XAPP1146 (v1.0) May 5, 2010

Embedded Platform Software and Hardware In-the-Field Upgrade Using Linux

Author: Brian Hill

Summary

This application note describes an in-the-field upgrade of the Spartan®-6 FPGA bitstream, Linux kernel, and loader flash images, using the presently running Linux kernel. Upgrade files are obtained from a CompactFlash storage device or over the network from an FTP server.

Included Designs

Included with this application note is one reference design built for the Xilinx® SP605 Rev C board. The reference design is available in the following ZIP file available at:

https://secure.xilinx.com/webreg/clickthrough.do?cid=144349

Introduction

New features and bug fixes often necessitate upgrading flash images to replace the existing FPGA bitstream, bootloader, Linux kernel, or file system. This presents a challenge to provide a convenient mechanism for end users to perform this task. This application note provides a reference design and an example methodology to perform an in-the-field flash upgrade. New images can be retrieved from a CompactFlash device or from a network server. The running Linux image performs the flash upgrade.

Target Audience

This application note best serves users who are already adept at building and using Linux.

Hardware and Software Requirements

The hardware and software requirements for this reference design are:

- Xilinx SP605 Rev C board
- RS-232 serial communication utility (HyperTerminal)
- Xilinx Platform Studio v11.4
- Xilinx ISE® Design Suite v11.4
- Xilinx Open-Source Linux
- Suitable MicroBlaze® processor toolchain and Linux root file system
- (optional) GIT revision control software
- Ethernet cable
- FTP server

Reference Design Specifics

The supplied MicroBlaze processor reference design is configured to boot from Block RAM. The design also contains a DDR3 memory controller, IIC master, interrupt controller, tri-mode Ethernet MAC, 16550 UART, GPIO, and SPI master (SPI flash) IP cores.

Address Map

Table 1: **Reference Design Address Map**

Peripheral	Instance	Base Address	High Address
lmb_bram_if_cntlr	LocalMemory_Cntlr_D/I	0x00000000	0x00001FFF
mpmc	DDR3_SDRAM	0x50000000	0x57FFFFFF
xps_ll_temac	Soft_TEMAC	0x81000000	0x8100FFFF
xps_iic	IIC_EEPROM	0x81600000	0x8160FFFF
xps_intc	Interrupt_Cntlr	0x81800000	0x8180FFFF
xps_spi	SPI_FLASH	0x83400000	0x8340FFFF
xps_sysace	SysACE_CompactFlash	0x83600000	0x8360FFFF
xps_timer	Dual_Timer_Counter	0x83C00000	0x83C0FFFF
xps_uart16550	RS232_Uart_1	0x84000000	0x8400FFFF
mdm	Debug_Module	0x84400000	0x8440FFFF
mpmc	DDR3_SDRAM (SDMA)	0x84600000	0x8460FFFF

Support Files

The reference design includes the files described in Table 2 that support this application note.

Table 2: **Reference Design File Descriptions**

File Name	Description
hardware/	The EDK system for this application note.
ready_for_download/	
download.bit	FPGA bitstream.
xapp1146.cmd	Instructions for iMPACT (Xilinx software tool).
xapp1146.opt	Commands for XMD.
simpleImage.xilinx-xapp1146	Bootable Linux image.
linux/	
dotconfig	Linux kernel configuration.
initramfs_minimal.cpio.gz	Linux ramdisk image.
xilinx-xapp1146.dts	Device tree hardware description.
scripts/	
build_rom.pl	Generates a flash image suitable for use with the enclosed loader application.
mk_download.bin.sh	Converts download.bit to a file suitable for programming into flash with Linux.
upgrade.sh	Script which performs a flash upgrade.
upgrade-image/	
manifest	Upgrade description file.

Table 2: **Reference Design File Descriptions *(Cont'd)***

File Name	Description
upgrade.tgz	Upgrade images.
software/	
zlib-1.2.3/	Compression library (source and libz.a).
SDK_Hardware_Export/	Hardware system exported to SDK.
SDK_Projects/	
standalone_0/	Stand-alone BSP used by loader1 and loader2.
loader1/	Simple stage 1 boot loader.
loader2/	
loader2.c	Simple stage 2 boot loader.
loader2.ld	Manually modified linker script. The loader .text is linked to an offset past the start of DDR.
compiledate.c	Automatically generated file with a date string.
datemaker.sh	Script that generates compiledate.c for each build.
spi_flash.c	Read from W25Q64BV flash.
spi_flash.h	

Executing the Reference Design

Using HyperTerminal or a similar serial communications utility, map the operation of the utility to the physical COM port to be used. Then connect the UART of the board to this COM port. In HyperTerminal, set Bits per Second to **9600**, Data Bits to **8**, Parity to **None**, and Flow Control to **None**.

Executing the Reference System Using the Pre-Built Bitstream and the Compiled Software Application

To execute the system using files in the ready_for_download/ directory in the project root directory, follow these steps:

1. Change directories to the ready_for_download directory.
2. Use iMPACT to download the bitstream by using the following command:

```
impact -batch xapp1146.cmd
```

3. Invoke XMD and connect to the processor using the following command:

```
xmd -opt xapp1146.opt
```

4. Download and run the Linux executable using the following commands:

```
dow simpleImage.xilinx-xapp1146
run
```

5. Proceed to the Programming the Flash with Linux section, using the upgrade files provided in the ready_for_download/upgrade-image/ area.

Executing the Reference Design from XPS for Hardware

To execute the system for hardware using XPS, follow these steps:

1. Open system.xmp in XPS.
2. Select **Hardware→Generate Bitstream** to generate a bitstream for the system.
3. Select **Device Configuration→Download Bitstream** to download the bitstream.
4. Invoke XMD and connect to the processor using the following command:

```
xmd -opt xapp1146.opt
```

5. Download and run the Linux executable using the following commands:

```
dow simpleImage.xilinx-xapp1146
run
```

6. Proceed to the Programming the Flash with Linux section, using the upgrade files provided in the `ready_for_download/upgrade-image/` area.

Obtaining the Software

The user must obtain source code for the Linux kernel and the Linux kernel BSP generator to complete the tasks described in this application note. These are available on the Xilinx public GIT server and access portal, http://git.xilinx.com. GIT is a distributed revision control system. Installation and usage of GIT are beyond the scope of this application note; refer to the Xilinx application note XAPP1107, *Getting Started Using Git* for additional information.

Obtaining the Software with GIT

Users who do not have GIT installed or who choose not to use GIT should proceed to the Obtaining a Snapshot of the Software without GIT section.

Users who already have GIT properly installed can obtain the latest versions of the required software with the following commands:

1. Obtain the latest Linux 2.6 kernel

```
$ mkdir <project area>
$ cd <project area>
$ git clone git://git.xilinx.com/linux-2.6-xlnx.git
```

(OPTIONAL) Revert to the version used with this application note. This version has been demonstrated to work as described in this document without modification. Perform after cloning the tree.

```
$ cd linux-2.6-xlnx
$ git checkout 6e95b504
```

2. Obtain the latest device tree generator

```
$ cd <project area>
$ git clone git://git.xilinx.com/device-tree.git
```

(OPTIONAL) Revert to the version used with this application note. This version has been demonstrated to work as described in this document without modification. Perform after cloning the tree.

```
$ cd device-tree
$ git checkout a2444d9a
```

Obtaining a Snapshot of the Software without GIT

A snapshot of the source tree can be obtained from git.xilinx.com as a compressed tar file.

The user can navigate to the desired revisions used to create this application note via the following links:

http://git.xilinx.com/cgi-bin/gitweb.cgi?p=device-tree.git;a=summary

http://git.xilinx.com/cgi-bin/gitweb.cgi?p=linux-2.6-xlnx.git;a=summary

Obtaining a Toolchain Compiler

To build any of the software used in this application note, the user requires an appropriate MicroBlaze processor toolchain (compiler, linker, etc.). Linux also requires a Root File System. If the user does not already have these resources available, consult http://xilinx.wikidot.com/. Toolchain installation is beyond the scope of this application note.

Flash Organization

The onboard W25Q64BV SPI flash must be logically divided into separate areas to contain the various objects needed to boot Linux in a stand-alone fashion.Table 3 shows the division chosen in this application note.

Table 3: **Flash Partitions**

	Offset	Size
FPGA Bitstream	0x00000000	0x0016B000 (1452K)
Linux Kernel	0x0016B000	0x00675000 (6612K)
Loader	0x007E0000	0x00020000 (128K)

Linux requires an explicit definition of all flash sections. This explicit definition is represented by Linux as partitions of the flash device, much like fixed disk or any other mass storage partition. This configuration is presented in Prepare the Device Tree for Linux.

Generate the Linux BSP

The device tree is a single text file that describes the hardware devices present in the system. The device tree generator is used to create this BSP.

Note: The user might want to begin with the provided `xapp11406/ready_for_download/linux/xilinx-xapp1146.dts` device tree rather than create one with the device tree generator.

The Linux BSP is generated within the Xilinx SDK. Users should create their own SDK workspace and specify the hardware system provided with this application note, `software/SDK_Hardware_Export/hw/system.xml`. If not automatically prompted when creating a new workspace, the hardware design can be selected by **Hardware Design** → **Import Hardware Design**. There are many ways of starting SDK, creating a new workspace, and choosing a hardware design. Here is one example of how to do all three things simultaneously:

```
C:\> xps_sdk.bat -workspace c:\xapp\xapp1146\software\my_new_sdk_workspace
-hwspec c:\xapp\xapp1146\software\SDK_Hardware_Export\hw\system.xml
```

Note: Use of SDK is beyond the scope of this application note, and only the minimal information required is presented here. The above `xps_sdk.bat` command is specific to Microsoft Windows; Linux users must use `xps_sdk` instead.

1. Copy the `device-tree/`directory obtained in Obtaining the Software to the `software/` directory provided with this application note.
2. Add `software/` as a user repository to SDK: **Tools** →**Software Repositories**. See Figure 1.

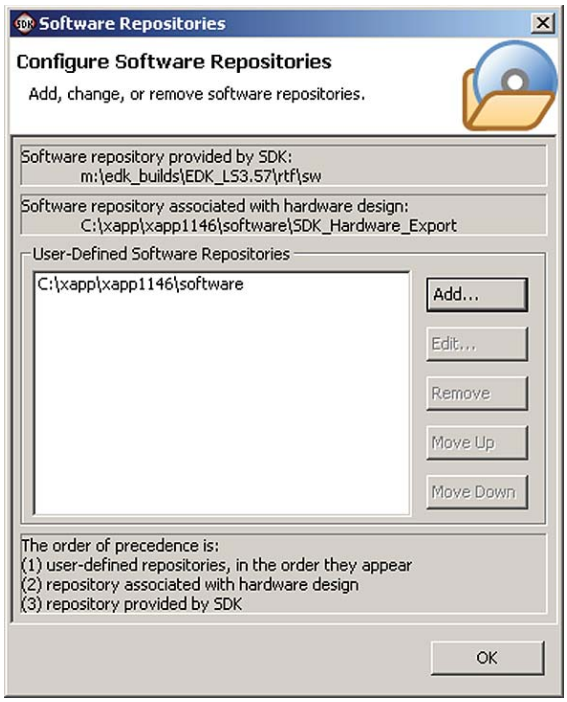


Figure 1: **SDK User Software Repository**

3. Choose **File** →**New** →**Board Support Package**.
4. Choose **Board Support Package Type: device-tree** and enter the Project name of **Linux_0**. Click **Finish**. See Figure 2.

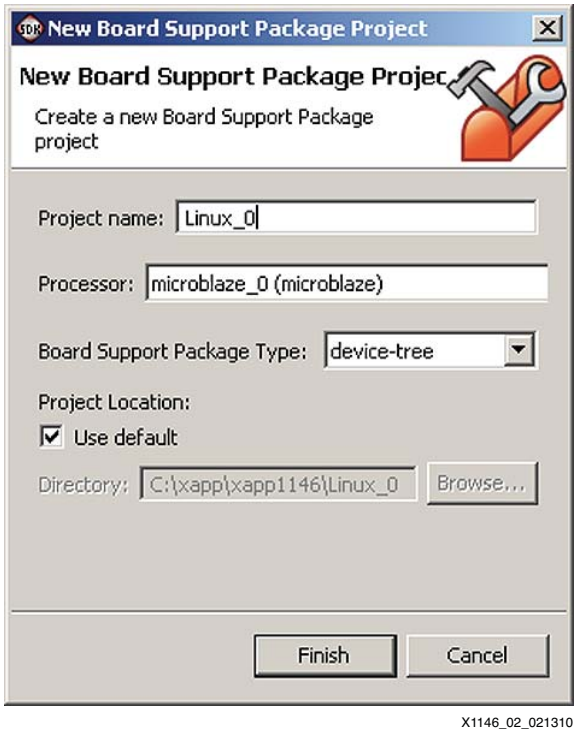


Figure 2: **Generate Linux BSP**

5. Enter **RS232_Uart_1** in the console device field on the Board Support Package Settings window and click **OK**. The file `<user's SDK workspace>/Linux_0/microblaze_0/libsrc/device-tree_v0_00_x/xilinx.dts` is generated.
6. Copy `xilinx.dts` to `<project area>/linux-2.6-xlnx/arch/microblaze/boot/dts/xilinx-xapp1146.dts`.

Prepare the Device Tree for Linux

The device tree file `<project area>/linux-2.6-xlnx/arch/microblaze/dts/xilinx-xapp1146.dts` is edited to specify the proper kernel command line. The unique Ethernet MAC address is also specified in this file. The MAC address assigned to the user's board is found on a sticker on the bottom of the board.

The proper modifications are shown in **bold**:

```
chosen {
        bootargs = "console=ttyS0 ip=192.168.1.10
mtdparts=spi32766.0:1452k(bits),6612k(zImage),128k(loader)";
    } ;
...
        SPI_FLASH: xps-spi@83400000 {
            compatible = "xlnx,xps-spi-2.01.b", "xlnx,xps-spi-2.00.a";
            interrupt-parent = <&Interrupt_Cntlr>;
            interrupts = < 4 2 >;
            reg = < 0x83400000 0x10000 >;
            xlnx,family = "spartan6";
            xlnx,fifo-exist = <0x1>;
            xlnx,num-ss-bits = <0x1>;
            xlnx,num-transfer-bits = <0x8>;
            xlnx,sck-ratio = <0x20>;
            spi_flash@0 {
                compatible = "stm,m25p40", "m25p80";
                reg = <0>;
                spi-max-frequency = <10000000>;
            };
        } ;
```

The flash organization shown in Flash Organization is specified here. An IP address of 192.168.1.10 is statically assigned. An SPI flash is specified on the SPI bus.

Patch the Linux Kernel

The Linux kernel source must be patched to make use of the onboard Winbond 25Q64BV SPI Flash device.

The M25P80 MTD driver can correctly operate the W25Q64BV device. The JEDEC ID for the W25Q must be added to the table of supported devices.

1. Edit `drivers/mtd/devices/m25p80.c` and make the modifications shown in **bold**:

```
static struct flash_info __devinitdata m25p_data [] = {
/* Winbond -- w25x "blocks" are 64K, "sectors" are 4KiB */
    { "w25x10", 0xef3011, 0, 64 * 1024, 2, SECT_4K, },
    { "w25x20", 0xef3012, 0, 64 * 1024, 4, SECT_4K, },
    { "w25x40", 0xef3013, 0, 64 * 1024, 8, SECT_4K, },
    { "w25x80", 0xef3014, 0, 64 * 1024, 16, SECT_4K, },
    { "w25x16", 0xef3015, 0, 64 * 1024, 32, SECT_4K, },
    { "w25x32", 0xef3016, 0, 64 * 1024, 64, SECT_4K, },
    { "w25x64", 0xef3017, 0, 64 * 1024, 128, SECT_4K, },
    { "w25q64", 0xef4017, 0, 64 * 1024, 128, SECT_4K, },
};
```

The drivers for most SPI slaves in the kernel do not presently support the Open Firmware device tree. The modalias table is used to map the compatible entry "stm,m25p40" chosen for use in the device tree (see Prepare the Device Tree for Linux) to a driver that has registered itself with the name "m25p80", as found in `linux-2.6-xlnx/drivers/mtd/devices/m25p80.c`:

```
static struct spi_driver m25p80_driver = {
        .driver = {
                .name   = "m25p80",
                .bus    = &spi_bus_type,
                .owner  = THIS_MODULE,
        },
```

2. Edit `drivers/of/base.c` and observe the text shown in **bold**:

```
static struct of_modalias_table of_modalias_table[] = {
    { "fsl,mcu-mpc8349emitx", "mcu-mpc8349emitx" },
    { "mmc-spi-slot", "mmc_spi" },
    { "stm,m25p40", "m25p80" },
};
```

Build the Linux Kernel

Copy the Ramdisk Image

1. Copy the provided ramdisk image to the kernel tree:

```
$ cp <edk project>/ready_for_download/linux/initramfs_minimal.cpio.gz
<project area>/linux-2.6-xlnx/
```

Configure the Kernel

The Linux kernel is configured to include the appropriate drivers needed to access the onboard flash.

2. Indicate which toolchain is to be used. The below works with a properly installed MicroBlaze processor toolchain:

```
$ export CROSS_COMPILE microblaze-unknown-linux-gnu-
$ cd <project area>/linux-2.6-xlnx
```

3. Copy the default SP605 kernel configuration to use as a starting point.

```
$ cp arch/microblaze/configs/xilinx_mmu_defconfig .config
```

4. Build and run the kernel menu configuration application.

```
$ make ARCH=microblaze menuconfig
```

Note: The user can choose to begin with the provided `xapp1146/ready_for_download/linux/dotconfig` configuration file instead of performing the configuration process.

Submenus are chosen with <**Enter**>, options are modified with <**Space Bar**>.

5. Enable **Device Drivers** →**Memory Technology Device (MTD)** support using <**Space Bar**>, making an asterisk [*] appear. See Figure 3.
6. Choose **Device Drivers** →**Memory Technology Device (MTD)** support using <**Enter**>.
   a. Enable **MTD partitioning support**.
   b. Enable **Command line partition table parsing**.
   c. Enable **Direct char device access to MTD devices**.
   d. Enable **Caching block device access to MTD devices**.

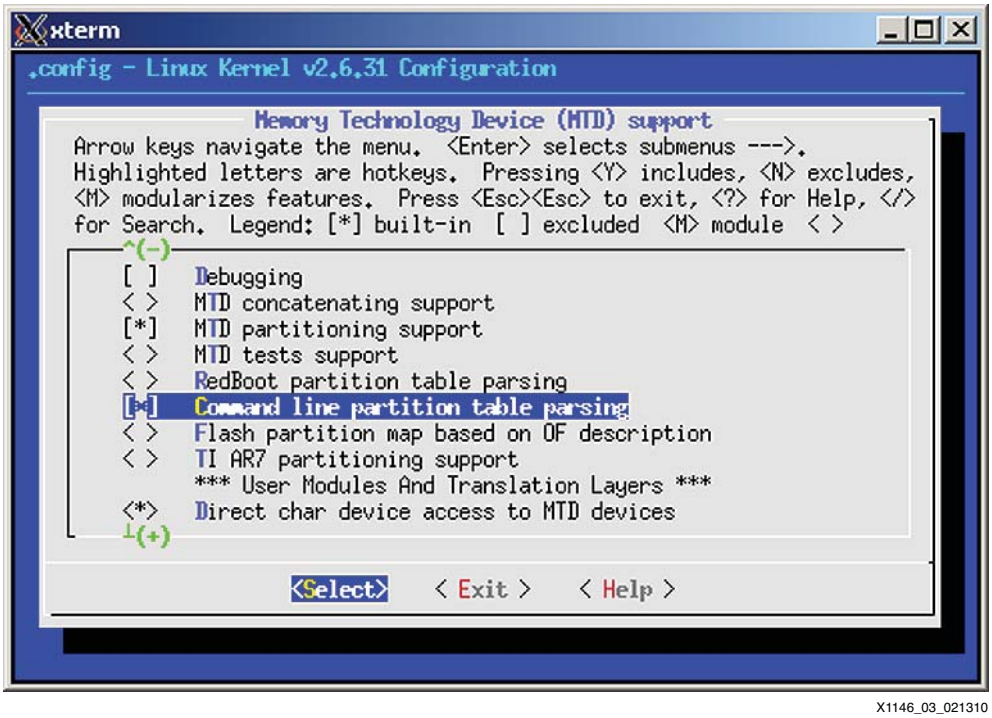
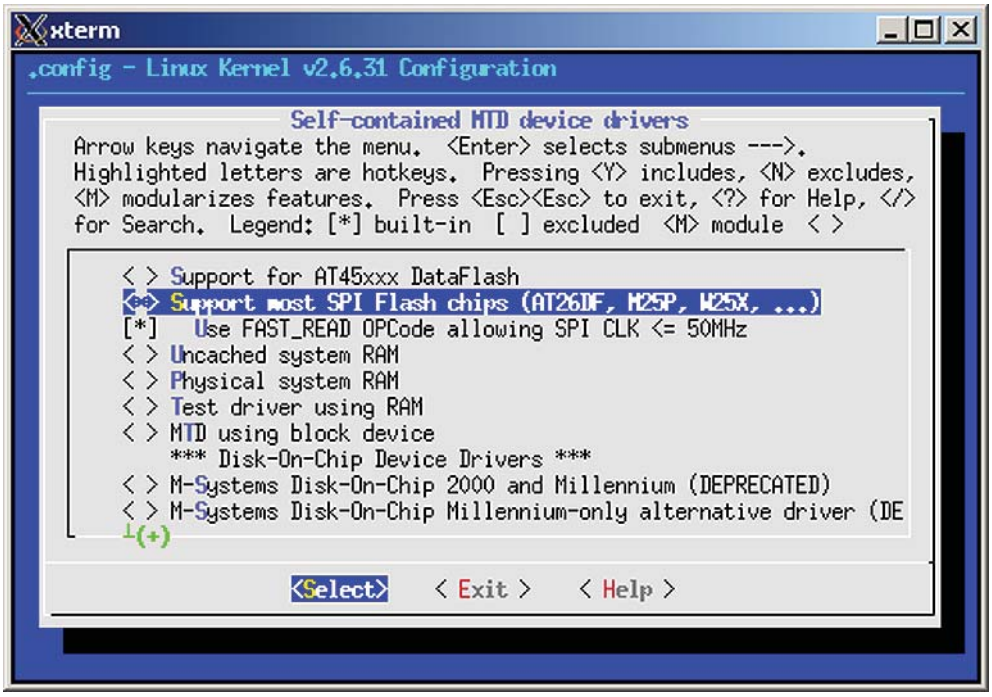


Figure 3: **MTD Support**

7. Choose **Device Drivers** →**MTD Support** →**Self-contained MTD device drivers**.
   a. Enable **Support most SPI Flash chips (AT26DF, M25P, M25X, ...)**. See Figure 4.



X1146_04_021310

Figure 4: **MTD Flash Device Drivers**

8. Enable **Device Drivers →SPI support**.
   a. Enable **Device Drivers →SPI support →Xilinx SPI controller**.
9. Exit and save the configuration.
10. Compile the kernel:

```
$ make ARCH=microblaze simpleImage.xilinx-xapp1146
```

Note: A prebuilt image, `simpleImage.xilinx-xapp1146`, is provided in the `ready_for_download` area.

The new image is created in:
`linux-2.6-xlnx/arch/microblaze/boot/simpleImage.xilinx-xapp1146`.

The Loader

SPI Flash Boot Methodology

The processor cannot natively fetch and execute instructions from SPI flash. In the hardware system provided with this application note, the MicroBlaze processor boot vector (`0x00000000`) is mapped to 8K of block RAM. A small loader, loader1, runs at startup in this 8K block RAM. The block RAM is initialized to contain loader1 during FPGA configuration (it is part of the `download.bit` image). The function of loader1 is to copy the loader2 image from SPI flash and run it. Loader2, which runs in DDR, is larger and more capable than loader1, which must fit in the 8K of available block RAM. Loader2 is capable of uncompressing a gzip archive. The Linux kernel stored in SPI flash has been compressed to maximize the 8 MB of total available space, and loader2 uncompresses this kernel image to the appropriate location.

Note: If the user does not want to compress the kernel, then the bootloading process can be accomplished with loader1 alone. Also, if the user wants to increase the amount of block RAM in the hardware build to 64K, loader2 can be run from block RAM without the need to use loader1. This application note is focused on using the minimum block RAM and SPI flash storage possible, requiring the multi-stage boot process described here.

Rather than a loader that parses the ELF headers of the Linux simpleImage directly, the simpleImage is converted to an ordinary binary file, and a header is prepended to indicate

where this binary blob should be copied. This allows the loader (loader 2) to be very small and simple. See Figure 5.

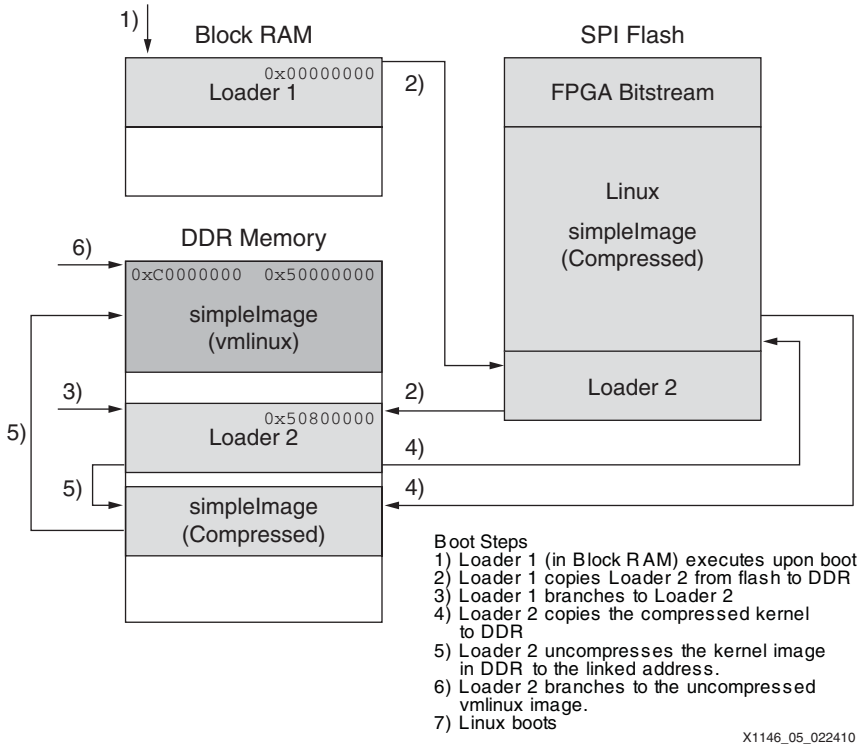


Figure 5: **MicroBlaze Processor SPI Flash Boot Process**

Generate a Binary Image of the Linux Kernel ELF File

An absolute memory image of the Linux simpleImage is used in the flash, not the ELF file output by the linker. The Object Copy utility is used to copy segments from the ELF file to a binary image.

```
$ mb-objcopy -O binary simpleImage.xilinx-xapp1146 linux.bin
```

The generated file `linux.bin` has no relocation information, so the loader does not know where it should be copied from flash.

The readelf Utility

The readelf utility is used to display the ELF headers of an executable in a textual format. This data shows how the simpleImage should be relocated to DRAM. The data needed for the flash loader is shown in **bold**:

```
$ mb-readelf -e simpleImage.xilinx-xapp1146
ELF Header:
  Magic:   7f 45 4c 46 01 02 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, big endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           <unknown>: baab
  Version:                           0x1
  Entry point address:               0x50000000
  Start of program headers:          52 (bytes into file)
  Start of section headers:          4391568 (bytes into file)
```

```
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         41
  Section header string table index: 38

Section Headers:
  [Nr] Name      Type      Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]           NULL      00000000 000000 000000 00      0   0  0
  [ 1] .text     PROGBITS  c0000000 001000 258bd0 00  AX  0   0 16
  ...
  [37] .bss      NOBITS    c0431000 43009c 02d874 00  WA  0   0 16
  ...
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)

Program Headers:
  Type Offset   VirtAddr   PhysAddr   FileSiz  MemSiz   Flg Align
  LOAD 0x001000 0xc0000000 0x50000000 0x2fcfb8 0x2fd000 RWE 0x1000
  LOAD 0x2fe000 0xc02fd000 0x502fd000 0x11c6c  0x11c6c  RW  0x1000
  LOAD 0x310000 0xc0310000 0x50310000 0x12009c 0x14e874 RWE 0x1000
```

The data provided by readelf that is needed by the loader is described in Table 4.

Table 4: **Data Provided by readelf**

Data	Description
LOAD	The address where the executable begins and its size. The simpleImage begins at `0x50000000`.
Entry Point	The address of the first instruction of the executable.
.bss	The Block Started by Symbol (BSS) is not present within the ELF file. This segment is the location of uninitialized global data. The loader should zero this memory.

The build_rom.pl Script

The script `build_rom.pl` provided with this application note generates a binary image of the ELF file using objdump, compresses the binary image with gzip, parses the output of readelf, and prepends a header suitable for use with a simple loader to the binary image. The file format is shown in Table 5.

Table 5: **Loader Image Header Format**

0	"XLNX"
1	Entry point address
2	BSS address
3	BSS size
4	Load address
5	Uncompressed load size
6	Compressed load size
7	<compressed data follows header>

Generate the Linux Flash Image

The flash image for the Linux kernel is generated with the build_rom script:

```
$ <edk project>/ready_for_download/scripts/build_rom.pl
Parsing readelf output for simpleImage.xilinx-xapp1146
Entry: 0x50000000
BSS:   0x50431000 SIZE: 186484
LOAD:  0x50000000
Generating BIN image from elf:
Uncompressed BIN size:     4391068
Compressing BIN image:
Compressed BIN image size: 2565198
Appending header:
```

The first seven words of the generated binary file contain the expected header information:

```
$ hexdump -C simpleImage.xilinx-1146.bin |head
00000000  58 4c 4e 58 50 00 00 00  50 43 10 00 00 02 d8 74
00000010  50 00 00 00 00 43 00 9c  00 27 24 4e 1f 8b 08 08
00000020  1b ff 0a 4b 02 03 74 6d  70 2e 62 69 6e 00 ec bd
00000030  0f 54 5c c7 95 27 7c fb  8f a4 b6 d5 13 3d a0 8d
00000040  db 0e 63 b5 22 ec b4 c7  ec f8 81 98 0c f3 2d b3
00000050  69 2f d8 66 13 be 5d d6  e6 24 64 86 99 b4 96 46
00000060  c6 31 67 87 ac f8 14 9c  65 92 96 68 db 64 c2 7e
00000070  c3 1c 63 1b cf 70 36 ad  55 4b 66 14 f6 1c d6 d6
00000080  27 c3 4c 7f ed d6 08 db  78 a2 ef 1b 76 ad 75 18
00000090  47 3b 69 8d e5 f8 01 ca  86 d8 ac dd 40 c3 db df
```

Import the Software into SDK

Note: This section assumes that the user has performed the steps pertaining to SDK provided in Generate the Linux BSP.

The provided software needs to be imported into the user's own SDK workspace.

1. In SDK, choose **File** →**Import**.
2. Select **Existing SDK Products into Workspace** and click **Next**. See Figure 6.

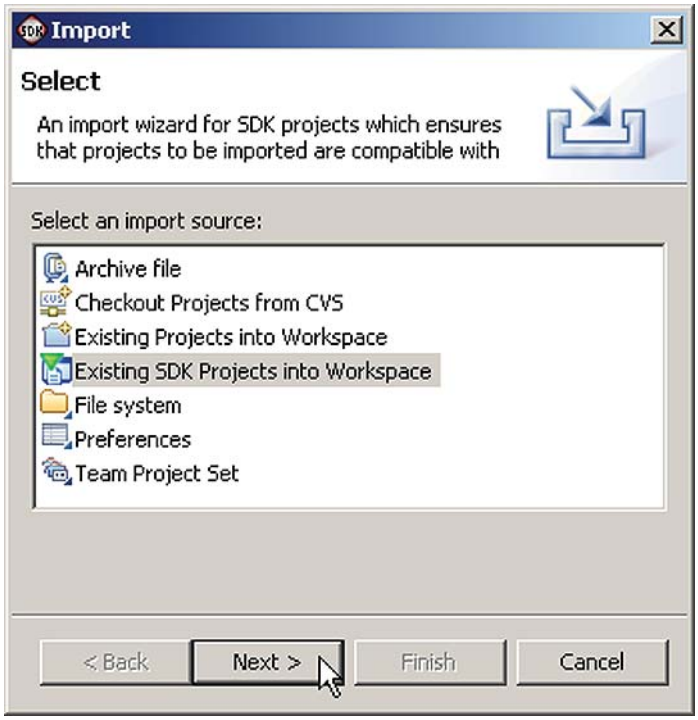


Figure 6: **Import Existing SDK Projects into Workspace**

3. Choose **Select root directory:** and browse to the provided `software/SDK_Projects` directory.
4. Select **loader1**, **loader2**, and **standalone_0** and click **Finish**. See Figure 7.

Note: Choose **Yes** if prompted to overwrite any existing files.

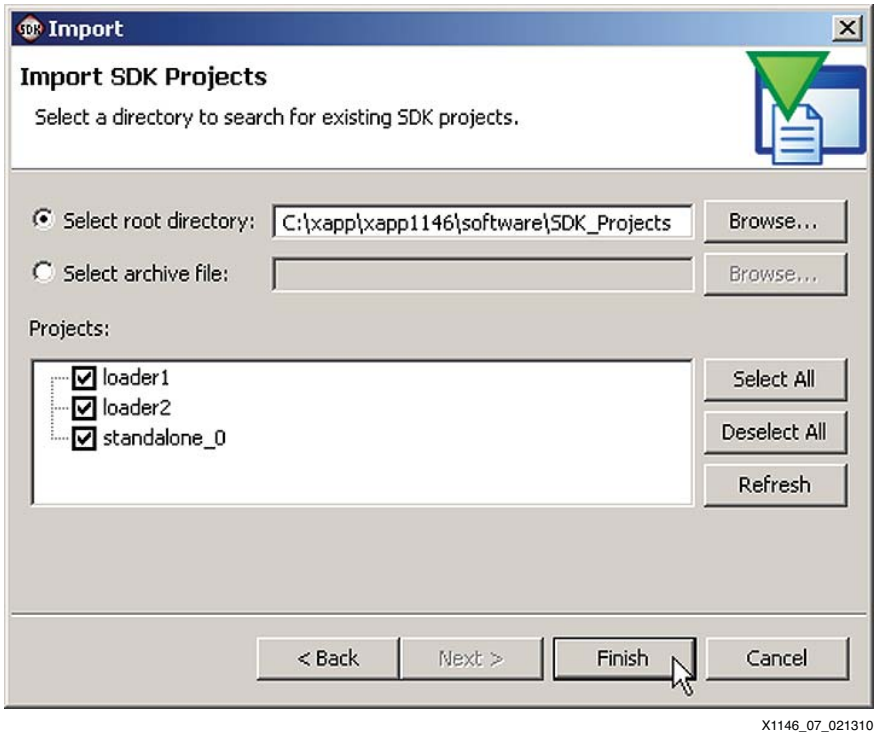


Figure 7: **Select Existing SDK Projects Directory**

SDK automatically builds the imported software. If the user's SDK workspace is not located immediately under the provided software/ directory, as shown in Generate the Linux BSP, *the loader2 application will fail to build.* This is addressed in the loader2 section.

loader1

The simple stage 1 boot loader, loader1 (provided with this application note), is a Xilinx stand-alone BSP application. The linker script provided with this application was created with the *Generate Linker Script* feature in SDK, specifying that all segments apart from the heap and stack should be in block RAM. The heap and the stack are assigned to DDR memory.

Fit loader1 in 8K

Because they consume lots of memory, loader1 avoids the use of the derivatives of printf(). The application and the stand-alone BSP are configured to compile optimized for size **`-Os`**. These actions alone are not sufficient for loader1 to fit in the 8K available. The BSP and application files are compiled with the **`-ffunction-sections`** option. This causes the compiler to place each function in its own section in the generated object file. For example, main() is no longer in the .text segment, but rather in .text.main. The application is linked with the **`-Wl,-gc-sections`** parameter. Ordinarily, if any single function from an object file is used, the linker includes the entire object file in the linked output. The **`-gc-sections`** option only includes those segments that are used; the rest are discarded. Because everything has been compiled with function-sections, each function is in its own segment; the linker discards all functions not used by the loader1 application. This greatly reduces the size of the final executable.

Residing in block RAM, loader1 is provided as part of the FPGA configuration in `download.bit`. A `download.bit` with an embedded ELF file can only be indirectly created within SDK by choosing to configure the FPGA using SDK. Alternately, the loader1 ELF can be used as an ELF-only project to create a `download.bit` in XPS. If the user does not want to configure the FPGA in SDK for the sole purpose of creating this file, or does not want to use SDK for this process, the following commands are used:

```
$ cd <unzipped project area>/xapp1146/software
$ data2mem -bm SDK_Export/hw/system_bd.bmm -bt SDK_Export/hw/system.bit
-bd SDK_Projects/loader1/Release/loader1.elf tag microblaze_0 -o b
download.bit
```

A `download.bit` containing loader1 is created in the `<unzipped project area>/xapp1146/software` directory.

loader2

The simple stage 2 boot loader, loader2 (provided with this application note), is a XIlinx stand-alone BSP application. As noted in the Import the Software into SDK section, the user might need to modify the build properties of this project in SDK for it to build properly. The zlib library is used by loader2 to uncompress gzip images. The location of this library and included files need to be specified. The existing location specified will not work for the user if the SDK Workspace directory is not located immediately under the provided `software/` directory, as shown in the Generate the Linux BSP section. If necessary, this can be remedied with the following steps:

1. Right-click on the **loader2** project and choose **Properties**.
2. In the loader2 Properties window, choose **C/C++ Build**.
   a. Choose **Include Paths**. See Figure 8.

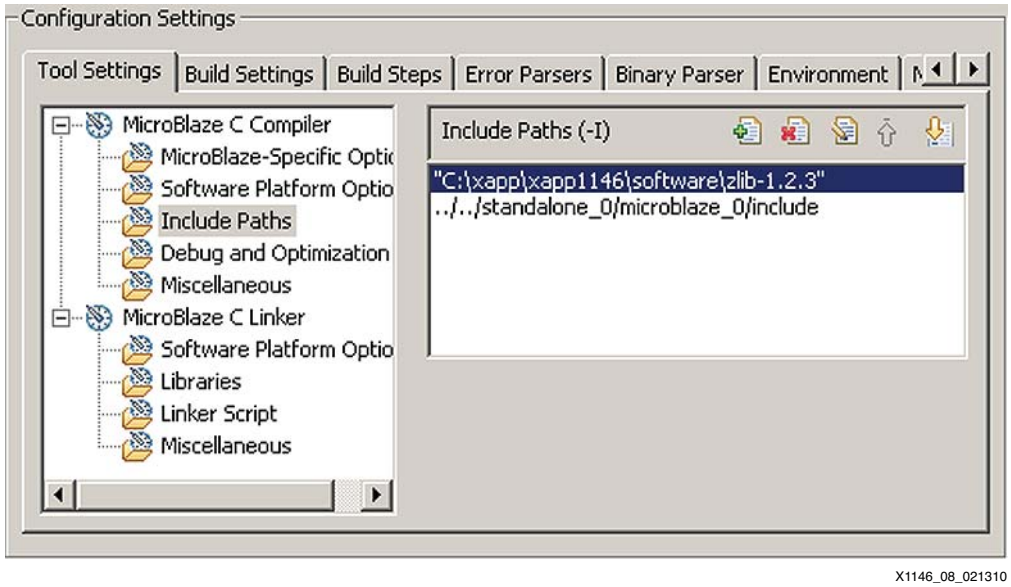


Figure 8: **Configure SDK Include Path**

b. Delete (**X**) any present references to zlib.

c. Add (**+**) the present location of the `software/zlib-1.2.3` directory provided with this application note.Choose **Libraries**.

d. Delete (**X**) any present references to zlib from the Library path.

e. Add (**+**) the present location of the `software/zlib-1.2.3` directory provided with this application note. See Figure 9.

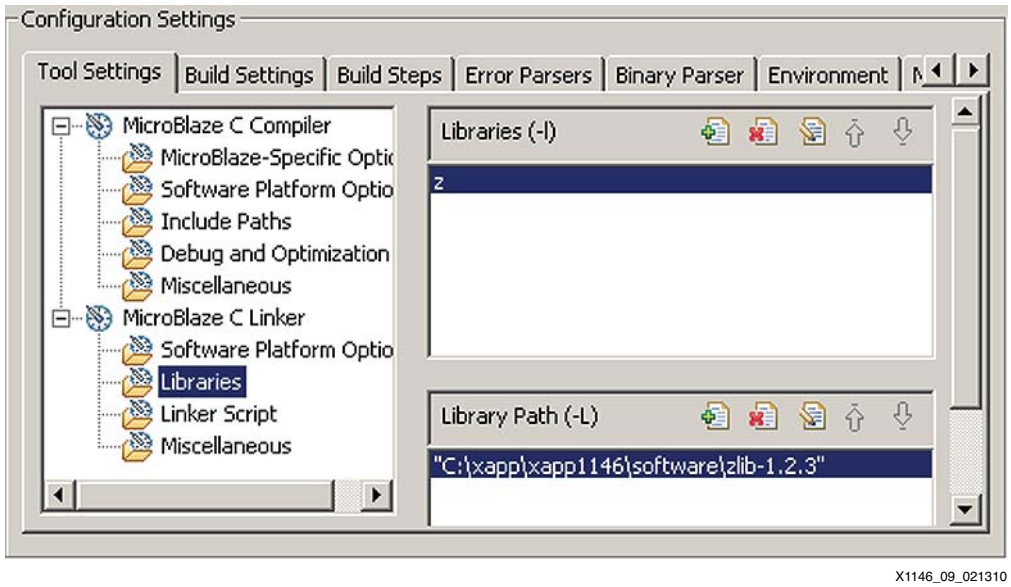


Figure 9: **Configure SDK Libraries**

f. Click **OK**. Loader2 builds.

The linker script provided with this application note was created with the *Generate a Linker Script* feature in SDK, specifying that all segments should be in DDR.

The SDK linker file generator links items at the beginning of the selected memory. The Linux kernel is relocated to the beginning of memory by loader2; therefore, loader2 cannot reside at

this location. The generated linker script has been edited so that loader2 is placed at an offset beyond the expected size of the Linux kernel.

Stand-alone BSP applications are meant to run without the assistance of any operating system or loader. Code is placed at the MicroBlaze processor reset vector (0x00000000) directly. Loader2 cannot operate in this manner. The application start is changed from the processor boot vector to instead immediately precede the rest of the loader2 application (with no “empty” space in between) by editing the linker script.

loader2.ld:

```
MEMORY
{
   LocalMemory_Cntlr_I_LocalMemory_Cntlr_D : ORIGIN = 0x00000050, LENGTH =
0x00001FB0
/*
   DDR3_SDRAM_MPMC_BASEADDR : ORIGIN = 0x50000000, LENGTH = 0x08000000
 */
   /* Loader 2 must not start at the beginning of DDR as this is the
    * location where it will copy the kernel.
    */
   DDR3_SDRAM_MPMC_BASEADDR : ORIGIN = 0x50800000, LENGTH = 0x07800000
}

...

SECTIONS
{

/* ORIGINAL HERE, MODIFIED BELOW:
 .vectors.reset 0x00000000 : {
    *(.vectors.reset)
 }
 .vectors.sw_exception 0x00000008 : {
    *(.vectors.sw_exception)
 }
 .vectors.interrupt 0x00000010 : {
    *(.vectors.interrupt)
 }
 .vectors.hw_exception 0x00000020 : {
    *(.vectors.hw_exception)
 }
*/

/* MODIFIED VECTORS. Nothing is at the hardware vectors any longer. */
.vectors.reset : {
   *(.vectors.reset)
} > DDR3_SDRAM_MPMC_BASEADDR
.vectors.sw_exception : {
   *(.vectors.sw_exception)
} > DDR3_SDRAM_MPMC_BASEADDR
.vectors.interrupt : {
   *(.vectors.interrupt)
} > DDR3_SDRAM_MPMC_BASEADDR
.vectors.hw_exception : {
   *(.vectors.hw_exception)
} > DDR3_SDRAM_MPMC_BASEADDR
```

The application and the stand-alone BSP are configured to compile optimized for size **-Os**. After editing the loader.ld file, the preparations are completed. The loader must be compiled, and the loader ELF file created. The user should be familiar enough with the Xilinx SDK software to accomplish this without further instruction.

An image of the loader suitable for programming into flash is generated with the objcopy utility.

```
$ mb-objcopy -O binary loader2.elf loader2.bin
```

The SDK project has already been configured to automatically generate the image file `loader2.bin` when the application is built.

The FPGA Bitstream

The Spartan-6 FPGA can be configured with an SPI flash. The same flash that holds Linux, the Linux loader, and the Linux file system is used to configure the FPGA. This allows the design to be entirely stand-alone, eliminating the need to configure the FPGA with iMPACT software (by Xilinx).

1. Set the SP605 configuration switches so that the FPGA is configured with SPI_UP configuration 0. SW1 is set to '10'.

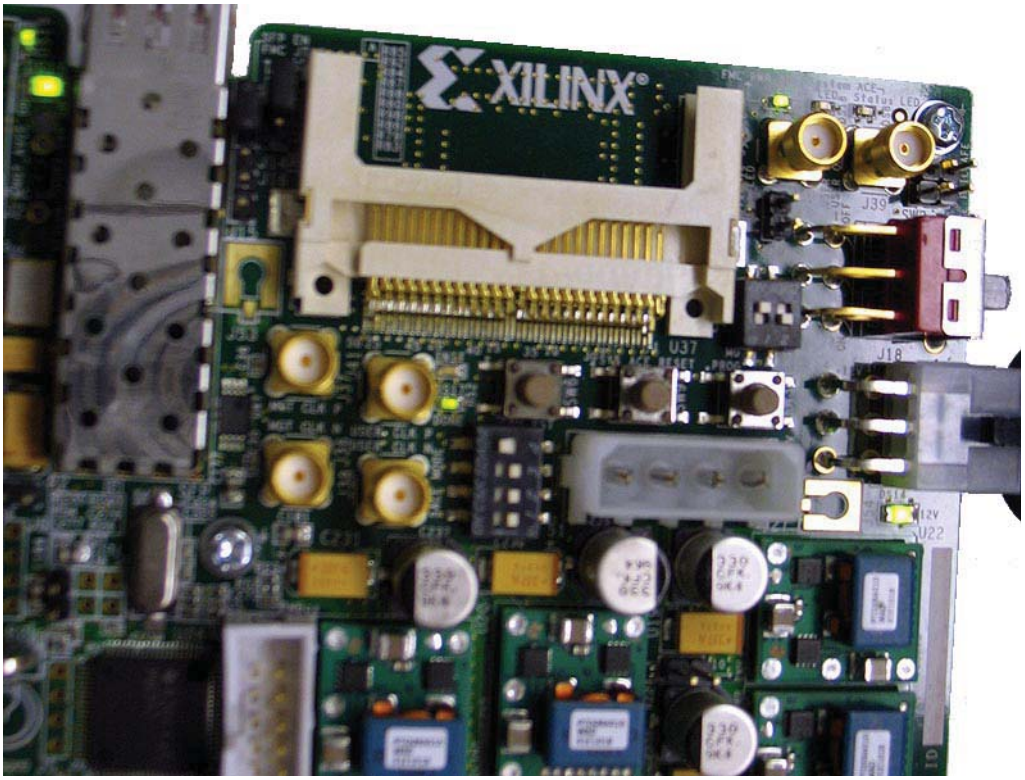


Figure 10: **SP605 M0 M1 Settings for SPI Configuration**

2. An image file of suitable format is prepared from the `download.bit` file generated in Loader1.

```
$ cd <unzipped project area>/xapp1146/software
$ promgen -w -p bin -spi -c FF -o download.bin -u 0 download.bit
```

A previously generated download.bin is available in the `ready_for_download/upgrade-image/upgrade.tgz` archive.

Note: The bitstream must use the Configuration Clock as the Startup Clock. This has already been specified in the EDK project file `etc/bitgen.ut`. The following line is an excerpt from the `bitgen.ut` file, indicating the applicable modification:

```
-g StartUpClk:CCLK
```

Programming the Flash with Linux

The previously generated `download.bin`, `loader2.bin`, and `simpleImage.xilinx-xapp1146.bin` files are ready to be programmed into flash at the offsets indicated in Flash Organization. This application note only discusses using Linux to upgrade the flash.

Retrieving Upgrade Images Over the Network

The upgrade images are retrieved over the network. As seen in Prepare the Device Tree for Linux, a static IP address of 192.168.1.10 is assigned to the SP605.

IP configuration and addressing are beyond the scope of this application note. While more complex configurations are possible, the user should directly connect the SP605 to an FTP server that has been manually configured with the IP address of 192.168.1.1 to successfully perform the tasks outlined here.

Manual Flash Programming

To program the flash with new flash images, the files must be made available to the running Linux image. There are numerous ways this can be accomplished, such as FTPing the files over the network, the System ACE™ tool, or USB mass storage. This application note only discusses files retrieved over the network.

Fetch the image to be programmed using the wget utility.

```
:/> cd /tmp
:/tmp> wget ftp://192.168.1.1/download.bin
Connecting to 192.168.1.1 (192.168.1.1:21)
download.bin 100% |*******************************|  1449k --:--:-- ETA
:/tmp>
```

Erase the Partition

The FPGA bitstream is programmed first. Before programming the new image, it is necessary to erase the appropriate flash region. The first flash partition corresponds to the bitstream:

```
:/> cat /proc/mtd
dev:    size   erasesize  name
mtd0: 0016b000 00001000 "bits"
mtd1: 00675000 00001000 "zImage"
mtd2: 00020000 00001000 "loader"
```

Erase MTD0:

```
:/> flash_eraseall /dev/mtd0
Erasing 4 Kibyte @ 16a000 -- 99 % complete.
```

Program Download.bin into the Flash

```
:/> cd /tmp
:/tmp> cp download.bin /dev/mtd0
```

Automated Flash Upgrade

The script `upgrade.sh` provided with this application note automates the upgrade procedure. It can use upgrade images over the network. If a URL is provided, the manifest file is retrieved over the network from the specified location. The sample manifest provided with this application note is shown below:

```
version: 1.0
tarball: upgrade.tgz
image: mtd0 download.bin
image: mtd1 simpleImage.xilinx-xapp1146.bin
image: mtd2 loader.bin
```

The manifest file specifies a version number (1.0). This version coincides with the file /version in the Linux file system:

```
root:/> cat /version
version: 1.0
```

The tarball field indicates which compressed tar file on the network server contains the upgrade images. In this instance, `upgrade.tgz` is used.

The image: fields denote which flash partition is programmed with which image file. The image files are located within the compressed tar image.

Generate the Tarball

To generate the tarball:

1. Place all the image files in a subdirectory images:

```
$ ls images/
download.bin  loader2.bin  simpleImage.xilinx-xapp1146.bin
```

2. Create a compressed tar file from the images:

```
$ cd images
$ tar -czvf ../upgrade.tgz *
download.bin
loader2.bin
simpleImage.xilinx-xapp1146.bin
```

Note: Previously generated `manifest` and `upgrade.tgz` files are provided in the `<EDK project>/ready_for_download/upgrade-image/` directory.

Upgrade the Images Over the Network

The `upgrade.sh` script uses the wget utility to obtain the manifest and tarball files. Any URL supported by wget (FTP, HTTP) should function.

Note: Once the upgrade has begun, the board must not be powered off until the upgrade is completed.

1. Place the manifest and tarball files on the FTP server and run the `upgrade.sh` script on the SP605:

```
:/> upgrade.sh ftp://192.168.1.1
Network upgrade from ftp://192.168.1.1
Connecting to 192.168.1.1 (192.168.1.1:21)
manifest 100%    |*******************************|   127  --:--:-- ETA
Upgrade manifest version 1.0 found
Currently installed version: 1.0
Proceed? (y/n)
y
Connecting to 192.168.1.1 (192.168.1.1:21)
upgrade.tgz 100%    |*******************************|  2446k --:--:-- ETA
Extracting: /tmp/upgrade.tgz
IMAGES: download.bin simpleImage.xilinx-xapp1146.bin loader2.bin

Upgrading bitstream
Erasing MTD0
Erasing 4 Kibyte @ 16a000 -- 99 % complete.
Programming MTD0

Upgrading Linux kernel
Erasing MTD1
Erasing 4 Kibyte @ 674000 -- 99 % complete.
Programming MTD1

Upgrading loader
Erasing MTD2
```

```
Erasing 4 Kibyte @ 1f000 -- 96 % complete.
Programming MTD2
```

Power Cycle the SP605

This is the expected console output, which would be seen if the power cycle is successful.

```
Xilinx Loader1 (11/23/09 14:08):
Xilinx Loader1: done.

Xilinx Loader2 (11/23/09 14:00):
Entry:        0x50000000
BSS:          0x50431000
BSS Size:     0x0002D874
Load Addr:    0x50000000
Load Size(u): 0x0043009C
Load Size(c): 0x0027244E
Copy compressed image from flash offset 0x0016B000 to DDR 0x5080BD00
Uncompress image:
Uncompress image: done. Uncompressed 4391068 bytes.
Launch:

Linux version 2.6.31-12323-g6e95b50-dirty (bhill@xaqbhill40) (gcc version
4.1.2)
<kernel boot messages follow>
```

Reference Design Matrix

Table 6: **Reference Design Information**

Developer Name	Xilinx
Target Devices	Spartan-6 devices
Source Code Provided	No custom IP is used in the reference design
Source Code Format	N/A
Design Uses Code/IP from an Existing Reference Design / Application Node, Third Party, or CORE Generator Software	No
Simulation	
Functional Simulation Performed	No
Timing Simulation Performed	No
Testbench Used for Functional Simulations Provided	N/A
Testbench Format	N/A
Simulator Software Used/Version	N/A
SPICE/IBIS Simulations	No
Implementation	
Synthesis Software Tools Used/Version	XST
Implementation Software Tools Used/Version	EDK design suite, version 11.4
Static Timing Analysis Performed	No
Hardware Verification	
Hardware Verified	Yes
Hardware Used for Verification	SP605 development board

Additional Information

1. UG111, *Embedded System Tools Reference Manual*
2. http://xilinx.wikidot.com (Xilinx Open Source documentation)
3. XAPP1140, *Embedded Platform Software and Hardware In-the-Field Upgrade Using Linux* (for Virtex-5 FXT devices)

Revision History

The following table shows the revision history for this document.

Date	Version	Description of Revisions
05/05/10	1.0	Initial Xilinx release.

Notice of Disclaimer