Ronetix Application Note AN009: DaVinci DM355 Linux Kernel Debugging and NAND Flash Programming

1. Overview

This article shows you how to debug Linux kernel and completely program the DM355 EVM board using PEEDI. This method can be used for restoring the factory images to the board as well as end product manufacture programming. First it is described how to do it manually and after that how to configure PEEDI to do it by a single push of a button.

2. Prerequisites

- a PEEDI
- a DM355 board
- UBL, U-BOOT, KERNEL and a tarball archive of the files needed to be included in the YAFFS2 partition
- A host machine running an FTP server

3. Setting PEEDI and dumping the YAFFS

First, you need to prepare your CFG file. Since the CFG file itself is not that simple, you can download it from https://download.ronetix.at/peedi/cfg_examples/arm9/tms320dm355-DM355EVM.cfg. The following covers only the FLASH settings. As you need to program 4 different images (UBL, U-BOOT, Linux kernel and the Root file system), you need to define four FLASH profiles in the CFG. You can set the CORE0_PATH parameter to point to where the files are located. This way in the FLASH profiles or with the commands you can use only the file name, not the whole path. Later, if you decide to use the PEEDI in stand-alone mode, you can copy the files onto an MMC/SD card, and you will need to change only the CORE0_PATH. For now, I will set this parameter to point my FTP server root directory.

```
[PLATFORM_ARM]
…
CORE0_FLASH0 = NAND_UBL
CORE0_FLASH1 = NAND_UBOOT
CORE0_FLASH2 = NAND_KERNEL
CORE0_FLASH3 = NAND_ROOTFS
…
CORE0_PATH = "ftp://user:[email protected]/"

[NAND_UBL]
CPU = TMS320DM355
CHIP = NAND_FLASH
DATA_BASE = 0x02000000 ; data
CMD_BASE = 0x02000010 ; commands (CLE)
ADDR_BASE = 0x0200000A ; addresses (ALE)
FILE = ubl.bin, BIN, 64*2048
OOB_INFO = DAVINCI_ECC_HW10_512
BURST_MODE = YES
DAVINCI_UBL_DESC_TYPE = 0
DAVINCI_UBL_DESCRIPTOR_MAGIC = 0xA1ACED00
```

```
DAVINCI_UBL_DESCRIPTOR_ENTRY_POINT = 0x20
DAVINCI_UBL_DESCRIPTOR_LOAD_ADDR = 0
DAVINCI_UBL_MAX_IMAGE_SIZE = 28*1024

[NAND_UBOOT]
CPU = TMS320DM355
CHIP = NAND_FLASH
DATA_BASE = 0x02000000 ; data
CMD_BASE = 0x02000010 ; commands (CLE)
ADDR_BASE = 0x0200000A ; addresses (ALE)
FILE = u-boot-DM355.bin, BIN, 8*64*2048 ; 64 pages per block
;FILE = u-boot-DM355.bin, BIN, 8*128*2048; 128 pages per block
OOB_INFO = DAVINCI_ECC_HW10_512
BURST_MODE = YES
DAVINCI_UBL_DESC_TYPE = 1
DAVINCI_UBL_DESCRIPTOR_MAGIC = 0xB1ACED22
DAVINCI_UBL_DESCRIPTOR_ENTRY_POINT = 0x81080000
DAVINCI_UBL_DESCRIPTOR_LOAD_ADDR = 0x81080000
DAVINCI_UBL_MAX_IMAGE_SIZE = 1024*1024

[NAND_KERNEL]
CPU = TMS320DM355
CHIP = NAND_FLASH
DATA_BASE = 0x02000000 ; data
CMD_BASE = 0x02000010 ; commands (CLE)
ADDR_BASE = 0x0200000A ; addresses (ALE)
FILE = uImage, BIN, 0x00400000
OOB_INFO = DAVINCI_ECC_HW10_512
BURST_MODE = YES

[NAND_ROOTFS]
CPU = TMS320DM355
CHIP = NAND_FLASH
DATA_BASE = 0x02004000 ; data
CMD_BASE = 0x02004010 ; commands (CLE)
ADDR_BASE = 0x0200400A ; addresses (ALE)
FILE = rootfs.bin, BIN, 0x20800000
OOB_INFO = YAFFS
BURST_MODE = YES
```

As you can see from the first two profiles – PEEDI can auto program the image descriptors using info given by the user. My board has 2GB NAND chip. This device is actually 2x1GB NAND devices in one chip, so in hardware point of view there are two NAND chips each with its own chip select. However, the Linux NAND driver merges this into a single 2GB linear device, but for PEEDI there are still 2x1GB devices. In this example, the ROOTFS is placed on the second device. For Linux, the address is 0x60800000, but for PEEDI you will need to use the address calculated from the beginning of the second device, so you must use 0x20800000 (0x60800000 – 1GB). You can also note that to drive the second device chip select, other NAND Command, Address and Data address registers are used.

The “CPU = TMS320DM355” parameter tells PEEDI to check for a Bad Block table and if not found, to create it on the last good block. PEEDI also creates a mirror BBT. Now we can program into the NAND chip UBL, U-BOOT and KERNEL to let the system boot and then load the YAFFS via the Ethernet. Using **flash set** command, you can select the FLASH profile you want to use and then **flash program** to program the NAND:

```
peedi> flash set 0
peedi> flash erase
peedi> flash program
peedi> flash set 1
peedi> flash program
peedi> flash set 2
peedi> flash program
```

Now disconnect the board from PEEDI and reset it, it should boot normally. Please read this guide on how to program the YAFFS root file system:
Put YAFFS Image to Flash

Now connect the board back to PEEDI. Since there is no right tool to make YAFFS images for the DM355 CPU (too many complications caused by the HW ECC), you can dump the YAFFS image from the NAND and use it later for a quick restore or a manufacture programming. Using the **flash set 3** and **flash query** PEEDI commands, check the list of the bad blocks and make sure there are no any bad blocks in the YAFFS NAND region:

```
peedi> flash set 3
peedi> flash query
```

First, you need to check where the YAFFS region ends. You could dump the whole partition but it is pointless to dump a lot of FF's if the partition is not full. First, check out the sum size of all files included in the tarball. Then read the NAND at address which is the base address of the partition plus the size just to make sure there is data on that address. Then find the last NAND page in the partition that has been written with YAFFS data using the **flash read** command. Lets assume that the page at address 0x2B900000 gives all FF's, i.e. it is empty:

```
peedi> flash read 0x2B800000
```

Considering that the file system begins at 0x20800000, it means you need to dump 0xB000000 bytes. Now you can dump this area, but YAFFS image files include the OOB data, this OOB data is not included in the NAND memory space so in reality you need to dump a little bit more data correcting the page size from 2048 to 2112:

```
peedi> flash dump 0x20800000 0xB000000/2048*2112 rootfs.bin
```

Just to make sure everything is OK you can verify the just dumped image:

```
peedi> flash verify rootfs.bin 0x20800000
```

4. Programming the board

Now you are ready for the complete burn process. Next time you want to restore your board or just program a new one, you can skip the whole previous part and jump directly to this point here, which programs everything:

```
peedi> flash set 0
peedi> flash erase
peedi> flash program
peedi> flash set 1
```

```
peedi> flash program
peedi> flash set 2
peedi> flash program
peedi> flash set 4
peedi> flash erase
peedi> flash program
```

Disconnect the board and verify it boots normally and mounts the root file system. Now you can put all the commands into a script for an easy automated execution:

```
[prog_all]
flash set 0
flash erase
flash program
flash set 1
flash program
flash set 2
flash program
flash set 4
flash erase
flash program
```

And put the script in the [ACTIONS] list so it can be started using the PEEDI buttons:

```
[ACTIONS] ; user defined scripts
;AUTORUN = 1 ; executed on every target connect
1 = prog_all
```

So the script can be started in three ways:

- by using the PEEDI **run $prog_all** command
- by selecting and starting script 1 using the front PEEDI buttons
- if the AUTORUN = 1 line is uncommented, the script will be started every time a board is connected to PEEDI.

The way 1 is useful for a board restore. Ways 2 and 3 are useful for production programming, since nonqualified personnel can easily be instructed on how to work with PEEDI.

5. Linux kernel debugging

For Linux kernel debugging, you need an ELF of the kernel compiled with debug info and optimization switched off (-g -O0). The board should be set to load and start the kernel either from the NAND or any other source. For debugging, best is loading it via the Ethernet, this way you don’t need to program the NAND from build to build. Once you build the kernel, locate the vmlinux file (it should be in the root directory of the Linux kernel project) and use the nm tool to find the address of the start_kernel function like this:

```
nm vmlinux | grep start_kernel
```

Now use CFG without INIT section so after PEEDI connects to the CPU it does not make any initialization and make sure breakpoint mode is set to SOFT (we are going to debug in RAM). After PEEDI is connected, use “break add hard ADDRESS” to add a hardware breakpoint at the start_kernel address. Now “go” to start the target and wait until it breaks at the breakpoint. Next you can start insight in the kernel root directory pointing the kernel ELF image:

```
arm-elf-insight vmlinux  ; Now connect to PEEDI in the insight console:
```

target remote YOUR_PEEDI_IP:2000
Next, use the “si” command just to refresh the view. From now on you can add breakpoints, step, go, stop and so on the target. To start execution, use the C button or the ‘continue’ command in the insight console or the resume button in eclipse. Once you do the previous, you can automate your work by putting this in the INIT section:

```
break add hard ADDRESS ; add hardware break at start_kernel
go ; start the CPU
wait 30000 stop ; wait for the target to break with timeout of 30 sec, it might need more time, check the board boot up time
break del all ; remove the break
beep 500 500 ; beep to signal ready for debug
```

And you can also put the target remote and si command in a .gdbinit file, so these are auto executed on gdb/insight start. Of course, you can use eclipse too for debugging.

6. Conclusion

At first glance, things might seem too complicated, but it gets easier after the first iteration. On the other hand, all the complicated setting is made once and after that everything is easy as one push of a button.

Share on LinkedinShare on FacebookShare on Reddit on TwitterShare Via EmailShare on Whatsapp Share