Sitara Uniflash Flash Programming with Linux

Contents

Overview

eMMC Programming Process
- Overview
- Modify the Provided Flasher Script
- Build the Flasher Image
  - Build SPL (also called MLO) and U-Boot for Flashing
  - Download and Install RAM Filesystem
  - Build Linux Kernel for Flashing
- Prepare Files to be Transferred to the Host PC which has UniFlash
  - Prepare the Flasher Image
  - Prepare the Production Image
  - Use Linux Host to Validate and Fine Tune
- Use Uniflash to Flash a Board
- Optimization

Archived Versions

Overview

This information describes how a small Linux system 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.

Linux is particularly useful when you need to partition and format a memory, like a large eMMC device. Or, when the flexibility and capability of Linux is needed. U-boot can also be used to perform many of these steps and may be faster and easier, depending on what needs to be done. Using U-Boot is discussed more here.

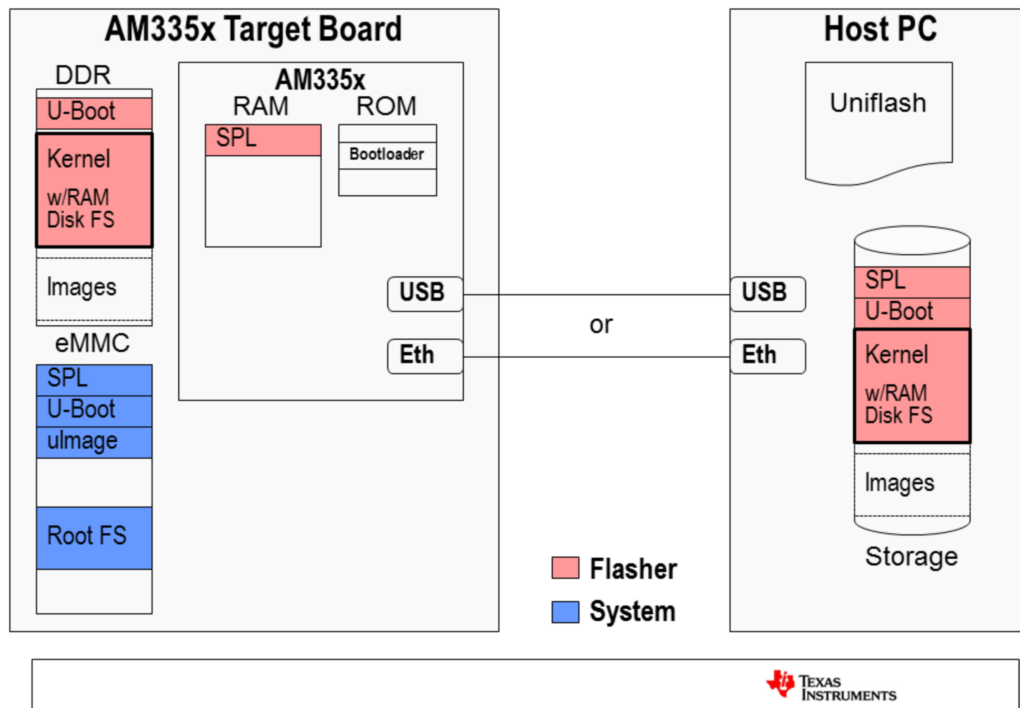
eMMC Programming Process

The above steps lay out a process that works well for NAND, NOR, and (Q)SPI type flash memories. eMMC is another option that is growing rapidly in popularity. Since it is a managed NAND, much like SD/MMC cards, we can treat it as such and take advantage of some of the differences. One of these differences is that we can partition and format it like a hard disk, and only copy over the files that are needed at that time. This is different than NAND for example, where we would have to write the entire NAND partition or space, even if a lot of it is blank data. For large memories where a lot of the space may be unused at the begining, a lot of programming time can be saved. On a production line where time is money (quite literally), these savings can be substantial. This process could also be applied to other environments like NAND, and have some advantages as well as the entire Linux OS is available to the flasher program.

A specific example of this process applied to the Beaglebone Black can be found 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 Linux 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. This Linux image is likely different than the Linux 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 Kernel shown on the host PC (and shown in red) represent the flasher image, which will be downloaded over Ethernet or USB by the AM335x/AM437x ROM Bootloader at boot. This image will initialize the target board as necessary to do the flashing operation. It will be set up to fetch a script, flasher.sh, via TFTP from the host PC. This script will partition and format the eMMC, download the necessary files from the host PC via TFTP (images in the above diagram), and write these files to the eMMC (the blue system files above).

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 (flasher program), which is composed of the SPL, U-Boot, and Kernel, a root filesystem set up as an initramfs RAM disk. 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). An initialization script inside the root FS will begin execution and request the flasher.sh script from the server. These files are built to boot from either USB or ethernet, which is likely very different than the system they will flash (which is probably built to boot from eMMC in this case).
2. The image to be written which will be copied from the host PC. Once on the target, these files will be written to the appropriate places in flash as determined by the flasher program above (mainly by the flasher script). This image will also likely contain a SPL, U-Boot, Kernel (uImage or zImage) and Root Filesystem as it is a Linux system similar to the flasher program. 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 (server).

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 and Linux, so it is really a simple adaptation of the code that has already been adapted to the board for the production image. 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 Sitara Linux SDK U-Boot and Kernel ported to the desired platform. All patches and code changes referred to below will assume this as the base.

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

- Gather the files together that make up the production image.
- Modify a Flasher Script: A template is provided that should meet most of your needs. However, feel free to adapt this as needed.
- Apply a U-Boot patch to enable TFTP boot by default.
- Build SPL/U-Boot with a specific make target depending on if the Ethernet or USB interface is used. This is different than the SPL/U-Boot being programmed into Flash. The latter is probably set up to boot from eMMC. The flasher image needs to be built to boot from the desired interface, USB or ethernet.
- Build a Linux kernel that includes the flasher FS that contains an init script for fetching the flasher script and the other utilities that it will need.
- **It would be good to test all of this from Linux to fine tune the process, if needed, before transferring it to Windows. It is much easier and efficient to debug and fine tune this process using a Linux host.**
- 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).
- Set up a Windows machine with Uniflash to be the server at the desired location and 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.

Modify the Provided Flasher Script

When the flasher image boots up, an init script, fetcher.sh, will request a flasher script, flasher.sh, from the server and execute it. The flasher script will need to partition and format the eMMC (or other flash device), TFTP the images from the Host PC, and copy the production files to the necessary locations in the flash storage.

- An example flasher script, flasher.sh (https://gforge.ti.com/gf/download/frsrelease/1303/7969/flasher.tar.gz), is provided. This script:
  - Sets up the appropriate network device, ethernet or USB.
  - Looks for a valid eMMC device.
  - Partitions that device using SFdisk.
  - Formats the partitions using mkfs.
  - Mounts the partitions.
  - TFTPs the necessary files from the Host PC to RAM.
  - Copies the necessary files from the Flasher filesystem (in RAM) to the newly created partitions on the eMMC.
- Edit this script as needed to suit your needs. Here are some things to consider:
  - How many and how large do you need the partitions to be? The example uses 2 partitions to be consistent with the SDK.
  - What formats do you want to use for the partitions? The example uses FAT for the boot partition and ext3 for the rootfs partition again the defaults used by the SDK.
  - Which files for the production image do you need to copy and where?
  - How large are the images? When the flasher.sh TFTPs the images, it is limited to the RAM on the device to how much it can store at once. If the images are larger than this, they will have to be broken into smaller chunks for transfer and programming.

Build the Flasher Image

Now that you have an production system 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, Kernel, customized flasher filesystem, and customized flasher script to meet the needs of your production system.

Build SPL (also called MLO) and U-Boot for Flashing

This process uses the U-Boot sources that have been board ported to the custom target board. More information about building U-Boot can be found here (http://processors.wiki.ti.com/index.php/Linux_Core_U-Boot_User's_Guide)

- A patch need to be applied to the U-Boot source tree for the given platform, AM335x or AM437x. This patch changes the default boot behavior to TFTP over either USB or ethernet.
  - AM335x - [U-Boot Patch to Add TFTP Boot Configurations (https://gforge.ti.com/gf/download/frsrelease/1303/7971/am335x_uboot_flasher_patches_psdkl_2_0_0_0.tar.gz)]
  - AM437x - [U-Boot Patch to Add TFTP Boot Configurations (https://gforge.ti.com/gf/download/frsrelease/1278/7772/%30%30%30%31%2d%41%4d%34%33%37%78%2d%55%2d%42%6f%6f%74%2d%41%64%64%2d%54%46%54%50%2d%62%6f%6f%74%2d%63%6f%6d%6d%61%6e%64%2d%63%6f%6e%66%69%67%75%72%61%74%69%6f%6e%2e%70%61%74%63%68)]
- Use the below commands to build the images from the u-boot source tree (assumes TI Linux Processor SDK 1.0):
  - make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- O=flasher <build target from below table>
  - make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- O=flasher

Build Target Configurations

Platform	Interface	Build Target
AM335x	Ethernet	am335x_evm_tftpboot_config
AM335x	USB	am335x_evm_usbspl_tftpboot_config
AM437x	Ethernet	am43xx_evm_tftpboot_config
AM437x	USB	am43xx_evm_usbspl_tftpboot_config

- This command makes a directory called flasher to contain the flasher image separate from the image to be flashed.
  - To keep the images unique from the images to be flashed, go into the flasher directory and copy these images:
    - Copy u-boot.img to u-boot-restore.img.
    - In the directory flasher/spl copy u-boot-spl.bin to u-boot-spl-restore.bin.

Note: This assumes that the developer has kept the build targets for their platform.

Download and Install RAM Filesystem

A very simple, small filesystem is provided to use as an initramfs within the kernel.

- Download the tiny flasher filesystem (https://gforge.ti.com/gf/download/frsrelease/1303/7970/am335x_tiny_filesystem.tar.gz).
- Untar it to a directory of your choice. The directory will be fed to the Linux kernel in the following steps.

> **NOTE**
>
> This filesystem will automatically attempt to download and execute flasher.sh from the host via TFTP. This is done by running /etc/init.d/fetcher.sh as an init script. Feel free to examine this file and make sure it meets the needs of your setup.

Build Linux Kernel for Flashing

We need to build a kernel that contains the flasher filesystem (including the production image files and init script) as an initramfs RAM disk. This change is mainly to kernel configuration. This is a short overview as it assumes some experience with the kernel gained working on the production image. If you'd like to see more information about building the Linux kernel inside the SDK, it can be found here (http://processors.wiki.ti.com/index.php/Linux_Kernel_Users_Guide).

- Set up your path:
  export PATH="<sdk install dir>/linux-devkit/sysroots/i686-arago-linux/usr/bin:$PATH"
  Where <sdk install dir> should be replaced with the directory where the SDK was installed.

> **NOTE**
>
> The next step will delete any saved .config file in the kernel tree as well as the generated object files. If you have done a previous configuration and do not wish to lose your configuration file you should save a copy of the configuration file before proceeding.

- Clean the kernel

  make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- mrproper

- Configure the kernel:
  - It is often easiest to start with a base default configuration and then customize it for you use case if needed. In the Linux kernel a command of the form:

    make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- <config>

    will look in the arch/arm/configs directory for the configuration file (<config> in the line above) use that file to set default configuration options.

  To build the SDK configuration for the AM335x/AM437x the command would be:

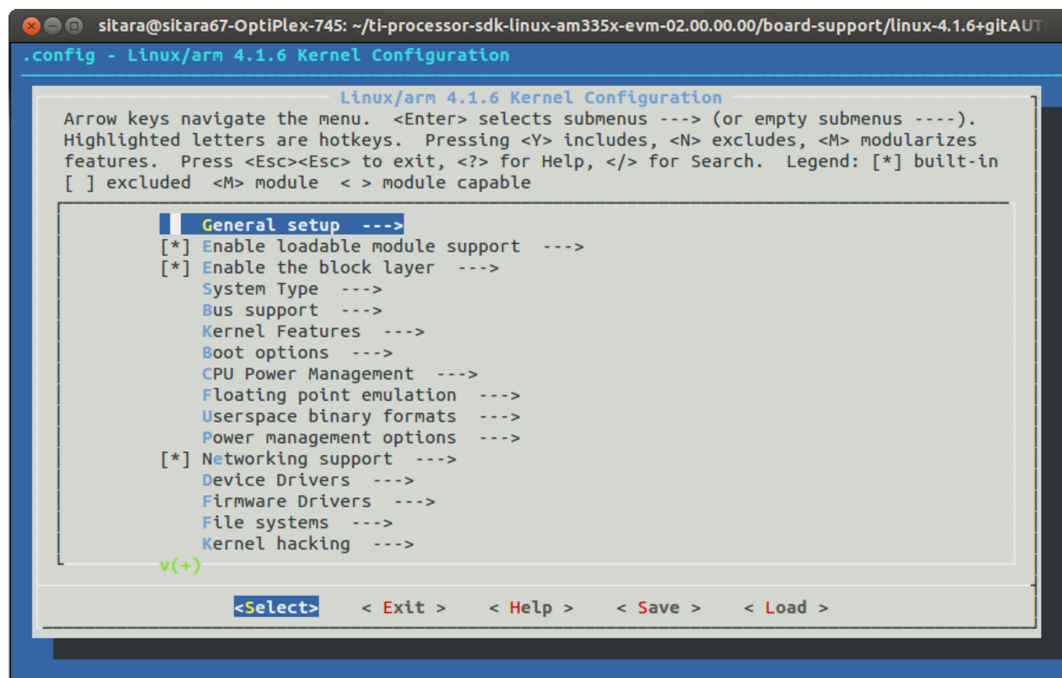
  make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- tisdk_am335x-evm_defconfig

  This should be fine for the flasher build. If you've developed your own config, you might prefer to use it.

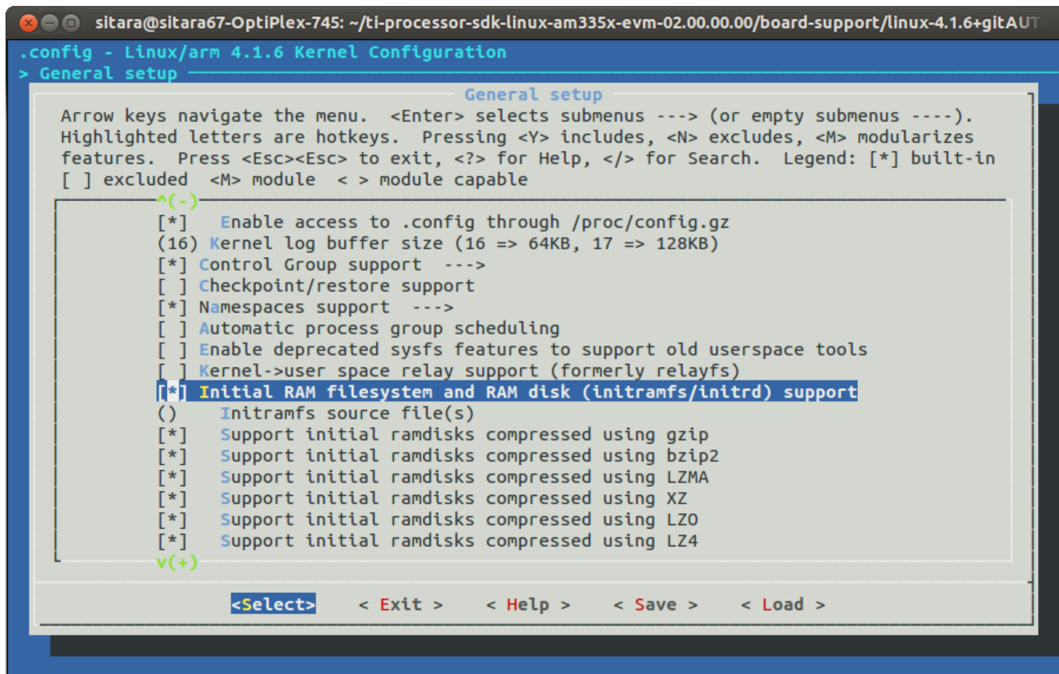
- Customize the configuration to build the initramfs RAM Disk that contains the root FS. Use your favorite method to edit the configuration. We'll use menuconfig here:

  make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- menuconfig

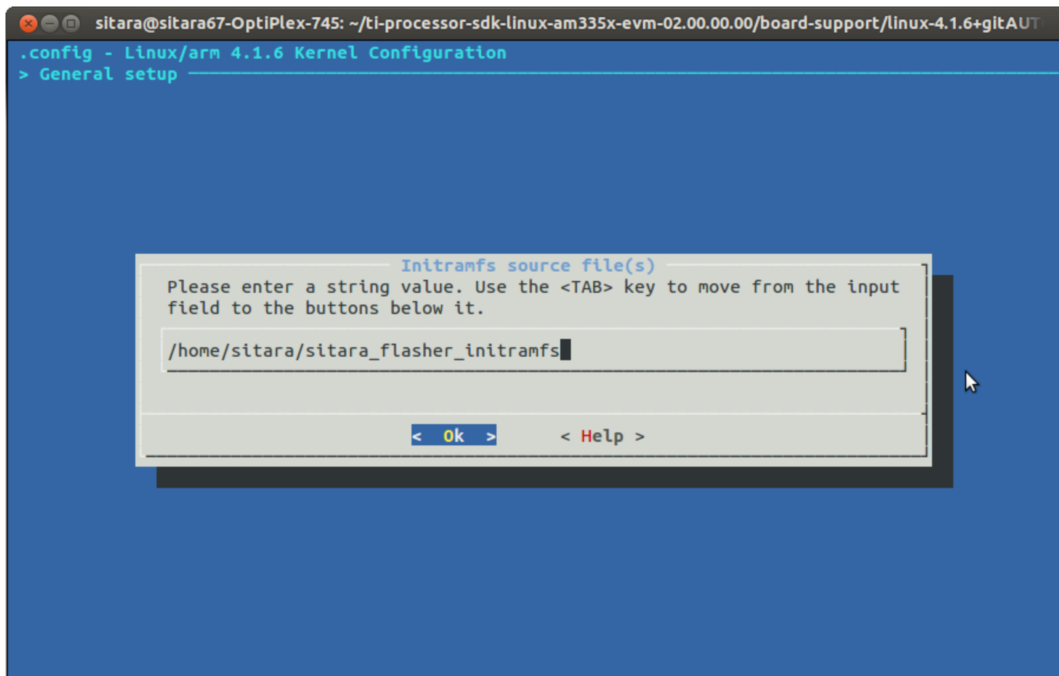
  - You should see something like this:



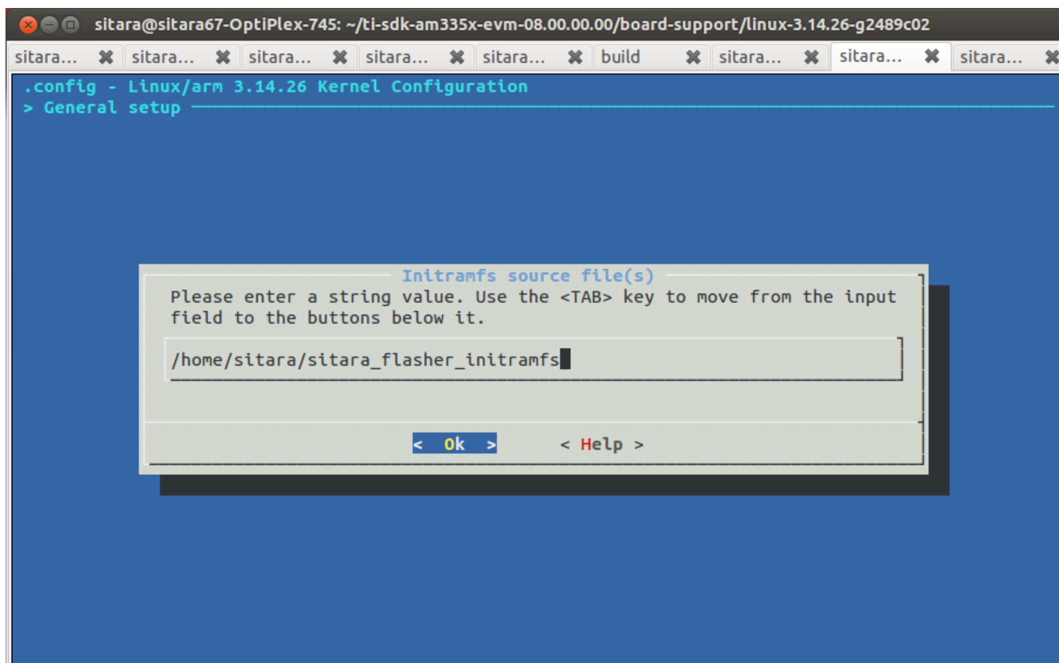
  - Hit **Enter** to go into **General Setup**.
  - Scroll down to **Initial RAM Filesystem and RAM disk (initramfs/initrd) support** and use the **spacebar** to make sure that it is enabled.



- Move down to the next line, **Initramfs Source Files**.

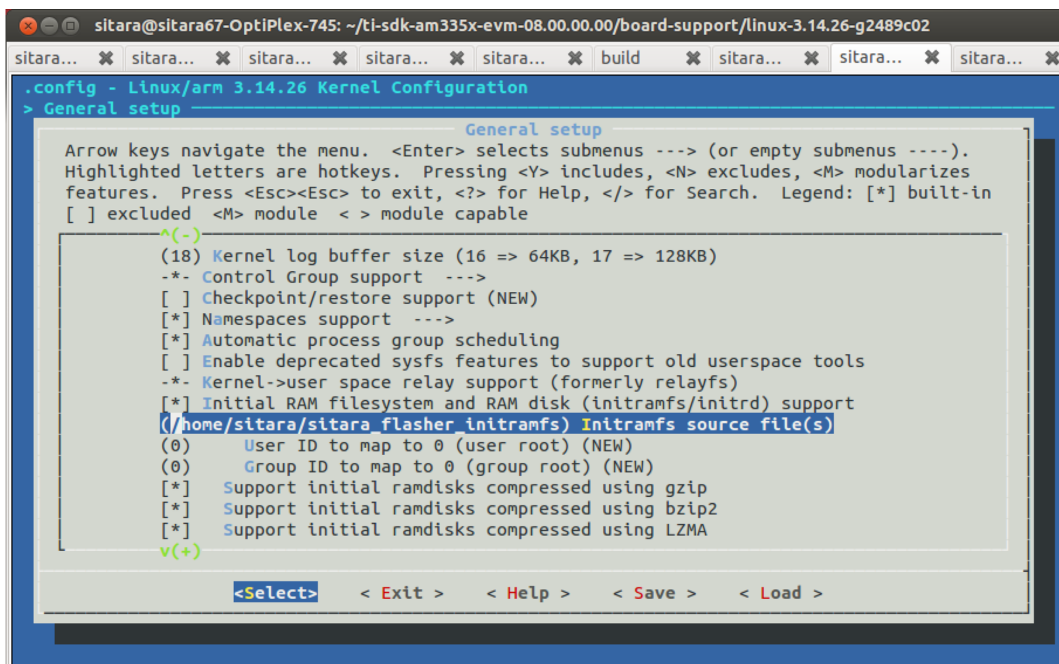


- Press **Enter** to set it:



> **NOTE**
>
> The path used should be the top directory of the filesystem that you created which contains the production images and the init script.

- Select **Ok** to save the path and you should see something like this:



- Select **Exit** to leave menuconfig and save the new configuration.

- If using USB RNDIS to boot the target board, the appropriate USB support needs to be built into the kernel. Refer to the Beaglebone Black example for instructions on how to build this support into the kernel.
- Finally, build the kernel :

  make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf-

Prepare Files to be Transferred to the Host PC which has 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.

Prepare the Flasher Image

- SPL (ex. u-boot-spl-restore.bin) - SPL that loads U-Boot
- U-Boot (ex. u-boot-restore.img) – U-Boot that loads the Linux Kernel from the appropriate interface

- Linux Kernel containing initramfs RAM disk
- A Device Tree DTB for the board, to be passed to the kernel from U-Boot
- flasher.sh (https://gforge.ti.com/gf/download/frsrelease/1303/7969/flasher.tar.gz) - Script to copy production image to eMMC (or other flash device), fetched by init script in flasher FS

Prepare the Production Image

The production image that needs to be flashed is usually made up of a few different pieces. These pieces need to be placed on the Host PC in the directory used by the Uniflash program. We recommend creating a tarball that corresponds to the partitions to be programmed. For example:

- A file for the boot partition containing:
  - SPL (Secondary Program Loader) that can be named MLO, u-boot-spl.bin, etc.
  - U-Boot image which is usually name u-boot.img.
  - Kernal image typically named zImage.
  - A DTB file for the board.
- Root Filesystem that is usually packaged into a filesytem which is simply zipped up into a tar.gz file.

> **NOTE**
>
> The file names can vary depending on how they were built, and should correlate to the flasher init script.

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.

Use Linux Host to Validate and Fine Tune

It is much easier to debug and fine tune this process, particularly for the U-Boot and Linux images, on a Linux host. This page (http://processors.wiki.ti.com/index.php/Ubuntu_12.04_Set_Up_to_Network_Boot_an_AM335x_Based_Platform) provides useful information to help with setting up a Linux host for this procedure. It is also a more efficient way to do U-Boot/Kernel development. Once the images have been validated to work well using a Linux host, it should be much easier to get everything to work with Windows.

Use Uniflash to Flash a Board

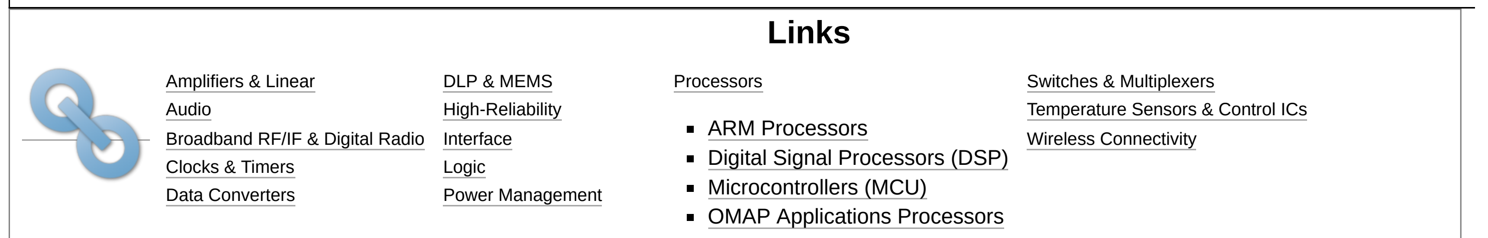
Once the files have been developed and transferred to the Windows PC that will play the role of the server, you are ready to use Uniflash to program a board. This process is covered in detail in the Sitara Uniflash Quick Start Guide.

Optimization

Archived Versions

- ***Sitara Linux SDK 08.00.00.00*** (http://processors.wiki.ti.com/index.php/?title=Sitara_Linux_AM335x_Flash_Programming_Linux_Development&oldid=199727)

{{ 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 Linux** 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 Linux** 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 Linux** 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 Linux** 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 Linux** 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 Linux** 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 Linux** 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 Linux** here.*	*For tec please questio http://e. Please comme article **S Flash F with Li*** }}



Retrieved from "https://processors.wiki.ti.com/index.php?title=Sitara_Uniflash_Flash_Programming_with_Linux&oldid=214152"

This page was last edited on 31 March 2016, at 10:49.

Content is available under Creative Commons Attribution-ShareAlike unless otherwise noted.