Preparing a Uboot image for Altera’s Cyclone V SoC FPGA

General

While preparing the Xillinux distribution for Cyclone V SoC, it turned out more difficult than expected to build an SD card image from scratch. This post outlines the essentials for preparing a custom U-boot based preloader and framework for loading Linux (and possibly other images). This covers the “HPS first” type of boot from an SD (MMC) device on a SocKit board.

The details are related to working on a Linux machine (Fedora Core 12) using Quartus II 13.0sp1. The Quartus version is significant — the U-boot version that comes with e.g. 13.1 is different, even regarding issues like the baud rate on the UART port.

The boot process

First, some background on the boot stages:

- The ARM processor loads a hardcoded boot routine from an on-chip ROM, and runs it. There is of course no way to change this code.
- The SD card’s partition table is scanned for a partition with the partition type field having the value 0xa2. Most partition tools will consider this an unknown type.
- The 0xa2 partition is expected to contain raw boot images of the preloader, as explained below. Since there’s a 60 kB limit on this stage, the full U-boot loader can’t fit. Rather, the SPL (”Secondary Program Loader”) component of U-boot is loaded into the processor.
- The U-boot SPL, which functions as the preloader, contains board-specific initialization code, which sets up the processor’s registers to reflect the settings made in Qsys: That the correct UART is used, the DDR memory becomes usable and the pins designated as GPIO start to behave like such, etc. One side-effect is that the four leftmost LEDs on a SocKit board, which are connected directly to the HPS, are turned off. This is a simple visible indication that the SPL has loaded.
- The SPL loads the “full U-boot” image into memory, and runs it. The image resides in the 0xa2 partition, immediately after the SPL’s boot images (details below).
- U-boot launches, counts down for autoboot, and executes its default boot command (unless a key is pressed on the console, allowing an alternative boot through the shell). See below on how to change this and other environment variables.
- In a typical setting, U-boot loads three files from the first partition of the SD device, which is expected to be FAT: The kernel image as uImage (in U-boot image format), the device tree as socfpga.dtb, and the FPGA bitstream as soc_system.rbf.
- The kernel is launched.

So how can one tell if something is going on? Well, as mentioned above, there’s the four LEDs going off when the SPL has loaded. The following output (or similar) is expected on the console by the SPL (UART at 57600 8N1):

```
U-Boot SPL 2012.10 (Nov 04 2013 - 19:29:22)
SDRAM: Initializing MMR registers
SDRAM: Calibrating PHY
SEQ.C: Preparing to start memory calibration
SEQ.C: CALIBRATION PASSED
DESIGNWARE SD/MMC: 0
```

If it stops at this point, or this appears more than once, there’s a problem with loading and launching the full U-boot software.

A split second later, the following (or similar) should appear:

```
U-Boot 2012.10 (Nov 04 2013 - 19:29:32)

CPU   : Altera SOCFPGA Platform
BOARD : Altera SOCFPGA Cyclone 5 Board
DRAM:  1 GiB
MMC:   DESIGNWARE SD/MMC: 0
In:    serial
```

Related pages

- Xillinux for SoCKit board (deprecated)
- U-Boot programming: A tutorial -- Part I
- U-Boot programming: A tutorial -- Part II
- U-Boot programming: A tutorial -- Part III
- Setting up a device tree entry on Altera’s SoC FPGAs

```
Out:   serial
Err:   serial
Net:   mii0
Hit any key to stop autoboot:  5
```

(and counting down to zero)

This short video clip shows what it looks like when booting Xillinux. As mentioned earlier, the four LEDs to the left go off when the SPL is loaded. The four other LEDs go off when the FPGA is loaded. The one LED that blinks is toggled by FPGA logic (a "heartbeat" LED).

In fact, the LEDs aren't really lit to begin with -- they reflect a floating condition. What makes them go off is that a clear logic level is set. It's like someone takes control.

Note that in this clip there is nothing connected to the UART USB port. The two LEDs to the right of the power button, that flash a few times when the power goes on, are related to the UART and will show considerable activity if something is connected to the UART port.

What’s where

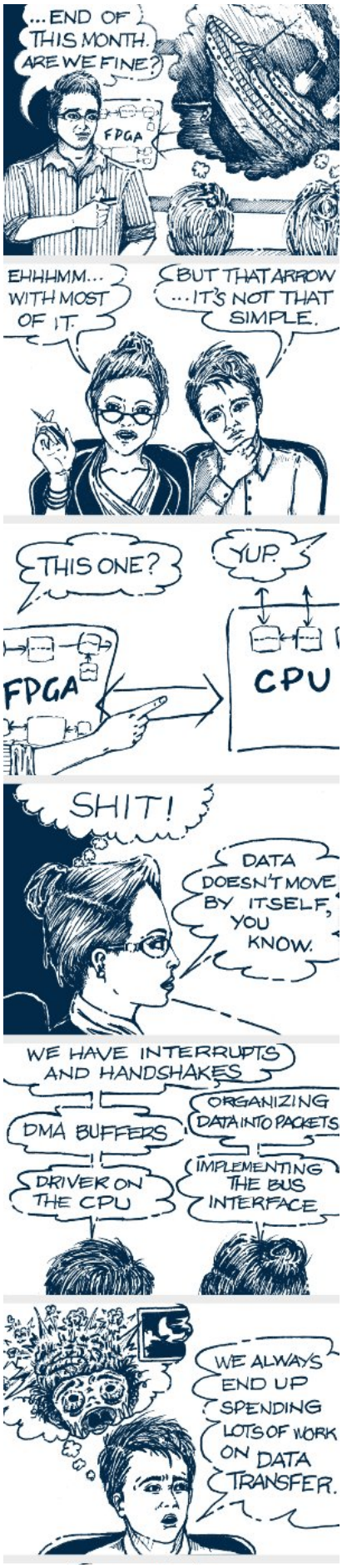
The partition table of a functioning system may look as following:

```
# fdisk -lu /dev/sda

Disk /dev/sda: 7948 MB, 7948206080 bytes
245 heads, 62 sectors/track, 1021 cylinders, total 15523840 sectors
Units = sectors of 1 * 512 = 512 bytes

   Device Boot      Start         End      Blocks   Id  System
/dev/sda1              62      167089       83514    b  W95 FAT32
/dev/sda2          167090      182279        7595   a2  Unknown
/dev/sda3          182280    15508989     7663355   83  Linux
```

Note the Id of /dev/sda2, which is 0xa2. This turns /dev/sda2 into the initial boot partition. Its content is described in Appendix A of the Cyclone V handbook, volume 3, but there’s no need to know the structure of the preloader. There are four preloader images, located as 4x64 kB raw blocks, making the first segment 256 kB long. In all known settings, it’s the same 64 kB image, repeated four times.

One can recognize a preloader segment by its signature:

```
# hexdump -v -C -n 192 /dev/sda2
00000000  1a 00 00 ea 14 f0 9f e5  14 f0 9f e5 14 f0 9f e5  |................|
00000010  14 f0 9f e5 14 f0 9f e5  14 f0 9f e5 14 f0 9f e5  |................|
00000020  20 00 ff ff 24 00 ff ff  28 00 ff ff 2c 00 ff ff  | ...$...(...,...|
00000030  30 00 ff ff 00 01 ff ff  38 00 ff ff 78 56 34 12  |0.......8...xV4.|
00000040  41 53 30 31 00 00 88 29  00 00 a6 01 07 00 00 ea  |AS01...)........|
00000050  40 00 00 01 20 a6 00 00  1c a6 00 00 b8 a7 00 00  |@... ...........|
00000060  1c a6 00 00 de c0 ad 0b  de c0 ad 0b de c0 ad 0b  |................|
00000070  40 00 00 eb 00 00 0f e1  1f 00 c0 e3 d3 00 80 e3  |@...............|
00000080  00 f0 29 e1 10 0f 11 ee  02 0a c0 e3 10 0f 01 ee  |..).............|
00000090  a8 00 9f e5 10 0f 0c ee  05 00 00 eb 12 00 00 eb  |................|
000000a0  9c d0 9f e5 07 d0 cd e3  00 00 a0 e3 08 06 00 eb  |................|
```



```
000000b0  1e ff 2f e1 00 00 a0 e3  17 0f 08 ee 15 0f 07 ee  |../.............|
```

The "AS01" signature at offset 0x40 indicates a preloader boot section. It appears three more times, with 64 kB intervals.

The "full U-boot" image is found at offset 0x40000 in the partition. This is the reason some variables (e.g. spl.boot.SDMMC_NEXT_BOOT_IMAGE and CONFIG_PRELOADER_SDMMC_NEXT_BOOT_IMAGE) in the sources are given the value 0x40000. The SPL loader is told to start reading the next step this far into the partition with the 0xa2 ID (the SPL component of U-boot doesn't know where it was loaded from, so it rescans the partition table).

And indeed,

```
# hexdump -v -C -s 0x40000 -n 192 /dev/sda2
00040000  27 05 19 56 7a 6a 81 22  52 77 d9 93 00 03 c9 cc  |'..Vzj."Rw......|
00040010  01 00 00 40 00 00 00 00  8f c6 72 88 11 02 05 00  |...@......r.....|
00040020  55 2d 42 6f 6f 74 20 32  30 31 32 2e 31 30 20 66  |U-Boot 2012.10 f|
00040030  6f 72 20 73 6f 63 66 70  67 61 5f 63 79 63 6c 6f  |or socfpga_cyclo|
00040040  16 00 00 ea 14 f0 9f e5  14 f0 9f e5 14 f0 9f e5  |................|
00040050  14 f0 9f e5 14 f0 9f e5  14 f0 9f e5 14 f0 9f e5  |................|
00040060  40 02 00 01 a0 02 00 01  00 03 00 01 60 03 00 01  |@...........`...|
00040070  c0 03 00 01 20 04 00 01  60 04 00 01 78 56 34 12  |.... ...`...xV4.|
00040080  40 00 00 01 d4 6c 03 00  d4 6c 03 00 50 b4 06 00  |@....l...l..P...|
00040090  cc c9 03 00 de c0 ad 0b  de c0 ad 0b de c0 ad 0b  |................|
000400a0  4f 04 00 eb 00 00 0f e1  1f 00 c0 e3 d3 00 80 e3  |O...............|
000400b0  00 f0 29 e1 10 0f 11 ee  02 0a c0 e3 10 0f 01 ee  |..).............|
```

The 0x27, 0x05, 0x19, 0x56 sequence is U-boot's native image signature. Those curious to know what the first 32 bytes mean, can look in include/image.h in U-boot's sources. Or use U-boot's own tool, which resides in the sources, and is built automatically when U-boot is compiled:

```
$ uboot-socfpga/tools/mkimage -l uboot-socfpga/u-boot.img
Image Name:   U-Boot 2013.01.01 for socfpga bo
Created:      Mon Nov  4 19:39:59 2013
Image Type:   ARM U-Boot Firmware (uncompressed)
Data Size:    257216 Bytes = 251.19 kB = 0.25 MB
Load Address: 01000040
Entry Point:  00000000
```

Building the preloader / U-boot

One of the possible motivations to build your own U-boot binary is to change the environment variables by editing CONFIG_BOOTCOMMAND and/or CONFIG_EXTRA_ENV_SETTINGS in U-boot's sources (uboot-socfpga/include/configs/socfpga_common.h, or socfpga_cyclone5.h in the same directory). This is however not necessary in most cases, since U-boot reads its defaults from its raw storage beginning at sectors 1 (offset 0x200), immediately after the MBR.

Using the "setenv" and "saveenv" commands at U-boot prompt, it's easy to change the power-up defaults to anything desirable.

To revert to the hardcoded defaults, go (at U-boot's prompt):

```
SOCFPGA_CYCLONE5 # env default -a
SOCFPGA_CYCLONE5 # saveenv
```

The process for setting up the boot loader is somewhat tangled, mainly because the SPL's initialization phase depends on the processor's settings made within Qsys. To make things somewhat more exotic, some C sources and their headers are generated as a byproduct of fully implementing the FPGA project which includes the processor (i.e. running quartus_asm for obtaining a .sof).

In other words, it's **not enough** to "generate" the Qsys project, but it must be used in an FPGA project, which in turn refers to the .qip file representing the top-level module of the processor. In fact, the end-to-end flow is

- "Generate" the Qsys project, from the Qsys GUI or with ip-generate
- Compile the Quartus project using the Qsys element all the way
- Building the preloader (that is, the U-boot project)

On the bright side, most of the really tangled parts are handled automagically with a simple Makefile.

The first thing to do is invoking an Altera Embedded shell, so that the execution paths are set up properly. This is done by issuing a command like

```
$ /my/software/altera/13.0sp1/embedded/embedded_command_shell.sh
```

Now go

```
bsp-create-settings --type spl --bsp-dir build \
  --preloader-settings-dir ../../../synth/hps_isw_handoff/soc_system_hps_0/ \
  --settings build/settings.bsp --set spl.boot.WATCHDOG_ENABLE false
```

This creates a new directory, "build/", and populates it with some files. Most notably, a Makefile.

The –preloader-settings-dir must point at a directory containing the handoff sources. It should be found in the same directory to which the .sof file was written. If you didn't generate a .sof file, don't be surprised if you can't find the handoff files. See above.

Note that these handoff files are the link between the processor settings made in Qsys and the initialization routine that sets the processor's registers, so that these settings actually happen. In other words, if you've changed the function of one of the processor's direct I/O pins, you'll need to reimplement the Qsys project, then the Quartus project, and then the U-boot project (the SPL part in particular).

Now to the magic: To implement the preloader (U-boot SPL) just go

```
$ make -C build
```

which means "change directory to build/ and run the 'all' target in the Makefile". This compiles the U-boot SPL component, and generates the 256 kB worth of preloader image as build/preloader-mkpimage.bin. And also compiles U-boot's utitlies under tools/, including mkimage (which is, by the way, necessary to compile the Linux kernel as an uImage).

As a matter of fact, so much happens as a result of this command, that one may forget to ask: Where did the sources for U-boot come from? Surely some esoteric git repo? Well, no. Among the peculiarities of this build process, a tarball is opened from /path/to/altera/13.0sp1/embedded /host_tools/altera/preloader/uboot-socfpga.tar.gz. It's not as bad as it sounds, because:

- This U-boot version is good for use with SocKit. Well, almost. The environment variables need to be changed as explained below.
- This happens once, and the sources may be edited and recompiled
- A custom tarball can be used instead by issuing

```
$ make -C build TGZ=/path/to/my/uboot-socfpga.tar.gz
```

instead. Note that the tarball's name should be consistent with the directory it generates, so don't just rename the tarball.

Another little peculiarity is that the 'all' target in the relevant Makefile stands for 'spl' and 'mkpimage-spl'. So the "full U-boot" image isn't built. To achieve this, simply go

```
$ make -C build uboot
```

which builds uboot-socfpga/u-boot.img.

Did it work? Great. Now edit uboot-socfpga/include/configs/socfpga_cyclone5.h so its environment variables reflect the following:

```
fpgaload=fatload mmc 0:${mmcloadpart} 0x2000000 soc_system.rbf; fpga load 0 ${fpgad
filesize=6AEBD0
bootcmd=run mmcload; run fpgaload; run mmcboot
```

And issue "make uboot" again. ("filesize" is the hexadecimal number for the file length of the .rbf file).

To wrap this up, verify that build/preloader-mkpimage.bin is **exactly** 262144 bytes long (a.k.a. 256 kB) and concatenate the two files into a partition image:

```
$ cat build/preloader-mkpimage.bin build/uboot-socfpga/u-boot.img > build/boot-part
```

And finally, write this to the 0xa2 partition.

```
# dd of=/dev/sda2 bs=512 if=boot-partition.img ; sync

997+1 records in
997+1 records out
510476 bytes (510 kB) copied, 0.3039 seconds, 1.7 MB/s
```

The numbers may vary, of course. The “sync” command in the end makes sure that the data is flushed to the SD device.

Note: /dev/sda2 is given here for illustration. If you’re not sure which device file to write to, don’t. Picking the wrong device file can wipe your computer’s hard disk.

Please keep in mind, that if the SD card has been used with U-boot previously, U-boot may remember the previous environment variables. It’s therefore recommended to use the short procedure described above for reverting to the default environment.

Further reading

- Xillinux for SoCKit board (deprecated)
- U-Boot programming: A tutorial -- Part I
- U-Boot programming: A tutorial -- Part II
- U-Boot programming: A tutorial -- Part III
- Setting up a device tree entry on Altera’s SoC FPGAs