Sitara Uniflash Flash Programming with U-Boot

Contents

Overview

Using U-Boot for NAND, NOR, and (Q)SPI Flash Programming Process

- Overview
- Preparing the Images to be Flashed
  - Concatenate Images to be Flashed into a Single File
- Build the Flasher Image
  - Build SPL (MLO) and U-Boot for Flashing
  - Create a Debrick Script
    - Debrick Script Modification
    - Creating debrick.scr from text source
- Prepare Files to be Transferred to the Host PC with UniFlash
- Flash a Board

Archived Versions

Overview

This information describes how U-Boot can be used to program flash memory on a board. This process is particularly helpful for first-time or production programming when there is no information in the flash. This information can also be useful to developers who create the process or need to flash new images frequently.

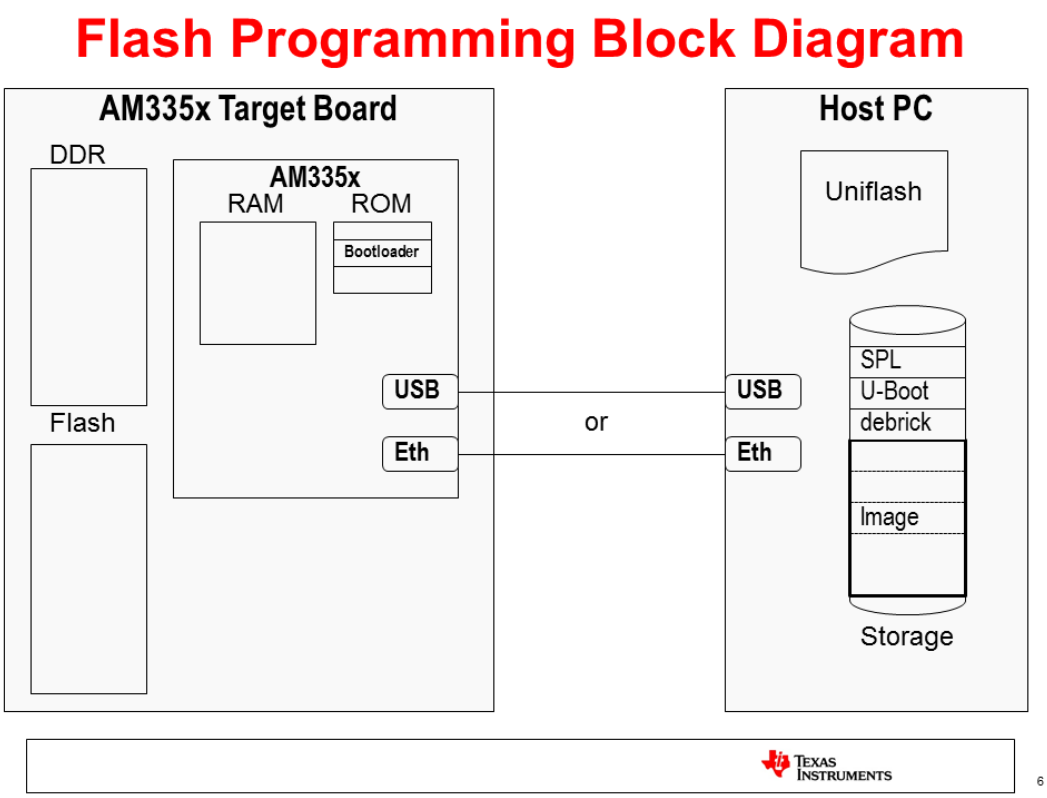
U-Boot is particularly useful when you want an easy, fast way to program non-managed flash memories like raw NAND, NOR, and (Q)SPI memories. Linux can also be used to perform many of these steps and may offer additional capabilities, depending on what needs to be done. Using Linux is discussed more here.

Using U-Boot for NAND, NOR, and (Q)SPI Flash Programming Process

The below information covers the process for NAND, NOR, and (Q)SPI based flash memories. The process for eMMC, which usually uses Linux, is a bit different and is documented here.

Overview

This process is focused on a target board that has never been flashed or programmed. Therefore, the only AM335x/AM437x code that can be executed is contained in the Internal ROM which can enable certain peripherals to download programs to execute. In order to program flash attached to an AM335x/AM437x, a dedicated U-Boot image that has been ported to the target board (enabling board specific features and capabilities like pin muxing) and built to run a special script that can be used for flash programming (debrick.scr). This U-Boot image is likely different than the U-Boot image that has been developed to run out of the flash once it has been programmed. However, the same board port code can likely be re-used and will not require much additional development, if any. Here is an block diagram of what we are looking at creating.



The SPL, U-Boot, and debrick script shown on the host PC represent the flasher image, which will be downloaded over Ethernet or USB by the AM335x ROM Bootloader at boot. This image will initialize the target board as necessary to do the flashing operation. Then it will download and execute the debrick script which will be written to download the Image shown on the host PC and place it in RAM. Using the image downloaded to RAM, the debrick script will program the flash memory. Most likely, this image will also contain a SPL and U-boot, as well as a Kernel and Filesystem that make up the Linux image to be executed on the board once it has been flashed.

In attempt to be as clear as possible, there are two separate program images that we are referring to:

1. The image to write the flash on the target board, which is composed of the SPL, U-Boot, and debrick files indicated. These will be pulled over by the bootloader in ROM when the target board is powered on (assuming the boot settings are set up to boot from USB or Ethernet).
2. The image to be written. This is shown as "Image" and is pulled over as one request to improve performance. Once on the target, it will be broken up and written to the appropriate places in flash as determined by the flasher program above (mainly by the debrick script). This image will also likely contain a SPL and U-Boot (like the flasher program, but different), as well as a Kernel (uImage or zImage) and Root Filesystem. This is the full Linux image that will execute out of flash once it has been written and will vary depending the needs of the target board.

In general, a Client/Server model is being used where the AM335x/AM437x based target board is the client and the host PC is the server. The server's role in this setup is very simple, to serve up files that the target or client requests. This creates a clear line of separation and allows the client to be as simple or complex as necessary, without changing what needs to happen on the host PC side.

Here is an overview of the process to create the flasher program and the image to be programmed:

- Building the program necessary to "flash" the board. This code is built from U-Boot, so it is really a simple adaptation of the code that has already been adapted to the board. This adaptation is best performed by the original code developers. Here are some basic assumptions:
  - The platform developer(s) has a working SPL, U-Boot, Linux Kernel and File System ready for flashing.
  - The platform developer has the Processor SDK Linux U-Boot ported to the desired platform.

Here is a summary of the steps to be followed by the platform developer(s).

- Develop the image to be programmed to the flash.
  - Create flash-image.out file: A utility is provided to concatenate all the images(SPL, U-Boot, Linux Kernel, Root File System, etc) to be flashed into one large image.
- Develop the flashing applications. This is a combination of U-Boot and debrick script that will be executed to do the flash programming.
  - Modify U-Boot to execute from either Ethernet or USB. This will probably involve bulding U-Boot with different make commands to enable different interfaces. **Make sure you can get to a U-Boot prompt using this code before moving forward. Use this prompt to help develop and validate the debrick script.**
  - Create Debrick Script: A template is provided along with instructions on how to modify and build the debrick script. This script is essentially the U-Boot commands needed to program the flash. A working U-Boot prompt, per above, is a great starting point to fine tune this process. This is also very useful in debugging the actual image that is being programmed.
- Zip images into a file to transfer to the Windows platform (and the point of programming, which is likely to be a production environment and different than the development location). This package will contain both the flasher and the image to be programmed.
- Set up a Windows machine with Uniflash to be the server at the desired location.
- Program the target board(s) using the image provided from the above steps and a Windows Host PC set up to serve these images as needed by the target board for flash programming. This task is designed to enable anyone with fairly basic PC skills to accomplish and is documented here.

Preparing the Images to be Flashed

The program developer builds a set of SPL and U-Boot to enable loading a U-Boot script called debrick.scr as part of its default environment through a specific make target for U-Boot. The debrick.scr allows the target board to self-flash. When executed in a U-Boot context the script will pull a single image that contains all the images to be flashed from the Windows machine to the target board over the connected interface and perform the flashing for each image to the target.

Concatenate Images to be Flashed into a Single File

The production image that needs to be flashed is usually made up of a few different pieces. It is better to bundle these pieces together into a single image that can be transferred from the host to the target in one transfer to minimize transer overhead. In a typical Sitara Processor system there are 5 pieces:

- SPL (Secondary Program Loader) that can be named MLO, u-boot-spl.bin, etc.
- U-Boot image which is usually name u-boot.img.
- Linux Kernel (zImage)
- Device Tree File
- Root Filesystem that is usually packaged into a filesytem lik UBIFS and named rootfs.ubi or simply zipped up into a tar.gz file.

> **NOTE**
>
> The file names can vary depending on how they were built.

There can be more images; the only dependency is how many images the target flasher program has been designed to support and the needs of the production system.

The user runs the Flash Cat utility (https://gforge.ti.com/gf/download/frsrelease/1303/7973/flash_cat_util.tar.gzv) on the assembled images to create a single image.

Example: **: ./flash_cat_util.out MLO-am335x-evm u-boot-am335x-evm.img zImage-am335x-evm.bin am335x-evm.dtb ubi.img**

There are two files created by the utility:

- flash-image.out: This is the concatenated image of all the images to be flashed.
- flash-image-data: This file contains information required for the Debrick Scrtipt.

> **NOTE**
>
> The number of images is not limited to 5 in this example. To add more images, keep adding the names to the command line when calling the utility. Make sure that the order matches the order that the restore flash u-boot debrick script uses.

Looking inside the results of the flash-data-image file, each file name has an offset into the flash-image.out along with the length of the image. These numbers will be used to fill out the debrick script.

```
Texas Instruments Flash Image Concatenation tool - Copyright 2016

Image Name, offset into DDR Image will be after transfer and Length of the Image
Place in the Debrick text file

Image MLO-am335x-evm Offset 0x0 Length 0x10f18
Image am335x-evm.dtb Offset 0x11000 Length 0x9bab
Image u-boot-am335x-evm.img Offset 0x1b000 Length 0x5b3b8
Image zImage-am335x-evm.bin Offset 0x76800 Length 0x3152f0
Image ubi.img Offset 0x38c000 Length 0x2ee0000
```

> **NOTE**
>
> The file names, offsets, and lengths are all dependent on the image being built. These are just examples.

To build the Flash-Cat Utility (a binary is supplied with the package).

```
gcc -W flash_cat_util.c -o flash_cat_util.out
```

Build the Flasher Image

Now that you have an image to flash in a format that it can be transferred from host to target easily and efficiently, it is time to build the flasher image that will actually download the image and burn it to flash. This flasher program will consist of a board ported SPL, U-Boot, and customized debrick script.

Build SPL (MLO) and U-Boot for Flashing

This process uses the U-Boot sources that have been board ported to the target board to build a custom U-Boot that will load the debrick.scr created in the above steps and flash the images provided.

- Patches will need to be applied to enable the necessary build commands.
- Download the patch set for your platform:

Patch Links

Platform	Patch Set
AM335x	Patches (https://gforge.ti.com/gf/download/frsrelease/1303/7974/u-boot-restore-flash-patches-psdkl-2_0_0_0.tar.gz)

- Extract the patches to a folder
- Apply the patches to the u-boot sources

  git am /path/to/patches

- Use the appropriate command to build the images from the u-boot source tree (assumes TI Processor SDK Linux 2.0.x.x). The build target will changed depending on whether you are booting from Ethernet or USB, and what type of flash you will be programming:

Build Targets

Boot Source	Memory Type	Build Target
Ethernet	NAND, SPI	am335x_evm_restore_flash
USB	NAND, SPI	am335x_evm_restore_flash_usbspl

And use the below build command for the build:

make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- O=flash-restore *build target from table above*

For example, to build a flasher for USB flashing of NAND or SPI Flash:

make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- O=flash-restore *am335x_evm_restore_flash_usbspl*

This command makes a directory called flash-restore to contain the flasher image separate from the image to be flashed.

- To keep the flasher images unique from the images to be flashed, go into the flash-restore directory and copy these images:
  - Copy u-boot.img to u-boot-restore.img
  - In the directory flash-restore/spl copy u-boot-spl.bin to u-boot-spl-restore.bin.

> **NOTE**
>
> This assumes that the developer has kept the SDK build targets for their platform. If they were changed, this change will need to be accounted for.

- **Boot the board to be flashed using the desired interface and the newly created U-Boot (SPL and U-Boot). Verify that you get to a U-Boot prompt and that the flash commands that you need for programming are available as expected. Use this prompt to develop and fine tune the debrick script below, and validate the image to be flashed (if necessary).**

Create a Debrick Script

This script will be loaded by SPL/U-Boot and will control the actual transfer and flashing of the images. This script needs to be customized to the specific images to be programmed and the type of memory to be programmed.

- Linux developer creates debrick.txt script using the template provided
- Currently only a NAND version is supplied. Examples for NOR and SPI will be added soon.

Debrick Script Modification

The debrick script will need to be adapted to each particlular situation. As you will see, the debrick script is the engine that actually transfers the files via TFTP from the host to the target and programs them into the flash memory. A template file has been provided to serve as an example. Here are some details to help guide modifying the debrick script to meet your particular needs.

1. The template file provided is called debrick_nand (https://gforge.ti.com/gf/download/frsrelease/1303/7975/debrick.txt/open)
2. Add the names of the images to be flashed into the device, search for this entry “Name of Images” and enter the names that were developed with the product. The names are examples only and depend on what you put in the image itself.

```
# Name of Images to be flashed, MLO image will have 3 backups
setenv Image1_Name MLO-am335x-evm
setenv Image2_Name device-tree-am335x-evm.dtb
setenv Image3_Name u-boot-am335x-evm.img
setenv Image4_Name zImage-am335x-evm.bin
setenv Image5_Name ubi.img
```

3. Add the offsets used into the NAND for each image. Search for “Image offsets”. These offsets have to match what is in the Linux device tree (am335x_evm.dtb for example) for the NAND partitions of the target board. They rely on the setting in the U-Boot configuration (am335x_evm.h for example).

```
# Image offsets into NAND - these are defined in the EVM configuration header
# and should align with the Linux kernel.
setenv Image1_NAND_Offset NAND.SPL
setenv Image2_NAND_Offset NAND.SPL.backup1
setenv Image3_NAND_Offset NAND.SPL.backup2
setenv Image4_NAND_Offset NAND.SPL.backup3
setenv Image5_NAND_Offset NAND.u-boot-spl-os
setenv Image6_NAND_Offset NAND.u-boot
setenv Image7_NAND_Offset NAND.kernel
setenv Image8_NAND_Offset NAND.file-system
```

4. Set the Image DDR locations, these are calculated using the data from flash-image-data file. Take the offsets listed in the file and add them to the load address that u-boot uses.

```
# Image offsets in ddr
# This has to be calculated from the load address
# this assumes the load address is 0x82000000
# Use the load address and the offset provided from the flash-cat utility to calculate
setenv Image1_DDR_ADDR 0x82000000
setenv Image2_DDR_ADDR 0x82011000
setenv Image3_DDR_ADDR 0x8201B000
setenv Image4_DDR_ADDR 0x82076800
setenv Image5_DDR_ADDR 0x8238C000
```

5. Set the Image length for each image, again using the data from the flash-image-data file, here the lengths have been converted to hex.

```
# These numbers come from the flash-cat utility
setenv Image1_Length 0x10F18
setenv Image2_Length 0x9BAB
setenv Image3_Length 0x5B3B8
setenv Image4_Length 0x3152F0
setenv Image5_Length 0x2EE0000
```

6. For each Image that is to be programmed, a section like this needs to be added, please note that the example debrick.txt has 8 images already setup (3 redundant copies of MLO).

```
# # ------------------------------------------------------------------
# # Add a section for each Image to be flashed
# # For each Image to be flashed you will need to set the
# # following environments:
# # - Source Address in the Single Image
# # - Storage Offset into the Storage device
# # - Length of the image
# # - Image Name
# # ------------------------------------------------------------------

# # Flash Image MLO
# #------------------------------------------------------------------
setenv source_addr ${Image1_DDR_ADDR}
setenv storage_offset ${Image1_NAND_Offset}
setenv image_length ${Image1_Length}
setenv ImageName ${Image1_Name}
run FlashImage
# #------------------------------------------------------------------
```

> **NOTE**
>
> The debrick.scr uses tftpget to “get” a file at the end of several steps in the process. These files are not expected to actually exist on the server. This is a simple way for the target board to indicate progress to Uniflash.

Creating debrick.scr from text source

Once it has been modified to meet your needs, the debrick.txt file needs to be converted into a .scr file with the below command:

mkimage -A arm -O U-Boot -C none -T script -d debrick.txt debrick.scr

Prepare Files to be Transferred to the Host PC with UniFlash

Provide the below files to the Flash Programming process. These will need to be placed in the c:\AM335x_Flashtool\images directory (by default) or wherever Uniflash is configured to look for them.

- flash-image.out – concatenated target files to be flashed
- debrick.scr – customized script to burn target files to flash
- SPL (ex. u-boot-spl-restore.bin) - flash restore SPL that loads U-Boot
- U-Boot (ex. u-boot-restore.img) – flash restore U-Boot that loads debrick.scr

Flash a Board

A host computer configured to serve as a DHCP/BOOTP and TFTP server is necessary to flash a board. This can either be a Linux computer configured with standard tools or a Windows computer with Uniflash.

Once the files have been developed and transferred to the Host PC that will play the role of the server, you are ready to use either standard tools or Uniflash to program a board. The process with Uniflash is covered in detail in the Sitara Uniflash Quick Start Guide.

Archived Versions

- ***Sitara Linux SDK 06.00.00.00*** (http://processors.wiki.ti.com/index.php/?title=Sitara_Uniflash_Flash_Programming_with_U-Boot&oldid=205792)

{{ 1. switchcategory:MultiCore= • For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum • For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum Please post only comments related to the article **Sitara Uniflash Flash Programming with U-Boot** here.	Keystone= • For technical support on MultiCore devices, please post your questions in the C6000 MultiCore Forum • For questions related to the BIOS MultiCore SDK (MCSDK), please use the BIOS Forum Please post only comments related to the article **Sitara Uniflash Flash Programming with U-Boot** here.	C2000=*For technical support on the C2000 please post your questions on The C2000 Forum. Please post only comments about the article **Sitara Uniflash Flash Programming with U-Boot** here.*	DaVinci=*For technical support on DaVincoplease post your questions on The DaVinci Forum. Please post only comments about the article **Sitara Uniflash Flash Programming with U-Boot** here.*	MSP430=*For technical support on MSP430 please post your questions on The MSP430 Forum. Please post only comments about the article **Sitara Uniflash Flash Programming with U-Boot** here.*	OMAP35x=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **Sitara Uniflash Flash Programming with U-Boot** here.*	OMAPL1=*For technical support on OMAP please post your questions on The OMAP Forum. Please post only comments about the article **Sitara Uniflash Flash Programming with U-Boot** here.*	MAVRK=*For technical support on MAVRK please post your questions on The MAVRK Toolbox Forum. Please post only comments about the article **Sitara Uniflash Flash Programming with U-Boot** here.*	*For tec please questio http://e Please comme article **S Flash P with U-* }}

Clocks & Timers	Logic	Digital Signal Processors (DSP)
Data Converters	Power Management	Microcontrollers (MCU)
		OMAP Applications Processors