_PER */
    unsigned int pdPerState :2;

    /* State of ICSS memory when power domain is in retention */
    unsigned int pdPerIcssMemRetState :1;
```

```
    /* State of other memories when power domain is in retention */
    unsigned int pdPerMemRetState :1;

    /* State of OCMC memory when power domain is in retention */
    unsigned int pdPerOcmcRetState :1;

    /* State of ICSS memory when power domain is ON */
    unsigned int pdPerIcssMemOnState :2;

    /* State of other memories when power domain is ON */
    unsigned int pdPerMemOnState :2;

    /* State of OCMC memory when power domain is ON */
    unsigned int pdPerOcmcOnState :2;

    /* Wake sources */
    /* USB, I2C0, RTC_ALARM, TIMER1, UART0, GPIO0_WAKE0, GPIO0_WAKE1, WDT1, ADC_TSC */
    unsigned int wakeSources :13;

    unsigned int reserved :1;

    /* Command id to uniquely identify the intented deep sleep state */
    unsigned int cmdID:16;

    /* Delay for RTC alarm timeout. Default = 2secs */
    unsigned char rtcTimeoutVal :4;

}pmAttributes;
```

Save Peripheral and IO Pad Context

On power down of peripheral domain the context of the pripheral modules is lost. To retain the context on resume from sleep mode, the context of the necessary modules under peripheral domain has to be saved in DDR. On resume the saved context has to restored from DDR.

```
Note: Save Peripheral and IO Pad Context of all active peripherals in system for Deep Sleep0 only.
      For other sleep modes it is recommended to Save Peripheral and IO Pad Context of the peripheral reconfigured for wakeup.
```

Halt Peripherals

Disable or Halt any active transactions and configure system interfaces or modules to idle state.

Wake CM3

Currently CM3 will be in sleep state executing WFI instruction. An interrupt will cause it to wake from wfi. Here the mailbox interrupt is generated to wake CM3 from WFI. A8 waits for sync (txev) from CM3.

Clear Mailbox

After getting the sync from CM3, the MPU (A8) clears the mailbox, by reading the message and clearing the new message status. Since in SA, CM3 is not capable of clearing the mailbox, A8 clears it.

Configure wakeup sources

Configure the designated wakeup peripheral. Configuration of wakeup source depends on the sleep mode.

For sleep modes other than standby only sources under wakeup domain can be configured as wake source. Configure the peripheral in smart-idle-wakeup mode and disable the wakeup peripheral (Timer is an exception, timer module should not be disabled it timer is expected to wakeup the device). Possible wakeup sources are,

- GPIO0 bank
- dmtimer1_1ms (timer based wakeup)
- USB2PHY (USB resume signaling from suspend) – Both USB ports supported.
- TSC (touch screen controller, ADC monitor functions)
- UART0 (Infra-red support)
- RTC (RTC Alaram)
- I2C0

Any source can be configured for resume from standby. Any peripheral interrupt can cause resume from standby. Configure the wake source for interrupt and donot disable the peripheral.

```
Note: Not all sources are demonstrated in StarterWare.
```

Disable modules

EMIF shall not be disabled at this stage. Only modules not configured for wakeup shall be disabled for standby mode. For other sleep modes all modules have to be disabled.

Reduce peripherals frequency

Since the VDD_MPU voltage will be reduced (to OPP 50 value) before entering Deep sleep, the operating frequency of MPU and peripherals have to be reduced to OPP 50 vlaues.

Reduce VDD voltages

The VDD_MPU voltage is reduced to 0.95v to reduce leakage during deep sleep state.

Disable Interrupts

Diasble IRQ interrupts. This will ensure that DDR is not accessed when trying to execute the ISR.

Save MPU context

Save MPU context in OCMC ram, which will be retained during deep sleep mode.

Save EMIF context

Ensure that before disabling the EMIF module its context is saved.

```
Note: Save EMIF Context is required for Deep Sleep0 and Standby Mode.
```

DDR Self-Refresh

Put DDR into self-refresh mode to retain the DDR contents in sleep mode.

Disable EMIF Clock

For further optimisation of power disable EMIF module. Ensure that EMIF context is saved before disabling module.

```
Note: EMIF has to be disabled for Deep Sleep0 and Standby Mode.
```

Configure DDR for low power

1. Configure DDR I/O for weak pull down.
2. Configure DDR to Dynamic Power Down.
3. Disable VTP.
4. Enable SRAM LDO Ret Mode.

Bypass PLLs

Configure PLLs to Bypass mode.

WFI

When A8 (A8 module configured to disable) executes wfi, CM3 gets an interrupt and starts deep sleep entering process. CM3 disables all the power domain, configures the memory retention and Master OSC (disable or not) based on the parameters passed by A8 along with the deep sleep command.

```
Note: When A8 is connected to debugger and executes wfi, CM3 will not receive interupt.
       Only when debugger is not connected CM3 will receive the interrupt.
```

Wakeup from Deep Sleep

The following steps are to be followed before executing the steady state functionalities.

Re-Lock PLLs

On wakeup relock the PLLs.

Revert low power configurations for DDR

1. Enable SRAM LDO Ret Mode.
2. Enable VTP.
3. Revert DDR Dynamic Power Down.
4. Disable weak pull down for DDR I/O.

Enable EMIF Clock

Enable EMIF Modules before restoring the EMIF context.

```
Note: EMIF has to be enabled for Deep Sleep0 and Standby Mode.
```

Restore EMIF Context

Restore the EMIF context saved before entring sleep.

```
Note: Restore EMIF Context is required for Deep Sleep0 and Standby Mode.
```

Exit DDR from self-refresh mode

Configure DDR to exit from self-refersh.

Restore MPU context

Restore MPU context from OCMC ram, which are retained during deep sleep mode.

Configure Vector table

Configure the vector table base address in CP15 register.

Enable Interrupts

Enable IRQ interrupts. The system is now capable of handling interrupts.

Enable all the mdoules

Enable all the modules (except EMIF) and configure PLLs to the required configuration.

Disable Wakeup

Disable the wakeup source, so that the normal interrupt is not registered with CM3 as wakeup interrupt. If the wake source is not disabled, this might cause the device to wake up immediately after entering the Deep sleep (in the next cycle).

Restore VDD voltages

The VDD_MPU is voltage is restored to the value before sleep.

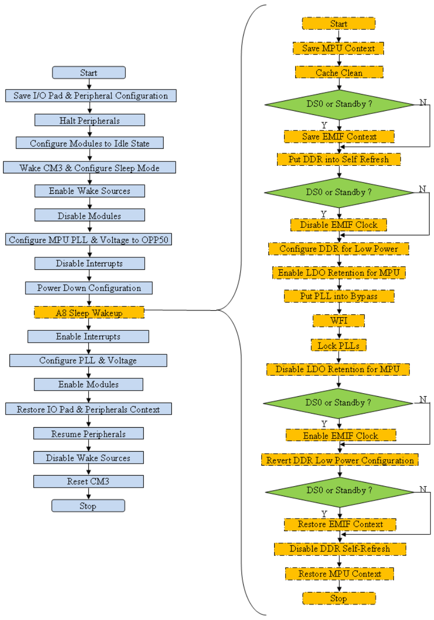
Reinitalize the peripherals

The peripherals in PD_PER power domain need to be reinitialized, since PD_PER is switched off during deep sleep.

```
Note: Restore peripheral context and I/O pad configuration of active peripherals in system for Deep Sleep0.
      For other sleep modes it is recommended to Restore Peripheral and IO Pad Context of the peripheral reconfigured for wakeup.
```

Now the system is ready for next cycle of sleep/wake sequence.

Code flow for suspend/resume or sleep/wakeup sequence - on (A8) MPU side




```
    Note: A8 comunicates with CM3 for Wake CM3 & Configure Sleep Mode, WFI and Reset CM3.
                Wake CM3 & Configure Sleep Mode - to configure Sleep Mode
                WFI - to enter sleep mode
                Reset CM3 - to clear the Sleep Mode configuration
```

RTC Only mode

Refer TRM section 8.1.4.3.5 for RTC Only mode.
To execute RTC Only mode

- Configure RTC module clock to external CLK_32K_RTC clock.
- RTC alarm2 is configured to put the system in RTC Only mode.
- RTC alarm and/or external wake pin are configured to wakeup system from RTC Only.
- Refer TRM section 20.3.5.34 to enable RTC Only wake.

Known Issue

- In RTC Only mode distorted image is observed on LCD with EVM 1.5x.
- On EVM 1.5x, there is stray voltage observed on VDDSHV1/3/5/6 supplies due to a board hardware issue.

Executing Example Applications

In StarterWare package the power management examples are integrated with OOB. The "PM" slide in OOB demonstrate enter/exit of different sleep modes. Steps to execute the sample application and the expected behaviour are given below.

1. Load the OOB application on to target.
2. If the image(.out) is loaded through debugger, the debugger has to be disconnected after executing (run) the image. Otherwise the device will not be able to enter sleep state. Alternatively the user can load the binary file [3] form MMC/SD or SPI or NAND.
3. Navigate to the "PM" slide and trigger the sleep mode (in the examples supplied the trigger sources supported is touch screen)
4. Now the device will be in sleep state consuming low power. The power consumption across different rails (on EVM) can be calculated by measuring the drop across sense resisters in different rails. Refer power consumption for more details.

5. To wake the device from sleep state, trigger the wake source. Listed below are supported wake sources for EVM, EVM-SK & Beaglebone

Wake Source	EVM	EVM-SK	Beaglebone	Remarks
Touch Screen	Supported	Supported	Not Supported	
Timer1	Supported	Supported	Supported	Not supported for Standby
Uart0	Supported	Supported	Supported	For standby external peripheral to be configured for GPIO wake
GPIO0	Supported (SW9)	Supported (SW3)	Not Supported	
RTC Alarm	Supported	Not Supported	Not Supported	Supported from SoC Version 2.0 and Standby
Timer6	Supported	Supported	Supported	Wakeup with modules in peripheral domain supported for Standby only
Ext Wake Pin	Supported (SW7)	Not Supported	Not Supported	Supported for RTC Only mode

Note

- Wake source selection is not applicable for RTC only

Now the device will be in normal state. This cycle can be repeated.

Converting CM3 binary to header file

The following steps are to be followed to convert CM3 binary to header file.

1. The source for binToc is located at StarterWare\\tools\binToC.
2. Compile the above to get the converter executable
3. conversion command: a.exe <CM3_binary.bin> <CM3_image.h>
4. Use the generated header in your application

Power Consumption

This section indicates the power measured for all power rails when system is in DeepSleep0, DeepSleep1 and Standby. The measurements are done using EVM 1.5x with OOB application from latest StarterWare Software version.

Power Rail	Deep Sleep0 (milliwatts)	Deep Sleep1 (milliwatts)	Standby (milliwatts)
VDD_CORE*	1.96	6.10	16.31
VDD_MPU	0.22	0.22	0.22
VDDS_RTC	0.04	0.04	0.04
VDDS_DDR	0.05	0.05	0.05
VDDS	0.86**	0.35	0.35
VDDS_SRAM_CORE_BG	0.17	0.17	1.94
VDDS_SRAM_MPU_BB	0.01	0.01	0.01
VDDS_PLL_DDR	0.00	0.00	0.00
VDDS_PLL_CORE_LCD	0.00	0.00	0.00
VDDS_PLL_MPU	0.00	0.00	0.00
VDDS_OSC	0.00	0.00	1.23
VDDA_1P8V_USB0_1	0.00	0.00	0.00
VDDS_A3P3V_USB0_1	0.07	0.07	0.07

VDDA_ADC	0.00	0.00	0.00
VDDSHV1	0.07	0.47	0.47
VDDSHV2	0.07	0.07	0.08
VDDSHV3	0.08	0.11	0.10
VDDSHV4	0.04	0.09	0.10
VDDSHV5	0.06	0.18	0.18
VDDSHV6	0.38	0.69	0.65
Total Power (Sum of all Rails)	**4.08**	**8.63**	**21.93**

Note

- With the following modifications power can be further reduced
  - (*) - Configuring VDD_CORE to 0.95V. This configuration has to be done in CM3. It is not implemented in current release.
  - (**) - Configuring CONTROL_CONF_ECAP0_IN_PWM0_OUT iopad as GPIO with pull up/down disabled. With this modification distortion is observed on LCD in sleep state.

References

[1] http://www.ti.com/lit/pdf/spruh73
[2] http://arago-project.org/git/projects/?p=am33x-cm3.git;a=snapshot;h=750362868d914702086187096ec2c67b68eac101;sf=tgz
[3] http://processors.wiki.ti.com/index.php/AM335X_StarterWare_Booting_And_Flashing

StarterWare Graphics

StarterWare Graphics Library

The graphics library from StellatisWare is integrated with StarterWare. The off-screen LCD support is added for 16 bpp and 24 bpp LCD. The graphics lib supports drawing various shapes like vertical, horizontal & slanting lines, filled and empty circle and rectangle, push button, check box, radio button, list box, slider, drawing image and it also supports drawing strings in various fonts. Currently it supports 153 different fonts.

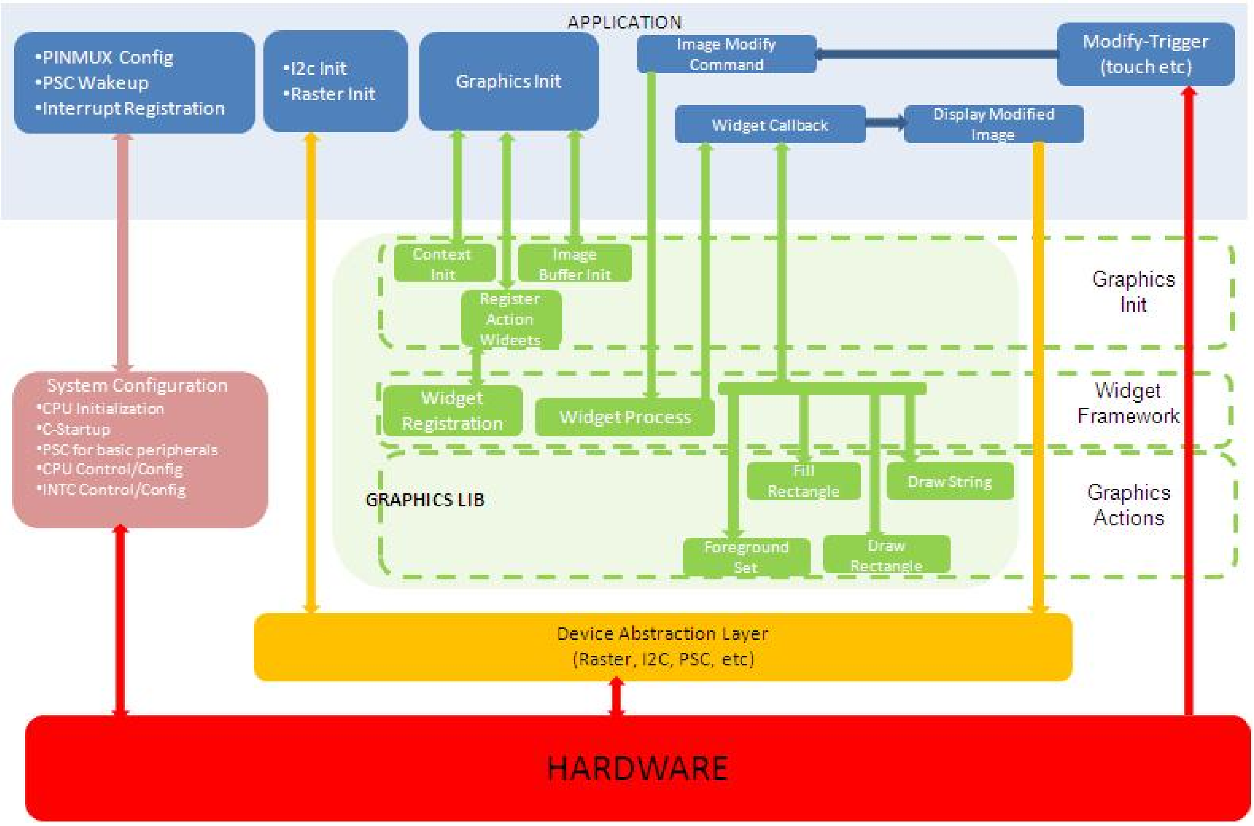
It also supports widget tree data structure, where all the widgets like push button, radio button, check box, list box, slider can be added. If touch screen is integrated, the coordinates can be sent as a message to the tree. Based on the coordinates the message will be sent to the corresponding button. The button can be configured to call the user specific handler function when the button is touched. For the graphics library API reference, refer Stellaris Graphics Library User's Guide from here [1]. While referring the document, the following points are to be noted from SitaraWare context,

- The discussion on tool chain support has to be ignored at this point in time.
- In addition to the Off-screen buffer display drivers provided in 1 bit-per-pixel (BPP) format, 4 BPP format and 8 BPP format, StarterWare supports 16BPP and 24BPP format also.

Graphics Library based Example Applications

The graphics library example application, demonstrate how graphics library APIs can be used to easily draw/paint/write on the off screen LCD panel. It provides examples which show the ease of use in displaying dynamic content on the LCD panel.

- The modules used in the example applications are
  - Graphics Library - for dynamic content/image modifications
  - Raster DAL - for LCD display
  - I2C - for touch detection



Programming

- Since the examples use the raster DAL for display on the panel, the programming, in addition to LCD initialization, consists of
  - Initializing off-screen image buffer by calling the API GrOffScreenXXBPPInit() (XX- indicated corresponding BPP)
  - Initializing the drawing context by calling the API GrContextInit()

Demonstration Application

This application provides a demonstration of the capabilities of the 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.

1. The first panel provides some introductory text and basic instructions for operation of the application.
2. The second panel shows the available drawing primitives: lines, circles, rectangles, strings, and images.
3. 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.
4. The fourth panel shows the check box widget, which provides a means of toggling the state of an item. Four 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.
5. 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.
6. The sixth panel shows the push button widget. Two columns of push buttons are provided. Each push button has a red LED to its left, which is toggled via an application callback each time the push button is pressed.
7. 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.
8. 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

On loading the graphics example binary on to the target, the banner page is displayed and on touching anywhere on the LCD the first slide is shown. Other slides can be navigated with the previous and next button at the bottom of the LCD.

Maze Game Application

The Maze game is ported from StellarisWare(Quickstart Application of EK-LM3S2965). A game in which a blob-like character tries to find its way out of a maze. The character starts in the middle of the maze and must find the exit, which will always be located at one of the four corners of the maze. Once the exit to the maze is located, the character is placed into the middle of a new maze and must find the exit to that maze; this repeats endlessly. The Graphics library is used to display the all the elements(maze, player, stars, score) in the game.

- **Executing The Example Application**

The game is started by pressing the select button on the right side of the LCD. During game play, the select button will fire a bullet in the direction the character is currently facing, and the navigation push buttons on the left side of the LCD will cause the character to walk in the corresponding direction.

Populating the maze are a hundred spinning stars that mindlessly attack the character. Contact with one of these stars results in the game ending, but the stars go away when shot.

Score is accumulated for shooting the stars and for finding the exit to the maze. The game lasts for only one character, and the score is displayed on the virtual UART at 115,200, 8-N-1 during game play and will be displayed on the screen at the end of the game.

The application also contains a screen saver. The screen saver will only become active if two minutes have passed without the user action while waiting to start the game (that is, it will never come on during game play). Qix-style bouncing lines are drawn on the display by the screen saver.

After two minutes of running the screen saver, the display will be turned off and the user LED will blink. Either mode of screen saver (bouncing lines or blank display) will be exited by touching the touch screen on LCD. The select button will then need to be pressed again to start the game.

Porting StellarisWare graphics application to StarterWare

Please refer here to know more on the various aspects to be considered while porting StellarisWare graphics application to StarterWare.

pnmtoc Tool

This section explains how to convert an image to C structure definition that can be passed to the GrImageDraw() function. This involves two steps,

- Convert image (jpg, bmp, jpeg etc...) to .pnm format using Imagemagick tool, which can be downloaded from here [2]

convert.exe file.bmp -colors 65536 file.pnm

This command converts a bmp image to pnm file with maximum of 65536 colors.

- pnmtoc (convert pnm file to C array). The tool is run from the command line, and its usage is as follows:

Windows host: pnmtoc.exe [-c] [-f] file.pnm > file.c

Linux host: ./pnmtoc [-c] [-f] file.pnm > file.c

Where the arguments mean:

-c Specifies that the image should be compressed. If compression is bypassed it would result in a larger C array.

-f specifies whether 24bpp should be forced

file.ppm Specifies the input image file.

This pnmtoc tool is located in */tools/pnmtoc/ (pnmtoc.exe - to be used in windows command prompt, pnmtoc - to be used in linux)*

Note: The icons used in OOB demo are created with command "*pnmtoc -f file.pnm > file.c*"

The resulting C image array definition is written to standard output. The output should be redirected into a file so that it can then be used by the application.This will result in an array called g_pucImage that contains the image data from file.ppm. If file.ppm contains only two colors, the 1 BPP image format is used; if it contains 16 or less colors, the 4 BPP image format is used; if it contains 256 or less colors, the 8 BPP image format used; if it contains more than 256 colors the 16 BPP image format used; and if it contains more than 65536 colors an error is generated.

References

[1] http://focus.ti.com/docs/toolsw/folders/print/sw-grl.html#Technical%20Documents
[2] http://www.imagemagick.org/script/install-source.php?ImageMagick=lif9tvsr9obctgorhl98eahhd6

StarterWare ConsoleUtilities

Background

Previous versions of StarterWare were supporting only UART based console utilities, where the I/O operations were redirected to the serial console through UART interface. This imposes a limitation where, the I/O operations can not be performed without the UART interface. To overcome this limitation, generic console utilities have been developed. The generic console utilities are wrapper functions which are compliant with the standard I/O utility functions. These utilities will allow user to select either debugger console or UART console. Configuring the console type will redirect all the console input/output to the selected console type.

Console Utils APIs

The below table provides the list of Console Utils APIs present in StarterWare and corresponding functionally equivalent standard I/O APIs.

S.No	Console Utils API	Standard I/O API
1	ConsoleUtilsPrintf	printf
2	ConsoleUtilsScanf	scanf
3	ConsoleUtilsGets	gets - with buffer overrun check
4	ConsoleUtilsPuts	puts
5	ConsoleUtilsGetChar	getchar
6	ConsoleUtilsPutChar	putchar

Note: UARTGetc API functionality is not compliant with Standard I/O functionality of 'getchar'. To read a character, 'getchar' requires an ENTER key press along with the character, where as UARTGetc does not require ENTER key press.

Debug Console Support

In the StarterWare applications the default console is configured as uart. The support for the debugger console is provided through a mechanism called "Semihosting". More information about Semihosting and procedure to enable Semihosting can be at Semihosting.

Code Modifications

Following mandatory steps should be performed before using any of the Console Utils APIs.

1. Initialization of Console. Following API should be called to init the Console.

```
    ConsoleUtilsInit();
```

2. Selecting the type of Console. Console selection can be achieved through the following run time macros

```
CONSOLE_UART     : For selecting UART console
CONSOLE_DEBUGGER : For selecting the debug console
```

Following API should be called to select the console type

```
ConsoleUtilsSetType(CONSOLE_UART);
```

```
            or

   ConsoleUtilsSetType(CONSOLE_DEBUGGER);
```

Note:

1. In the StarterWare applications the default console is configured as uart.

2. Semihosting should be enabled first to support redirection of I/O operations to debugger console. Without enabling the semihosting, if the runtime macro is configured as "CONSOLE_DEBUGGER", then an error message will be displayed on the UART console. The procedure to enable semihosting has been given at StarterWare_Semihosting.

Deprecated APIs

The following list of APIs have been deprecated in the current version of StarterWare. All the references to the above APIs have been replaced with "ConsoleUtilsPrintf" and "ConsoleUtilsScanf" instead.

```
1. UARTPutHexNum
2. UARTGetHexNum
3. UARTPutNum
4. UARTGetNum
5. UARTPrintf
```

Note:

1.The deprecated API will give a warning for GCC and CCS when used anywhere in the code.

2. IAR does not give any warning for deprecated API, because Deprecated attribute/pragma is not supported by the compiler

Known issues / Limitations

1. ConsoleUtilsPrintf and ConsoleUtilsScanf do not support I/O operations of float data types (format specifier %f is not supported).
2. Redirection to debugger console is supported with CCS5.4 only because SemiHosting is enabled with CCS 5.4 and the previous versions do not support semihosting.
3. Non-blocking getc is not supported in ConsoleUtils since that does not match with any standard IO functions. If such a functionality is needed with UART, UARTgetc must be directly used.

StarterWare Semihosting

Background

Semihosting is a mechanism that enables code running on the target to communicate with the host pc that is running a debugger, providing IO facilities. For more information on Semihosting refer semihosting [1]. The below sections discuss semihosting support in StarterWare and steps to be followed to enable it. In StarterWare the semihosting has been validated with Linaro GCC and TMS470 tool chain. The standard I/O Functions which are tested and supported in StarterWare on a Debug Console are Printf, Scanf, Puts, fGets, Getchar, PutChar.

Code Update

In the StarterWare applications the default console is configured as uart. This can be changed to redirect to debugger console.

```
ConsoleUtilsSetType(CONSOLE_UART);

''Change to''

ConsoleUtilsSetType(CONSOLE_DEBUGGER);
```

For more information on Console utility API please refer Console Utils..

Please refer below sections on tool chains which explains how to compile the application to enable semihosting.

Tool chains

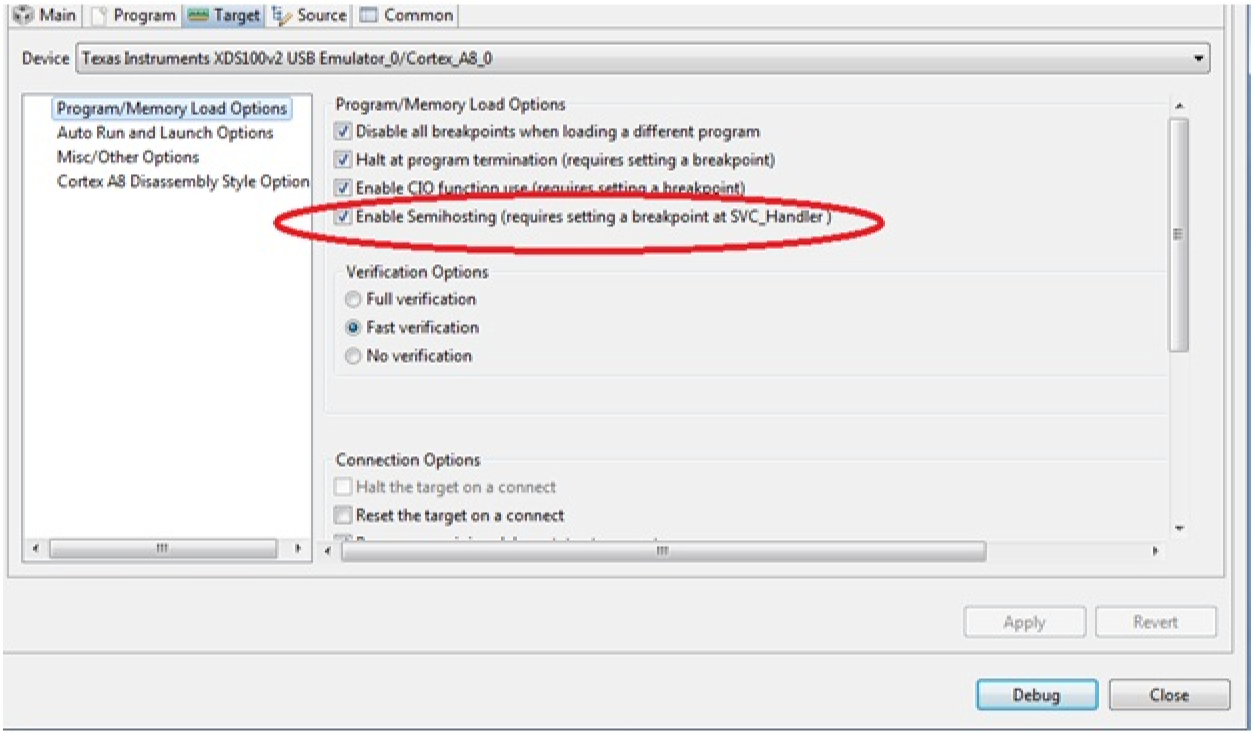
Linaro GCC

StarterWare supports linaro gcc tool chain. To enable semihosting follow the below steps.

- Compile the application using the command

```
    make CONSOLE=SEMIHOSTING
```

- The linker command file (.lds) in StarterWare supports the necessary sections needed for semihosting. So no change is needed here.
- In CCS ensure the semihosting option is selected as shown below.

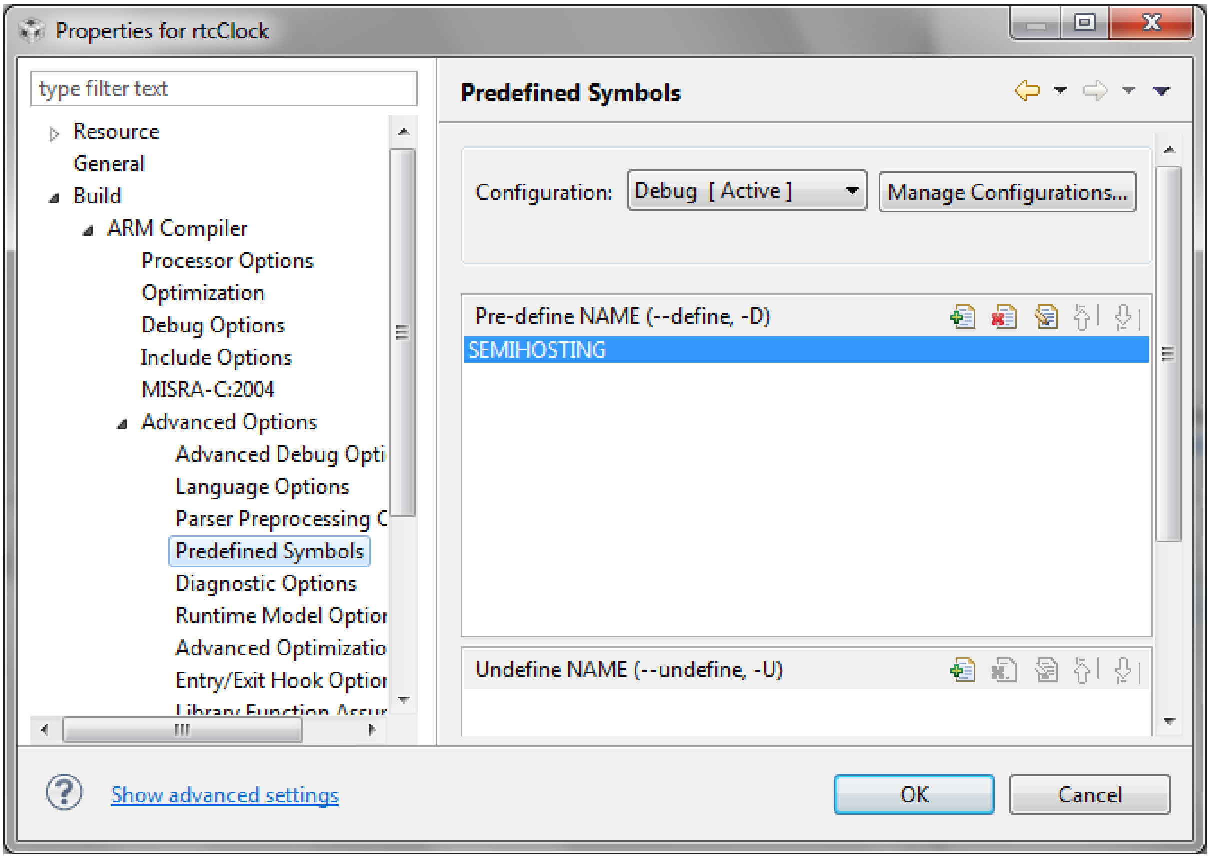


- After compilation load the elf (.out) file using CCS and IO functions can be handled via CCS console.
- A sample output from execution of RTC example on AM335x EVMSK is shown bleow.

TI ARM Tool chain

Follow the below steps to enable semihosting,

- In addition to enabling semihosting as shown in above section, while building StarterWare example the "utils" project has to be complied with predefined symbol "SEMIHOSTING".
- Other steps for loading and executing are as usual and no change is required.



Emulators Supported

The following emulators are supported in CCS 5.4 for semihosting,

- xds 560
- xds 100v2

Known issues / Limitations

1. SemiHosting is enabled with CCS 5.4 only and the previous versions do not support semihosting.
2. xds 510 emulator will not work for semihosting in CCS 5.4
3. Console works slowly while using XDS100V2 Emulator.
4. SemiHosting with IAR toolchain is not tested with StarterWare

Additional Information

- http://processors.wiki.ti.com/index.php/Semihosting

References

[1] http://processors.wiki.ti.com/index.php/Semihosting

StarterWare NeonVFP

Introduction:

AM335x ARM MPU Subsystem includes the SIMD capable NEON engine and VFP coprocessor. The VFP coprocessor implements the VFPv3 architecture and is fully compliant with IEEE 754 standard. The Neon engine with the SIMD architecture is used to accelerate media codecs, 2D/3D graphics and image Processing.

For more information and help on Neon, VFP coprocessor and SIMD concept refer processors.wiki.ti.com/index.php/Cortex-A8 [1]

The below sections discuss about Neon and VFP coprocessors support provided in StarterWare and the different configurable options provided to measure the Neon/VFP engine performance.

StarterWare Support for Neon and VFP coprocessors:

The following support is provided in StarterWare for Neon and VFP coprocessors.

- Enabling Neon/VFP engine during system initialization. The ARM cortex-A8 comes up with Neon and VFP engines disabled at power up. Support is provided to enable this feature during startup. In the system initialization code the init.s file is updated with assembly code to enable the coprocessors as per the syntax of the different toolChains supported.

The updated init file is located at

```
            \system_config\armv7a\<compiler>\init.s for different compilers
```

- IRQ handler context save/restore for Neon/VFP registers

The AM335x IRQ handler is updated to save and restore the Neon/VFP registers on the stack during an interrupt.

The exceptionhandler at following location is modified

```
          \system_config\armv7a\<compiler>\exceptionhandler.s
```

- A basic benchmarking Application *neonVFPBenchmark* is provided for measuring the performance of the functions on Neon and VFP coprocessors. It measures the performance of functions in terms of time required to execute the functions.
- The Neon and VFP coprocessor registers are saved and restored in demo example as part of sleep wakeup sequence.

Benchmarking Application

neonVFPBenchmark example

- neonVFPBenchmark example application provides support for basic benchmarking of functions by giving their execution time in micro-seconds. It provides performance numbers of functions for the following cases

1: For Neon engine performance

2: For VFP engine Performance

3: SoftFloat (without any coprocessors support where floating point operations is implemented using library helper calls by the compiler)

- The example framework permits the users with the flexibility to plug in their own function and have its performance measured, this can be done by updating the structure ***benchmarkFunction*** with details about the function to be benchmarked.
- The performance numbers of functions included in example are displayed on the selected Console and the data is displayed in the below format in addition to the ticks clocked by the Timer.

```
        <Function Name> <TimeTaken in micro seconds> <No of Iterations>
```

- The example gives the performance numbers of functions performing Float point Additions and Multiplications for all toolChains supported i.e. **GCC**, **IAR** and **CCS** with **Neon** enabled, with **VFP** enabled and **SoftFloat** case.
- Additionally for GCC compiler the application measures performance of Sine and Cosine Maths functions from third_party library for Neon engine implemented using with and without Neon Intrinsics.

Default Example settings

- The example provides performance numbers for Neon engine performance by default for all the toolChains.
- To get the Performance numbers for VFP coprocessor and SoftFloat case refer to the building the example section below.
- All the functions included in the example for performance benchmarking are executed 100000 times, the user has the flexibility to change this to get performance results for different iteration values.

Building the Example Application

- This section describes the build configuration settings which have to be changed for the neonVFPBenchmark example application to get the performance numbers needed for different cases like with Neon engine enabled, with VFP enabled and Softfloat.
- By default the example Application is compiled with Neon compiler option for all the toolchains.
- To get the performance numbers for different cases the example has to be compiled with options customized for each toolChain.
- The below section describes the build configuration settings which have to be changed for GCC, CCS and IAR toolchains.

GCC Compiler Options

This section describes the build configurations required for GCC compiler to build the example application for getting performance benchmarking numbers for different cases.

Below are the steps required to change the build configurations

The below commands are to be executed in the command line at the appropriate build location for different cases.

Ex: **~/build/armv7a/gcc/am335x/evmAM335x/neonVFPBenchmark**

a: To get performance numbers for Neon engine performance

```
$make clean+

$make FPU=NEON
```

b: To get performance numbers for VFP engine performance

```
$make clean+

$make FPU=VFP
```

c: To get performance numbers without any coprocessor support

```
$make clean+

$make FPU=SOFT
```

CCS Compiler Options

This section explains how to enable different build configurations required to be enabled for neonVFPBenchmark application to get the performance numbers for different cases.

Steps to measure the performance numbers for Neon

- Import the system_config and neonVFPBenchmark project into CCSv 5.4 for details on importing a CCS project refer the following link processors.wiki.ti.com/index.php/AM335X_StarterWare_Environment_Setup [2]
- Build the neonVFPBenchmark example project.
- Run the .out generated on to the target.
- The Benchmark numbers are displayed on to the configured console for Neon.

Steps to measure the performance numbers for VFP

- Import the system_config and neonVFPBenchmark project into CCSv 5.4
- Right Click on Project Name-> Go to Show Build Settings.
- Select VFPv3 Option from processor options dropdown for both ***system_config*** *and* **neonVFPBenchmark** *project.*
- Refer to the screeshots below for changing the compiler options.
- Neon option can be disabled for the neonVFPBenchmark project to get the VFP performance numbers. Go to Build Settings-> ARM Compiler-> Advanced Options -> Runtime Model Options-> Generate SIMD instructions targeting neon and uncheck the generates simd instructions checkbox.
- First Build the system_config project with VFPv3 option selected.
- Then build the neonVFPBenchmark project with VFPv3 option selected.
- Run the .out on the Target.
- The Benchmark numbers for VFP are displayed on the configured console.

Note:

1. After Running the neonVFPBenchmark application with VFP. Disable the VFP option enabled for system_config by following the steps listed above.
2. Failure to do step 1 results in build errors for other dependent projects in Starterware as they are not compliant with VFP calling conventions.
3. Neon option has to remain enabled for system_config project to avoid build errors as it uses NeonVFP assembly instructions.
4. To get better performance numbers it is recommended to have both Neon and VFP options enabled for neonVFPBenchmark project.

Steps to measure the Performance numbers for SoftFloat.

- For the neonVFPBenchmark project disable the Neon and/or VFP option if enabled.
- Refer to screenshots below for details on how to enable/disable Neon/VFP option for a project.

- Rebuild the project with the new settings.
- Ensure that the system_config project is built with Neon option to avoid build warnings.
- Run the .out on the target to get the performance numbers displayed on the configured console.

Screenshot for changes required for Enabling/Disabling VFP for system_config and NeonVFPBenchmark application.

Screenshot for enabling/disabling Neon option for a project.

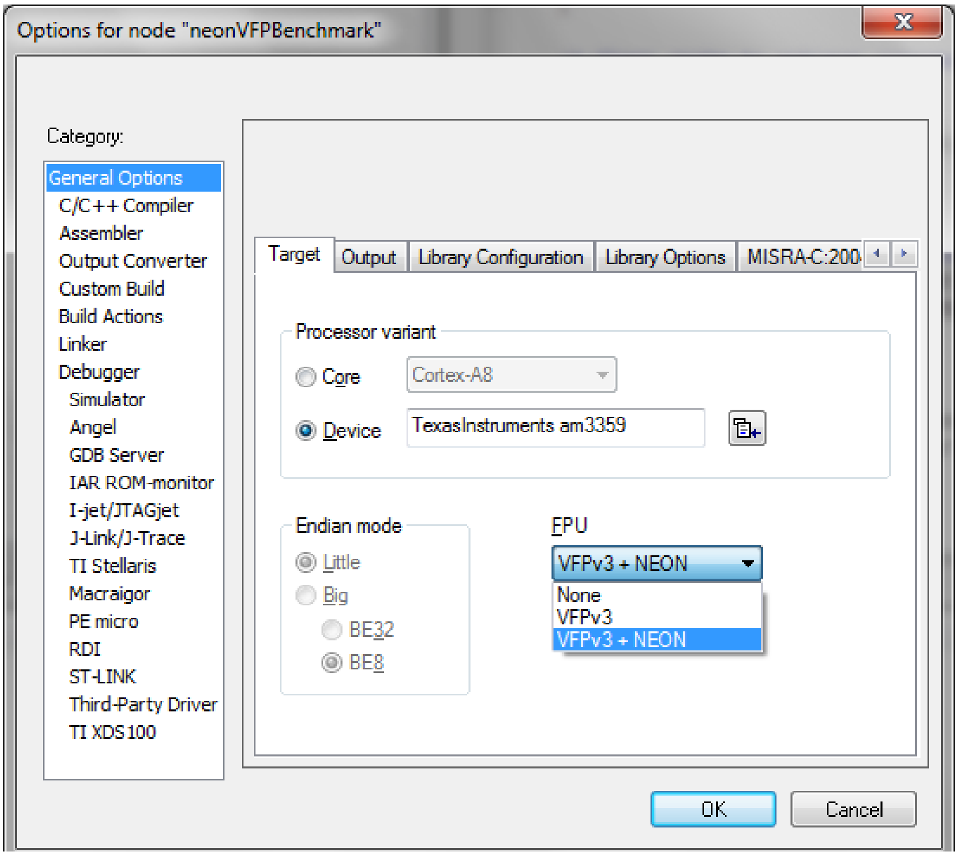
IAR Compiler Options

This section describes about the different build configurations for Benchmarking example for IAR compiler.

To get the performance numbers for VFP and SoftFloat options follow the steps listed below.

- Import the neonVFPBenchmark Project into the IAR workspace. Refer this link on how to import an IAR project processors.wiki.ti.com/index.php/AM335X_StarterWare_Environment_Setup [2]
- Right click on the project name and select the General Options then under FPU drop down menu select the FPU for which you would like to measure the performance.
- The possible options include VFPv3 + Neon, VFPv3 and None which represent the three different cases for which the example demonstrates the performance numbers.
- Build the example project with newly configured settings.
- Load the .out on the target and get the performance numbers displayed on the configured console.

Refer the ScreenShot below for selecting the appropriate option.



Known Issues and Limitations

1. IAR compiler does not support Auto-Vectorization feature.
2. The Timer used to measure the performance of functions overflows on 170 seconds. So any function which takes more time than this overflow limit cannot be measured accurately.
3. StarterWare presently does not have support for printing float point values on the console.

Additional Links

1. www.arm.com/products/processors/technologies/neon.php [3]
2. gcc.gnu.org/onlinedocs/gcc/ARM-NEON-Intrinsics.html [4]
3. Refer ARM Info Center for Cortex-A8 website for more details on Neon and VFP coprocessors

References

[1] http://processors.wiki.ti.com/index.php/Cortex-A8
[2] http://processors.wiki.ti.com/index.php/AM335X_StarterWare_Environment_Setup
[3] http://www.arm.com/products/processors/technologies/neon.php
[4] http://gcc.gnu.org/onlinedocs/gcc/ARM-NEON-Intrinsics.html

PortingStellarisWareGraphicsApplicationToSitaraWare

IMPORTANT - The content of this page is due to change quite frequently and thus the quality and accuracy are not guaranteed until this message has been removed. For suggestion/recommendation, please send mail to your local FAE.

Background

This wiki captures the re-use strategy/information of StellarisWare [1]applications on to StarterWare [2](Formerly SitaraWare).

Though the StellarisWare has been a reference in developing StarterWare all the APIs DAL (device abstraction layer functions) are not compatible with StellarisWare. Few APIs have been added, few have been removed and few got changes, all due to hardware changes. Please refer to the Migration Notes for comparison of device abstraction layers of StellarisWare and StarterWare.

So, we want to explicitly say that the applications from StellarisWare will not work as is in StarterWare context.

But we have reused some middleware from StellarisWare almost as is. An example for that is Graphics library. The changes were to support 16bpp buffer, as AM18XX EVM support LCDC/LCD.

Porting StellarisWare graphics application to StarterWare

Notes on StellarisWare Graphics/display

StellarisWare has two kinds of graphics library examples-

- one which updates the display by sending commands and data to the LCD/LCDC and the following sequence explains the typical sequence

1. The application decides what shape/action to be done on the buffer.
2. It calls the graphics library with appropriate parameters (buffer, shape, co ordinates etc)
3. Graphics library processes the request,

- the low-level display driver sends a sequence of command and data to the LCD controller to update
- the display then the flush command is sent so that the changes are updated in the display.

- the other in which the buffer in system memory is updated and is sent to off-chip LCDC/LCD to display. The following sequence explains the typical sequence

1. The application decides what shape/action to be done on the buffer.
2. It calls the graphics library with appropriate parameters (buffer, shape, co ordinates etc)
3. Graphics processes the request and gets back to Application.
4. Application programs the off-chip LCDC indirectly(eg- using gpio pins) to display the processed buffer.
5. The off-chip LCD controller will update the LCD with the new frame buffer contents

The StellarisWare HW and SW supports only 1bpp, 4bpp, and 8bpp only.

Notes on StarterWare graphics/display

The support for 16bpp(5-6-5 format) and 24bpp are added to graphics library . The following sequence explains the typical sequence

1. The application decides what shape/action to be done on the buffer.
2. It calls the graphics library with appropriate parameters (buffer, shape, co ordinates etc)
3. Graphics processes the request and gets back to Application.
4. Application programs the LCDC (APIs) to display the processed buffer.
5. The LCD controller will automatically update the LCD with the new frame buffer contents

Critical information for porting StellarisWare graphics application to StarterWare

Though at the outset we say that graphic library is re-used, it is not possible to re-use a existing StellarisWare graphics based application as is. This is because as we mentioned earlier in "background" section, the API of DAL will not be compatible. In addition to that, the following aspects also have to be looked in when trying to port StellarisWare graphics application on StarterWare.

Please note that, "application" in this context means everything except DAL (which abstract the device through set of APIs). In the diagram below shows multiple sub component that make up application.

		Sitaraware reuse?
Core CPU	Stellaris SOC's uses Cortex M3 cores. Sitara SOC's uses different ARM cores in different SOC.	The vector table (to handle interrupt and exceptions), assembly code specific to a CPU core may need to be changed according to the core processor used.
SOC specific	Interrupt controller, Pin Multiplexing, Power and sleep control, pepiphrals supported etc... are specific to an SOC.	These modules has to be configured according to the SOC used.
EVM	The EVM configurations for an SOC may differ between different EVM's.Eg- the specific instance of display controller or any peripehral extended on the EVM.	The EVM specific code has to be changed.
Core Application	Core logic involved in the application.	The Application code can be re-used. If there are any device driver calls which are not supported in Sitara SOC's, the application has to be modified.
Graphics lib API's	The API's exposed from graphics library used in StellarisWare.	The API's are not changed. So they can be reused as-is. But the low level call back functions required to update the display may differ according to the display used.
Display controller	The display interfaced on the EVM may differ across different EVM's.	So the display driver code has to be updated according to the display used.
Device abstraction layer API's	The API's exposed by the DAL. Eg- the API's exposed for uart, i2c, spi etc.	The API's exposed by the DAL of SitaraWare will differ from the DAL of Stellarisware . So all the DAL API calls has to be updated.

It is very evident that except two components which is marked in green, others require good attention for porting/changing accordingly.

Summary

Porting from StellarisWare to StarterWare is require enough consideration and porting work.

References

[1] http://focus.ti.com/mcu/docs/mculuminarysoftcode.tsp?sectionId=95&tabId=2489&familyId=1755&toolTypeId=1&DCMP=Stellaris&HQS=Other+OT+stellarisware
[2] http://processors.wiki.ti.com/index.php/StarterWare

Migration Notes

IMPORTANT - The content of this page is due to change quite frequently and thus the quality and accuracy are not guaranteed until this message has been removed. For suggestion/recommendation, please send mail to your local FAE.

Introduction

This document gives a comparison of Device Abstraction Layers of StellarisWare and SitaraWare. This document is relevant only for applications migrating from StellarisWare to SitaraWare.

Why Is SitaraWare Device Abstraction Layer Different?

- Hardware difference
  - The peripheral implementations of StellarisWare and SitaraWare are completely different
  - Some functionalities of a peripheral in StellarisWare are not available in the SitaraWare hardware
  - Same functionality is achieved by the hardwares by different means
- Some API names are dependent on register names/functionality in StellarisWare
  - API names will be ambiguous if the same name is given in SitaraWare if the register names differ

What SitaraWare Offers

- Implements APIs in such way as to offset migration where ever possible.
- Additional or enhanced set of APIs to expose new functionalities of the peripherals.

Please refer to SitaraWare API Reference Guide for more details on the API's added in SitaraWare (which are not in StellarisWare). The API reference guide is in zip format. So please unzip and save the .chm document to local disk to view its contents.

Peripherals

GPIO

StellarisWare API Name	Functionality in StellarisWare	API supported in SitaraWare ?	Deviations	Remarks
GPIODirModeSet	Sets the direction and mode of the specified pin(s).	Yes.	A parameter 'pinNumber' has replaced 'ucPins' parameter present in StellarisWare. The former specifies the serial number of the pin while the latter specified the bit mask(s) of the pin(s) to be addressed.	Serial numbers of pins provided as a parameter instead of bit masks because there are 144 pins, divided into 9 banks. An additional bank number would be added as argument if bit masks were provided.
GPIODirModeGet	Gets the direction and mode of a pin.	Yes.	Not Applicable	Not Applicable
GPIOPinWrite	Writes a value to the specified pin(s).	Yes.	A parameter 'pinNumber' has replaced 'ucPins' parameter present in StellarisWare. The former specifies the serial number of the pin while the latter specified the bit mask(s) of the pin(s) to be addressed.	Same as 'GPIODirModeSet'. 'GPIOBankPinsWrite' is provided to write to multiple pins.
GPIOPinRead	Reads the values present of the specified pin(s).	Yes.	A parameter 'pinNumber' has replaced 'ucPins' parameter present in StellarisWare. The former specifies the serial number of the pin while the latter specified the bit mask(s) of the pin(s) to be addressed.	Same as 'GPIODirModeSet'.
GPIOIntTypeSet	Sets the interrupt type for the specified pin(s).	Yes.	A parameter 'pinNumber' has replaced 'ucPins' parameter present in StellarisWare. The former specifies the serial number of the pin while the latter specified the bit mask(s) of the pin(s) to be addressed.	Same as 'GPIODirModeSet'.
GPIOIntTypeGet	Gets the interrupt type for a pin.	Yes.	Not Applicable	Not Applicable
GPIOPinIntStatus	Gets interrupt status for the specified GPIO port.	Yes.	A parameter 'pinNumber' has repalced 'bMasked' parameter present in StellarisWare. The former specifies the serial number of the pin while the latter specified whether to return raw or masked interrupt status.	No concept of masked status
GPIOPinIntClear	Clears the interrupt for the specified pin(s).	Yes.	A parameter 'pinNumber' has replaced 'ucPins' parameter present in StellarisWare. The former specifies the serial number of the pin while the latter specified the bit mask(s) of the pin(s) to be addressed.	Same as 'GPIODirModeSet'.

GPIOPadConfigSet	Sets the pad configuration for the specified pin(s).	No.	Not Applicable.	Pad configuration is part of application design
GPIOPadConfigGet	Gets the pad configuration for a pin.	No.	Not Applicable.	Pad configuration is part of application design
GPIOPinIntEnable	Enables interrupts for the specified pin(s).	No.	Not Applicable.	Only enable/disable of interrupts per bank. Thus, an API GPIOBankIntEnable is added
GPIOPinIntDisable	Disables interrupts for the specified pin(s).	No.	Not Applicable.	Only enable/disable of interrupts per bank. Thus an API GPIOBankIntDisable is added
GPIOPortIntRegister	Registers an interrupt handler for a GPIO port.	No.	Not Applicable.	ISR registration done using IntRegister API, defined in /include/interrupt.h
GPIOPortIntUnregister	Removes an interrupt handler for a GPIO port.	No.	Not Applicable.	ISR unregistration done using IntUnregister API, defined in /include/interrupt.h
GPIOPinConfigure	Configures the alternate function of a GPIO pin.	No.	Not Applicable.	Stellaris provides register support to configure GPIO pins for alternate functions. But AM1808 does not support any such features.
GPIOPinTypeADC	Configures pin(s) for use as analog-to-digital converter inputs.	No.	Not Applicable.	Same as 'GPIOPinConfigure' .
GPIOPinTypeCAN	Configures pin(s) for use as a CAN device.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeComparator	Configures pin(s) for use as an analog comparator input.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeEPI	Configures pin(s) for use by the external peripheral interface.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeEthernetLED	Configures pin(s) for use by the Ethernet peripheral as LED signals.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeGPIOInput	Configures pin(s) for use as GPIO inputs.	No.	Not Applicable.	.Same as 'GPIOPinTypeADC'.
GPIOPinTypeGPIOOutput	Configures pin(s) for use as GPIO outputs.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeGPIOOutputOD	Configures pin(s) for use as GPIO open drain outputs.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeI2C	Configures pin(s) for use by the I2C peripheral.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeI2S	Configures pin(s) for use by the I2S peripheral.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.

GPIOPinTypePWM	Configures pin(s) for use by the PWM peripheral.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeQEI	Configures pin(s) for use by the QEI peripheral.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeSSI	Configures pin(s) for use by the SSI peripheral.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeTimer	Configures pin(s) for use by the Timer peripheral.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeUART	Configures pin(s) for use by the UART peripheral.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeUSBAnalog	Configures pin(s) for use by the USB peripheral.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.
GPIOPinTypeUSBDigital	Configures pin(s) for use by the USB peripheral.	No.	Not Applicable.	Same as 'GPIOPinTypeADC'.

I2C

StellarisWare API Name	Functionality in StellarisWare	API supported in SitaraWare ?	Deviations	Remarks
I2cIntRegister	Registers an interrupt handler for the I2C module.	No	Not Applicable.	Not implemented because intr registration is static and part of the application
I2cIntUnRegister	Unregisters an interrupt handler for the I2C module.	No	Not Applicable.	Not implemented because intr unregistration is static and part of the application
I2cMasterbusy	Indicates whether or not the I2C Master is busy.	No	Not Applicable.	There is no single bit that indicates if the I2C peripheral is busy or not
I2CMasterIntClear	Clears I2C Master interrupt sources.	No	Not Applicable.	There is no single bit that clears all the interrupt status and also not all status bits can be cleared by write
I2CMasterIntDisable	Disables the I2C Master interrupt.	No	Not Applicable.	There is no single bit that disables all the interrupts. Also, it is not easy to decide which is a master interrupt and not
I2CMasterIntEnable	Enables the I2C Master interrupt.	No	Not Applicable.	There is no single bit that enables all the interrupts. Also, it is not easy to decide which is a master interrupt and not
I2CSlaveEnable	Enables the I2C slave block.	No	Not Applicable.	These is no separate slave block
I2CSlaveDisable	Disables the I2C slave block.	No	Not Applicable.	There is no separate slave block
I2CSlaveIntClear	Clears I2C slave interrupt sources.	No	Not Applicable.	There is no single bit that clears all the interrupt status and also not all status bits can be cleared by write

I2CSlaveIntDisable	Disables the I2C Slave interrupt.	No	Not Applicable.	There is no single bit that disables all the interrupts. Also, it is not easy to decide which is a master interrupt and not
I2CSlaveIntEnable	Enables the I2C Slave interrupt.	No	Not Applicable.	There is no single bit that enables all the interrupts. Also, it is not easy to decide which is a slave interrupt and not
I2CSlaveStatus	Gets the I2C Slave module status	No	Not Applicable.	There is not such register which gives the status of the slave, as there is no separate slave block
I2CMasterInitExpClk	Initializes the I2C Master block.	Yes	Parameters	Divisor calculation is different and needs addtional arguments.
I2CMasterDataGet	Receives a byte that has been sent to the I2C Master.	Yes	No	Not Applicable.
I2CMasterDataPut	Transmits a byte from the I2C Master	Yes	No	Not Applicable.
I2CMasterDisable	Disables the I2C master block.	Yes	No	Not Applicable.
I2CMasterEnable	Enables the I2C Master block.	Yes	No	Not Applicable.
I2CMasterErr	Gets the error status of the I2C Master module.	Yes	No	Not Applicable.
I2CSlaveDataGet	Receives a byte that has been sent to the I2C Slave.	Yes	No	Not Applicable.
I2CSlaveDataPut	Transmits a byte from the I2C Slave	Yes	No	Not Applicable.
I2CSlaveIntDisableEx	Disables individual I2C Slave interrupt sources.	Yes	No	Not Applicable.
I2CSlaveIntEnableEx	Enables individual I2C Slave interrupt sources	Yes	No	Not Applicable.
I2CMasterSlaveAddrSet	Sets the address that the I2C Master will place on the bus.	Yes	Parameters	Slave Address register dosen't have any bit to specify,whether master is trying to read or write to slave as in case of stellarisware.
I2CMasterBusBusy	Indicates whether or not the I2C bus is busy.	Yes	No	Not Applicable.
I2CMasterControl	Controls the state of the I2C Master module.	Yes	No	Not Applicable.
I2CSlaveIntClearEx	Clears I2C Slave interrupt sources.	Yes	No	Not Applicable.
I2CSlaveIntStatusEx	I2CSlaveIntStatusEx	Yes	No	Not Applicable.
I2CSlaveIntStatus	Gets the current I2C Slave interrupt status.	Yes	Parameters	Sitara processor has only one interrupt status register .It dosen't have raw and masked interrupt status register as in case of Stellarisware
I2CSlaveInit	Initializes the I2C Slave block.	No	Not Applicable.	No separate slave block.
I2CMasterIntStatus	Gets the current I2C Master interrupt status.	Yes	Parameters	Sitaraware has only one interrupt status register .It dosen't have raw and masked interrupt status register as in case of Stellarisware

Timer

StellarisWare API Name	Functionality in StellarisWare	API Suppported in SitaraWare ?	Deviations	Remarks
TimerEnable	Timer Enable only enables either timer A or Timer B or both.	Yes	parameters	Hardware Difference; Mode of enabling is handled in a different way
TimerDisable	Disables the Timer A or Timer B or Both	Yes	Not Applicable	NA
TimerConfigure	Selects the operating mode.	Yes	parameters	Hardware Difference; Timer peripheral is configured in a different way
TimerControlLevel	Controls the output level.	No	Not Applicable	Hardware Difference
TimerControlTrigger	Enables/Disables the trigger output	No	Not Applicable	Hardware Difference
TimerControlEvent	Controls the event type	No	Not Applicable	Hardware Difference
TimerControlStall	Controls stall handling	No	Not Applicable	Hardware Difference
TimerControlWaitOnTrigger	Daisy chain of all timer modules	No	Not Applicable	Hardware Difference
TimerRTCEnable	Enables RTC	No	Not Applicable	RTC is a separate peripheral
TimerRTCDisable	Disables RTC	No	Not Applicable	RTC is a separate peripheral
TimerPrescaleSet	Sets 8 bit prescalar	No	Not Applicable	Hardware Difference; Also, API name is register dependent
TimerPrescaleGet	Gets 8 bit prescalar	No	Not Applicable	Hardware Difference; Also, API name is register dependent
TimerPrescaleMatchSet	Set the Prescale Match Register.	No	Not Applicable	Hardware Difference; Also, API name is register dependent
TimerPrescaleMatchGet	Get the Prescale Match Register.	No	Not Applicable	Hardware Difference; Also, API name is register dependent
TimerLoadSet	Set the Load Register.	No	Not Applicable	Hardware Difference; Also, API name is register dependent
TimerLoadGet	Get the Load Register.	No	Not Applicable	Hardware Difference; Also, API name is register dependent
TimerValueGet	Reads timer Counter value	No	Not Applicable	Hardware Difference; Also, API name is register dependent
TimerMatchSet	Sets the Match Register	No	Not Applicable	Hardware Difference; Also, API name is register dependent
TimerMatchGet	Reads the Match Register	No	Not Applicable	Hardware Difference; Also, API name is register dependent
TimerIntRegister	Registers an ISR	No	Not Applicable	ISR registering done by application in SitaraWare using IntRegister() API
TimerIntUnregister	Unregisters an ISR	No	Not Applicable	ISR unregistering done by application in SitaraWare using IntUnregister() API

TimerIntEnable	Enables interrupts	Yes	Not Applicable	Not Applicable
TimerIntDisable	Disables interrupts	Yes	Not Applicable	Not Applicable
TimerIntStatus	Get the interrupt status	No	Not Applicable	Hardware Difference; Also, API name is register dependent
TimerIntClear	Clears the interrupt status	No	Not Applicable	Hardware Difference; Also, API name is register dependent

UART

StellarisWare API Name	Functionality in StellarisWare	API supported in SitaraWare ?	Deviations	Remarks
UARTConfigSetExpClk	Sets the configuration of a UART.	Yes.	A parameter named 'overSampRate' is added.	Over sampling rate is required to be passed to this function to calculate the divisor value. This divisor value is written to the Divisor Latch Registers.
UARTConfigGetExpClk	Gets the current configuration of a UART.	Yes.	Not Applicable	Not Applicable
UARTEnable	Enables transmitting and receiving.	Yes.	StellarisWare enables the FIFO alongwith enabling the UART instance. But SitaraWare enables the UART instance alone.	Enabling FIFO is done in UARTFIFOEnable() function.
UARTFIFODisable	Disables transmitting and receiving.	Yes.	StellarisWare disables FIFO mode of operation alongwith disabling the UART instance. But SitaraWare disables the UART instance alone.	Disabling FIFO is done in UARTFIFODisable() function.
UARTFIFOEnable	Enables the transmit and receive FIFOs.	Yes.	Not Applicable	Not Applicable
UARTFIFODisable	Disables the transmit and receive FIFOs.	Yes.	Not Applicable	Not Applicable
UARTFIFOLevelSet	Sets the FIFO level at which interrupts are generated.	Yes.	Stellaris allows provision to program the transmitter trigger level also. But AM1808 does not support this feature. Thus, 'ulTxLevel' is omitted in SitaraWare.	Not Applicable
UARTFIFOLevelGet	Gets the FIFO level at which interrupts are generated.	No.	Not Applicable	In AM1808, FIFO Control Register(FCR) which controls the FIFO settings shares its address with Interrupt Identification Register(IIR). FCR thus becomes a write-only register. Reading from the address gives the value of IIR. Thus one cannot read the FIFO trigger level being set.

UARTDMAEnable	Enables the DMA mode of operation.	Yes.	Not Applicable	While passing a value for 'flags' variable from application, pass a bit mask which programs the FIFO enable bit, Receiver FIFO trigger level bits and DMA enable bit. In effect, preserve the status of FIFO Enable bit and Receiver Trigger level bits.
UARTDMADisable	Disable UART DMA mode of operation.	Yes.	Not Applicable	While passing a value for 'flags' variable from application, pass a bit mask which programs the FIFO enable bit, Receiver FIFO trigger level bits and DMA enable bit. In effect, preserve the status of FIFO Enable bit and Receiver Trigger level bits.
UARTParityModeSet	Sets the type of parity for transmission.	Yes.	Not Applicable	Not Applicable
UARTParityModeGet	Gets the type of parity currently being used.	Yes.	Not Applicable	Not Applicable
UARTSpaceAvail	Determines if there is any space in the transmit FIFO.	Yes.	Return type of this function in StellarisWare is Boolean. In SitaraWare, we have replaced it with unsigned integer type.	An 8 bit value or a 32 bit value has to be processed by ARM registers which are 32 bit in length. Thus, defining boolean as unsigned or signed character would not make difference in the way the returned value is processed. Thus, we made the return type as unsigned integer.
UARTCharsAvail	Determines if there are any characters in the receive FIFO.	Yes.	Return type of this function in StellarisWare is Boolean. In SitaraWare, we have replaced it with unsigned integer type.	Same as UARTSpaceAvail
UARTCharPutNonBlocking	Sends a character to the specified port.	Yes.	Return type of this function in StellarisWare is Boolean. In SitaraWare, we have replaced it with unsigned integer type.	Same as UARTSpaceAvail
UARTCharGetNonBlocking	Receives a character from the specified port.	Yes.	Not Applicable	Not Applicable
UARTCharPut	Waits to send a character from the specified port.	Yes.	Not Applicable	Not Applicable
UARTCharGet	Waits for a character from the specified port.	Yes.	Not Applicable	Not Applicable
UARTIntEnable	Enables individual UART interrupt sources.	Yes.	Not Applicable	Not Applicable
UARTIntDisable	Disables individual UART interrupt sources.	Yes.	Not Applicable	Not Applicable

UARTIntStatus	This function gets the current interrupt status. This either checks the raw interrupt status or masked interrupt status depending on a parameter passed by the application.	Yes.	In SitaraWare, this function returns the interrupt identificaiton number of the highest priority UART interrupt that has been generated.	AM1808 has no register to store the status of each of the UART interrupts supported. The Interrupt Identification Register(IIR) in AM1808 tells whether an interrupt is pending to be serviced or not and also the interrupt ID of the highest priority interrupt that has been generated.
UARTBreakCtl	Causes a BREAK command to be sent to the receiver.	Yes.	The datatype of 'bBreakState' variable is changed from boolean to unsigned integer.	Same as UARTSpaceAvail
UARTModemControlSet	Sets the states of the DTR or RTS modem handshake outputs from the UART.	Yes.	In SitaraWare, this function enables the AFE and/or RTS bits in Modem Control Register(MCR).	The Modem Control Register(MCR) in AM1808 has a bit named AFE to enable Autoflow control operation of the UART. There is no Data Terminal Ready(DTR) bit in MCR. In effect this API provides equivalent functionality as in StellarisWare.
UARTModemControlClear	Clears the states of the DTR or RTS modem handshake outputs from the UART.	Yes.	In SitaraWare, this function disables the AFE and/or RTS bits in Modem Control Register(MCR).	Same as UARTModemControlSet
UARTModemControlGet	Returns the current states of each of the two UART modem control signals, DTR and RTS.	Yes.	This function in SitaraWare gets the status of AFE and RTS bits in the MCR.	Same as UARTModemControlSet
UARTModemStatusGet	Gets the states of the RI, DCD, DSR and CTS modem status signals.	Yes.	Not Applicable	Not Applicable
UARTRxErrorGet	This function returns the status of four error bits: Framing error, Overrun error, Parity error and Break indicator bits.	Yes.	Not Applicable	Not Applicable
UARTEnableSIR	Enables SIR (IrDA) mode on the specified UART.	No	Not Applicable	Register does not exist
UARTDisableSIR	Disables SIR (IrDA) mode on the specified UART.	No	Not Applicable	Register does not exist
UARTBusy	Determines whether the UART transmitter is busy or not.	No	Not Applicable	This functionality is implemented by UARTSpaceAvail function. Thus implementing this function would make either of the functions redundant.

UARTIntRegister	Registers an interrupt handler for a UART interrupt.	No	Not Applicable	Interrupt regsitering and unregistering are done by APIs which control the ARM Interrupt Controller(AINTC). Refer to 'system_config/am1808/src/interrupt.c' to find APIs for registering and unregistering interrupts.
UARTIntUnregister	Unregisters an interrupt handler for a UART interrupt.	No	Not Applicable	Same as UARTIntRegister
UARTIntClear	Clears UART interrupt sources.	No	Not Applicable	Refer to 'system_config/am1808/src/interrupt.c' to find APIs to clear the status of interrupts.
UARTRxErrorClear	Clears all reported receiver errors.	No	Not Applicable	There is no provision in AM1808 to clear the status of error bits.
UARTSmartCardEnable	Enables ISO 7816 smart card mode on the specified UART.	No	Not Applicable	Functionality does not exist
UARTSmartCardDisable	Disables ISO 7816 smart card mode on the specified UART.	No	Not Applicable	Functionality does not exist
UARTTxIntModeSet	Sets the operating mode for the UART transmit interrupt.	No	Not Applicable	AM1808 does not provide the feature to configure the Transmitter trigger level. The transmitter empty interrupt is generated when the transmit FIFO is empty.
UARTTxIntModeGet	Returns the current operating mode for the UART transmit interrupt.	No	Not Applicable	Same as to UARTTxIntModeSet
UARTFlowControlSet	Sets the UART hardware flow control mode to be used.	No	Not Applicable	Flow control is already implemented in the function UARTModemControlSet
UARTFlowControlGet	Returns the UART hardware flow control mode currently in use	No	Not Applicable	A similar functionality is supported by UARTModemControlGet

Universal Serial Bus

All the USB APIs (Usbdcdc.h,Usbdevice.h, Usbdeviceprivate.h, Usblib.h, Usbdcomp.h, Usblibpriv.h, Usb.h) Except the ones listed in the below table are used as is in Sitaraware

StellarisWare API Name	Functionality in StellarisWare	API Suppported in SitaraWare ?	Deviations	Remarks
USBIntRegister	Registers an interrupt handler for the USB controller	No	Not Applicable	Intr registration is part of application
USBIntUnregiste	Unregisters an interrupt handler for the USB controller.	No	NA	Intr unregistration is part of application
USBHostSuspend	Puts the USB bus in a suspended state.	No	NA	Host mode is not supported in current release
USBHostReset	Handles the USB bus reset condition.	No	NA	Host mode is not supported in current release
USBHostResume	Handles the USB bus resume condition.	No	NA	Host mode is not supported in current release

USBHostSpeedGet	Returns the current speed of the USB device connected.	No	NA	Host mode is not supported in current release
USBIntStatus	Returns the status of the USB interrupts.	No	Registers	This function uses the core registers, but in sitara interrupts are managed through wrapper registers
USBIntDisable	Disables the sources for USB interrupts.	No	Registers	Same as above
USBIntEnable	Enables the sources for USB interrupts.	No	Registers	Same as above
USBIntDisableControl	Disable control interrupts on a given USB controller.	No	Registers	Same as above
USBIntEnableControl	Enable control interrupts on a given USB controller.	No	Registers	Same as above
USBIntStatusControl	Returns the control interrupt status on a given USB controller.	YES	Registers	Modified to return the status from wrapper registers
USBIntDisableEndpoint	Disable endpoint interrupts on a given USB controller.	YES	NA	Need to do the same action at wrapper level also
USBIntEnableEndpoint	Enable endpoint interrupts on a given USB controller	YES	NA	Need to do the same action at wrapper level also
USBIntStatusEndpoint	Returns the endpoint interrupt status on a given USB controller	YES	Registers	Modified to return the status from wrapper registers
USBHostEndpointStatusClear	Clears the status bits in this endpoint in host mode.	No	NA	Host mode is not supported in current release
USBHostEndpointDataToggle	Sets the value data toggle on an endpoint in host mode.	No	NA	Host mode is not supported in current release
USBHostEndpointConfig	Sets the base configuration for a host endpoint.	No	NA	Host mode is not supported in current release
USBHostEndpointDataAck	Acknowledge that data was read from the given endpoint's FIFO in host mode.	No	NA	Host mode is not supported in current release
USBHostRequestIN	Schedules a request for an IN transaction on an endpoint in host mode.	No	NA	Host mode is not supported in current release
USBHostRequestStatus	Issues a request for a status IN transaction on endpoint zero	No	NA	Host mode is not supported in current release
USBHostAddrSet	Sets the functional address for the device that is connected to an endpoint in host mode.	No	NA	Host mode is not supported in current release
USBHostAddrGet	Gets the current functional device address for an endpoint.	No	NA	Host mode is not supported in current release
USBHostHubAddrSet	Set the hub address for the device that is connected to an endpoint	No	NA	Host mode is not supported in current release
USBHostHubAddrGet	Get the current device hub address for this endpoint.	No	NA	Host mode is not supported in current release
USBHostPwrConfig	Sets the configuration for USB power fault.	No	NA	Host mode is not supported in current release
USBHostPwrFaultEnable	Enables power fault detection.	No	NA	Host mode is not supported in current release

USBHostPwrFaultDisable	Disables power fault detection.	No	NA	Host mode is not supported in current release
USBHostPwrEnable	Enables the external power pin.	No	NA	Host mode is not supported in current release
USBHostPwrDisable	Disables the external power pin.	No	NA	Host mode is not supported in current release
USBOTGSessionRequest	Starts or ends a session.	No	NA	OTG mode is not supported in current release
USBEndpointDMAChannel	Sets the DMA channel to use for a given endpoint.	No	NA	DMA mode is not supported in current release

Graphics Library

All the StellarisWare Graphics Library API's are applicable as-is to SitaraWare. In addition to the API's supported in StellarisWare (1 BPP, 4 BPP and 8 BPP) for Off-screen buffer display drivers, SitaraWare has additional API's to support 16BPP Off-screen buffer display drivers.

StellarisWare API Name	Functionality in StellarisWare	API supported in SitaraWare ?	SitaraWare API Name	Functionality
Not Available	Not Applicable	NA	GrOffScreen16BPPInit	This function initializes a display structure, preparing it to draw into the supplied image buffer. The image buffer is assumed to be large enough to hold an image of the specified geometry.
Not Available	Not Applicable	NA	GrOffScreen16BPPColorTranslate	This function translates a 24-bit RGB color into a value that can be written into the display's frame buffer in order to reproduce that color, or the closest possible approximation of that color.
Not Available	Not Applicable	NA	RGB16	Merges 8 bit red, green and blue components in to 16bpp(5-6-5) format.
Not Available	Not Applicable	NA	GrOffScreen16BPPRectFill	This function fills a rectangle on the display. The coordinates of the rectangle are assumed to be within the extents of the display, and the rectangle specification is fully inclusive (in other words, both sXMin and sXMax are drawn, along with sYMin and sYMax).
Not Available	Not Applicable	NA	GrOffScreen16BPPLineDrawH	This function draws a horizontal line on the display. The coordinates of the line are assumed to be within the extents of the display.
Not Available	Not Applicable	NA	GrOffScreen16BPPLineDrawV	This function draws a vertical line on the display. The coordinates of the line are assumed to be within the extents of the display.
Not Available	Not Applicable	NA	GrOffScreen16BPPPixelDraw	This function sets the given pixel to a particular color. The coordinates of the pixel are assumed to be within the extents of the display.

Not Available	Not Applicable	NA	GrOffScreen16BPPPixelDrawMultiple	This function draws a horizontal sequence of pixels on the screen, using the supplied palette. For 1 bit per pixel format, the palette contains pre-translated colors; for 4 and 8 bit per pixel formats, the palette contains 24-bit RGB values that must be translated before being written to the display.

Article Sources and Contributors

Quick Start Guide StarterWare 02.00.XX.XX (supports AM335x) *Source*: http://processors.wiki.ti.com/index.php?oldid=156361 *Contributors*: Baskaran, GuruduttBharadwaj, Jeethan, Sujith, Sujithkv, Vishwa, X0189513

StarterWare Getting Started 02.00.XX.XX *Source*: http://processors.wiki.ti.com/index.php?oldid=156368 *Contributors*: Baskaran, GuruduttBharadwaj, Jeethan, Rsjsouza, Sujithkv, X0189513

AM335X StarterWare Booting And Flashing *Source*: http://processors.wiki.ti.com/index.php?oldid=156548 *Contributors*: Baskaran, GuruduttBharadwaj, NageswariSrinivasan, Sujith, Sujithkv, Vishwa, X0182916, X0189513

StarterWare Project Creation *Source*: http://processors.wiki.ti.com/index.php?oldid=156603 *Contributors*: Baskaran, GuruduttBharadwaj, X0189513

StarterWare 02.00.01.01 User Guide *Source*: http://processors.wiki.ti.com/index.php?oldid=156663 *Contributors*: Baskaran, Jeethan, Paianant, X0182916, X0189513

StarterWare CPSW *Source*: http://processors.wiki.ti.com/index.php?oldid=156607 *Contributors*: Baskaran, Vishwa, X0182916

StarterWare CPSW Port lwIP *Source*: http://processors.wiki.ti.com/index.php?oldid=108540 *Contributors*: Sujithkv

StarterWare USB *Source*: http://processors.wiki.ti.com/index.php?oldid=128280 *Contributors*: Baskaran, BijuC, Jcoombs, Jeethan, Snsirishk, Sujithkv, Vineeth Balan, Vishwa, Ypeels

StarterWare MMC *Source*: http://processors.wiki.ti.com/index.php?oldid=99370 *Contributors*: Baskaran, Vishwa

StarterWare MMCSD Driver *Source*: http://processors.wiki.ti.com/index.php?oldid=99371 *Contributors*: Baskaran, Vishwa

StarterWare NAND Driver *Source*: http://processors.wiki.ti.com/index.php?oldid=99374 *Contributors*: Baskaran, Vishwa

StarterWare ADC *Source*: http://processors.wiki.ti.com/index.php?oldid=99368 *Contributors*: Baskaran, Vishwa

StarterWare DMTimer *Source*: http://processors.wiki.ti.com/index.php?oldid=116416 *Contributors*: Baskaran, ChrisRing, Jeethan, Vishwa

StarterWare EHRPWM *Source*: http://processors.wiki.ti.com/index.php?oldid=99133 *Contributors*: Sujithkv

StarterWare ELM *Source*: http://processors.wiki.ti.com/index.php?oldid=99373 *Contributors*: Baskaran, Vishwa

StarterWare GPIO V2 *Source*: http://processors.wiki.ti.com/index.php?oldid=99369 *Contributors*: Baskaran, Vishwa

StarterWare GPMC *Source*: http://processors.wiki.ti.com/index.php?oldid=99372 *Contributors*: Baskaran, Vishwa

StarterWare HSI2C *Source*: http://processors.wiki.ti.com/index.php?oldid=110223 *Contributors*: Baskaran, GuruduttBharadwaj, Vishwa

StarterWare LCDC *Source*: http://processors.wiki.ti.com/index.php?oldid=99366 *Contributors*: Baskaran, Jcoombs, Vishwa

StarterWare McASP *Source*: http://processors.wiki.ti.com/index.php?oldid=115808 *Contributors*: GuruduttBharadwaj, Jcoombs, Sujithkv

StarterWare Audio Application *Source*: http://processors.wiki.ti.com/index.php?oldid=83353 *Contributors*: Jcoombs

StarterWare McSPI *Source*: http://processors.wiki.ti.com/index.php?oldid=116434 *Contributors*: Baskaran, Jeethan, Vishwa

StarterWare RTC *Source*: http://processors.wiki.ti.com/index.php?oldid=98450 *Contributors*: GuruduttBharadwaj, Jcoombs

StarterWare Touchscreen *Source*: http://processors.wiki.ti.com/index.php?oldid=98447 *Contributors*: GuruduttBharadwaj

StarterWare UART/IrDA/CIR *Source*: http://processors.wiki.ti.com/index.php?oldid=99360 *Contributors*: Baskaran, Vishwa

StarterWare Watchdog Timer *Source*: http://processors.wiki.ti.com/index.php?oldid=116437 *Contributors*: GuruduttBharadwaj, Jeethan

StarterWare DCAN *Source*: http://processors.wiki.ti.com/index.php?oldid=116412 *Contributors*: Baskaran, GuruduttBharadwaj, Jeethan, Sujithkv

AM335x StarterWare Power management *Source*: http://processors.wiki.ti.com/index.php?oldid=156729 *Contributors*: Baskaran, GuruduttBharadwaj, X0182916

StarterWare Graphics *Source*: http://processors.wiki.ti.com/index.php?oldid=112981 *Contributors*: Baskaran, GuruduttBharadwaj

StarterWare ConsoleUtilities *Source*: http://processors.wiki.ti.com/index.php?oldid=156643 *Contributors*: MugdhaK, Paianant, X0182916, X0189513

StarterWare Semihosting *Source*: http://processors.wiki.ti.com/index.php?oldid=156628 *Contributors*: Baskaran, Paianant

StarterWare NeonVFP *Source*: http://processors.wiki.ti.com/index.php?oldid=156644 *Contributors*: Paianant

PortingStellarisWareGraphicsApplicationToSitaraWare *Source*: http://processors.wiki.ti.com/index.php?oldid=98468 *Contributors*: Baskaran

Migration Notes *Source*: http://processors.wiki.ti.com/index.php?oldid=70417 *Contributors*: Baskaran

Image Sources, Licenses and Contributors

Image:TIBanner.png *Source*: http://processors.wiki.ti.com/index.php?title=File:TIBanner.png *License*: unknown *Contributors*: Nsnehaprabha

Image:StarterWareFramework.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:StarterWareFramework.JPG *License*: unknown *Contributors*: Jcoombs

Image:HP USB SD Card Format Tool v2p0p6 Settings.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:HP_USB_SD_Card_Format_Tool_v2p0p6_Settings.jpg *License*: unknown *Contributors*: X0189513

Image:Bootmodeswitches.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Bootmodeswitches.jpg *License*: unknown *Contributors*: Mike Tadyshak

Image:BootMode.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:BootMode.JPG *License*: unknown *Contributors*: Vishwa

Image:AM335x Boot.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:AM335x_Boot.JPG *License*: unknown *Contributors*: Vishwa

Image:New AM335x CCS Project.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:New_AM335x_CCS_Project.jpg *License*: unknown *Contributors*: GuruduttBharadwaj

Image:DCANCableSetup.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:DCANCableSetup.jpg *License*: unknown *Contributors*: Jeethan

Image:Oob slides ug.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:Oob_slides_ug.jpg *License*: unknown *Contributors*: Baskaran, Sujithkv

Image:StarterWare Ethernet.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:StarterWare_Ethernet.jpg *License*: unknown *Contributors*: Jcoombs

Image:USB Design.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:USB_Design.JPG *License*: unknown *Contributors*: Baskaran

Image:Sitaraware USB.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Sitaraware_USB.JPG *License*: unknown *Contributors*: Baskaran

Image:Cdc11.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Cdc11.JPG *License*: unknown *Contributors*: Baskaran

Image:Cdc12.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Cdc12.JPG *License*: unknown *Contributors*: Baskaran

Image:Cdc13.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Cdc13.JPG *License*: unknown *Contributors*: Baskaran

Image:Cdc14.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Cdc14.JPG *License*: unknown *Contributors*: Baskaran

Image:Control1.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Control1.JPG *License*: unknown *Contributors*: Baskaran

Image:Cdc data flow.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Cdc_data_flow.JPG *License*: unknown *Contributors*: Baskaran

Image:RX mode1.jpg *Source*: http://processors.wiki.ti.com/index.php?title=File:RX_mode1.jpg *License*: unknown *Contributors*: Baskaran

Image:TX mode.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:TX_mode.JPG *License*: unknown *Contributors*: Baskaran

Image:Bulk.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Bulk.JPG *License*: unknown *Contributors*: Jcoombs

Image:Bulk enum.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Bulk_enum.JPG *License*: unknown *Contributors*: Jcoombs

Image:Hid Mouse.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Hid_Mouse.JPG *License*: unknown *Contributors*: Baskaran

Image:HID Mouse Flow.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:HID_Mouse_Flow.JPG *License*: unknown *Contributors*: Baskaran

Image:Mouse app flow.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Mouse_app_flow.JPG *License*: unknown *Contributors*: Baskaran

Image:HID Report flow.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:HID_Report_flow.JPG *License*: unknown *Contributors*: Baskaran

Image:Msc.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Msc.JPG *License*: unknown *Contributors*: Baskaran

Image:Msc flow.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Msc_flow.JPG *License*: unknown *Contributors*: Baskaran

Image:Composite_HID_CDC.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Composite_HID_CDC.JPG *License*: unknown *Contributors*: Vishwa

Image:Composite_CDC_CDC.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Composite_CDC_CDC.JPG *License*: unknown *Contributors*: Vishwa

Image:DevMgr1.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:DevMgr1.JPG *License*: unknown *Contributors*: Vishwa

Image:DevMgr12.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:DevMgr12.JPG *License*: unknown *Contributors*: Vishwa

Image:HyperTrm1.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:HyperTrm1.JPG *License*: unknown *Contributors*: Vishwa

Image:HyperTrm2.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:HyperTrm2.JPG *License*: unknown *Contributors*: Vishwa

Image:HID host.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:HID_host.JPG *License*: unknown *Contributors*: Jcoombs

Image:Hidhostexample1.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Hidhostexample1.JPG *License*: unknown *Contributors*: Jcoombs

Image:MscHost.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:MscHost.JPG *License*: unknown *Contributors*: Jcoombs

Image:MMCSDFrameWork.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:MMCSDFrameWork.JPG *License*: unknown *Contributors*: Sujithkv

Image:NANDFrameWork.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:NANDFrameWork.JPG *License*: unknown *Contributors*: Jeethan

Image:StarterWare_AudioSlide0.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:StarterWare_AudioSlide0.JPG *License*: unknown *Contributors*: Jcoombs

Image:StarterWare_AudioSlide1.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:StarterWare_AudioSlide1.JPG *License*: unknown *Contributors*: Jcoombs

Image:StarterWare_AudioSlide3.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:StarterWare_AudioSlide3.JPG *License*: unknown *Contributors*: Jcoombs

Image:StarterWare_AudioSlide4.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:StarterWare_AudioSlide4.JPG *License*: unknown *Contributors*: Jcoombs

Image:StarterWare_AudioSlide5.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:StarterWare_AudioSlide5.JPG *License*: unknown *Contributors*: Jcoombs

Image:StarterWare_AudioSlide6.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:StarterWare_AudioSlide6.JPG *License*: unknown *Contributors*: Jcoombs

Image:SA deep sleep state flow.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:SA_deep_sleep_state_flow.JPG *License*: unknown *Contributors*: Baskaran

Image:SA A8 CM3.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:SA_A8_CM3.JPG *License*: unknown *Contributors*: Baskaran

Image:PM_Code_Flow.png *Source*: http://processors.wiki.ti.com/index.php?title=File:PM_Code_Flow.png *License*: unknown *Contributors*: X0182916

Image:SitaraWareGraphicsApplication.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:SitaraWareGraphicsApplication.JPG *License*: unknown *Contributors*: Baskaran

Image:Semihosting.jpeg *Source*: http://processors.wiki.ti.com/index.php?title=File:Semihosting.jpeg *License*: unknown *Contributors*: Mandeep

Image:StarterWare semihosting compilation.png *Source*: http://processors.wiki.ti.com/index.php?title=File:StarterWare_semihosting_compilation.png *License*: unknown *Contributors*: Baskaran

Image:IAR neon.png *Source*: http://processors.wiki.ti.com/index.php?title=File:IAR_neon.png *License*: unknown *Contributors*: Paianant

Image:Port.JPG *Source*: http://processors.wiki.ti.com/index.php?title=File:Port.JPG *License*: unknown *Contributors*: Baskaran

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.
BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

License

1. Definitions

a. "**Adaptation**" means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.
b. "**Collection**" means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined below) for the purposes of this License.
c. "**Creative Commons Compatible License**" means a license that is listed at http://creativecommons.org/compatiblelicenses that has been approved by Creative Commons as being essentially equivalent to this License, including, at a minimum, because that license: (i) contains terms that have the same purpose, meaning and effect as the License Elements of this License; and, (ii) explicitly permits the relicensing of adaptations of works made available under that license under this License or a Creative Commons jurisdiction license with the same License Elements as this License.
d. "**Distribute**" means to make available to the public the original and copies of the Work or Adaptation, as appropriate, through sale or other transfer of ownership.
e. "**License Elements**" means the following high-level license attributes as selected by Licensor and indicated in the title of this License: Attribution, ShareAlike.
f. "**Licensor**" means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.
g. "**Original Author**" means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.
h. "**Work**" means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.
i. "**You**" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.
j. "**Publicly Perform**" means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.
k. "**Reproduce**" means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections;
b. to create and Reproduce Adaptations provided that any such Adaptation, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";
c. to Distribute and Publicly Perform the Work including as incorporated in Collections; and,
d. to Distribute and Publicly Perform Adaptations.
e. For the avoidance of doubt:

   i. **Non-waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;
   ii. **Waivable Compulsory License Schemes.** In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor waives the exclusive right to collect such royalties for any exercise by You of the rights granted under this License; and,
   iii. **Voluntary License Schemes.** The Licensor waives the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. Subject to Section 8(f), all rights not expressly granted by Licensor are hereby reserved.

4. Restrictions

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested. If You create an Adaptation, upon notice from any Licensor You must, to the extent practicable, remove from the Adaptation any credit as required by Section 4(c), as requested.
b. You may Distribute or Publicly Perform an Adaptation only under the terms of: (i) this License; (ii) a later version of this License with the same License Elements as this License; (iii) a Creative Commons jurisdiction license (either this or a later license version) that contains the same License Elements as this License (e.g., Attribution-ShareAlike 3.0 US)); (iv) a Creative Commons Compatible License. If you license the Adaptation under one of the licenses mentioned in (iv), you must comply with the terms of that license. If you license the Adaptation under the terms of any of the licenses mentioned in (i), (ii) or (iii) (the "Applicable License"), you must comply with the terms of the Applicable License generally and the following provisions: (I) You must include a copy of, or the URI for, the Applicable License with every copy of each Adaptation You Distribute or Publicly Perform; (II) You may not offer or impose any terms on the Adaptation that restrict the terms of the Applicable License or the ability of the recipient of the Adaptation to exercise the rights granted to that recipient under the terms of the Applicable License; (III) You must keep intact all notices that refer to the Applicable License and to the disclaimer of warranties with every copy of the Work as included in the Adaptation You Distribute or Publicly Perform; (IV) when You Distribute or Publicly Perform the Adaptation, You may not impose any effective technological measures on the Adaptation that restrict the ability of a recipient of the Adaptation from You to exercise the rights granted to that recipient under the terms of the Applicable License. This Section 4(b) applies to the Adaptation as incorporated in a Collection, but this does not require the Collection apart from the Adaptation itself to be made subject to the terms of the Applicable License.
c. If You Distribute, or Publicly Perform the Work or any Adaptations or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and (iv) , consistent with Ssection 3(b), in the case of an Adaptation, a credit identifying the use of the Work in the Adaptation (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Adaptation or Collection, at a minimum such credit will appear, if a credit for all contributing authors of the Adaptation or Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.
d. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Adaptations or Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation. Licensor agrees that in those jurisdictions (e.g. Japan), in which any exercise of the right granted in Section 3(b) of this License (the right to make Adaptations) would be deemed to be a distortion, mutilation, modification or other derogatory action prejudicial to the Original Author's honor and reputation, the Licensor will waive or not assert, as appropriate, this Section, to the fullest extent permitted by the applicable national law, to enable You to reasonably exercise Your right under Section 3(b) of this License (right to make Adaptations) but not otherwise.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Adaptations or Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.
b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.
b. Each time You Distribute or Publicly Perform an Adaptation, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.
c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.
d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.
e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.
f. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.