Project report: Advanced System and Architectures IEEE 1588 PTP synchronization on an embedded platform

Giovanni Simoni
Register 142955
giovanni.simoni@roundhousecode.com

August 1, 2011

Contents

1 Overview

The Freescale MPC8313E-RDB is provided with a U-Boot software, which by default loads a Linux kernel image with a Busybox environment. The command lines of both systems can be accessed through the serial port. In order to communicate through the serial port, an *USB to serial* adapter can be used: on a recent enough GNU/Linux operating system the procedure should be totally seamless.

The board comes along with two CD-ROMs providing the CodeWarrior IDE and some kind of loading system based on the LTIB[1] software, however I'm a suspicious guy, I tend to mistrust anything trying to simplify my life beyond a certain threshold. This is true especially when the software requires to run the *rpm* command with *root* privileges on my Debian-like OS!

I ran the program into a CentOS *chroot jail*, but I still got confused on what the software is supposed to do. Instead of trying understanding it I preferred to do some retro-engineering of the system and go for the *bare-metal* approach I'm going to describe in this document.

The idea is simple: with both U-Boot and Linux the Ethernet interfaces can be enabled, and this can be exploited to conveniently load programs on the board. The development process can be achieved with the classic *editor* + *compiler* combo, which in my opinion is definitely the simplest way of getting stuff done.

The CD-ROMs however are not totally useless, since one of them contains some precious files that usually nobody want to produce by themselves. More on that in the next section.

1.1 Naming conventions of this document

- The Freescale MPC8313E-RDB will be simply referred as "*board*";
- The personal computer or laptop on which compilations and configurations are achieved is referred as "*host*";
- Listings to be executed on the host will have the "host$" prompt:

```
host$ echo Hello
```

- Listings to be executed on the board with the Busybox + Linux environment will have the "board$" prompt:

```
board$ echo Hello
```

- Listings to be executed on the board with the U-Bootsystem will have the "=>" prompt:

```
=> echo Hello
```

[1] http://www.bitshrine.org/ltib/

1.2 What are we running?

During the process realization I aimed at time efficiency: instead of producing from scratch all the tools I preferred to hack and modify the existent environment.

The approach is reasonable, since the system was not that outdated as I initially thought, anyways one might want to update it. The information contained in this document can be very useful also for this kind of operation.

2 Retrieving the tools

As mentioned in Section 1 useful tools and configurations are provided in the CD-ROM:

- A fully fledged cross-compiler for the Power-PC architecture can be found in the **pkgs** directory: **mtwk-lnx-powerpc-gcc-3.4.3-glibc-2.3.3-0.28-1.i686.rpm**;
- The **images** directory contains all the binary objects to be loaded on the *flash memory* of the board:
  - A pre-compiled binary image of the *kernel*: **uimage**;
  - An image of the *Busybox-based filesystem*: **rootfs.ext2.gz.uboot**;
  - The *device tree blob*: **mpc8313erdb.dtb**

  These three items are stored by default inside the board flash memory (Subsection 6.1 explains the technique I used to verify this fact).

The configurations files they can be found inside the **ltib.tar.gz** tarball, in a directory named **config**. They can be used in case of partial or total rebuilding of the system.

Warning: about configurations

If an upgrade to newer versions of the software is required, please mind the fact that old configuration files are usually not suitable for new versions of the software

2.1 The cross-compiler

The cross-compiler is packaged for a Red-hat system, and the *rpm* file uses **/opt/mtwk** as installation directory. On a Debian system you may want to convert it with the alien tool into a *deb* package (it seems also safe to install it directly).

Warning: Compiler issues?

Before discovering the available Linux kernel image I tried to recompile it with the given cross-compiler and kernel configuration. The process gave me some troubles due to a "*internal compiler error*". This has been verified with the 2.6.20 (the version I've found pre-installed on the board).

2.2 Other useful tools

- The mkimage tool is needed to build binary images:
  - Available for Debian-like distributions in the **u-boot-tools** package;
  - Available for Red-hat-like distributions in the **uboot-tools** package (EPEL repositories).
- The communication via serial port can be achieved by using Minicom, available for both Debian-like and Red-hat-like distributions as **minicom**;
- *BootP* and *TFTP* daemons are needed, and many solutions are available for both of them.
  - I used the server packaged as **tftpd** under Debian-like distributions and **tftp-server** under Red-hat-like distributions;
  - I used the server packaged as **bootp** under Debian-like distributions. I'm not sure about the package name under Red-hat.

3 A simple program uploading technique

When I booted the board with the untampered factory settings I noticed that the installed Busybox + Linux system is provided with a Netcat-like tool[2].

The very first test I ran was trying to execute a simple *Hello World* C program on the board as follows:

1. Cross-compile the program:

```
host$ cat > hello.c << EOF
> #include <stdio.h>
> int main ()
> {
>     printf("Hello world!\n");
>     return 0;
> }
> EOF
host$ $ CC=powerpc-linux-gcc make hello
powerpc-linux-gcc    hello.o   -o hello
```

2. Feed a Netcat in server mode with the executable:

```
host$ nc -l 9000 < hello
```

3. Retrieve the program from the board and execute it:

```
board$ nc 192.168.1.1 > hello
board$ chmod +x ./hello
board$ ./hello
Hello world
```

[2] http://nc110.sourceforge.net/

Warning: Interrupt from keyboard

The installed Busybox + Linux system may not support the *termination from keyboard* sequence (CTRL+C); this is probably due to the serial interface. It's a good practice to launch programs like Netcat, which waits for some event, as *background process*.
Also if you plan to test the network with `ping`, for the same reason I suggest to use the `-c` option in order to get a bounded execution.

This technique is a very simple way of running a program on the board, but since the filesystem works in RAM, the program will be lost at the first board reboot. On the other hand this doesn't require to manipulate the system image stored in the flash memory, so it's very quick.

By using this trick you should be able to achieve software testing in a convenient way, while the flashing operation can be delayed to the deployment phase.

4 Hacking binary images

After a proper testing phase one may want to deploy the application in a stable environment, possibly with some automatisms.

An Embedded Linux environment needs three main items: *kernel*, a *filesystem image* and a *device tree*. Those three guys, as explained in Section 2, are provided by Freescale.

If we want to permanently install programs on the board we basically need to modify the filesystem, while the other two elements can be used as they are (as long as the functionality we need aren't placed in kernel space).

4.1 The filesystem image

The filesystem is named **rootfs.ext2.gz.uboot**, thus what I expected is just a *gzipped ext2* filesystem enveloped into some U-Boot wrapper:

```
host$ file rootfs.ext2.gz.uboot
rootfs.ext2.gz.uboot: u-boot legacy uImage, uboot ext2 ramdisk
rootfs, Linux/PowerPC, RAMDisk Image (gzip), 2831355 bytes,
Fri Aug 24 17:01:41 2007, Load Address: 0x00000000, Entry Point:
0x00000000, Header CRC: 0x9B7D6AEB, Data CRC: 0x14B719EB
```

Searching on the web I've found out that this kind of image is produced by the mkimage tool, which envelops stuff inside a 64 bytes header. Not a big deal for our friend dd, right?

So this is an example shell session you can use to mount the filesystem on some directory:

```
host$ dd if=rootfs.ext2.gz.uboot bs=64 skip=1 of=rootfs.ext2.gz
44239+1 records in
44239+1 records out
2831355 bytes (2.8 MB) copied, 0.0881506 s, 32.1 MB/s
host$ gunzip rootfs.ext2.gz
host$ mkdir userland
host$ su -c 'mount -t ext2 -o rw,loop rootfs.ext2 userland'
Password:
host$ ls userland/
bin/  etc/   lib/      lost+found/  opt/   root/  sys/  usr/
dev/  home/  linuxrc@  mnt/         proc/  sbin/  tmp/  var/
```

Bingo!

Warning: Using su or sudo

Depending on your distribution's policy, the execution of a command with *root* privileges could require the su or the sudo command. For instance, if you are on a Ubuntu system you are likely to be using sudo for the mount command:

```
host$ sudo mount -t ext2 -o rw,loop rootfs.ext2 userland
```

Now we need to install some program, say the *hello world* program we compiled in Section 3, inside the image. We just need to copy it:

```
host$ cp /tmp/hello userland/bin
```

4.2 The startup script

At this point we need a little magic: during bootstrap the Linux kernel relies on a temporary root filesystem, which usually contains the modules to be loaded before starting the actual operating system. When the modules are loaded the `pivot_root` command is used to swap the temporary root with the actual root.

Many different solutions are supported here, but the *initramfs* is definitely the most easy and immediate: it's shaped as a *gzipped cpio* archive. More information on this topic can be easily found on the web.

After the bootstrap, the Linux kernel will try to execute the file named **/init**, so what we want to do is skip the root pivoting and remain in the *initramfs* environment. If we want to automatically execute the *hello world* program at bootstrap we just need something like this:

```
host$ cat > userland/init << EOF
> #!/bin/ash
>
> mount -t proc /proc proc
> mount -t sysfs none /sys
>
> hello
>
> echo 'Ready to go! Welcome!'
> /bin/ash --login
> EOF
```

Now we can build an *initramfs* image starting from the modified filesystem by using the `cpio` command, and subsequently envelop it into the wrapper for U-Boot by using mkimage

```
host$ cd userland
host$ find . | cpio -o -H newc | gzip --best > ../initramfs.img
host$ cd ..
host$ mkimage -A powerpc -O Linux -T ramdisk -C gzip -n Sylvie -a 0 -e
      0 -d initramfs.img rootfs.hello.uboot

Image Name:   Sylvie
Created:      Wed Jul 27 17:31:47 2011
Image Type:   PowerPC Linux RAMDisk Image (gzip compressed)
Data Size:    2800916 Bytes = 2735.27 kB = 2.67 MB
Load Address: 00000000
Entry Point:  00000000
```

Our image, the file **rootfs.hello.uboot** is now ready to be loaded on the board. This aspects are covered in Section 5.

5 Uploading binary images

The U-Boot software supports the network booting through *BootP* + *TFTP*. This allows to transfer a binary object, like a *filesystem image* or a *kernel*, from a server to the board.

The default system, installed by Freescale, defines a boot procedure which loads an embedded operating system from the flash memory. I modified such setting, and now it behaves as follows:

1. After a basic bootstrap the board queries the network with the *dhcp*;
2. The host answers providing:
   - The network configuration (IP address + subnet mask);
   - The name of a U-Boot script to be loaded and executed.

3. Basing on the script two different things can happen:
    - The default system booting procedure is activated (*normal booting*);
    - A new *initramfs* is loaded into memory and copied inside the flash memory, replacing the previous image of the filesystem (*update booting*).

   In both cases the assigned IP number is propagated also to the Linux system trought the kernel's command line options.

This procedure particularly well suited for a network environment: by modifying the settings of the *BootP* + *TFTP* server it's possible to redefine the behavior of the deployed boards in a centralized way.

Many tutorials on the Internet cover in detail how to properly configure the daemons. This section contains the dump of my configuration files and an explanation on how I implemented my idea.

But first of all a little overview on how to mess with the network. . .

5.1 Network setup

The board is provided with two network interfaces: the Vitesse VSC7385 and the Marvell. In U-Boot those are respectively enumerated as `TSEC0` and `TSEC1`, and only one of them at a time can be activated. This setting is bound by the environment variable named `ethact`.

In order to select the Marvell card the following command is needed:

```
=> setenv ethact TSEC1
```

Warning: Variable reset problem

As specified by the U-Boot manual, the internal U-Boot environment can be saved by using the `saveenv` command. This works correctly for any variable including `ethact`, but for some reason the active NIC gets re-assigned when U-Boot starts or gets reset.

During my experiments I noticed that U-Boot allows to both set manually an IP address for the board or use a *BootP* server in order to make the procedure automated. The status of the network connection can be verified by using a simple version of the *ping* program embedded into U-Boot.

By running a *sniffer* I discovered that U-Boot doesn't answer to *arp requests*: the MAC address must be set manually on the host. This problem doesn't show up if a *BootP* server is used.

Note: How to deal with network-manager

The network-manager program, very common on recent GNU/Linux distributions, deactivates Ethernet interfaces on which there's no carrier. This can be obnoxious, since the *ARP cache* will be cleaned each time you reset the board. You may want to temporarily deactivate network-manager or manually set the *ARP* address when needed.

5.2 *BootP* and *TFTPd*

Both the choosen versions of the packages (see Subsection 2.2) rely on Xinetd[3]

5.2.1 *TFTPd*

The *TFTPd* configuration is trivial as it should be. We just need to tell Xinetd how to start the daemon and we are done:

```
host$ more /etc/xinetd.d/tftp
service tftp
{
     id = tftp-dgram
     type = UNLISTED
     disable = no
     socket_type = dgram
     protocol = udp
     wait = yes
     server = /usr/sbin/in.tftpd
     server_args = -s /opt/tftpd
     per_source = 1
     user = root
     port = 69
}
```

Some comments:

- My own setting uses the directory **/opt/tftpd** has been chosen as root for the server. All the binaries mentioned in Section 2 are stored there;
- The name of the program to be executed is `in.tftpd`;
- The protocol works on UDP port 69 (according to the well-known ports, see **/etc/services**).

[3] a secure replacement for the Inet daemon. http://xinetd.org

5.2.2 *BootP*

The configuration of the *BootP* daemon is mainly splitted among Xinetd settings and *BootP*'s own configuration file: **/etc/bootptab**.

The former is pretty general, as for *TFTPd*:

```
host$ more /etc/xinetd.d/bootp
service bootps
{
    id = bootp-dgram
    type = UNLISTED
    disable = no
    socket_type = dgram
    protocol = udp
    wait = yes
    user = root
    server = /usr/sbin/bootpd
    per_source = 1
    port = 67
}
```

while **/etc/bootptab** contains the actual loading system logic. My one is the following:

```
host$ more /etc/boottab
.default:\
    :sm=255.255.255.0:\
    :sa=192.168.1.1:\
    :gw=192.168.1.1

.noupdate:\
    :tc=.default:\
    :bf=bootseq_run.img

.update:\
    :tc=.default:\
    :bf=bootseq_update.img

board:\
    :ht=ether:\
    :ha=00fa10fafa10:\
    :tc=.update:\
    :ip=192.168.1.200
```

Some comments:

- The `.default` label identifies general settings of the network;
- The `tc` keyword indicates inheritance of the settings from another label;

- The two behaviors, described at the very beginning of this section, are managed trough the `.noupdate` and the `.update` labels. *BootP* allows to define a *boot file* which is automatically loaded by the *client*:
  1. The `.noupdate` label defines as boot file the script which simply runs the system as it is;
  2. The `.update` label defines as boot file the script which first updates the system and then runs it.

  Further details on Subsection 5.3.
- Finally, for each board (in my case just one of them, I named it just "*board*") we can define the IP address and the kind of operation the board will do on bootstrap (in this case the system will be updated).

5.3 Startup modes

5.3.1 Bootstrap modification

U-Boot conventionally uses, as default bootstrap procedure, the sequence of operations listed inside the `bootcmd` environment variable.

The default U-Boot setting by Freescale sets `bootcmd` as follows:

```
bootcmd=run run_vscld1; run ramargs addtty;bootm fe100000 fe300000
  fe700000
run_vscld=tftp 40000 /tftpboot/vsc7385_load.bin;go 40004
ramargs=setenv bootargs root=/dev/ram rw
addtty=setenv bootargs ${bootargs} console=ttyS0,${baudrate}
baudrate=115200
```

Basically what they are doing is just loading the operating system having the kernel flashed in location `0xfe100000`, the filesystem image in location `0xfe300000` and the device tree at `0xfe700000`.

I moved such sequence into another variable named `vendor_bootcmd`:

```
=> setenv vendor_bootcmd ${bootcmd}
```

Then I modified the `bootcmd` environment variable as follows:

```
=> setenv bootcmd "setenv loadaddr 0x100000; dhcp; source 0x100000"
```

Do you remember the files I mentioned in Paragraph 5.2.2? In the **/etc/boottab** configuration, **bootseq_update.img** and **bootseq_run.img** are set as *boot file* for the `.update` and the `.noupdate` label respectively.

This is what happens with the new setting:

1. The `loadaddr` variable is set to a certain RAM address (I chose arbitrarily `0x100000`). Such variable is conventionally used by the U-Boot `tftp` command as target memory area for the image loaded from the network;

2. The `dhcp` command loops on the network interfaces and probes for a *BootP* service. The *boot file* pointed by *BootP* is downloaded from the *TFTP* server;
3. The `source` command executes the code placed at location `0x100000` (i.e. where the image has been loaded).

5.3.2 The internal scripts

At this point the question is: what are those two files supposed to do? A nice functionality of U-Boot is the support for a basic scripting system.

bootseq_run just requires the execution of the default bootstrap procedure:

```
host$ more bootseq_run
run vendor_bootcmd
```

bootseq_update loads the **rootfs.ptpd.uboot** image from the *TFTP* server and writes it into the flash memory, exactly where the default bootstrap procedure expects to find it. After that the default bootstrap procedure is executed.

```
host$ more bootseq_update
setenv kernel 0xfe100000
setenv initramfs 0xfe300000
setenv devtree 0xfe700000
setenv filesize 0

tftp ${loadaddr} /rootfs.ptpd.uboot
erase ${initramfs} +0x${filesize}
cp.b 0x${loadaddr} ${initramfs} 0x${filesize}

run vendor_bootcmd
```

The two script files are encapsulated inside binary images with the mkimage program.

```
host$ mkimage -T script -C none -n SylvieRun -d bootseq_run
      bootseq_run.img

Image Name:   SylvieRun
Created:      Mon Aug  1 10:54:48 2011
Image Type:   PowerPC Linux Script (uncompressed)
Data Size:    27 Bytes = 0.03 kB = 0.00 MB
Load Address: 00000000
Entry Point:  00000000
Contents:
Image 0: 19 Bytes = 0.02 kB = 0.00 MB
```

```
    host$ mkimage -T script -C none -n SylvieUpdate -d bootseq_update
          bootseq_update.img

    Image Name:   SylvieUpdate
    Created:      Mon Aug  1 10:54:48 2011
    Image Type:   PowerPC Linux Script (uncompressed)
    Data Size:    242 Bytes = 0.24 kB = 0.00 MB
    Load Address: 00000000
    Entry Point:  00000000
    Contents:
    Image 0: 234 Bytes = 0.23 kB = 0.00 MB
```

And of course, in order to be loaded, the created images must be placed into the *TFTP* root directory.

5.4 Summing up...

A script can make things easyer:

```
#!/bin/sh

BIN_RESULT=rootfs.ptpd.uboot
FILESYSTEM=ramfs.img
LOAD=0
ENTRY=$LOAD

rm ${BIN_RESULT}
(cd mnt; find . | cpio --create --format newc | gzip --best > \
    ../${FILESYSTEM})

mkimage -A powerpc -O Linux -T ramdisk -C gzip -n Sylvie -a ${LOAD} \
    -e ${ENTRY} -d ${FILESYSTEM} ${BIN_RESULT}
mkimage -T script -C none -n SylvieRun -d bootseq_run bootseq_run.img
mkimage -T script -C none -n SylvieUpdate -d bootseq_update \
    bootseq_update.img
```

but remember to mount the filesystem before launching it, or you'll end up with an empty filesystem!

6 Some useful tricks

6.1 Images checking

When I first analyzed the original system on the board, I wasn't sure about which of the binary images were stored inside the flash memory.

Of course the file names are self-explanatory enough, but the CD-ROM provides many different files of the same type. There are, for instance, many *device tree blobs*:

- **mpc8313erdb.dtb**
- **mpc8313erdb_usbgadget_external_phy.dtb**
- **mpc8313erdb_usbgadget_internal_phy.dtb**
- **mpc8313erdb_usbhost_external_phy.dtb**
- **mpc8313erdb_usbotg_external_phy.dtb**

How to choose the right one?

The U-Boot system provides a command named `crc`[4] which as computes the *CRC-32 checksum* of a certain memory area.

The size of all *.dtb* files is 12 Kilobytes (`0x3000` in hexadecimal) thus, once the address of the installed *device tree* is known, its *CRC* can be obtained as follows:

```
=> crc fe300000 0x3000
CRC32 for fe300000 ... fe302fff ==> cb588b5e
```

Now we can use a checksum program (like `jacksum`) to obtain the *CRC-32* of all images:

```
host$ for i in *dtb; do jacksum -a crc32 $i; done
2678061774  12288   mpc8313erdb.dtb
1614041190  12288   mpc8313erdb_usbgadget_external_phy.dtb
161761636   12288   mpc8313erdb_usbgadget_internal_phy.dtb
2371930621  12288   mpc8313erdb_usbhost_external_phy.dtb
3656584488  12288   mpc8313erdb_usbotg_external_phy.dtb
```

If we convert in decimal the value extracted from U-Boot we get:

```
host$ echo 'print 0x9f9fface, "\n"' | perl
2678061774
```

which corresponds to the **mpc8313erdb.dtb** file.

This trick can be used to check consistency of uploaded images.

[4] http://www.denx.de/wiki/view/DULG/UBootCmdGroupMemory#Section_5.9.2.2.