1 Introduction

When creating a new embedded Linux design, getting a printed circuit board (PCB) to boot for the first time can have many unique challenges. These challenges can be hardware related: incorrect connections, improper voltages, bad component values, etc. or software related: incorrect device tree, improper drivers, bad configuration values etc. One common challenge: how to properly load and configure software drivers during boot

It is important for the software running on the new design to recognize the unique set of peripherals and components used on the PCB so that the drivers can be properly loaded and configured. This can be accomplished in many different ways: pulling GPIO pins, programming hardware fuses, storing values in non-volatile memory, etc. This is an area where having non-volatile storage, like EEPROM, either inside a System-in-Package, as in the OSD335x-SM, or on the board can help. The Linux images from BeagleBoard.org used for the OSD335x Family of devices, identify designs by a unique code, a **board ID**, that is stored within an EEPROM attached to the I2C0 bus. This board ID is then used within U-Boot to properly configure the system.

This article will discuss: how the board ID is used in U-Boot (Section 3); how to modify U-Boot to ignore the board ID (Section 4); and how to program the board ID in an EEPROM either before boot (Section 6) or within U-Boot (Section 5). Your preferred method to program the board ID, depends on the hardware in your system. For example, if you can boot from a microSD card, then modifying U-Boot to program the board ID (either Section 4.2 or Section 5) might be your preferred method. Similarly, if you have an external I2C programmer, you would prefer the programming method outlined in Section 6.1. If you would like to program the EEPROM over USB from a host PC (Section 6.2), software has been provided in the associated zip file and can also be found in Sections 8 and 9.

Table of Contents

2 Revision History

Revision Number	Revision Date	Changes	Author
1	6/11/2018	Initial Release	Erik Welsh

3 Understanding the Board ID

A board ID can be used for many different functions:

- Distinguishing between different hardware designs
- Allowing inventory management and board tracking, such as differentiating between manufacturers
- Identifying and tracking failures for yield analysis
- Compatibility checking for software
- Encoding board revisions for debug

During boot, it is especially important to distinguish between different hardware designs so that software drivers can be properly configured and loaded. As part of the ***Linux boot process for the OSD335x Family of devices***, U-Boot uses the board ID to determine the printed circuit board (board) on which it is running. This makes U-Boot more flexible and allows a single U-Boot image to be used for many different development platforms. You can see that there are many functions defined in the U-Boot file *"./board/ti/am335x/board.h"* that are used to control what functions are performed during the boot process (you can find the files by viewing the U-Boot source code, see reference section on page 18). For example, the following function is used to determine if the board is the Beagleboard.org® PocketBeagle®:

```
static inline int board_is_pb(void)
{
        return board_ti_is("A335PBGL");
}
```

This function uses a common board detect infrastructure defined in *"./board/ti/common/board_detect.c"* and *"./board/ti/common/board_detect.h"* to determine the board. The board detect infrastructure in turn relies on a specific data structure, a *ti_am_eeprom* struct. This structure is found at the beginning of an EEPROM on an I2C bus (I2C0 in the case of the AM335x devices):

```
/**
 * struct ti_am_eeprom - This structure holds data read in from the
 *                     AM335x, AM437x, AM57xx TI EVM EEPROMs.
 * @header: This holds the magic number
 * @name: The name of the board
 * @version: Board revision
 * @serial: Board serial number
 * @config: Reserved
 * @mac_addr: Any MAC addresses written in the EEPROM
 *
 * The data is this structure is read from the EEPROM on the board.
 * It is used for board detection which is based on name. It is used
 * to configure specific TI boards. This allows booting of multiple
 * TI boards with a single MLO and u-boot.
 */
struct ti_am_eeprom {
        unsigned int header;
        char name[TI_EEPROM_HDR_NAME_LEN];
        char version[TI_EEPROM_HDR_REV_LEN];
        char serial[TI_EEPROM_HDR_SERIAL_LEN];
        char config[TI_EEPROM_HDR_CONFIG_LEN];
        char mac_addr[TI_EEPROM_HDR_NO_OF_MAC_ADDR][TI_EEPROM_HDR_ETH_ALEN];
} __attribute__ ((__packed__));
```

In the definition above, the *header* field is a "magic number", i.e. a constant, *TI_EEPROM_HEADER_MAGIC,* with a value of *0xEE3355AA*. This structure can be generically referred to as a "board ID" even though it has many different pieces of information. If you read the contents of an EEPROM from the Octavo Systems development board OSD3358-SM-RED, you can see that it follows the above data structure format:

```
Address    Value                                            ASCII Value
00000000   AA 55 33 EE  41 33 33 35  42 4E 4C 54  4F 53 30 30  .U3.A335BNLTOS00
00000010   00 00 00 00  00 00 00 00  00 00 00 00  FF FF FF FF  ................
```

Unfortunately, the reliance on the board ID during U-Boot can cause problems when first booting a board after manufacturing. By default, all EEPROMs are initially unprogrammed (i.e. all bytes have a value of 0xFF) when placed on a board. Therefore, when U-Boot first comes up and reads the unprogrammed board ID, it will read a value that does not match any board causing the software to hang because U-Boot is unable to know how to configure any peripheral to continue the boot. Unfortunately, the software hang occurs before the U-Boot console is active which can be mistaken for hardware bring-up problems (i.e. power is applied to the board, but nothing happens).

To mitigate this issue, you can either modify U-Boot to ignore the board ID information within the EEPROM (i.e. hard code the board ID), or you can program the EEPROM to have the correct board ID information for the given system.

4 Modifying U-Boot to Ignore the Board ID

One way to overcome an unprogrammed EEPROM is to modify U-Boot itself so that either the board ID is ignored by U-Boot or that the value of an unprogrammed EEPROM is recognized as a "blank" board. While modifying U-Boot can be a good short-term solution to work around this condition, it is not necessarily a good long-term solution to have U-Boot either ignore the board ID or for the board to be recognized as "blank". Doing this in a production software image can be problematic in the case that revisions of the system have components that must be handled differently during boot. It also limits the reusability of the production software image across multiple products. Therefore, it is recommended to only ignore the board ID during the prototyping phase of a design and to program the EEPROM with a valid board ID during production (See Section 5).

To modify U-Boot, you first must be familiar with the process needed to build U-Boot. You can find instructions on how to do this at ***https://eewiki.net/display/linuxonarm/BeagleBone+Black*** Once you are familiar with the process, there are two methods you can follow to update U-Boot to bypass the board ID checks.

4.1 Hard-Coding the Board ID

In this first method, you can manually modify U-Boot to hard code a function within "*./board/ti/am335x/board.h*" so that the board has a fixed identity (i.e. the board ID is "hard coded"). For example, to make U-Boot always identify the board as "BeagleBone® Black", you will need to make the following modification:

```
static inline int board_is_bone_lt(void)
{
//      return board_ti_is("A335BNLT");  -- Hard code board ID to BeagleBone Black
        return 1;
}
```

By returning "1" (i.e. true), this function will make U-Boot believe that the board is BeagleBone® Black and follow the boot process for BeagleBone® Black. If this works for your system (i.e. you have similar components to the BeagleBone® Black for booting, like an eMMC on MMC1 and an SD card on MMC0), then you can use this to bypass the board ID check.

4.2 Recognizing a "Blank" Board ID

Another method to ignore the board ID, is to update U-Boot to recognize the unprogrammed EEPROM value as "blank". To do this, a patch has been created that can help with this process. As part of the U-Boot build process there are two patches that must be downloaded and applied to the U-Boot code base:

```
wget -c https://rcn-ee.com/repos/git/u-boot-patches/v2018.03/0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch
wget -c https://rcn-ee.com/repos/git/u-boot-patches/v2018.03/0002-U-Boot-BeagleBone-Cape-Manager.patch
```

In addition to these patches, another patch can be downloaded that will configure U-Boot to recognize the unprogrammed EEPROM value as a "blank" board and potentially program the EEPROM with a board ID. You can find this patch at:

https://github.com/RobertCNelson/Bootloader-Builder/blob/master/patches/v2018.03-rc1/0002-NFM-Production-eeprom-assume-device-is-BeagleBone-Bl.patch

> Caveat:
>
> The version of U-Boot and the patches listed above may not have be the same in the future. The correct versions and names should be listed in the instructions to build U-Boot.

If you look at the patch, you can see how the files in U-Boot will be modified to ignore the board ID by recognizing the unprogrammed EEPROM as a "blank" (i.e. "A335BLNK") board and then potentially program a board ID value into the EEPROM. If you look at lines 153 to 167 of the patch:

```
@@ -190,6 +190,14 @@
                                "setenv fdtfile am335x-boneblack.dtb; " \
                        "fi; " \
                "fi; " \
+               "if test $board_name = A335BLNK; then " \
+                       "if test -e mmc 0:1 /boot/.eeprom.txt; then " \
+                               "load mmc 0:1 ${loadaddr} /boot/.eeprom.txt;" \
+                               "env import -t ${loadaddr} ${filesize};" \
+                               "echo Loaded environment from /boot/.eeprom.txt;" \
+                               "run eeprom_program; " \
+                       "fi;" \
+                       "setenv fdtfile am335x-boneblack-emmc-overlay.dtb; fi; " \
                "if test $board_name = BBBW; then " \
                        "setenv fdtfile am335x-boneblack-wireless.dtb; fi; " \
                "if test $board_name = BBG1; then " \
```

You can see that if the board is "blank" (i.e. "A335BLNK"), then the boot process will check to see if the file *"/boot/.eeprom.txt"* exists in the root file system of the boot image. If it does, then it will automatically run the *eeprom_program* command, defined in *0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch*, utilizing the variables set in the *"/boot/.eeprom.txt"* file. If you look at the *eeprom_program* command (lines 1607 to 1617 of *0001-am335x_evm-uEnv.txt-bootz-n-fixes.patch*):

```
+       "eeprom_program="\
+               "if test $board_eeprom_header = bbb_blank; then " \
+                       "run eeprom_dump; run eeprom_blank; run eeprom_bbb_header; run eeprom_dump;
reset; fi; " \
+               "if test $board_eeprom_header = bbbl_blank; then " \
+                       "run eeprom_dump; run eeprom_blank; run eeprom_bbb_header; run
eeprom_bbbl_footer; run eeprom_dump; reset; fi; " \
+               "if test $board_eeprom_header = bbbw_blank; then " \
+                       "run eeprom_dump; run eeprom_blank; run eeprom_bbb_header; run
eeprom_bbbw_footer; run eeprom_dump; reset; fi; " \
+               "if test $board_eeprom_header = os00_blank; then " \
+                       "run eeprom_dump; run eeprom_blank; run eeprom_bbb_header; run
eeprom_os00_footer; run eeprom_dump; reset; fi; " \
+               "if test $board_eeprom_header = beaglelogic_blank; then " \
+                       "run eeprom_dump; run eeprom_blank; run eeprom_beaglelogic; run eeprom_dump;
reset; fi;  \0" \
```

You can see that depending on the value of the variable *board_eeprom_header*, the appropriate board ID value will be programmed into the EEPROM. As of this writing, the acceptable values for *board_eeprom_header* are:

- `board_eeprom_header=bbb_blank`
- `board_eeprom_header=bbbl_blank`
- `board_eeprom_header=bbbw_blank`
- `board_eeprom_header=os00_blank`
- `board_eeprom_header=beaglelogic_blank`

If you need to add in a custom board ID value to be programmed into the EEPROM, it is straight forward to extend the code from the patches (i.e. the *eeprom_program* command and the *EEPROM_PROGRAMMING* #define). In this way, you can use the updated U-Boot image to program all of your systems so that they can have a valid board ID.

5 Programming the Board ID Within U-Boot

If you used the “hard-coding method” or did not use the automated EEPROM programming functions of the “blank method” in Section 4, you can still program the board ID manually in U-Boot. Once you have been able to boot to the U-Boot console (i.e. you have bypassed the board ID check), it is straight forward to program values in the EEPROM corresponding to the board ID structure. From the U-Boot prompt, you only need to use the *i2c* command to program the EEPROM with the appropriate value. The commands below can be used to program the board ID for the OSD3358-SM-RED board.

```
// Set i2c device
i2c dev 0

// Set the EEPROM header “magic number”:  0xAA5533EE
i2c mw 0x50 0x00.2 aa
i2c mw 0x50 0x01.2 55
i2c mw 0x50 0x02.2 33
i2c mw 0x50 0x03.2 ee

// Set the EEPROM name (bytes 0 – 4):  “A335”
i2c mw 0x50 0x04.2 41
i2c mw 0x50 0x05.2 33
i2c mw 0x50 0x06.2 33
i2c mw 0x50 0x07.2 35

// Set the EEPROM name (bytes 4 – 7):  “BNLT”
i2c mw 0x50 0x08.2 42
i2c mw 0x50 0x09.2 4e
i2c mw 0x50 0x0a.2 4c
i2c mw 0x50 0x0b.2 54

// Set the EEPROM version:  “OS00” – OSD3358-SM-RED development platform
i2c mw 0x50 0x0c.2 4f
i2c mw 0x50 0x0d.2 53
i2c mw 0x50 0x0e.2 30
i2c mw 0x50 0x0f.2 30
```

Caveat:

The version field of early revisions of the OSD3358-SM-RED had a value of “BBNR” (i.e. 42 42 4e 52)

Each of the values passed to the I2C write command (e.g. 42 or 4e) is a hexadecimal ASCII value (***https://www.asciitable.com/***). Once the *name* and *version* fields of the EEPROM data structure are written, you can check that the programming was successful using an I2C read command:

```
=> i2c dev 0
Setting bus to 0

=> i2c md 0x50 0x00.2 20
0000: aa 55 33 ee 41 33 33 35 42 4e 4c 54 4f 53 30 30    .U3.A335BNLTOS00
0010: 00 00 00 00 00 00 00 00 00 00 00 00 ff ff ff ff    ................
```

At this point, the board has been successfully programmed so that it can now boot a default Linux image from BeagleBoard.org®. However, if you would also like to add a serial number to the EEPROM, you may do so in the next twelve (12) bytes (above example has a serial number of: 000000000000).

6 Programming the Board ID Outside of U-Boot

If there is no removable media, like an microSD card, in a system, it can be difficult to modify and load U-Boot to program an EEPROM. However, it is possible to program the board ID directly without modifying U-Boot by either using an external I2C programmer or over USB from a host PC.

6.1 Using an External I2C Programmer

To program the board ID using an external I2C programmer, there are two requirements:

1. The I2C0 pins must be accessible (i.e. the pins must be brought out to a header or test points) so that they can be connected to an external I2C programmer
2. The AM335x within the OSD335x family of devices should be held in reset (i.e. WARMRSTN should be held low).

Making the I2C0 pins accessible is straight forward but must be added during the hardware design phase of your system. If the I2C0 pins are not accessible, it will make using an external I2C programmer difficult to impossible.

Also, the AM335x within the OSD335x family of devices should be held in reset when using an external I2C programmer. This will guarantee that there is only one master on the I2C0 bus and that there will be no conflicts with the AM335x trying to control the bus. This requires that a reset button or header exists that can hold the WARMRSTN pin low.

6.2 Over USB From a Host PC

To program the board ID over USB from host PC, you can use a custom bare-metal program that will write values to the EEPROM. This method requires:

- AM335x StarterWare 02.00.01.01
- GCC cross compiler
- CCS UniFlash v3.4.1
- Target system to boot from USB
- Optional: UART-to-USB adapter and Terminal Emulator (such as Putty[2])

For the target system to boot from USB, the boot mode must try to use the USB peripheral to download a boot image. For example, if the SYSBOOT[4:0] pins have a value of 11000b, i.e. the boot mode selected by the SD Boot button on the OSD3358-SM-RED, then the processor will try SPI0, MMC0, USB0, and UART0, in that order, to boot. This will allow programming over USB0.

6.2.1 Installing the Tools

To install StarterWare and the GCC cross compiler, please follow the instructions from Sections 3 (*Installing StarterWare for AM335x*) and 4 (*Installing Linaro GCC Compiler*) of the ***Bare Metal Applications on OSD335x using U-Boot*** application note which can be found ***here***. The instructions provided below assume that both of these tools were installed on a Linux system (Ubuntu 16.04 was used in the example). However, the instructions should be similar for a Windows system as long as the GCC compiler is used.

To install the CCS UniFlash first follow the *Installation Instructions* from:

http://processors.wiki.ti.com/index.php/CCS_UniFlash_v3.4.1_Release_Notes

The instructions provided below assume that this tool was installed on a Windows system (Window 10 was used in the example). However, the instructions should be similar for a Linux system.

Once the UniFlash tool is installed, you will need to modify the configuration files so that the tool uses more appropriate IP addresses. By default, the configuration files use the 192.168.100 subnet. However, the USB RNDIS connection on the example system was configured to use the 192.168.0 subnet. Therefore, the following configuration files need to be modified:

File **uniflash_3.4/third_party/sitara/opendhcp.ini**:

- Line 22:
  - Original version:
    - `192.168.100.1`
  - New version:
    - `192.168.0.1`
- Lines 117 – 119:
  - Original version:
    - `DHCPRange=192.168.100.2-192.168.100.254`
    - `NextServer=192.168.100.1`
    - `Router=192.168.100.1`

- New version:
  - `DHCPRange=192.168.0.2-192.168.0.254`
  - `NextServer=192.168.0.1`
  - `Router=192.168.0.1`

File **uniflash_3.4/third_party/sitara/opentftp.ini**:

- Line 23:
  - Original version
    - `192.168.100.1`
  - New version
    - `192.168.0.1`

Depending on your USB RNDIS connection configuration, you should adjust the default subnet accordingly. If you do not wish to modify the configuration files, you can always manually modify the subnet values when running the tool.

6.2.2 Compiling the Binary

Next, you will need to modify the ***hsi2c_eeprom*** example in order to create a program that will program a custom value into the EEPROM to set the board ID. As part of the installation instruction for the GCC cross compiler, you should have set the LIB_PATH environment variable. This variable must be set for the makefiles for the GCC build to work correctly.

1. Replace the *hsi2cEeprom.c* file with the one provided (see Appendix A: hsi2cEeprom.c)
   a. Change directories to: *AM335X_StarterWare_02_00_01_01/examples/beaglebone/hsi2c_eeprom*
   b. Replace the contents of *hsi2cEeprom.c* with the provided code. This code uses a polling I2C method to read values from and write values to the EEPROM.

2. Modify the *hsi2cEeprom.c* file with your appropriate board ID value
   a. On approximately line 85, there are two *#define* variables that tell the program the number of bytes to write the EEPROM (***NUM_BYTES_TO_WRITE***) as well as the values to write to the EEPROM (***EEPROM_VALUE_TO_WRITE***).
   b. Currently, these values are set to the OSD3358-SM-RED board ID. These values should be updated to program your board ID.

3. Replace the *hsi2c_eeprom* linker command file with the one provided(See Appendix B: hsi2cEeprom.lds):
   a. Change directories to: *AM335X_StarterWare_02_00_01_01/build/armv7a/gcc/am335x/beaglebone/hsi2c_eeprom*
   b. Replace the contents of the linker command file, *hsi2cEeprom.lds*, with the provided code. This will place the program within the internal AM335x memory so that it can be used directly as a Secondary Program Loader (SPL) without having to initialize DDR.

4. Build the program
    a. In the same directory as the linker command file: *AM335X_StarterWare_02_00_01_01/build/armv7a/gcc/am335x/beaglebone/hsi2c_eeprom*
    b. Execute the following commands:
        i. `make clean`
        ii. `make`

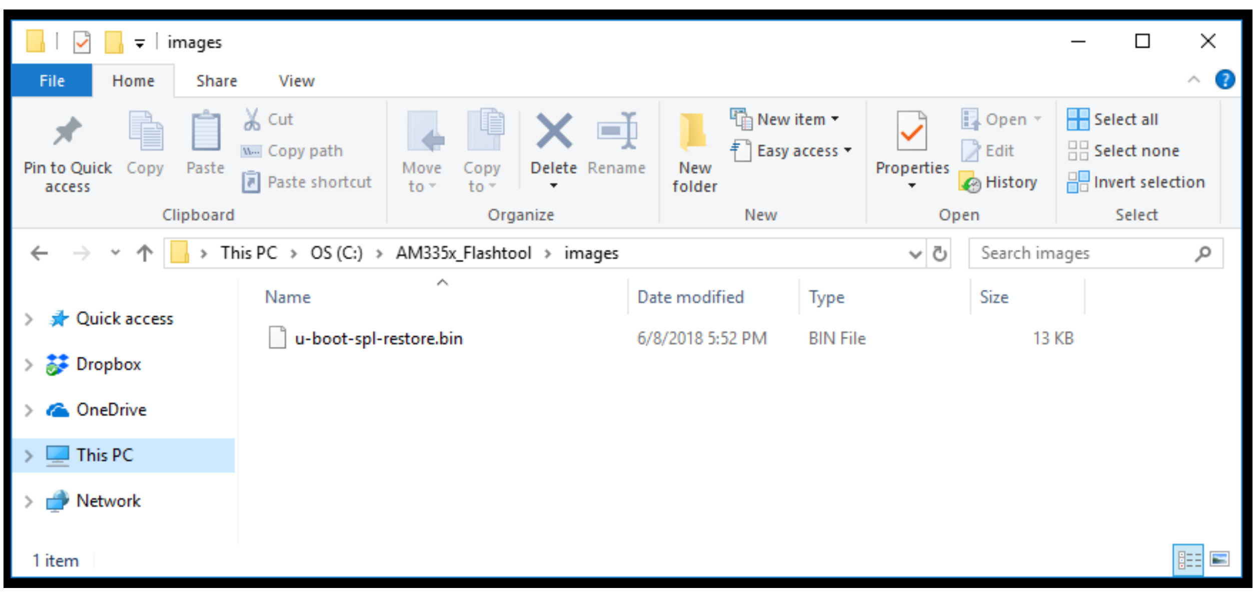
5. Collect the compiled binary to program into the device
    a. Change directories to: *AM335X_StarterWare_02_00_01_01/binary/armv7a/gcc/am335x/beaglebone/hsi2c_eeprom/Debug*
    b. Collect the following file to be used to program the device: *hsi2cEeprom.bin*

If you need to change the value programmed into the EEPROM for the board ID, you can update the *hsi2cEeprom.c* file from Step 2 and then recompile the executable using Step 4.

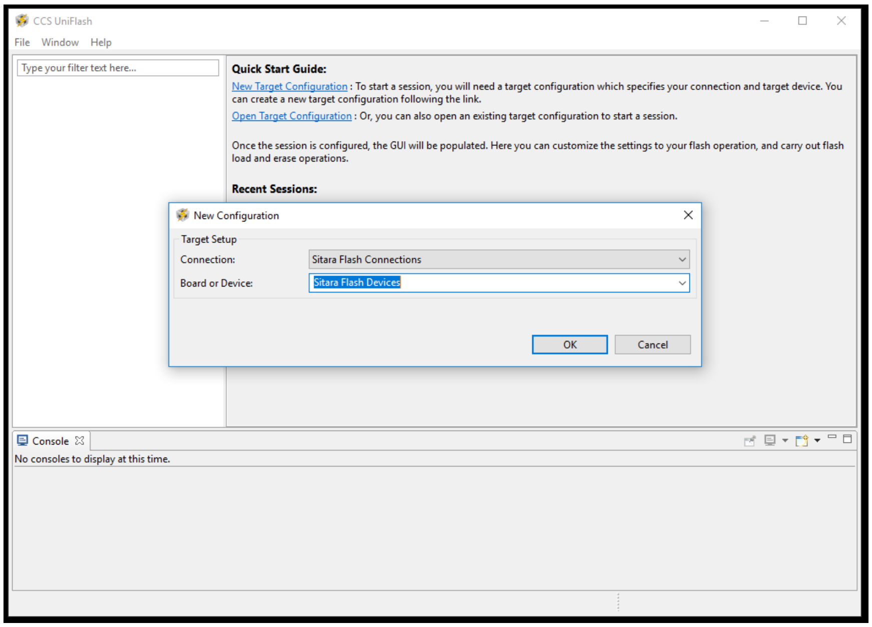
6.2.3 Programming the Device

Now that you have the executable program, *hsi2cEeprom.bin*, that will program the board ID into the EEPROM, you need to use the UniFlash tool to load and run the program.

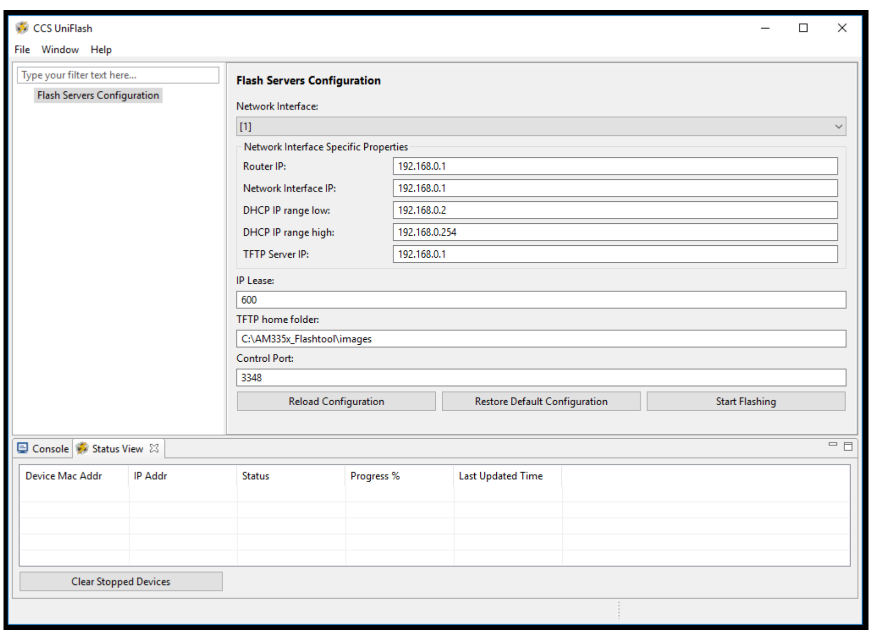
1. Update the UniFlash file to load into the device
    a. Open folder: *C:\AM335x_Flashtool\images*
    b. Copy *hsi2cEeprom.bin* to this directory
    c. Change the name of the file to *u-boot-spl-restore.bin*



2. Setup the UniFlash tool
    a. Open the UniFlash tool
    b. Select “New Target Configuration”
    c. Select “Sitara Flash Connections” for “Conection”
    d. Select “Sitara Flash Devices” for “Board or Device”



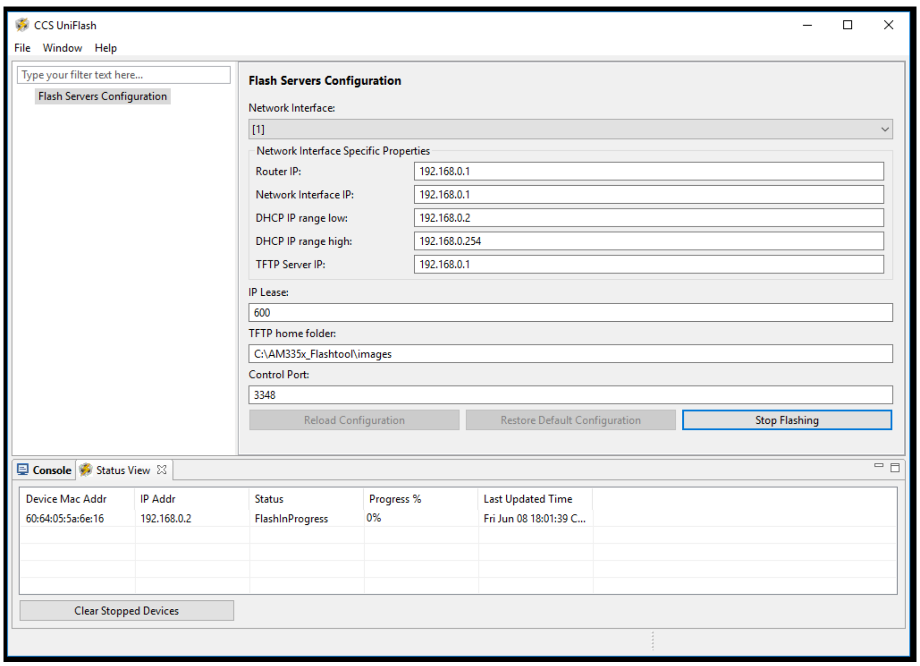
    e. At this point, you should see the following settings in the tool (These can be adjusted if they are not correct for your setup).



3. Optional: View program output
    a. Connection UART-to-USB adapter to the UART0 interface of the target system
    b. Start a terminal emulator program (such as Putty)

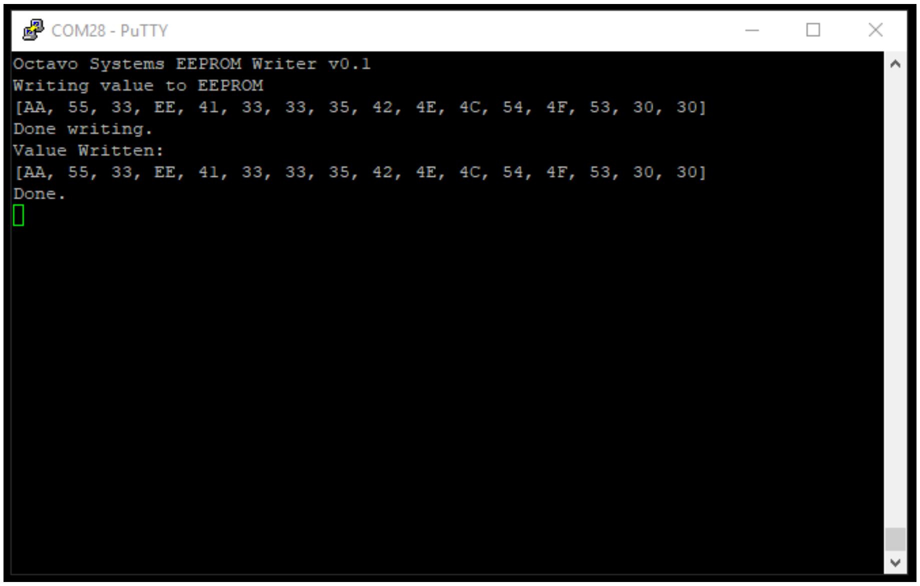
This will allow you to see the output of the program on UART0 but is optional.

4. Load and execute the program
    a. Connect a USB cable from the appropriate USB interface of the target system to the host system that is running CCS UniFlash.
    b. Boot the board in the appropriate boot mode
        i. Note: In Windows, you should hear a notification that the target system has connected to the host using USB RNDIS. You should also be able to see this connection in the "Network Connections" window. By looking at this connection, you can debug any IP address issues you may have with the UniFlash IP settings.
    c. Click the "Start Flashing" button in UniFlash



    d. After a few seconds (up to a minute), you should see the target system populate a line in the "Status View". This will always show a "Progress %" of "0%" even though the program has been loaded and executed by the target system. You can view the download of the executable with a tool such as WireShark[3].

e. If you have the optional program output enabled, then you should see the following output on the UART0 interface (with the value that you set in the program instead of the default value pictured below).



f. Close UniFlash since the target system has been successfully programmed.

Caveat:

Please ensure that the EEPROM is not write protected when you try to write the board ID value into the EEPROM. For the OSD335x-SM, this means pulling the *EEPROM_WP* pin low.

7 References

For more information on software and optional tools, please refer to the following links:

1. U-Boot ***https://github.com/u-boot/u-boot***
2. Putty ***https://www.putty.org/***
3. WireShark ***https://www.wireshark.org/***

8 Appendix A: hsi2cEeprom.c

```
/**
 * \file  eeprom_writer.c
 *
 * \brief Application to write a blank EEPROM with a value.  This will allow
 *        customers to set the Board ID to configure U-Boot.
 *
 *        Application Configuration:
 *
 *            Modules Used:
 *                I2C0
 *                UART0
 *
 *            Configurable parameters:
 *                NUM_BYTES_TO_WRITE
 *                EEPROM_VALUE_TO_WRITE
 *
 *            Hard-coded configuration of other parameters:
 *                Bus frequency     - 100kHz
 *                Addressing mode   - 7bit
 *                I2C Instance      - 0
 *                Slave Address     - 0x50
 *                EEPROM memory address - 0x0000
 *
 *        Limitations:
 *        With no flashed data in EEPROM, if the application tries to read from
 *        EEPROM, then the data values read would be "0xFF", which indicates an
 *        invalid EEPROM data.
 */

/*
* Copyright (C) 2018 Octavo Systems, LLC - http://www.octavosystems.com/
*/
/*
* Copyright (C) 2010 Texas Instruments Incorporated - http://www.ti.com/
*/
/*
*  Redistribution and use in source and binary forms, with or without
*  modification, are permitted provided that the following conditions
*  are met:
*
*    Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
*
*    Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the
*    distribution.
*
*    Neither the name of Texas Instruments Incorporated nor the names of
*    its contributors may be used to endorse or promote products derived
*    from this software without specific prior written permission.
*
*  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
*  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
*  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
*  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
*  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
*  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
*  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
*  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
*  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
*  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
*  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*
*/
```

```
#include "hw_types.h"
#include "hw_control_AM335x.h"
#include "hw_cm_wkup.h"
#include "watchdog.h"
#include "evmAM335x.h"
#include "hsi2c.h"
#include "uartStdio.h"
#include "soc_AM335x.h"
#include "stdio.h"


/******************************************************************************
**                          INTERNAL MACRO DEFINITIONS
*******************************************************************************/

//
// MODIFY THIS CODE
//   - NUM_BYTES_TO_WRITE should match the number of data values provided in
EEPROM_VALUE_TO_WRITE
//   - NUM_BYTES_TO_WRITE should not exceed BUFFER_SIZE
//

#define  NUM_BYTES_TO_WRITE            (16)
#define  EEPROM_VALUE_TO_WRITE         {0xAA, 0x55, 0x33, 0xEE, 0x41, 0x33, 0x33, 0x35, 0x42,
0x4E, 0x4C, 0x54, 0x4F, 0x53, 0x30, 0x30}

//
// END MODIFY THIS CODE
//


/* I2C address of AT24C256 eeprom */
#define  I2C_SLAVE_ADDR                (0x50)

/* Higher byte address (i.e A8-A15) */
#define  E2PROM_ADDR_MSB               (0x00)

/* Lower byte address (i.e A0-A7) */
#define  E2PROM_ADDR_LSB               (0x00)

/* I2C instance        */
#define  I2C_0                         (0x0u)

/* System clock fed to I2C module - 48Mhz    */
#define  I2C_SYSTEM_CLOCK              (48000000u)

/* Internal clock used by I2C module - 12Mhz */
#define  I2C_INTERNAL_CLOCK            (12000000u)

/* I2C bus speed or frequency - 100Khz       */
#define  I2C_OUTPUT_CLOCK             (100000u)

/* I2C interrupt flags to clear      */
#define  I2C_INTERRUPT_FLAG_TO_CLR     (0x7FF)

/* Buffer Size */
#define  BUFFER_SIZE                   (1024)


/******************************************************************************
**                          INTERNAL FUNCTION PROTOTYPES
*******************************************************************************/
static void setup_platform(void);
static void print_data(unsigned char *data, unsigned int size);
static void setup_I2C(void);
static void setup_UART(void);
static void SetupI2CTransmit(unsigned int dcount);
static void SetupReception(unsigned int dcount);
```

```
/******************************************************************************
**                      INTERNAL VARIABLE DEFINITIONS
******************************************************************************/
volatile unsigned char dataToSlave[BUFFER_SIZE];
volatile unsigned char dataFromSlave[BUFFER_SIZE];
volatile unsigned int  tCount;
volatile unsigned int  rCount;


/******************************************************************************
**              FUNCTION DEFINITIONS
******************************************************************************/

int main(void)
{
    volatile unsigned char data[NUM_BYTES_TO_WRITE] = EEPROM_VALUE_TO_WRITE;
    volatile unsigned int  i, j;

    // Setup the platform
    setup_platform();

    // Initialize variables
    for (i = 0; i < (NUM_BYTES_TO_WRITE + 2); i++) {
        dataToSlave[i] = 0;
        dataFromSlave[i] = 0;
    }

    UARTPuts("Octavo Systems EEPROM Writer v0.1\n", -1);

    // Write data to the EEPROM
    //   - Setup Address
    //   - Copy data to write
    //   - Transmit data
    //
    UARTPuts("Writing value to EEPROM\n", -1);
    tCount         = 0;
    dataToSlave[0] = E2PROM_ADDR_MSB;
    dataToSlave[1] = E2PROM_ADDR_LSB;
    for (i = 0; i < NUM_BYTES_TO_WRITE; i++) {dataToSlave[i + 2] = data[i];}
    print_data((unsigned char *)&dataToSlave[2], NUM_BYTES_TO_WRITE);

    SetupI2CTransmit(NUM_BYTES_TO_WRITE + 2);

    UARTPuts("Done writing.\n", -1);

    // Wait for EEPROM to finish writing
    j = 0;
    for (i = 0; i < 1000000; i++) {
      j++;
    }

    // Read data back from EEPROM
    //   - Use the address from the current dataToSlave buffer
    rCount = 0;
    tCount = 0;

    SetupReception(NUM_BYTES_TO_WRITE);

    // Print data from EEPROM
    UARTPuts("Value Written:\n", -1);
    print_data((unsigned char *)dataFromSlave, NUM_BYTES_TO_WRITE);

    UARTPuts("Done.\n", -1);

    while(1);
}
```

```
/*
 * \brief This function Initializes WDT, UART, and I2C
 *
 * \param  none
 *
 * \return none
*/
void setup_platform(void)
{
    // Disable WDT
    HWREG(SOC_WDT_1_REGS + WDT_WSPR) = 0xAAAAu;
    while(HWREG(SOC_WDT_1_REGS + WDT_WWPS) != 0x00);

    HWREG(SOC_WDT_1_REGS + WDT_WSPR) = 0x5555u;
    while(HWREG(SOC_WDT_1_REGS + WDT_WWPS) != 0x00);


    /* Enable the control module */
    HWREG(SOC_CM_WKUP_REGS + CM_WKUP_CONTROL_CLKCTRL) =
            CM_WKUP_CONTROL_CLKCTRL_MODULEMODE_ENABLE;

    /* UART Initialization */
    setup_UART();
    UARTStdioInit();

    /* I2C Initialization */
    setup_I2C();
}


/*
** Print data
*/
static void print_data(unsigned char *data, unsigned int size) {
    unsigned char ch[2];
    unsigned char temp;
    unsigned int  i;

    if ((BUFFER_SIZE / 4) < size) {
        // Print error message
        UARTPuts("Too much data requested \n", -1);
        return;
    }

    ch[1] = 0;

    // Print first character
    UARTPuts("[", 1);

    for (i = 0; i < size; i++) {
        // Collect the Most Significant Nibble of the data byte
        temp = ((data[i] & 0xF0) >> 4);

        if (temp < 10) {
            // UARTPrintf("%c", (temp + 0x30));
            ch[0] = temp + 0x30;
            UARTPuts(ch, 1);
        } else {
            // UARTPrintf("%c", (temp + 0x37));
            ch[0] = temp + 0x37;
            UARTPuts(ch, 1);
        }

        // Collect the Least Significant Nibble of the data byte
        temp = (data[i] & 0x0F);

        if (temp < 10) {
            // UARTPrintf("%c", (temp + 0x30));
            ch[0] = temp + 0x30;
```

```
            UARTPuts(ch, 1);
        } else {
            // UARTPrintf("%c", (temp + 0x37));
            ch[0] = temp + 0x37;
            UARTPuts(ch, 1);
        }

        if (i < (size - 1)) {
            UARTPuts(", ", 2);
        }
    }

    // Print final character
    UARTPuts("]\n", -1);
}


/*******************************************************************************
**               I2C FUNCTION DEFINITIONS
*******************************************************************************/

/*
 * \brief Configure I2C0 on which the PMIC is interfaced
 *
 * \param  none
 *
 * \return none
 */
void setup_I2C(void)
{
    /* Enable the clock for I2C0 */
    I2C0ModuleClkConfig();

    I2CPinMuxSetup(I2C_0);

    /* Put i2c in reset/disabled state */
    I2CMasterDisable(SOC_I2C_0_REGS);

    /* Disable auto Idle functionality */
    I2CAutoIdleDisable(SOC_I2C_0_REGS);

    /* Configure i2c bus speed to 100khz */
    I2CMasterInitExpClk(SOC_I2C_0_REGS, I2C_SYSTEM_CLOCK, I2C_INTERNAL_CLOCK,
                                                    I2C_OUTPUT_CLOCK);

    /* Set i2c slave address */
    I2CMasterSlaveAddrSet(SOC_I2C_0_REGS, I2C_SLAVE_ADDR);

    /* Bring I2C out of reset */
    I2CMasterEnable(SOC_I2C_0_REGS);

    while(!I2CSystemStatusGet(SOC_I2C_0_REGS));
}


/*
 * \brief Clear the status of all interrupts
 *
 * \param  none.
 *
 * \return none
 */
void CleanupInterrupts(void)
{
    I2CMasterIntClearEx(SOC_I2C_0_REGS,  I2C_INTERRUPT_FLAG_TO_CLR);
}
```

```
/*
 * \brief Transmits data over I2C0 bus
 *
 * \param  none
 *
 * \return none
 */
void SetupI2CTransmit(unsigned int dcount)
{
    I2CSetDataCount(SOC_I2C_0_REGS, dcount);

    CleanupInterrupts();

    I2CMasterControl(SOC_I2C_0_REGS, I2C_CFG_MST_TX);

    I2CMasterStart(SOC_I2C_0_REGS);

    while(I2CMasterBusBusy(SOC_I2C_0_REGS) == 0);

    while((I2C_INT_TRANSMIT_READY == (I2CMasterIntRawStatus(SOC_I2C_0_REGS)
                                 & I2C_INT_TRANSMIT_READY)) && dcount--)
    {
        I2CMasterDataPut(SOC_I2C_0_REGS, dataToSlave[tCount++]);

        I2CMasterIntClearEx(SOC_I2C_0_REGS, I2C_INT_TRANSMIT_READY);
    }

    I2CMasterStop(SOC_I2C_0_REGS);

    while(0 == (I2CMasterIntRawStatus(SOC_I2C_0_REGS) & I2C_INT_STOP_CONDITION));

    I2CMasterIntClearEx(SOC_I2C_0_REGS, I2C_INT_STOP_CONDITION);
}


/*
 * \brief Receives data over I2C0 bus
 *
 * \param  dcount - Number of bytes to receive.
 *
 * \return none
 */

void SetupReception(unsigned int dcount)
{
    unsigned int num_addr_bytes = 2;

    // Transmit Address Bytes
    I2CSetDataCount(SOC_I2C_0_REGS, num_addr_bytes);

    CleanupInterrupts();

    I2CMasterControl(SOC_I2C_0_REGS, I2C_CFG_MST_TX);

    I2CMasterStart(SOC_I2C_0_REGS);

    while(I2CMasterBusBusy(SOC_I2C_0_REGS) == 0);

    while((I2C_INT_TRANSMIT_READY == (I2CMasterIntRawStatus(SOC_I2C_0_REGS)
                                 & I2C_INT_TRANSMIT_READY)) && num_addr_bytes--)
    {
        I2CMasterDataPut(SOC_I2C_0_REGS, dataToSlave[tCount++]);

        I2CMasterIntClearEx(SOC_I2C_0_REGS, I2C_INT_TRANSMIT_READY);
    }

    I2CMasterStop(SOC_I2C_0_REGS);

    while(0 == (I2CMasterIntRawStatus(SOC_I2C_0_REGS) & I2C_INT_ADRR_READY_ACESS));
```

```
    // Receive Data Bytes
    I2CSetDataCount(SOC_I2C_0_REGS, dcount);

    CleanupInterrupts();

    I2CMasterControl(SOC_I2C_0_REGS, I2C_CFG_MST_RX);

    I2CMasterStart(SOC_I2C_0_REGS);

    /* Wait till the bus if free */
    while(I2CMasterBusBusy(SOC_I2C_0_REGS) == 0);

    /* Read the data from slave of dcount */
    while((dcount--))
    {
        while(0 == (I2CMasterIntRawStatus(SOC_I2C_0_REGS) & I2C_INT_RECV_READY));

        dataFromSlave[rCount++] = I2CMasterDataGet(SOC_I2C_0_REGS);

        I2CMasterIntClearEx(SOC_I2C_0_REGS, I2C_INT_RECV_READY);
    }

    I2CMasterStop(SOC_I2C_0_REGS);

    while(0 == (I2CMasterIntRawStatus(SOC_I2C_0_REGS) & I2C_INT_STOP_CONDITION));

    I2CMasterIntClearEx(SOC_I2C_0_REGS, I2C_INT_STOP_CONDITION);
}


/*******************************************************************************
**               UART FUNCTION DEFINITIONS
*******************************************************************************/

/*
 * \brief This function is used to initialize and configure UART Module.
 *
 * \param  none.
 *
 * \return none
*/

void setup_UART(void)
{
    volatile unsigned int regVal;

    /* Enable clock for UART0 */
    regVal = (HWREG(SOC_CM_WKUP_REGS + CM_WKUP_UART0_CLKCTRL) &
                    ~(CM_WKUP_UART0_CLKCTRL_MODULEMODE));

    regVal |= CM_WKUP_UART0_CLKCTRL_MODULEMODE_ENABLE;

    HWREG(SOC_CM_WKUP_REGS + CM_WKUP_UART0_CLKCTRL) = regVal;

    UARTStdioInit();
}
```

9 Appendix B: hsi2cEeprom.lds

```
/*
* Copyright (C) 2010 Texas Instruments Incorporated - http://www.ti.com/
*/
/*
*  Redistribution and use in source and binary forms, with or without
*  modification, are permitted provided that the following conditions
*  are met:
*
*    Redistributions of source code must retain the above copyright
*    notice, this list of conditions and the following disclaimer.
*
*    Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the
*    distribution.
*
*    Neither the name of Texas Instruments Incorporated nor the names of
*    its contributors may be used to endorse or promote products derived
*    from this software without specific prior written permission.
*
*  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
*  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
*  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
*  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
*  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
*  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
*  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
*  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
*  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
*  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
*  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
*
*/

/* ld script for StarterWare AM335x. */

/*
** The stack is kept at end of the image, and its size is 128 MB.
** The heap section is placed above the stack to support I/O
** operations using semihosting. The size of the section is 2KB.
*/

MEMORY
{
    IRAM_MEM :    o = 0x402F0400,  l = 0x1FBFF      /* 127k internal Memory */
}

OUTPUT_FORMAT("elf32-littlearm", "elf32-littlearm", "elf32-littlearm")
OUTPUT_ARCH(arm)

SECTIONS
{

        .rsthand :
        {
            . = ALIGN(4);
            *bl_init.o      (.text)
        } >IRAM_MEM

        .text :
        {
            . = ALIGN(4);
            *(.text*)
            *(.rodata*)
        } >IRAM_MEM

        .data :
```

```
        {
            . = ALIGN(4);
            *(.data*)
        } >IRAM_MEM

        .bss :
        {
            . = ALIGN(4);
            _bss_start = .;
            *(.bss*)
            *(COMMON)
            _bss_end = .;
        } >IRAM_MEM

        .heap :
        {
            . = ALIGN(4);
            __end__ = .;
            end = __end__;
            __HeapBase = __end__;
            *(.heap*)
            . = . + 0x800;
            __HeapLimit = .;
        } >IRAM_MEM

        .stack :
        {
            . = ALIGN(256);
            __StackLimit = . ;
            *(.stack*)
            . = . + 0x800;
            __StackTop = .;
        } >IRAM_MEM
        _stack = __StackTop;

}
```