



TECHNICAL NOTE

Practical Aspects in U-Boot Programming

Document Version 1.0

13th of July, 2017

Authors

João Monteiro, Jinxin Song, Rafael Lopes, António Santos
joao.monteiro, jinxin.song, rafael.sardinha, antonio.santos @criticalsoftware.com

Abstract

U-Boot is widely used in embedded systems, not only as the bootloader itself, but also to support hardware bring-up. It has grown to an extent that it now provides support for a variety of target systems, making it suitable for low-level hardware testing. It can also be adapted to support custom boards, given the variety of drivers already implemented by default. However, the documentation related with u-boot is rare and sparse, making it very difficult for engineers to properly understand the u-boot structure, adapt it, or even add new custom functionalities. This document aims to fill this gap by providing self-contained practical hands-on information, ultimately easing the ramp-up phase for engineers interested in working with u-boot.



Critical Software, 2017

© 2017 Critical Software. This work is made available under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit <http://creativecommons.org/licenses/by/4.0/>.



Introduction

The lack of self-contained and concise documentation available to support engineers working with u-boot is evident. It is common sense today that the best way to learn about u-boot is through experimentation. Still, the authors understand this as a necessary, but insufficient premise. Good documentation can greatly speed-up ramp-up times and support correct coding decisions in further stages by providing an early, well established, ground-base knowledge.

The authors then decided to create, filter and compile all necessary information in a single *practical* document that reflects their hands-on experience. It will hopefully guide engineers through building, customizing and having u-boot easily running on their targets.

Practical experimentation behind this document was carried out with support of the BeagleBone Green board, endowed of an AM3358 TI Sitara SoC, with a 1GHz ARM Cortex A8 processor. However, the information provided herein can be easily extrapolated for other hardware targets.

Instructions are provided on how to run u-boot with qemu emulator so that the reader can test u-boot builds without a specific hardware target.

Particular effort was put on having hands-on, step-by-step, instructions in order to make this a *practical* and self-contained guide. If not found any more online, specific packages may be requested to the authors.



Index

1	Preparation	5
1.1	Setting Up a U-Boot Development Environment with ECLIPSE	5
1.2	Install the Cross-Compile Toolchain	10
1.3	Build U-Boot	11
1.4	QEMU Installation.....	12
1.5	Run U-Boot on QEMU (versatile target)	13
2	U-Boot Structure.....	16
2.1	Directory Hierarchy.....	16
2.2	Configuration Options.....	16
2.2.1	Compiling U-Boot Image.....	16
2.3	Boot Process	17
3	U-Boot Customization	19
3.1	Implementation of an External Memory SPI Driver.....	19
3.1.1	Hardware and Software Requirements.....	19
3.1.2	U-Boot Drivers (SPI Example)	19
3.1.3	U-Boot Commands (SPI Example)	19
3.1.4	U-Boot Driver Calls (SPI Example)	21
3.2	Creating an API (MR25H40 drive case)	23
3.2.1	Mram SPI Read	23
3.2.2	Mram SPI Write	24
3.3	Integrating Mram API Into U-Boot	26
3.4	Running U-Boot am335x Target on QEMU	28
3.4.1	Installing qemu-linaro.....	28
3.5	Running U-Boot am335x Target on Hardware	32
3.5.1	Creating a Bootable microSD Card.....	34
4	Integrating Standalone Applications.....	37
4.1	Understanding the Hello World Application	37
4.2	Exporting U-Boot Functions for Standalone Applications	38
4.2.1	Exporting Additional U-Boot Functions	39
4.3	Porting “Memtester” to a Standalone U-Boot Application	41
4.3.1	Running Standalone MemTester	43
4.3.2	Memtester Standalone Application Structure in Memory	44
5	Using U-Boot Commands	50
5.1	bdinfo.....	50



5.2	md	50
5.3	mw	51
5.4	usb	51
5.5	mmc	52
5.6	Usage Examples	53
5.6.1	USB R/W Access	53
5.6.2	SD R/W Access	57
6	References	58

1 Preparation

1.1 Setting Up a U-Boot Development Environment with ECLIPSE

The Development Environment will be installed using a Linux Virtual Machine under Windows environment. Alternatively, the same steps can be used in a Linux machine apart from the VM setup. Instructions will be provided step-wise.

NOTE: This section is mainly based on [1], with small modifications for our specific case.

[Step 1] Download and install Virtual Box from <https://www.virtualbox.org/>

[Step 2] Download xubuntu 12.04 32b image from <https://virtualboxes.org/images/xubuntu/>

[Step 3] Setup VirtualBox with the settings shown in Figure 1, or alternatively, the closest possible:

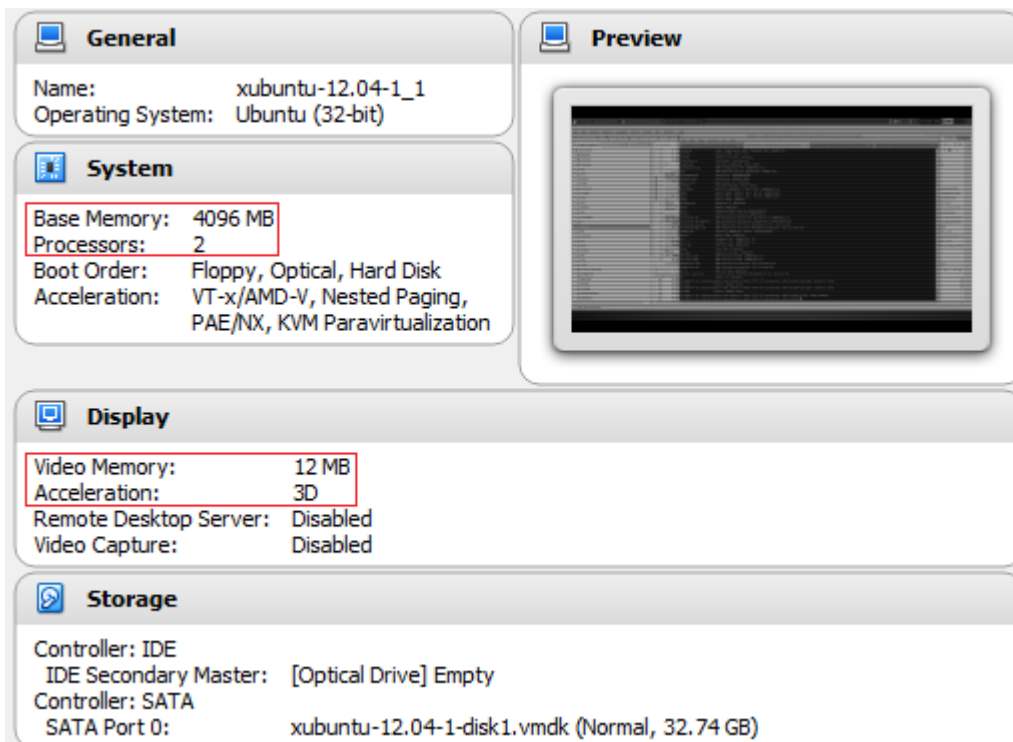


Figure 1: VirtualBox Settings

- Add a shared folder under Settings->Shared Folders with 'Auto-mount'
- Activate 'Bidirectional' setting under Settings->Generic->Advanced->Shared Clipboard/Drag'n'Drop

[Step 4] Log into your Linux VM (pwd:reverse) and open a command terminal.

[Step 5] Install Guest Additions on the Ubuntu Virtual Machine [4]:

- On the VirtualBox Menu, choose Devices -> 'Insert Guest Additions CD Image'



- Open a terminal and go to the mounted media folder:
\$ cd /media/VBOXADDITIONS_5.1.22_115126

Note that version may vary, which shall not be a problem.

- Launch installation

```
$ sudo -s
```

```
$/VBoxLinuxAdditions.run
```

Note: Ignore the 'Mesa' issue.

- Reboot the VM by clicking xubuntu (top right corner) -> Reboot -> Reboot

- Log-in and the screen shall maximize/adapt to your resolution. Also you should be able to copy-paste text between the VM and host.

[Step 6] Open a terminal window.

NOTE: It may be necessary to adjust the Keyboard language setting. Default is ENG.

[Step 7] Install Eclipse (used version is Mars II):

```
$sudo -s
```

```
$cd /opt/
```

```
$wget http://www.mirrorservice.org/sites/  
download.eclipse.org/eclipseMirror/technology/epp/downloads/release/mars/2/eclipse  
-cpp-mars-2-linux-gtk.tar.gz
```

```
$ tar -xvf eclipse-cpp-mars-2-linux-gtk.tar.gz
```

[Step 8] Add eclipse executable to PATH by default:

```
$nano /etc/profile
```

place the following line at the end of the file:

```
PATH=/opt/eclipse:$PATH
```

[Step 9] Create a project folder:

```
$mkdir /opt/projects/
```

```
$cd /opt/projects/
```

[Step 10] Download u-boot source:

```
$apt-get update
```

```
$apt-get install git
```

```
$git clone https://github.com/gvigelet/u-boot-2015.07.git
```

```
$cd u-boot-2015.07
```

[Step 11] Install JAVA:

```
$add-apt-repository ppa:openjdk-r/ppa
```

```
$apt-get update
```

```
$apt-get install openjdk-8-jdk
```

[Step 12] Open eclipse and set your default environment folder as shown in Figure 2:

```
$. /etc/profile  
$eclipse
```

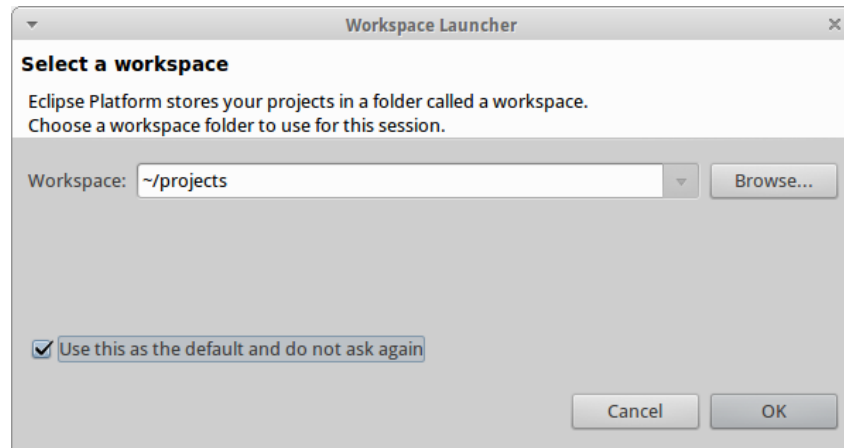


Figure 2: Eclipse Workspace Setup

[Step 13] Create a new project:

Click on File -> New C Project (Figure 3)

- Set project name u-boot-2015
- Check if location is set by default to /opt/projects/u-boot-2015.07
- Project type: Empty Project
- Toolchains: Linux GCC
- Click 'Finish'.

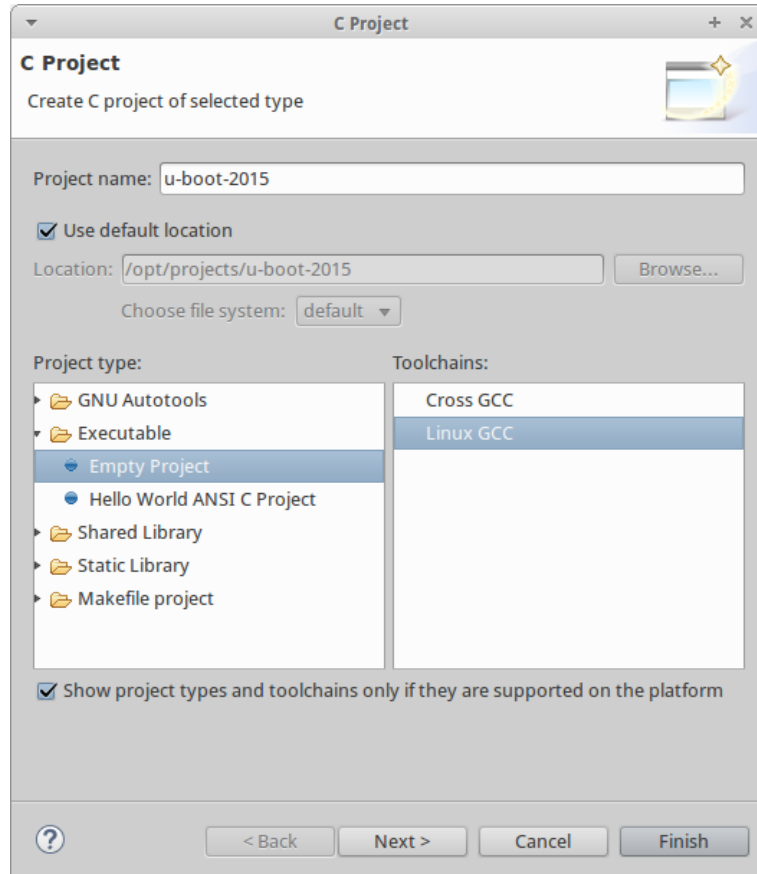


Figure 3: New Project Settings

[Step 14] Click to highlight the newly-created 'u-boot-2015' project on the 'Project Explorer' Window on the left.

[Step 15] Click on Project -> Properties -> C/C++ Build -> 'Behaviour' tab

- Leave the editable field under Build (Incremental build) blank. This is necessary, otherwise the IDE will issue a "make all" command when the project is built.
- Uncheck Build on resource save (Auto build)
- Change clean to distclean, otherwise it will issue the "make clean" command rather than the desired "make distclean".
- Click on the Builder Settings Tab
 - Change the default Build Directory path to /opt/projects/u-boot-2015.07
 - Uncheck General Makefiles automatically

[Step 16] Click on Project -> Properties -> C/C++ Build -> Environment

- Undefine CWD and PWD

- Add environment variable with name `CROSS_COMPILE` and value `arm-none-gnueabi-` for the variable (don't forget the last dash). The result should be as shown in Figure 4.

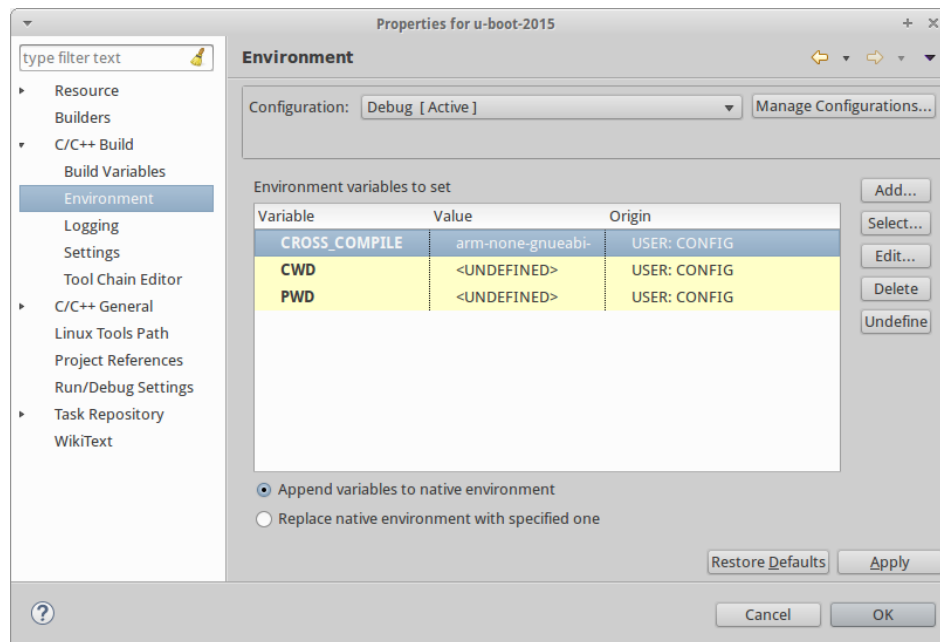


Figure 4: Project Environment Variable Settings

[Step 17] Click on Project -> Properties -> C/C++ Build -> Settings -> 'Binary Parsers' tab

- Check that 'GNU Elf Parser' is selected.
- Click OK to close the Properties window.

[Step 18] Go to Project -> Make Target -> Create

- Target Name: `distclean`
- Click OK.

[Step 19] Go to Project -> Make Target -> Create

- Target Name: `am335x_evm_config`
- Click OK.

[Step 20] Go to Project -> Make Target -> Build

- Check that both Build Targets `distclean` and `am335x_evm_config` exist, as shown in Figure 5.

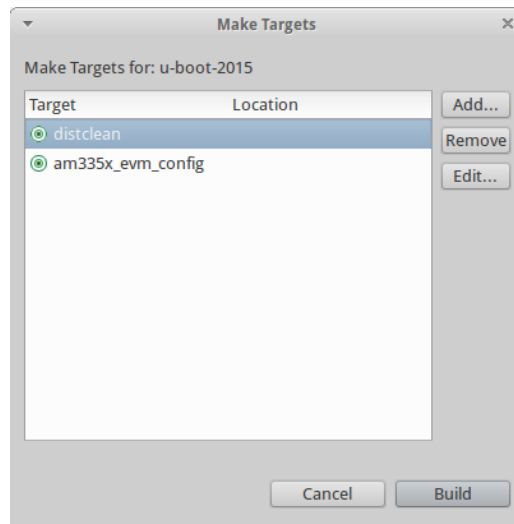


Figure 5: Make Targets

[Step 21] Close Eclipse.

It will not be possible to run the make targets just yet because for that, the cross-compile toolchain is necessary. This will be covered in sub-section 1.2.

1.2 Install the Cross-Compile Toolchain

[Step 1] Download and install arm-2010.09-50-arm-none-linux-gnueabi.bin:

```
$sudo -s
$mkdir ~/buffer
$cd ~/buffer
$wget https://sourcery.mentor.com/public/gnu_toolchain/arm-none-linux-gnueabi/arm-2010.09-50-arm-none-linux-gnueabi.bin
$chmod 777 arm-2010.09-50-arm-none-linux-gnueabi.bin
$./arm-2010.09-50-arm-none-linux-gnueabi.bin
```

[Step 2] If prompted about the 'DASH Shell not supported as system shell', run:

```
$sudo dpkg-reconfigure -plow dash
```

and choose 'No', press enter, and try again. An installation GUI will appear, just follow the instructions.

NOTE: In the 5th screen, modify the installation path to `/opt/CodeSourcery/Sourcery_G++_Lite` [2].

[Step 3] Add toolchain binaries to PATH:

```
$nano /etc/profile
```

- Add the following line at the end of file:

```
PATH=/opt/CodeSourcery/Sourcery_G++_Lite/bin:$PATH
```

```
Ctrl-X -> Y -> Enter
```

- Manually launch script:

```
$. /etc/profile
```

1.3 Build U-Boot

It should now be possible to run the make targets and build u-boot:

[Step 1] Re-open eclipse using the command prompt:

```
$eclipse &
```

[Step 2] Highlight the u-boot-2015.07 project on the 'Project Explorer'

[Step 3] Go to Project -> Make Target -> Build...

- Select distclean

- Click build. The snippet shown in Figure 6 should be obtained in the console:

```
18:30:04 **** Build of configuration Debug for project u-boot-2015 ****
make distclean
18:30:04 Build Finished (took 785ms)
```

Figure 6: Make distclean target

- Select am335x_evm_config. The snippet shown in Figure 7 should be obtained in the console:

```
HOSTCC scripts/kconfig/zconf.tab.o
HOSTLD scripts/kconfig/conf
#
# configuration written to .config
#
18:34:52 Build Finished (took 2s.523ms)
```

Figure 7: Make am335x_evm_config target

[Step 4] Highlight the u-boot-2015.07 project on the 'Project Explorer'

[Step 5] Select Project -> Build Project

- The command shall end successfully with the messages shown in Figure 8:

```
OBJCOPY spl/u-boot-spl.bin
MKIMAGE MLO
CFG spl/u-boot-spl.cfg
18:37:39 Build Finished (took 43s.650ms)
```

Figure 8: Output of Build Project.



This process will be explained in detail in section 2.

1.4 QEMU Installation

QEMU will allow us to test the u-boot build without the hardware. It is capable of emulating target architectures and systems. Since further qemu versions may require specific dependency versions, here we will use qemu **2.9.0**, which we know that works.

NOTE: This section is mainly based on [3] [5], with small modifications for our specific case.

[Step 1] Open a terminal and create a folder to hold qemu repo:

```
$sudo -s
$mkdir /opt/buffer
$cd /opt/buffer
```

[Step 2] Get the source code:

```
$wget http://download.qemu-project.org/qemu-2.9.0.tar.xz
$tar -xvf qemu-2.9.0.tar.xz
$cd qemu-2.9.0
$mkdir build
$cd build
$apt-get install zlib1g-dev libglib2.0-dev libpixman-1-dev libfdt-dev
$./../configure --prefix=/opt/qemu --target-list=arm-softmmu,arm-linux-user --enable-debug
$make -s
$make install -s
```

[Step 3] Add qemu executable to PATH

```
$nano /etc/profile
- Add the following line at the end of file:
PATH=/opt/qemu/bin:$PATH
Ctrl-X -> Y -> Enter
- Manually launch script:
$. /etc/profile
```



1.5 Run U-Boot on QEMU (versatile target)

At this point we will launch a basic u-boot target build with q-emu as example. The BeagleBone is not supported by the 'raw' version of QEMU, hence requiring further steps that will be described later on. Also as alternative, we will build u-boot using command line instead of the IDE in order to clarify both methods.

[Step 1] Go to the u-boot project directory

```
$cd /opt/projects/u-boot-2015.07
```

[Step 2] Create versatilepb target configuration and make u-boot.

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- versatilepb_config
```

```
$ make ARCH=arm CROSS_COMPILE=arm-none-linux-gnueabi- -s
```

The process shall finish with a warning that can be ignored:

```
===== WARNING =====  
Please convert this board to generic board.  
Otherwise it will be removed by the end of 2014.  
See doc/README.generic-board for further information  
=====
```

For now, we will focus on the generated u-boot.bin file [6], which is the u-boot binary that we will load with qemu.

[Step 3] Finally, start u-boot with qemu:

```
$qemu-system-arm -M versatilepb -nographic -kernel u-boot
```

The following should be displayed on the screen:

```
root@xubuntu-VirtualBox:/opt/projects/u-boot-2015.07# qemu-system-arm -M versatilepb -  
nographic -kernel u-boot  
audio: Could not init `oss' audio driver  
  
U-Boot 2015.07-gdd01efc (Jul 05 2017 - 09:53:57 -0400)  
  
DRAM: 128 MiB  
WARNING: Caches not enabled  
Flash: Flash protect error at address 37ec0000  
Flash protect error at address 37fc0000  
64 MiB  
*** Warning - bad CRC, using default environment  
  
In: serial  
Out: serial  
Err: serial  
Net: SMC91111-0  
Warning: SMC91111-0 using MAC address from net device  
Warning: Your board does not use generic board. Please read  
doc/README.generic-board and take action. Boards not  
upgraded by the late 2014 may break or be removed.  
VersatilePB #
```



NOTE: We've set arm-none-linux-gnueabi- as the prefix for the compiler (gcc), meaning that u-boot was cross-compiled for the arm architecture. Also, qemu was executed using qemu-system-arm, which is dedicated to the emulation of ARM machines.

Type help + ENTER in the u-boot command prompt for a list of available commands. You can now try and run some commands on your own, edit the source code, re-build and see the output, customize it, or whatever desired. This build can work as a test bench to help you further understand the bootloader.

```
VersatilePB # bdfinfo
arch_number = 0x00000183
boot_params = 0x00000100
DRAM bank   = 0x00000000
-> start     = 0x00000000
-> size      = 0x08000000
eth0name    = SMC91111-0
ethaddr     = 52:54:00:12:34:56
current eth = SMC91111-0
ip_addr     = <NULL>
baudrate    = 38400 bps
TLB addr    = 0x07FF0000
relocaddr   = 0x07FD1000
reloc off   = 0x06FD1000
irq_sp      = 0x07FACF3C
sp start    = 0x07FACF30
VersatilePB # md 0x00000000
00000000: ea0000be e59ff014 e59ff014 e59ff014 .....
00000010: e59ff014 e59ff014 e59ff014 e59ff014 .....
00000020: 07fd1060 07fd10c0 07fd1120 07fd1180 \.....
00000030: 07fd11e0 07fd1240 07fd12a0 deadbeef ....@.....
00000040: 00000000 00000000 00000000 00000000 .....
00000050: 00000000 00000000 00000000 00000000 .....
00000060: 00000000 00000000 00000000 00000000 .....
00000070: 00000000 00000000 00000000 00000000 .....
00000080: 00000000 00000000 00000000 00000000 .....
00000090: 00000000 00000000 00000000 00000000 .....
000000a0: 00000000 00000000 00000000 00000000 .....
000000b0: 00000000 00000000 00000000 00000000 .....
000000c0: 00000000 00000000 00000000 00000000 .....
000000d0: 00000000 00000000 00000000 00000000 .....
000000e0: 00000000 00000000 00000000 00000000 .....
000000f0: 00000000 00000000 00000000 00000000 .....
VersatilePB #
```

A list of supported machines (targets) by qemu-system-arm can be obtained by executing the following command in the linux terminal:

```
$ qemu-system-arm -machine help
```

```
root@xubuntu-VirtualBox:/opt/projects/u-boot-2015.07# qemu-system-arm -machine help
Supported machines are:
akita           Sharp SL-C1000 (Akita) PDA (PXA270)
ast2500-evb     Aspeed AST2500 EVB (ARM1176)
borzoi          Sharp SL-C3100 (Borzoi) PDA (PXA270)
```



canon-a1100	Canon PowerShot A1100 IS
cheetah	Palm Tungsten E aka. Cheetah PDA (OMAP310)
collie	Sharp SL-5500 (Collie) PDA (SA-1110)
connex	Gumstix Connex (PXA255)
[...]	

It can be seen right away that the 'beagle' is not listed. Running the 'beagle' target with qemu will be covered in section 3.4.

NOTE: It is suggested to go through the steps in section 1.1 in order to create an ECLIPSE development environment for the versatile target. It will only be necessary to change [Step 19] with Target Name: versatilepb_config. If you went through that section already, simply add this target configuration as stated in [Step 19]. You'll end-up with three targets:

- distclean
- am335x_evm_config
- versatilepb_config

2 U-Boot Structure

This section describes the U-Boot structure, including the U-Boot code layout, configuration options used for compilation, and the boot process. In its essence, this section will fit pieces of the puzzle left in the previous section.

It will be assumed that the reader has access to a linux environment. The commands used throughout were tested in a xubuntu 12.04 environment as detailed in section 1.

2.1 Directory Hierarchy

The U-Boot source code contains a number of directories. We will focus on the ones listed in Table 1.

Directory	Description
/arch/arm	Files generic to ARM architecture
/board	Board dependent files
/cmd	U-Boot commands functions
/common	Misc architecture independent functions
/configs	Board default configuration files
/drivers/spi	SPI device drivers

Table 1: U-Boot Directory Structure

2.2 Configuration Options

The target board considered herein is endowed of the AM3358 processor. Hence, the corresponding default configuration file “am335x_evm_defconfig” can be found under /configs.

NOTE: From this point on we will use the **arm-linux-gnueabi-** cross-compiler instead of **arm-none-gnueabi-** instructed in [Step 16] of section 1.1. This is because the latter is needed for the versatile target only.

To install **arm-linux-gnueabi-**, do:

```
$sudo apt-get install gcc-arm-linux-gnueabi
```

Follow the instructions in [Step 16], section 1.1, to change the **CROSS_COMPILE** variable to **arm-linux-gnueabi-**.

Now, by invoking command:

```
$make CROSS_COMPILE=arm-linux-gnueabi- am335x_evm_defconfig
```

a configuration file `<.config>` will be generated in the project root directory. This file is a starting point for the target build configuration, defining which drivers will be supported, in which:

- Configuration `_OPTIONS_` (begin with “`CONFIG_`”);
- Configuration `_SETTINGS_` (begin with “`CONFIG_SYS_`”).

In order to automatically elaborate the `<.config>` file, `Kconfig` starts by checking the configuration options defined in `<configs/am335x_evm_defconfig>`, and resolves all related configurations that depend on those ones. In the end, all the target configuration options are collected in the `<.config>` file.

2.2.1 Compiling U-Boot Image

By invoking the command:

```
$make CROSS_COMPILE=arm-linux-gnueabi-
```




as instructed in 1.3 and/or 0, GNU make will perform the following major tasks:

- Create `<include/config/auto.conf>` which is generated by Kconfig;
- Create `<include/autoconf.mk>` which is used in the U-Boot conventional configuration;
- Build object files based on the configurations defined in the above to files;
- Link all the object files to create two ELF files: `<u-boot>`, `<spl/u-boot-spl>`;
- Create raw binary files: `<u-boot.bin>`, `<spl/u-boot-spl.bin>` and link map files `<u-boot.map>`, `<spl/u-boot-spl.map>`;
- Create image files: `<u-boot.img>`, `<MLO>`.

When Make starts building object files, it starts in the u-boot root directory. Then, it recursively executes sub-make in each subdirectory using the Makefile in the corresponding folder. A source file under a sub-folder is only considered for compilation if the configuration option (`CONFIG_`) of the object is defined in either `<include/config/auto.conf>` or `<include/autoconf.mk>`. Taking Makefile under `<drivers>` as an example, 'obj-y' means that the corresponding object file is always built, while `obj-$(CONFIG_OPTION)` will be built into object file and further linked, depending on if the `CONFIG_OPTION` is defined.

In the current version of U-BOOT (2015-07), developers are in the process of migrating board headers to Kconfig. As a result, part of the configuration options (those defined in .config file) end up in `<include/config/auto.conf>` while the rest still stays in board headers, which are then collected into `<include/autoconf.mk>`. For example, all configurations defined in `<include/configs/am335x_evm.h>` and `<include/configs/ti_armv7_common.h>` can be found in `<include/autoconf.mk>`, in turn generated at build time.

Alternatively, by invoking make with:

```
$make CROSS_COMPILE=arm-linux-gnueabi- V=1
```

The GNU make will provide a verbose output of the make process, which can be useful for debug purposes.

NOTE: The build process can be integrated in a suitable IDE such as Eclipse. Instructions to setup the development environment are provided in section 1.1.

2.3 Boot Process

After power-up or reset, the processor loads the U-Boot boot loader in several steps [14]. Below is the boot process, specifically for AM335x targets:

1. ROM code is executed after power-on;
2. ROM looks for U-Boot SPL (Secondary Program Loader) whose filename is `<MLO>`, and copies the content of MLO to an on-chip SRAM at address defined in `CONFIG_SPL_TEXT_BASE` (0x402F0400), which is hardware-specific. This U-Boot SPL stage is required due to the limited size of this SRAM.
3. U-Boot SPL starts initialization tasks: execution starts with architecture-specific ASM instructions set in `<start.S>`. The latter is located in `<arch/arm/cpu/armv7>`. From here, three functions are called:
 - a. **lowlevel_init():**
 - setup PLL, mux, and clocks.
 - defined in `<arch/arm/cpu/armv7/lowlevel_init.S>`
 - b. **board_init_f():**
 - init hardware for execution from SDRAM.
 - defined in `<arch/arm/cpu/armv7/omapcommon/hwinit_common.c>`

- c. **board_init_r():**
- load the <U-Boot.img> at into SDRAM at address defined in CONFIG_SYS_TEXT_BASE (0x80800000) and execute from there.
 - defined in <common/spl/spl.c>
4. U-boot is loaded into RAM, started, and in turn configures the Ethernet MAC address, flash, serial console, and loads the settings. The U-Boot environment setting is a block of memory that is kept in NVM.
 5. By pressing a key from serial console, autoboot to next stage (e.g. linux kernel) is interrupted, and the U-Boot command line console is displayed. The result should be that of Figure 9.

```
U-Boot SPL 2015.07 (Jul 03 2017 - 11:07:43)
reading args
spl_load_image_fat_os: error reading image args, err - -1
reading u-boot.img
reading u-boot.img

U-Boot 2015.07 (Jul 03 2017 - 11:07:43 +0100)

        Watchdog enabled
I2C:   ready
DRAM:  512 MiB
NAND:  0 MiB
MMC:   OMAP SD/MMC: 0, OMAP SD/MMC: 1
reading uboot.env

** Unable to read "uboot.env" from mmc0:1 **
Using default environment

Net:   <ethaddr> not set. Validating first E-fuse MAC
Phy 0 not found
cpsw, usb_ether
Hit any key to stop autoboot:  0
U-Boot#
```

Figure 9: U-Boot command line console on Beagle board

3 U-Boot Customization

This section describes how to adapt u-boot to a specific hardware requirement. As example, we will develop and integrate a driver to perform R/W operations to a 4Mb SPI MRAM (MR25H40), considered attached to the McSPI module of the AM3358 processor.

Last, we investigate the possibility to run U-Boot on QEMU (a generic and open source machine emulator).

3.1 Implementation of an External Memory SPI Driver

3.1.1 Hardware and Software Requirements

The requirements for this example are the following:

Hardware perspective:

A 4Mb SPI Interface MRAM (MR25H40) chip is attached to the McSPI module of a AM335x processor.

Software (u-boot) perspective:

A SPI driver is needed to perform R/W test operations to the MRAM device.

3.1.2 U-Boot Drivers (SPI Example)

In directory <drivers>, there are commonly used device drivers which implement a set of functions for accessing certain on-board hardware devices. So for the current task, it is first necessary to check if the SPI driver is available for the McSPI module, otherwise it needs to be created.

From Makefile in directory <drivers/spi/Makefile>, we get:

```
[...]  
obj-$(CONFIG_MXS_SPI) += mxs_spi.o  
obj-$(CONFIG_OMAP3_SPI) += omap3_spi.o  
obj-$(CONFIG_SANDBOX_SPI) += sandbox_spi.o  
obj-$(CONFIG_SH_SPI) += sh_spi.o  
[...]
```

By checking <include/autoconf.mk> and <include/config/auto.conf>, we can see that CONFIG_OMAP3_SPI is defined, therefore, **omap3_spi.c** is compiled and made available for the McSPI module of the hardware specification. An alternative way to find if a driver is compiled or not is to check <u-boot.cfg> which is generated during Make process, providing all the preprocessor definitions used during compilation. This information can also be verified by the Makefile output.

3.1.3 U-Boot Commands (SPI Example)

In directory <cmd> there are API commands that can be called from the U-Boot shell, which typically perform calls to a device driver in order to execute higher-level functions. In our case, both the SPI driver and CONFIG_CMD_SPI are defined. An explanation follows on [how the SPI utility command interacts with the SPI driver](#).

As previously discussed, all U-Boot command functions are under directory <common>, with prefix cmd_. In the Makefile kept in directory <common/Makefile> we have:

```
[...]  
obj-$(CONFIG_CMD_SOFTSWITCH) += cmd_softswitch.o  
obj-$(CONFIG_CMD_SPI) += cmd_spi.o  
obj-$(CONFIG_CMD_SPIBOOTLDR) += cmd_spibootldr.o  
[...]
```

Therefore, with CONFIG_CMD_SPI defined, the SPI utility command in file <common/cmd_spi.c> will be compiled. At the end of this file, we have the following code:

```
[...]
Line 165: U_BOOT_CMD(
           sspi,    5,    1,    do_spi,
           "SPI utility command",
[...]
```

The U_BOOT_CMD is a macro used to fill in a cmd_tbl_t structure (Command Table), and is defined in <include/command.h>

```
[...]
#define U_BOOT_CMD(_name, _maxargs, _rep, _cmd, _usage, _help) \
    U_BOOT_CMD_COMPLETE(_name, _maxargs, _rep, _cmd, _usage, _help, NULL)
[...]
```

The five macro arguments stand for [16]:

_name:	the name of the command. Not a string but the text of the command name.
_maxargs:	the maximum number of arguments this function takes
_repeatable:	either 0 or 1 to indicate if autorepeat is allowed
_command:	Function pointer (*cmd)(struct cmd_tbl_s *, int, int, char *[]);
_usage:	Short description. This is a string
_help:	Long description. This is a string

The cmd_tbl_t structure created is named with a special prefix and placed by the linker in a special section using the linker lists mechanism <include/linker_lists.h>. This makes it possible for the final link to extract all commands compiled into any object code and construct a static array so the command array can be iterated over using the linker lists macros.

In our case, do_spi(cmd_tbl_t *, int, int, char * const) function is stored in the linker-generated array placed in the ".u_boot_list" section so that user can use the "sspi" command from U-Boot console.

The SPI utility command requires 5 arguments: "[<bus>:] <cs> [. <mode>] <bit_len> <dout>", which stands for:

<bus>	- Identifies the SPI bus\n"
<cs>	- Identifies the chip select\n"
<mode>	- Identifies the SPI mode to use\n"
<bit_len>	- Number of bits to send (base 10)\n"
<dout>	- Hexadecimal string that gets sent"

The do_spi() function parses the command line parameters, given as arguments:

```
[...]
Line 105: int do_spi (cmd_tbl_t *cmdtp, int flag, int argc, char * const argv[])
[...]
```

and then calls do_spi_xfer(bus, cs), in turn defined in line 43 of the same source file:

```
[...]
Line 43: static int do_spi_xfer(int bus, int cs)
[...]
```

From within this function, the driver API is called to perform the SPI transfer. As it can still be seen from the command code, 5 SPI driver functions are called:

```
[...]
Line 59: spi_setup_slave(bus, cs, 1000000, mode);
[...]
Line 66: spi_claim_bus(slave);
[...]
Line 69: spi_xfer(slave, bitlen, dout, din,
               SPI_XFER_BEGIN | SPI_XFER_END);
[...]
Line 86: spi_release_bus(slave);
[...]
Line 88: spi_free_slave(slave);
[...]
```

3.1.4 U-Boot Driver Calls (SPI Example)

All of the above SPI driver functions are already implemented in <drivers/spi/omap3_spi.c>:

spi_setup_slave():

```
[...]
Line 61: struct spi_slave *spi_setup_slave(unsigned int bus, unsigned int cs,
Line 62:         unsigned int max_hz, unsigned int mode)
[...]
```

spi_setup_slave(...) function sets up communications parameters for a SPI slave and return a pointer to the spi_slave structure which is a representation of the SPI slave.

The arguments have the following meanings:

Bus:	- Bus ID of the slave chip.
Cs:	- Chip select ID of the slave chip on the specified bus.
max_hz:	- Maximum SCK rate in Hz.
Mode:	- Clock polarity, clock phase and other parameters.

spi_claim_bus():

```
[...]
Line 141: int spi_claim_bus(struct spi_slave *slave)
[...]
```

spi_claim_bus(...) function claims the bus and prepares for communication with the given slave. This function will read the up communications parameters inside the slave and write the McSPI configuration to the module, such as setup clock divisor, wordlength, chipselect polarity, SPI mode and transmit & receive mode.

spi_xfer():

```
[...]
Line 404: int spi_xfer(struct spi_slave *slave, unsigned int bitlen,
Line 405:         const void *dout, void *din, unsigned long flags)
[...]
```

spi_xfer (...) function is the core function of the data transaction between the SoC McSPI module and SPI slave. This function writes “bitlen” bits out the SPI MOSI port and simultaneously clocks “bitlen” bits in the SPI MISO port.

The arguments have the following meanings:

slave:	- The SPI slave which will be sending/receiving the data.
bitlen:	- Number of bits to write and read.
dout:	- Pointer to a byte array of data to send out.
din:	- Pointer to a byte array that will be filled in with data received.
Flags:	- A bitwise combination of SPI_XFER_* flags



The available transfer flags are defined in <include/spi.h>:

```
[...]  
/* SPI transfer flags */  
#define SPI_XFER_BEGIN      0x01    /* Assert CS before transfer */  
#define SPI_XFER_END        0x02    /* Deassert CS after transfer */  
#define SPI_XFER_MMAP       0x08    /* Memory Mapped start */  
#define SPI_XFER_MMAP_END   0x10    /* Memory Mapped End */  
#define SPI_XFER_ONCE       (SPI_XFER_BEGIN | SPI_XFER_END)  
#define SPI_XFER_U_PAGE     (1 << 5)  
[...]
```

spi_release_bus():

```
[...]  
Line 217: void spi_release_bus(struct spi_slave *slave)  
[...]
```

spi_release_bus(...) function must be called once for every call to spi_claim_bus() after all transfers have finished. In this function, McSPI module will be reset by writing the reset configuration to appropriate registers of the module.

spi_free_slave():

```
[...]  
Line 134: void spi_free_slave(struct spi_slave *slave)  
[...]
```

spi_free_slave() function then will free the SPI slave structure.

Other SPI driver functions can be found in the McSPI driver source code file, however the sspi command only calls the 5 functions described above. In fact, during U-Boot linking stage, the linker will discard the unreferenced functions and produces two link map files <u-boot.map>, <spi/uboot-spi.map> which are useful to find out where the functions are being used.

For example, as can be seen in <u-boot.map>, spi_cs_is_valid, spi_cs_activate and spi_cs_deactivate functions are in fact discarded after linking:

```
[...]  
Line 14: Discarded input sections  
[...]  
.text.spi_cs_is_valid  
0x0000000000000000      0x4 drivers/spi/built-in.o  
.text.spi_cs_activate  
0x0000000000000000      0x2 drivers/spi/built-in.o  
.text.spi_cs_deactivate  
0x0000000000000000      0x2 drivers/spi/built-in.o  
[...]  
Line805: Linker script and memory map  
[...]  
.text.spi_init  
0x00000000808197ca      0x2 drivers/spi/built-in.o  
0x00000000808197ca      spi_init  
.text.spi_setup_slave  
0x00000000808197cc      0xbc drivers/spi/built-in.o  
0x00000000808197cc      spi_setup_slave  
.text.spi_free_slave  
0x0000000080819888      0x4 drivers/spi/built-in.o  
0x0000000080819888      spi_free_slave  
.text.spi_claim_bus  
0x000000008081988c      0xb4 drivers/spi/built-in.o  
0x000000008081988c      spi_claim_bus
```

.text.spi_release_bus	0x0000000080819940	0x1c drivers/spi/built-in.o
	0x0000000080819940	spi_release_bus
.text.omap3_spi_write	0x000000008081995c	0x124 drivers/spi/built-in.o
	0x000000008081995c	omap3_spi_write
.text.omap3_spi_read		

3.2 Creating an API (MR25H40 drive case)

After understanding the SPI driver function, the goal now is to implement the Read and Write functions to be able to perform data transactions from/to the MR25H40 device. These two functions will have the following prototypes:

```
ssize_t mram_spi_read(uchar *addr, int alen, uchar *buffer, int len)
```

```
ssize_t mram_spi_write(uchar *addr, int alen, uchar *buffer, int len)
```

where the four arguments stand for:

uchar *addr:	- the char pointer to the Mram Read/Write address.
int alen:	- the address size in bytes.
uchar*buffer:	- pointer to the buffer to read or write from.
int len:	- buffer size in bytes.

3.2.1 Mram SPI Read

From the MR25H40 datasheet [8], the Read Data Bytes (READ) command allows data bytes to be read starting at an address specified by the 24-bit address field. Only address bits 0-18 are decoded by the memory, which is because this device is organized as 524,288 words of 8 bits which can be addressed by 19bits. The data bytes are read out sequentially from memory until the read operation is terminated by bringing CS high. The entire memory can be read in a single command. The read operation at the bus level is shown in Figure 10.

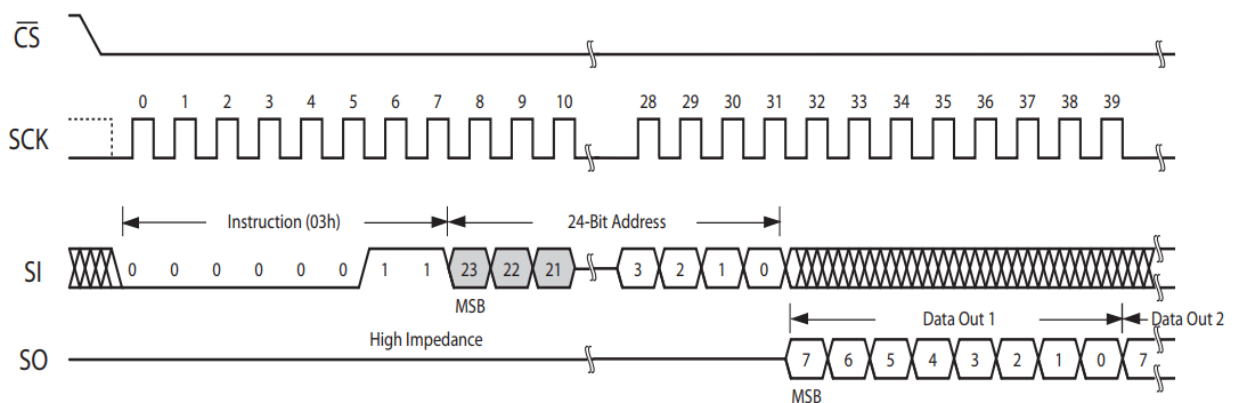


Figure 10: SPI Read operation from MR25H40

Therefore, to perform a read operation, we need to do the following steps:

1. Call `spi_setup_slave(...)` to setup configuration for MR25H40.

Here we need to determine the bus number, chip select number, SCK rate and SPI mode. By reading the data sheet of AM335x and MR25H40, we've chosen the bus number to be 0, chip select to be 0, SCK rate to 48MHz, SPI mode to 0 (CPOL=0, CPHA=0)

2. Call `spi_claim_bus(...)` to configure the McSPI module.

By calling `spi_claim_bus (...)`, we pass the `spi` slave struct which return by `spi_setup_slave(...)` in, and let this driver function to setup configuration for McSPI module and prepare for the data transactions.

3. Call `spi_xfer(...)` to perform SPI Read data transactions.

As can be seen in Figure 10, to read from MR25H40, we need to first bring CS low and write READ instruction code and 24bit read address through Serial Input and read out data from serial out.

To better clarify the Mram SPI read procedure, the call to `spi_xfer(...)` is made using the following steps:

```
/*write the READ instruction code to MR25H40, bring CS low;
 * SPI_XFER_BEGIN will indicate the spi_xfer function to bring CS low;
 * the spi_xfer din is set to NULL, and dout is the read command;
 * spi_xfer Will write through serial input with the read instruction code*/
spi_xfer(slave, 8, &cmd_read, NULL, SPI_XFER_BEGIN)

/*write the read address to MR25H40,*/
spi_xfer(slave, 24, read_address, NULL, 0);

/*read data from MR25H40 with the specific length and bring CS high*/
spi_xfer(slave, 8 * len, NULL, buffer, SPI_XFER_END)
```

4. Call `spi_release_bus(...)` to reset McSPI module, as explained in previous chapter.
5. Call `spi_free_slave(...)` to free up the memory of the `spi` slave struct created by `spi_setup_slave(...)`.

3.2.2 Mram SPI Write

Similarly, the Write Data Bytes (WRITE) command allows data bytes to be written, starting at an address specified by the 24-bit address. Again, only address bits 0-18 are decoded by the memory. The data bytes are written sequentially in memory until the write operation is terminated by bringing CS high. The entire memory can be written in a single command. The address counter will roll over to 0000h when the address reaches the top of memory.

The only difference compared to EEPROM or Flash is that data bytes can be written continuously without write delays or data polling. Back to back write commands can be executed without any write delay. The read operation at the bus level is shown in Figure 11.

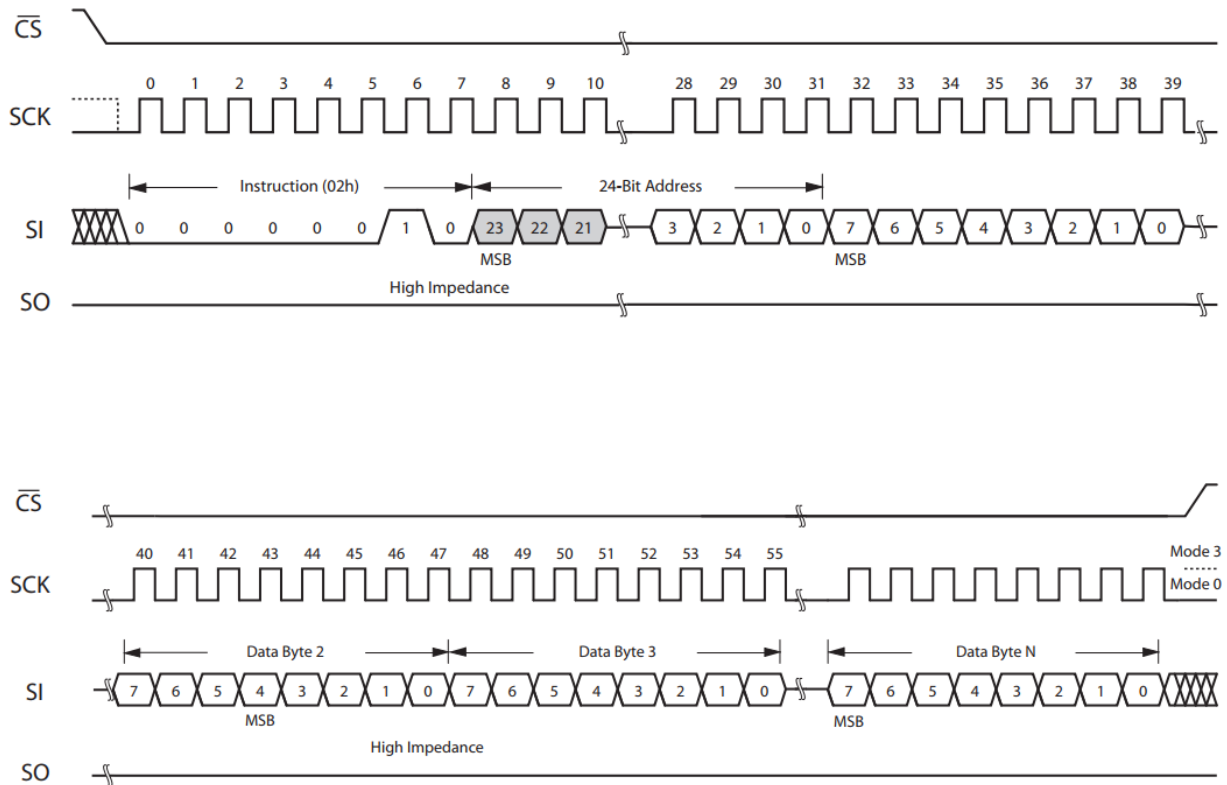


Figure 11: SPI Write operation from MR25H40

Similar to the Mram SPI read, the Mram SPI write calls to SPI driver API function with a similar procedure:

1. spi_setup_slave(...)
2. spi_claim_bus(..)
3. spi_xfer(...)
4. spi_release_bus(...)
5. spi_free_slave(...)

As can be seen from Figure 11, the difference is the call procedure of the spi_xfer(...):

First, we need to make sure the write enable latch bit is enabled by sending a write enable command. We can only send the write command and write address to the MRAM after the write is enabled.

Second, when we start writing data, the spi_xfer() dout argument should be set to NULL, and din should point to the byte buffer where the data needs to be sent.

```

/* bring CS low, write the Write Enable(WREN)instruction code to MR25H40
 * and then bring CS high. */
spi_xfer(slave, 8, &cmd_write_enable, NULL, SPI_XFER_BEGIN | SPI_XFER_END)

/* bring CS low, write the WRITE instruction code to MR25H40, */
spi_xfer(slave, 8, &cmd_write, NULL, SPI_XFER_BEGIN)

/*write the read address to MR25H40, bring CS low*/
spi_xfer(slave, 24, addr, NULL, 0);
    
```

```
/*read data from MR25H40 with the specific length and bring CS high*/  
spi_xfer(slave, 8 * len, NULL, buffer, SPI_XFER_END)
```

3.3 Integrating Mram API Into U-Boot

Since the Mram SPI Read and Write Functions are not implemented as U-Boot CMD, these two functions must be called from the driver API layer. However, the SPI Read and Write functions are implemented specifically for MR25H40 device, hence acting more like a driver for MR25H40. Thus, we create the file `<mr25h40.c>` and place it under directory `<drivers/mtd/spi>`.

To compile the new source code with U-Boot, the make file `<drivers/mtd/spi/Makefile>` needs to be modified:

```
[...]  
Line 24: obj-$(CONFIG_SPI_MR25H40) += mr25h40.o  
[...]
```

By adding this line, GNU Make will compile the API source if `CONFIG_SPI_MR25H40` is defined.

As stated above in sub-section 2.2.1, there are two ways of adding this configuration option:

1. Add it to the Kconfig file structure:

Add the following lines in to `<drivers/mtd/spi/Kconfig>`:

```
[...]  
config SPI_MR25H40  
    bool "MR25H40 SPI Flash support"  
    help  
        Enable SPI flash support for MR25H40 Device to perform read / write.  
[...]
```

and enable it in `<configs/am335x_evm_defconfig>`:

```
[...]  
Line 7: CONFIG_SPI_MR25H40=y  
[...]
```

2. Add to board specific header `<am335x_evm.h>`:

```
[...]  
#define CONFIG_SPI_MR25H40  
[...]
```

Finally, we can confirm that our changes have been successfully integrated into the U-Boot Image, either by checking the Make output, `<u-boot.map>` or `<u-boot.cfg>` after build:

Makefile Output:

```
[...]  
LD      drivers/mtd/built-in.o  
LD      drivers/mtd/onenand/built-in.o  
CC      drivers/mtd/spi/mr25h40.o  
LD      drivers/mtd/spi/built-in.o  
[...]
```

<u-boot.map>:

```
[...]
.text.mram_spi_read
        0x0000000000000000      0x7c drivers/mtd/spi/built-in.o
.text.mram_spi_write
        0x0000000000000000      0x98 drivers/mtd/spi/built-in.o
[...]
```

<u-boot.cfg>:

```
[...]
#define CONFIG_LIB_UUID
#define CONFIG_SPI_MR25H40 1
#define __FLT_MANT_DIG__ 24
#define CONFIG_G_DNL_MANUFACTURER "Texas Instruments"
[...]
```

Next, we can call the MRAM SPI driver functions from the U-Boot command line by creating two U-Boot console command `mramrd`, `mramwr` using the “U_BOOT_CMD” Marco.

```
[...]
U_BOOT_CMD(
    mramrd, 4,      1,      do_spi_mram_read,
    "SPI MRAM command,Read from MRAM MR25H40 device",
    "<bit_len> <addr> - Read <bit_len> bits from MRAM <addr>\n"
    "<bit_len> - Number of bits to receive (base 10)\n"
    "<addr> - MRAM Read address"
);

U_BOOT_CMD(
    mramwr, 5,      1,      do_spi_mram_write,
    "SPI MRAM command,Write to MRAM MR25H40 device",
    "<bit_len> <addr> <dout> - Send and receive bits\n"
    "<bit_len> - Number of bits to send (base 10)\n"
    "<addr> - MRAM Write address\n"
    "<dout> - Hexadecimal string that gets sent"
);
[...]
```

When we issue “`mramrd`” or “`mramwr`” command from the U-Boot command line, the corresponding `do_spi_mram_read()/do_spi_mram_write()` is called. These two functions can be seen as the interface between the command line (user) and the driver API. Within these functions, arguments collected from the console are parsed and MRAM SPI Read/Write are called. This is similar to `do_spi(...)` function as explained earlier in section 3.4.1. For convenience, these two commands are placed in source file `<common/cmd_spi.c>`.

Finally, we need to verify that the API was successfully included in the image by running it on the target. This will be done by:

1. Running our u-boot build on qemu;
2. Running our u-boot build on a BeagleBone Green target board.

The first will be covered in the following section.



3.4 Running U-Boot am335x Target on QEMU

QEMU is a versatile generic and open source machine emulator. If QEMU can emulate our hardware requirement, we could then test the U-Boot image before running on real hardware. Unfortunately, as seen in section 1.5, no similar hardware board can be found in the supported machines.

Another alternative is explored, Linaro QEMU, which is a branch of QEMU focused on improving support for ARM based systems. Linaro QEMU supports Beagleboard.

3.4.1 Installing qemu-linaro

The following instructions can be used to install qemu-linaro:

```
$sudo -s
$cd ~/buffer
$git clone https://github.com/firmadyne/qemu-linaro.git
$cd qemu-linaro
$git submodule update --init pixman
$git submodule update --init dtc
$./configure --prefix=/opt
NOTE: The above line is necessary in order to create the binary under /opt folder.
$make
$make install
```

Contrary to the versatilepb example in section 1.5, for the beagle target we need to mimic the actual hardware process, which is to load the software from a dedicated SD card.

A set of tools is also necessary to create the SD image for QEMU to boot, that can be obtained with:

```
$add-apt-repository ppa:linaro-maintainers/tools
$aapt-get update
$aapt-get install linaro-image-tools
```

To create the SD image for QEMU, both hardware pack and kernel image files are required. The hardware pack can be created based on an existing hwpack config file <linaro-beaglebone>. This configuration file contains the information about what packages and dependencies should be installed to create the hardware pack tarball.

In order to get the hardware pack (hwpack) config file, do:

```
$sudo apt-get install bzip2
$cd /opt/buffer
$bzip2 branch lp:~linaro-maintainers/linaro-images/hwpack.precise.linaro-beagleboard
```



And to create the hwpack based on the obtained config file, do:

```
$cd hwpack.precise.linaro-beagleboard  
$linaro-hwpack-create hwpacks/linaro-beagleboard 2
```

This last command will output a file named `hwpack_linaro-omap3_2_armhf_supported.tar.gz`. Since we are only interested in the U-Boot Stage, the kernel image can be just a stub file.

To create a stub image file, do:

```
$touch stub  
$tar -cvf stub.tar.gz stub
```

Now we are ready to create the SD image, replacing `$KERNEL` and `$HWPACK` with the tarball files we created:

```
$linaro-media-create --binary $KERNEL --hwpack $HWPACK --dev beagle
```

Specifically:

```
$KERNEL -> stub.tar.gz  
$HWPACK -> hwpack_linaro-omap3_2_armhf_supported.tar.gz
```

NOTE: Both `$KERNEL` and `$HWPACK` are in tarball format.

An image file named `<sd.img>` is created. Now, to finally boot the u-boot beagle target with qemu, run:

```
$/opt/bin/qemu-system-arm -M beagle -drive if=sd,cache=writeback,file=./sd.img -serial stdio
```

And the following will be obtained in the console:

```
[...]  
U-Boot SPL 2012.07 (Jun 20 2017 - 13:25:39)  
OMAP SD/MMC: 0  
reading u-boot.img  
U-Boot 2012.07 (Jun 20 2017 - 13:25:39)  
OMAP35XX-GP ES3.1, CPU-OPP2, L3-165MHz, Max CPU Clock 600 mHz  
OMAP3 Beagle board + LPDDR/NAND  
I2C: ready  
DRAM: 256 MiB  
NAND: 256 MiB  
MMC: OMAP SD/MMC: 0  
*** Warning - bad CRC, using default environment  
ERROR : Unsupport USB mode  
Check that mini-B USB cable is attached to the device  
In: serial  
Out: serial  
Err: serial  
Beagle Rev C4  
No EEPROM on expansion board  
Die ID #51454d5551454d55540000051454d55  
Net: No ethernet found.  
checking for preEnv.txt
```



```
reading preEnv.txt
** Unable to read file preEnv.txt **
Hit any key to stop autoboot: 0
OMAP3 beagleboard.org #
[...]
```

In fact, `linaro-media-create` is joining the `<MLO>` and `<u-boot.img>` with kernel image to create the complete SD image for QEMU to emulate. If we run QEMU with:

```
$/opt/bin/qemu-system-arm -M beagle -drive if=sd,cache=writeback,file=MLO -serial stdio
```

Only the MLO code will be executed on QEMU.

```
[...]
U-Boot SPL 2012.07 (Jun 20 2017 - 13:25:39)
OMAP SD/MMC: 0
** Partition 1 not valid on device 0 **
spl: fat register err - -1
### ERROR ### Please RESET the board ###
[...]
```

The messages mean that QEMU complains about the lack of next stage boot loading.

We tried to replace the u-boot image and MLO in the Linaro hwpack by our own u-boot build, without success. The MLO execution was interrupted, not reaching the state of loading the u-boot image. This may be due to differences in the used cross-compile toolchain and/or the source code version in the linaro package.

In order to identify the differences, we obtained the u-boot source version used in the Linaro hwpack ([git://git.linaro.org/boot/u-boot-linaro-stable.git](https://git.linaro.org/boot/u-boot-linaro-stable.git)) and cross-compiled it with our toolchain. It worked, and therefore we concluded that **the difference between U-Boot source code versions (2013.07 and 2015.07) is the cause for our version not being able to load on QEMU Linaro.**

As such, we compiled the MRAM SPI driver code together with the custom U-Boot command with U-Boot-2013.07, so that we could verify the integration with the emulator. Figure 12 depicts a screenshot of QEMU with "mramrd" and "mramwr" integrated into U-Boot.

```

ERROR : Unsupport USB mode
Check that mini-B USB cable is attached to the device
In:    serial
Out:   serial
Err:   serial
Beagle Rev C4
No EEPROM on expansion board
Die ID #51454d5551454d555400000051454d55
checking for preEnv.txt
Unknown command 'mmc' - try 'help'
Hit any key to stop autoboot:  0
OMAP3 beagleboard.org # help
?      - alias for 'help'
base   - print or set address offset
bdinfo - print Board Info structure
boot   - boot default, i.e., run 'bootcmd'
bootd  - boot default, i.e., run 'bootcmd'
bootm  - boot application image from memory
cmp    - memory compare
coninfo - print console devices and information
cp     - memory copy
crc32  - checksum calculation
echo   - echo args to console
editenv - edit environment variable
env    - environment handling commands
exit   - exit script
false  - do nothing, unsuccessfully
fdt    - flattened device tree utility commands
go     - start application at address 'addr'
help   - print command description/usage
imxtract - extract a part of a multi-image
itest  - return true/false on integer compare
loadb  - load binary file over serial line (kermit mode)
loads  - load S-Record file over serial line
loady  - load binary file over serial line (ymodem mode)
loop   - infinite loop on address range
md     - memory display
mm     - memory modify (auto-incrementing address)
mramrd - SPI utility command,Read from MRAM MR25H40 device
mramwr - SPI utility command,Write to MRAM MR25H40 device
mtest  - simple RAM read/write test
mw     - memory write (fill)
nand   - NAND sub-system
nandecc - switch OMAP3 NAND ECC calculation algorithm

```

Figure 12: U-Boot 2013.07 running in qemu-linaro with custom commands integrated.

The usage of a command can always be checked with:

```
$help $COMMAND
```

Figure 13 depicts the usage for our custom “mramrd” and “mramwr” commands.

```
OMAP3 beagleboard.org # help mramrd
mramrd - SPI utility command,Read from MRAM MR25H40 device

Usage:
mramrd <bit_len> <addr> - Read <bit_len> bits from MRAM <addr>
<bit_len> - Number of bits to receive (base 10)
<addr>    - MRAM Read address
OMAP3 beagleboard.org # help mramwr
mramwr - SPI utility command,Write to MRAM MR25H40 device

Usage:
mramwr <bit_len> <addr> <dout> - Send and receive bits
<bit_len> - Number of bits to send (base 10)
<addr>    - MRAM Write address
<dout>    - Hexadecimal string that gets sent
OMAP3 beagleboard.org # mramwr 16 0x5 FFFF
MRAM write 16 bits from address 0x5...
FFFF
MRAM write failed
OMAP3 beagleboard.org # mramrd 16 0x5
MRAM read 16 bits from address 0x5...
16 bits read:
0000
OMAP3 beagleboard.org #
```

Figure 13: Usage of custom commands (mramrd, mramwr) shown in u-boot

Since MR25H40 device is not present in neither QEMU nor Beagle Board, these two commands are expected to fail when we call them.

3.5 Running U-Boot am335x Target on Hardware

Now we move to next stage, which is to run U-Boot on real hardware, the Beagle board. The HW setup is shown in Figure 14.

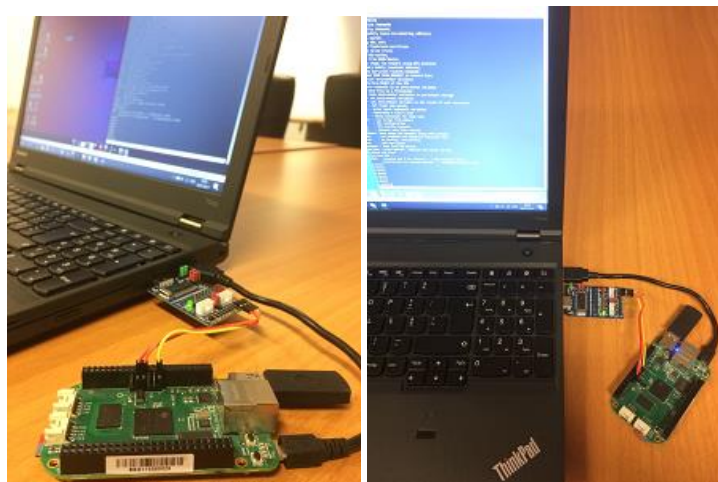


Figure 14: Physical Setup with the BeagleBone Green Board + FTDI

There was only one board available with multiple users needing access to it. The board was connected through FTDI to a serial port in a host PC and so, with SSH access, it was possible to use the u-boot serial console on the local PC.

```
$ssh -X svf@remotehostIp
```

In order to test a new u-boot image it was necessary to modify the uSD card inserted in the beagle and include the new u-boot .img. For this, the remote user would send the img file (created by default during build) to someone with physical access to the host computer and SD card in order to update it using a standard card reader and re-insert it on the beagle. Newly created MLO + u-boot.img files were passed to the host PC using:

```
$scp MLO svf@remotehostIp:/directory/to/store/image
```

```
$scp localu-boot.img svf@remotehostIp:/directory/to/store/image
```

To update the SD card, it was placed on a card reader and mounted on the Linux environment. Then, the files would be replaced in the first partition. The version running on the HW target was 2015.07, the same considered in section 1.

NOTE: Further u-boot builds only require u-boot .img to be passed and updated on the SD card.

Instructions to create a bootable SD card are given in section 4.

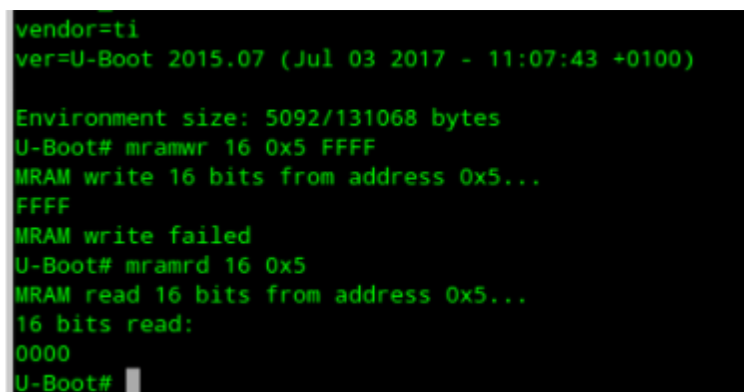
Access to the Beagle was made using a serial console in the Ubuntu VM (Putty) through an FTDI device set to 3V3, shown in Figure 14. The UART parameters are:

- **serial line:** /dev/ttyS0
- **baudrate:** 115200
- **data bits:** 8
- **Stop Bits:** 1
- **Parity Bit:** None
- **Flow Control:** XON/XOFF

It will also be necessary to setup a serial port in Virtual Box.

In Settings -> Serial Ports, check 'Enable Serial Port' with 'Port Number' set to COM1 and 'Port Mode' set to 'Host Device'. The 'Path/Address' field must be set to the FTDI COM number attributed by windows, which can be checked in the 'Device Manager' under 'Ports (COM & LPT)'. In our case, it was COM4.

Running this customized U-Boot image on beagle board, the output from "mramwr" and "mramrd" is exactly the same as that displayed on QEMU shown before. This can be seen in Figure 15.



```
vendor=ti
ver=U-Boot 2015.07 (Jul 03 2017 - 11:07:43 +0100)

Environment size: 5092/131068 bytes
U-Boot# mramwr 16 0x5 FFFF
MRAM write 16 bits from address 0x5...
FFFF
MRAM write failed
U-Boot# mramrd 16 0x5
MRAM read 16 bits from address 0x5...
16 bits read:
0000
U-Boot#
```

Figure 15: U-Boot 2015.07 running in Hardware with custom commands integrated.

NOTE: For the beagle case, the low-right corner ‘user’ button needs to be pressed at power-on time for the boot source to be redirected to the uSD card. The button is shown in Figure 16.



Figure 16: Button to press during power-up in order to switch the boot source to the uSD card.

3.5.1 Creating a Bootable microSD Card

In order to use the instructions below, the following items are necessary:

- A Linux u-boot development environment, similar to that of section 1, where both the u-boot .img and MLO can be created;
- A microSD (uSD) card;
- A USB SD card reader in order to format and further update the uSD card.

These steps need to be executed only once. They will format the SD card and create two partitions, one for the bootloader and another one for the Linux Kernel. We will be mainly focused on the first partition, which is the one concerning u-boot.

After innumerable tries, we were able to find a package from TI that contains a script that is really capable of formatting the uSD card with the exact requirements of our target processor. It is named “AM335X-LINUX-PSP-04.06.00.02” and is available from:

http://software-dl.ti.com/dsps/dsps_public_sw/psp/LinuxPSP/AM335x_04_06/index.html

We basically need the `mksd-am335x.sh` script contained inside this package. From it, we created a slightly different version, along with inputs from references [9], [10] and [11]. Reason is that in order to boot with u-boot alone, it isn't necessary to upload to the SD card the uImage and the file system associated with the OS. As such, the commands that reference these two ‘files’ have been removed from the script.

The custom script only requires 3 arguments: the device, the location of the MLO and u-boot .img files. As for the second partition required to load the OS, it is left empty.

The instructions to create a bootable microSD card are the following:



[Step 1] Create the script file:

```
$sudo -s
$cd /opt/buffer
$ nano mksd.sh
```

[Step 2] Paste the following code into it:

```
#!/bin/bash
if [[ -z $1 || -z $2 || -z $3 ]]
then
    echo "mksd Usage:"
    echo " mksd <device> <MLO> <u-boot.img> "
    echo " Example: mksd /dev/sdc MLO u-boot.img "
    exit
fi

if ! [[ -e $2 ]]
then
    echo "Incorrect MLO location!"
    exit
fi

if ! [[ -e $3 ]]
then
    echo "Incorrect u-boot.img location!"
    exit
fi

echo "All data on \"$1\" now will be destroyed! Continue? [y/n]"
read ans
if ! [ $ans == 'y' ]
then
    exit
fi

echo "[Partitioning $1...]"

DRIVE=$1
dd if=/dev/zero of=$DRIVE bs=1024 count=1024

SIZE=`fdisk -l $DRIVE | grep Disk | awk '{print $5}'`

echo DISK SIZE - $SIZE bytes

CYLINDERS=`echo $SIZE/255/63/512 | bc`

echo CYLINDERS - $CYLINDERS
{
echo ,9,0x0C,*
echo ,,,-
} | sfdisk -D -H 255 -S 63 -C $CYLINDERS $DRIVE

echo "[Making filesystems...]"
```



```
mkfs.vfat -F 32 -n boot "$1"1 &> /dev/null
mkfs.ext3 -L rootfs "$1"2 &> /dev/null

echo "[Copying files...]"

mount "$1"1 /mnt
cp $2 /mnt/MLO
cp $3 /mnt/u-boot.img
umount "$1"1

mount "$1"2 /mnt
chmod 755 /mnt
umount "$1"2

echo "[Done]"
```

[Step 3] Close and save changes:

```
Ctrl-X -> Y -> Enter
```

[Step 4] Identify the SD Device File

```
$ tail -f /var/log/kern.log
```

observe the log output which should be similar to this:

```
[...]
kernel: sd 35:0:0:0: [sdb] 61497344 512-byte logical blocks: (31.4 GB/29.3 GiB)
kernel: sd 35:0:0:0: [sdb] Write Protect is off
...
kernel: sdb: sdb1 sdb2 [...]
```

This means the SD card can be accessed under `/dev/sdb` (Our current card already has two partitions `sdb1` and `sdb2`)

NOTE: If the uSD does not appear automatically, it may be necessary to mount it again and repeat step2. We may mount it manually in alternative:

```
$mkdir /mnt/sdb

$mount /dev/sdb /mnt/sdb
```

[Step 5] Run `mkasd.sh`

```
$chmod 777 mkasd.sh

$ mkasd <device> <MLO> <u-boot.img>

example: mkasd.sh /dev/sdb MLO u-boot.img
```

NOTE: Remember to add the path to the files if they're not in the same folder as the `mkasd.sh` script.

This will format the uSD card with the exact requirements needed for the boot process and also, it will pass both MLO and u-boot image to the first partition automatically. This is what we need to properly run u-boot on the hardware target.

4 Integrating Standalone Applications

U-Boot supports standalone applications that are loaded dynamically. Such applications can have access to the U-Boot console I/O functions, memory allocation and interrupt services through the U-Boot Application Binary Interface (ABI) [14].

In this section we start by understanding the standalone “hello world” application already included with U-Boot source code. Then, we will port the source code of an external, widely-known application “Memtester” [12], into a standalone application, capable of being run with U-Boot, to demonstrate how to cross-compile a standalone application. The Beagle board will be used to validate the procedure in hardware.

4.1 Understanding the Hello World Application

The source code for “Hello World” application is in subdirectory <examples/hello_world.c>. It is automatically compiled when U-Boot is compiled. Looking at the make output we can find the following lines:

```
[...]  
CC      examples/standalone/stubs.o  
LD      examples/standalone/libstubs.o  
CC      examples/standalone/hello_world.o  
LD      examples/standalone/hello_world  
OBJCOPY examples/standalone/hello_world.srec  
OBJCOPY examples/standalone/hello_world.bin  
[...]
```

Using “make” with verbose mode, the following compiling and linking options can be found:

Compiling Options:

```
[...]  
-nostdinc -isystem /usr/lib/gcc/arm-linux-gnueabi/4.6/include -Iinclude -I  
./arch/arm/include -include ./include/linux/kconfig.h -D__KERNEL__ -D__UBOOT__ -DCONFIG  
_SYS_TEXT_BASE=0x80800000 -Wall -Wstrict-prototypes -Wno-format-security -fno-builtin -  
ffreestanding -Os -fno-stack-protector -fno-delete-null-pointer-checks -g -fstack-usage -  
Wno-format-nonliteral -fno-toplevel-reorder -D__ARM__ -Wa,-mimplicit-it=always -mthumb -  
mthumb-interwork -mabi=aapcs-linux -mword-relocations -mno-unaligned-access -ffunction-  
sections -fdata-sections -fno-common -ffixed-r9 -msoft-float -pipe -march=armv7-a  
[...]
```

The compiling options are the same as those used for compiling U-Boot image.

- “-nostdinc” specifies not to search standard system directories for header files, instead, search for headers defined after the “-isystem” switch;
- “-ffunction-sections”, “-fdata-sections”, “-fno-common”, “-ffixed-r9” are all platform specific compiling options.
 - “-ffunction-sections”, “-fdata-sections” are used to place each function and data item into its own section in the output file.
 - “-fno-common” is used to force uninitialized global variables to be placed in data section.
 - “-ffixed-r9” will treat the register r9 as a fixed register, and generated code should not never to it. This option plays an important role in the standalone application which will be discussed later.

Linking Options used for libstubs.o:

```
[...]  
arm-linux-gnueabi-ld.bfd -r -o examples/standalone/libstubs.o  
examples/standalone/stubs.o  
[...]
```

The “-r” switch generates relocatable output. It basically takes <stubs.o> as input file to generate <libstubs.o> which in turn serves as input to the linker, ld. This is useful for incremental link of several object files into the <libstubs.o>, as later if one of these object files changes, the linker will modify the existing executable rather than create a new one, thus saving link time.

Linking Options used for hello_world ELF file:

```
[...]  
arm-linux-gnueabi-ld.bfd -g -Ttext 0x80300000 -o examples/standalone/hello_world -e  
hello_world examples/standalone/hello_world.o examples/standalone/libstubs.o -L  
/usr/lib/gcc/arm-linux-gnueabi/4.6 -lgcc  
[...]
```

“-Ttext 0x80300000” specifies the address to load the standalone application for the Beagleboard. This address is hardware specific, and is defined in <arch/arm/config.mk>:

```
[...]  
Line 10: CONFIG_STANDALONE_LOAD_ADDR = 0x80300000  
[...]
```

“-e hello_world” specifies the entry point of the program to be “hello_world”, usually main().

“-L /usr/lib/gcc/arm-linux-gnueabi/4.6 -lgcc” specifies which library to link with and where to find the library. “libgcc.a” is the low level library which contains shared code for low level routines such as the arithmetic operations that the target processor cannot perform directly (multiplication, division, etc.).

As can be seen from the make output, two source files are compiled <hello_world.c> and <stubs.c>. Inside <hello_world.c> the entry function is defined as:

```
[...]  
Line 15: int hello_world (int argc, char * const argv[])  
[...]
```

Inside this function is the simple implementation of “hello_world”. The code itself has little into it, but we may ask:

“How low-level functions such as printf() and tstc() are getting called where are they defined?”

In fact, this is done by calling the U-Boot functions through ABI from the stand-alone application [13].

4.2 Exporting U-Boot Functions for Standalone Applications

The Application Binary Interface (ABI) is the interface between U-Boot and the standalone application. Through this ABI, the standalone application can access to functions which are already implemented into U-Boot at the level of machine code. However, for functions which are not available in U-Boot, one can always implement the missing functions in standalone application, or else , implement them into U-Boot and call through ABI.

Specifically, the U-Boot Functions are exported via a **jump table**, meaning that it contains all available function calls and pointers to them. The jump table can be accessed as the ‘jt’ field of the ‘global_data’ structure. The typedefine of ‘global_data’ structure is in <include/asm-generic/ global_data.h> as “gd_t”. ‘struct arch_global_data

arch' from the 'global_data' is the architecture-specific data structure whose structure declaration can be found in <arch/arm/include/asm/global_data.h>. Another important line in this latter file is:

```
[...]  
Line 85: #define DECLARE_GLOBAL_DATA_PTR register volatile gd_t *gd asm ("r9")  
[...]
```

This is the macro to declare the pointer to global data (`gd_t* gd`) and deliberately put it into register "r9". Choosing of the register is architecture dependent (U-Boot uses "r9" to hold the pointer to global data in ARM architecture), and this is also the reason why we need the "-ffixed-r9" switch (mentioned earlier) when compiling the source code to prevent deliberate use of this register.

The jump table is allocated and initialized in the `jumptable_init()` routine, defined in <common/exports.c> when booting into U-Boot:

```
[...]  
Line 6: DECLARE_GLOBAL_DATA_PTR  
[...]  
Line 17: #define EXPORT_FUNC(f, a, x, ...) gd->jt->x = f;  
Line 19: void jumptable_init(void)  
Line 20: {  
Line 21:     gd->jt = malloc(sizeof(struct jt_funcs));  
Line 22: #include <_exports.h>  
Line 23: }  
[...]
```

where the "jt_funcs" structure is defined in the included header <include/exports.h>

```
[...]  
Line 42: struct jt_funcs {  
Line 43: #define EXPORT_FUNC(impl, res, func, ...) res(*func)(__VA_ARGS__);  
Line 44: #include <_exports.h>  
Line 45: #undef EXPORT_FUNC  
Line 46: };  
[...]
```

<include/_exports.h> contains a list of exported functions wrapped into `EXPOR_FUNC` macro, some of which are exported depending on "CONFIG_XXX" options. The macro will be defined depending on which file this header is included. For example, in <common/exports.c> the `EXPOR_FUNC` macro will be set to initialize the jump table struct with the functions from <include/_exports.h>. In <include/exports.h>, The `EXPOR_FUNC` macro will be set to the type of function pointer which is included inside the jump table struct.

4.2.1 Exporting Additional U-Boot Functions

To export an additional u-boot function, say "long foobar(int i, char c)", to a stand-alone app, the following steps should be taken [13]:

- Append the following line at the end of the <include/_exports.h>
`EXPORT_FUNC(foobar, long, foobar, int, char)`

Parameters to `EXPORT_FUNC` are:

- (1) The function that is exported (default implementation)
- (2) Return value type
- (3) Name of the member in struct `jt_funcs`
- (4) Name that the standalone application will use.

The rest of the parameters are the function arguments

- Add the prototype for this function to the `<include/exports.h>`
`long foobar(int i, char c);`

Initialization with the default implementation is done in `jumtable_init()`, where the `EXPORT_FUNC` macro is expanded (defined) to `:gd->jt->foobar = another_foobar;`

- Optionally, if you want to export a function which depends on a `CONFIG_XXX`, add the following lines to the `<include/exports.h>`:

```
#ifndef CONFIG_FOOBAR
    EXPORT_FUNC(foobar, long, foobar, int, char)
#else
    EXPORT_FUNC(dummy, void, foobar, void)
#endif
```

So far, the jump table including the exported function has been implemented into U-Boot. At boot time, the `global_data` structure is allocated, and the jump table is allocated and initialized with the exported functions. For standalone applications to use these exported functions is mostly machine-independent: the only places written in assembly language are **stub** functions that perform the jump through the jump table. That said, to port this code to a new architecture, the only thing to be provided is the code in the `<examples/stubs.c>`. A code snippet is shown below.

```
[...]
/*
 * r9 holds the pointer to the global_data, ip is a call-clobbered
 * register
 */
Line 59 :#define EXPORT_FUNC(f, a, x, ...) \
Line 60: asm          volatile (      \
Line 61: "          .      globl " #x "\n"      \
Line 62: #x ":\n"          \
Line 63: "      ldr    ip      , [r9, %0]\n" \
Line 64: "      ldr    pc      , [ip, %1]\n" \
Line 65: "      : : "i"(offsetof(gd_t, jt)), "i"(FO(x)) : "ip");
[...]
Line 240: static
Line 241: #endif /* GCC_VERSION */
Line 242: void __attribute__((unused)) dummy(void)
Line 243: {
Line 244: #include <_exports.h>
Line 245: }
[...]
```

Lines 59 to 65 defines the macro to expand the `EXPORT_FUNC` into assembly language. In the “`dummy()`” function, the list of `EXPORT_FUNC` from `<_exports.h>` are expanded into assembly code. We can use

```
$ arm-linux-gnueabi-objdump -D stubs.o
```

to find out what is being compiled into the `<stubs.o>` :

```
[...]
examples/standalone/stubs.o:      file format elf32-littlearm
```


Disassembly of section .text.dummy:

```
00000000 <dummy >:
  0:  f8d9 c064      ldr.w  ip, [r9, #100] ; 0x64
  4:  f8dc f000      ldr.w  pc, [ip]

00000008 <getc>:
  8:  f8d9 c064      ldr.w  ip, [r9, #100] ; 0x64
  c:  f8dc f004      ldr.w  pc, [ip, #4]

00000010 <tstc>:
 10:  f8d9 c064      ldr.w  ip, [r9, #100] ; 0x64
 14:  f8dc f008      ldr.w  pc, [ip, #8]

00000018 <putc>:
 18:  f8d9 c064      ldr.w  ip, [r9, #100] ; 0x64
 1c:  f8dc f00c      ldr.w  pc, [ip, #12]

00000020 <puts>:
 20:  f8d9 c064      ldr.w  ip, [r9, #100] ; 0x64
 24:  f8dc f010      ldr.w  pc, [ip, #16]

00000028 <printf>:
 28:  f8d9 c064      ldr.w  ip, [r9, #100] ; 0x64
 2c:  f8dc f014      ldr.w  pc, [ip, #20]
[...]
```

As can be seen from the objdump output, all the exported functions are in section .text.dummy. This is because we included all of them into dummy() function in <stubs.c>.

Each of the exported functions contains two lines of assembly code defined by the EXPORT_FUNC macro. These are placed in the order they are defined in <_exports.h>. Taking the printf function as an example, when it is called from the standalone application, the above assembly codes which are under 00000028 <printf>: are executed. What it does is:

- Firstly, load IP (Intra Procedure call scratch register) register [15] from 100 bytes offset of the address in register r9. Since the address in r9 is the pointer to global data structure, and with 100 bytes offset, that is the address of the jump table structure.
- Secondly, load PC (Program Counter) register from 20 bytes offset of the address in IP register. Since we declared the jump table structure with function pointers from <_exports.h>, each of the function pointers takes 4bytes. This is exactly the 5th function (counting from zero) that found in <_exports.h>.

Now that the PC register is set with the address pointer values of the printf instruction code, the printf function in U-Boot is called from the standalone application. This is how standalone application call U-Boot functions through ABI.

4.3 Porting “Memtester” to a Standalone U-Boot Application

Memtester [12] is a useful utility application for detecting memory faults, widely used in embedded systems. It is portable and should compile and work on any Unix-like system.

The best and easiest approach to build a U-Boot standalone memtester application would be to integrate its source code into the existent hello_world example, with the same file structure. That is, the main entry of the memtester function present in the <memtester.c> can be ported to the <hello_world.c> and we can directly use the <stubs.c>



file from the `hello_world` example as our library for the exported functions. We also need to add all the missing U-Boot ABI functions into this file using the method explained in section 4.2. Moreover, the other source files from `memtester` should be compiled into object files and incremental link them into the `<libstubs.o>`.

For this, we then create a folder `<memtester>` under `<examples>` and copy all the `memtester` source files plus `<examples/standalone/stubs.c>`, `<examples/standalone/Makefile>` into it.

The first step is to facilitate the Makefile for the `memtester`. At this stage we will focus on the Makefile process, and so we can comment out all the code in the `memtester` source file, leaving only the main entry function in `<memtester.c>` and rename it to `memtester` to comply with Makefile linking command.

In the Makefile,

```
[...]  
extra-y      := hello_world  
[...]
```

defines the make target for the standalone application, and so, we need to change it into `memtester`. Then,

```
[...]  
LIBOBS-y += stubs.o  
[...]
```

defines the object file which is going to be linked into `<libstubs.o>`. Here we will append the `memtester` source “.cpp” files except the one with main entry function after this line.

The Makefile in directory `<examples/Makefile>` should also include the subdirectory we just created by adding

```
[...]  
subdir-y += memtester.  
[...]
```

Executing `make` command now, should prompt us the following output:

```
[...]  
CC      examples/memtester/tests.o  
CC      examples/memtester/stubs.o  
LD      examples/memtester/libstubs.o  
LD      examples/memtester/memtester  
OBJCOPY examples/memtester/memtester.srec  
OBJCOPY examples/memtester/memtester.bin  
[...]
```

Now, we need to uncomment the `memtester` source code and compile the files into the standalone application. The main challenge here is that for standalone application on U-Boot, we don't have the C standard library, so we need to remove all the C standard library headers such as `<stdlib.h>`, `<stdio.h>` which doesn't exist in U-Boot. We then need to fix the missing library functions by exporting U-Boot functions. If a given function isn't even implemented under U-Boot, we either need to implement it or to replace with similar functions from U-Boot.

The following U-Boot functions are exported by adding to `<_exports.h>`:

```
[...]  
EXPORT_FUNC(fprintf, int, fprintf, int, const char*, ...)  
EXPORT_FUNC(get_timer_masked, ulong, get_timer_masked, void)  
EXPORT_FUNC(raise, int, raise, int)
```

```
EXPORT_FUNC(strncmp, int, strcmp, __const char *, __const char *, size_t)
EXPORT_FUNC(strlen, size_t, strlen, __const char *)
[...]
```

The following additional changes were made:

- The `rand()` function from memtester source code is replaced with `'get_timer_masked()'`;
- The `putschar()` function from memtester source code is replaced with `putc()`;
- `fflush()` and `exit()` functions are removed;
- `getopt()` function is removed and replaced with a simple argument parser in main entry function;
- Code lines related with `mmap` memory are removed since no device is mapped into memory;

After fixing all the compilation errors, the standalone application is ready to be tested on board.

4.3.1 Running Standalone MemTester

We use `minicom` to communicate with the Beagle board via serial port, because `minicom` is capable of sending our application to it.

In order to run the application the U-Boot console command `'loads'` is used to load the S-Record file of the memtester standalone application `<memtester.srec>` into address `CONFIG_STANDALONE_LOAD_ADDR (0x80300000)`, defined in U-Boot precompile configurations.

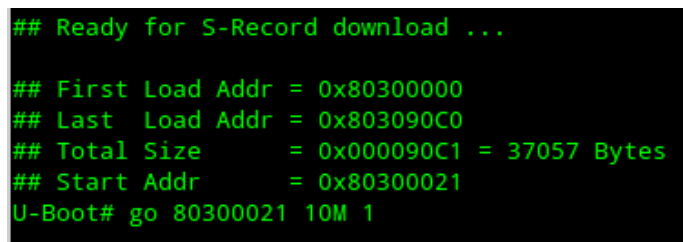
Typing `'loads'` on U-Boot Command console prompt us with the following output:

```
$loads
## Ready for S-Record download ...
```

Afterwards:

1. Press `CTRL + A`, then `Z`, on the linux machine, to open the `minicom` menu;
2. Press `S`, navigate to the folder where `<memtester.srec>` is;
3. Send the file via serial port.

When the transfer is done, U-Boot console will show the message shown in Figure 17.



```
## Ready for S-Record download ...
## First Load Addr = 0x80300000
## Last Load Addr = 0x803090C0
## Total Size      = 0x000090C1 = 37057 Bytes
## Start Addr     = 0x80300021
U-Boot# go 80300021 10M 1
```

Figure 17: Screenshot of memtester uploaded to BeagleBoard

Now the application is loaded at the address defined in `CONFIG_STANDALONE_LOAD_ADDR`. To run the application, command “go” is used followed by the application start address and the arguments to pass into it. Figure 18 shows the result of memtester with 1M memory tested in 1 loop.

```

U-Boot# loads
## Ready for S-Record download ...

## First Load Addr = 0x80300000
## Last Load Addr = 0x80309328
## Total Size      = 0x00009329 = 37673 Bytes
## Start Addr     = 0x80300021
U-Boot# go 80300021 1M 1
## Starting application at 0x80300021 ...
memtester version 4.3.0 (32-bit)
Copyright (C) 2001-2012 Charles Cazabon.
Licensed under the GNU General Public License version 2 (only).

sysconf(_SC_PAGE_SIZE) not supported; using pagesize of 8192
pagesizemask is 0xffffffffffe000
80300021 1M 1
80300021 1M 1,2
want 1MB (1048576 bytes)
got 1MB (1048576 bytes)
Loop 1/1:
  Stuck Address      : ok
  Random Value       : ok
  Compare XOR        : ok
  Compare SUB        : ok
  Compare MUL        : ok
  Compare DIV        : ok
  Compare OR         : ok
  Compare AND        : ok
  Sequential Increment: ok
  Solid Bits         : ok
  Block Sequential   : ok
  Checkerboard       : ok
  Bit Spread         : ok
  Bit Flip           : ok
  Walking Ones       : ok
  Walking Zeroes     : ok

Done.
## Application terminated, rc = 0x0
  
```

Figure 18: Screenshot of memtester running on BeagleBoard

4.3.2 Memtester Standalone Application Structure in Memory

As has been discussed in section 4.3.1, the resulting ‘Make’ output file `<memtester>` is an executable and linkable format (ELF) file which is the standard executable file type for Unix Systems. If we prompt:

```
$hexdump memtester
```

what we will see is the hexdump of the contents of ‘memtester’ binary file, which is the actual machine code that is going to be executed. Below are parts of the hexdump of ‘memtester’ ELF file:

```

[...]
00000000 457f 464c 0101 0001 0000 0000 0000 0000
00000010 0002 0028 0001 0000 0021 8030 0034 0000
00000020 d99c 0000 0002 0500 0034 0020 0003 0028
  
```

```

0000030 0012 000f 0001 0000 8000 0000 0000 8030
0000040 0000 8030 12a3 0000 12a3 0000 0005 0000
0000050 8000 0000 0001 0000 92a4 0000 92a4 8030
0000060 92a4 8030 0085 0000 0094 0000 0006 0000
0000070 8000 0000 e551 6474 0000 0000 0000 0000
0000080 0000 0000 0000 0000 0000 0000 0006 0000
0000090 0004 0000 0000 0000 0000 0000 0000 0000
00000a0 0000 0000 0000 0000 0000 0000 0000 0000
*
0008000 b508 4803 f000 f93c f44f 5000 bd08 bf00
0008010 0f46 8030 4601 4801 f000 b932 0f84 8030
0008020 e92d 4df0 4604 b088 460d 4877 2120 f000
0008030 f927 4876 f000 f924 4875 f000 f921 4875
[...]

```

Fortunately, the ARM GNU toolchain provides a useful binutils tool `arm-linux-gnueabi-readelf` to view this ELF file. A simple Executable ARM ELF file has the conceptual layout shown in Table 2 [17]:

ELF Header
Program Header Table
Text segment
Data segment
BSS segment
".symtab" section
".strtab" section
".shstrtab" section
Debug sections
Section Header Table

Table 2: ARM ELF File layout

The actual ordering of the file may differ from that shown in Table 2, since the only fixed position is that of the 'ELF header'. The position of all other parts of the file is defined by the Program and Section headers, where the former defines program positions and the latter section positions.

By calling:

```
$arm-linux-gnueabi-readelf -a memtester
```

all the information about the ELF is displayed, mainly including "ELF header", "program headers" and "section headers":

```

ELF Header:
  Magic:                7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
  Class:                ELF32
  Data:                 2's complement, little endian
  Version:              1 (current)
  OS/ABI:               UNIX - System V
  ABI Version:         0
  Type:                 EXEC (Executable file)
  Machine:              ARM
  Version:              0x1
  Entry point address:  0x80300021
  Start of program headers: 52 (bytes into file)
  Start of section headers: 55708 (bytes into file)
  Flags:                0x5000002, has entry point, Version5 EABI

```



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:	18
Section header string table index:	15

Above is the first part of the 'readelf' output, this is, the ELF header contents: the bytes next to the first line 'Magic' are exactly what is seen at the beginning of the hexdump. This is a constant sequence of bytes to establish the 'ELF' file format. The Header information is completely interpreted with the hexdump of the first 52 bytes (size of this header) of the ELF file.

"Start of program headers" indicates the byte offset of the program header table, which is present right after the first 52 bytes of the header information;

"Start of section headers" indicates the byte offset (55708 bytes) of the section header table inside the ELF file.

The next part of the 'readelf' output contains information of the section headers table. This provides information on where these sections are located inside the ELF file:

- The first column is the index of the sections. There are 18 sections in total (from 0 to 17), which is in accordance with the information from the first part;
- The second column refers to the name of each section; and the third column refers to the type of each section. The definition of each section type is:
 - SHT_NULL: This value marks the section header as inactive; it does not have an associated section. Other members of the section header have undefined values.
 - SHT_PROGBITS: The section holds information defined by the program, whose format and meaning are determined solely by the program.
 - SHT_SYMTAB and SHT_DYNSYM: These sections hold a symbol table.
 - SHT_STRTAB: The section holds a string table.
 - SHT_NOBITS: A section of this type occupies no space in the file but otherwise resembles SHT_PROGBITS. Although this section contains no bytes, the sh_offset member contains the conceptual file offset
- The third column refers to the load address of a section;
- The fifth column refers to the offset of a section inside the ELF file;
- The sixth column refers to the size of the section.

The meaning for the rest of the columns can all be found in [19], which will not be covered by this document.

Taking section header 1 as an example, it can be interpreted as: the .text section should be loaded at address 0x80300000; the section starts at offset 0x8000 in the ELF file; and the section has a size of 0xe84 bytes.

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	80300000	008000	000e84	00	AX	0	0	8



[2]	.rodata	PROGBITS	80300e84	008e84	00041f	01	AMS	0	0	1
[3]	.data	PROGBITS	803092a4	0092a4	000085	00	WA	0	0	4
[4]	.bss	NOBITS	8030932c	009329	00000c	00	WA	0	0	4
[5]	.comment	PROGBITS	00000000	009329	00002a	01	MS	0	0	1
[6]	.ARM.attributes	ARM_ATTRIBUTES	00000000	009353	000031	00		0	0	1
[7]	.debug_aranges	PROGBITS	00000000	009384	0000f8	00		0	0	1
[8]	.debug_info	PROGBITS	00000000	00947c	001755	00		0	0	1
[9]	.debug_abbrev	PROGBITS	00000000	00abd1	0004aa	00		0	0	1
[10]	.debug_line	PROGBITS	00000000	00b07b	000781	00		0	0	1
[11]	.debug_frame	PROGBITS	00000000	00b7fc	0002f4	00		0	0	4
[12]	.debug_str	PROGBITS	00000000	00baf0	000575	01	MS	0	0	1
[13]	.debug_loc	PROGBITS	00000000	00c065	0017ba	00		0	0	1
[14]	.debug_ranges	PROGBITS	00000000	00d81f	0000c8	00		0	0	1
[15]	.shstrtab	STRTAB	00000000	00d8e7	0000b3	00		0	0	1
[16]	.symtab	SYMTAB	00000000	00dc6c	000900	10		17	68	4
[17]	.strtab	STRTAB	00000000	00e56c	000402	00		0	0	1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
0 (extra OS processing required) o (OS specific), p (processor specific)

Some of the important section have the following meaning:

- .text section holds the executable instructions of the application;
- .rodata section holds read-only data.
- .bss section holds the uninitialized data; by default, the system will initialize the data with zeros when the application starts running, this section does not occupy any space in ELF file, this is also why the type of this section is NOBITS.
- .data section holds the initialized data.

To view the binary data of the section, we use the following call:

```
$arm-linux-gnueabi-readelf -x [section_name] memtester
```

Taking the .rodata section as an example, we can see the hexdump of it:

Hex dump of section '.rodata':										
0x80300e84	52616e64	6f6d2056	616c7565	00436f6d	Random Value.Com					
0x80300e94	70617265	20584f52	00436f6d	70617265	pare XOR.Compare					
0x80300ea4	20535542	00436f6d	70617265	204d554c	SUB.Compare MUL					
0x80300eb4	00436f6d	70617265	20444956	00436f6d	.Compare DIV.Com					
0x80300ec4	70617265	204f5200	436f6d70	61726520	pare OR.Compare					
0x80300ed4	414e4400	53657175	656e7469	616c2049	AND.Sequential I					
0x80300ee4	6e637265	6d656e74	00536f6c	69642042	crement.Solid B					
[...]										
0x80301274	74657374	2e2e2e0a	00080808	08080808	test.....					
0x80301284	08080808	20202020	20202020	20202008					
0x80301294	08080808	08080808	08080008	200800					

As can be seen, this section ranges from offset `0x80300e84` to `0x803012a2`, with a size of `0x41f` bytes. This conforms with what is shown in the Section Header Table shown above. The above hexdump also shows that this section contains all the string literals inside the memtester source code, since they are all read only data.

The next part of the 'readelf' output is dedicated to the Program Headers. The Program Header table is an array of structures, each describing a segment (or other information) needed by the system to prepare the program for execution [18]. In our specific case, there are three program headers, each representing a segment:

- The first column is the type of the segment;
- The second column is the offset of the segment inside the ELF file;
- The third and fourth columns give the virtual and physical address of the segment in memory;
- The fifth column "FileSiz" is the number of bytes in the ELF file;
- The sixth column "MemSiz" refers to the number of bytes in the memory;

Program Headers:						
Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg Align
LOAD	<code>0x008000</code>	<code>0x80300000</code>	<code>0x80300000</code>	<code>0x012a3</code>	<code>0x012a3</code>	R E <code>0x8000</code>
LOAD	<code>0x0092a4</code>	<code>0x803092a4</code>	<code>0x803092a4</code>	<code>0x00085</code>	<code>0x00094</code>	RW <code>0x8000</code>
GNU_STACK	<code>0x000000</code>	<code>0x00000000</code>	<code>0x00000000</code>	<code>0x00000</code>	<code>0x00000</code>	RW <code>0x4</code>

The section Header Table provides the information of each section (`.text`, `.rodata`, `.data`, etc.) from the Linking View, while the Program Header Table provides the information of each segment (Loadable Code Segment, Loadable Data Segment, etc.) from the execution view. A mapping from Section to Segment can also be found in the "readelf" output after Program Header:

Section to Segment mapping:		
Segment	Sections...	
<code>00</code>	<code>.text</code>	<code>.rodata</code>
<code>01</code>	<code>.data</code>	<code>.bss</code>
<code>02</code>		

This also explains why the FileSiz and MemSiz are different in the second Program Header. That is, the Loadable Data Segment contains both `.data` and `.bss` sections. Based on the Section Header Table, `.data` section has `0x85` bytes and `.bss` section has `0xc` bytes, which gives us a total of `0x91` bytes. However, since the alignment for both section is 4, the total bytes should be `0x94` bytes, and this is equals to MemSiz. On the other hand, in terms of byte sizes inside the ELF file, since the `.bss` section does not occupy any bits, the "file size" is only `0x85` bytes.

With the information from 'readelf' output, we can now associate the ELF layout with the hexdump of the ELF file, shown below where colors are used to separate ELF sections.

[...]		
<code>00000000</code>	<code>457f 464c 0101 0001 0000 0000 0000 0000</code>	<code>/*ELF Header */</code>
<code>0000010</code>	<code>0002 0028 0001 0000 0021 8030 0034 0000</code>	<code>/*byte offset 0 */</code>
<code>0000020</code>	<code>d418 0000 0002 0500 0034 0020 0003 0028</code>	<code>/*byte size 52(0x34) bytes */</code>
<code>0000030</code>	<code>0012 000f 0001 0000 8000 0000 0000 8030</code>	<code>/*Program Header Table */</code>
<code>0000040</code>	<code>0000 8030 103b 0000 103b 0000 0005 0000</code>	<code>/*byte offset 52(0x34) */</code>
<code>0000050</code>	<code>8000 0000 0001 0000 903c 0000 903c 8030</code>	<code>/* byte size 32(0x20) bytes */</code>
<code>0000060</code>	<code>903c 8030 0085 0000 0094 0000 0006 0000</code>	<code>/* number of Program Header: 3 */</code>
<code>0000070</code>	<code>8000 0000 e551 6474 0000 0000 0000 0000</code>	
<code>0000080</code>	<code>0000 0000 0000 0000 0000 0000 0006 0000</code>	
<code>0000090</code>	<code>0004 0000 0000 0000 0000 0000 0000 0000</code>	<code>/*filled with zeroes since the */</code>



```
00000a0 0000 0000 0000 0000 0000 0000 0000 0000 /*linker force the .text section */
* /*start at offset 0x8000 */
0008000 b508 4803 f000 f93c f44f 5000 bd08 bf00 /* .text Section */
0008010 0f46 8030 4601 4801 f000 b932 0f84 8030 /* byte offset 0x8000 */
0008020 e92d 4df0 4604 b088 460d 4877 2120 f000 /* byte size 0xE84 */
[...]
0008e80 fa0b bd02 6152 646e 6d6f 5620 6c61 6575 /*.rodata Section */
0008e90 4300 6d6f 6170 6572 5820 524f 4300 6d6f /*byte offset 0x8e84 */
0008ea0 6170 6572 5320 4255 4300 6d6f 6170 6572 /* byte size 0x41f */
[...]
00092a0 0820 0000 0e84 8030 04d5 8030 0e91 8030 /* .data Section */
00092b0 054d 8030 0e9d 8030 058f 8030 0ea9 8030 /*byte offset 0x92A4 */
00092c0 05d1 8030 0eb5 8030 0613 8030 0ec1 8030 /*byte size 0x85*/
[...]
0009320 0000 0000 5c2d 2f7c 4700 4343 203a 5528 /*.comment, .debug and sections */
[...] /*other sections */
000d990 6775 725f 6e61 6567 0073 0000 0000 0000 /*Section Header Table */
000d9a0 0000 0000 0000 0000 0000 0000 0000 0000 /*byte offset 55708(0xD99C) */
000d9b0 0000 0000 0000 0000 0000 0000 0000 0000 /*byte size 40(0x28) */
000d9c0 0000 0000 001b 0000 0001 0000 0006 0000 /* number of Section Header: 18 */
[...]
000dc60 0000 0000 0001 0000 0000 0000 0000 0000 /*.symtab section */
000dc70 0000 0000 0000 0000 0000 0000 0000 0000 /*byte offset 0xDC6C */
000dc80 0000 8030 0000 0000 0003 0001 0000 0000 /*byte size 0x900 */
[...]
000e560 02d8 8030 0000 0000 0010 0001 6d00 6d65 /*.strtab section */
000e570 6574 7473 7265 632e 2400 0064 7424 7300 /*byte offset 0xE56C */
000e580 7574 7362 632e 7400 7365 7374 632e 2e00 /*byte size 0x402 */
000e960 6573 7200 6961 6573 6600 6572 0065 /*end of the file */
```

5 Using U-Boot Commands

In this section we will cover the usage of common U-Boot commands to handle memory and mass-storage devices, already included in the standard build. The aim is to provide a use-case for commands to show how they can be used to exercise peripheral hardware. The detailed command structure is explained in sub-section 3.1.3.

5.1 binfo

This command is implemented under <common/cmd_binfo>. It prints the board info structure by making a call to the `do_binfo()` function:

```
int do_binfo(cmd_tbl_t *cmdtp, int flag, int argc, char * const argv[])
{
    bd_t *bd = gd->bd;

    print_num("mem start",          (ulong)bd->bi_memstart);
    print_lnum("mem size",          (u64)bd->bi_memsz);
    print_num("flash start",       (ulong)bd->bi_flashstart);
    print_num("flash size",        (ulong)bd->bi_flashsz);
    print_num("flash offset",      (ulong)bd->bi_flashoffset);

#ifdef CONFIG_SYS_SRAM_BASE
    print_num ("sram start",       (ulong)bd->bi_sramstart);
    print_num ("sram size",        (ulong)bd->bi_sramsz);
#endif

#ifdef CONFIG_CMD_NET
    print_eth(0);
    printf("ip_addr      = %s\n", getenv("ipaddr"));
#endif

    printf("baudrate    = %u bps\n", gd->baudrate);

    return 0;
}
```

`bd` is a pointer to the board information structure, used herein to provide memory information. Other info is provided, depending on the defined `CONFIG_` directives.

5.2 md

The memory display command calls the function `do_mem_md()`. It has the following parameters:

`$md [.b, .w, .l, .q] address [# of objects]`

- `.b` stands for bytes; `.w` stands for words and `.l` stands for long words, `.q` double long word (64b);
- `address`, is an address in the RAM;
- `# of objects`, "if another parameter, it is the length to display. Length is the number of objects, not number of bytes."



5.3 mw

The memory write command calls the function `do_mem_mw()`. This command has the following parameters:

```
$md [.b, .w, .l, .q] address [# of objects]
```

- `.b` stands for bytes; `.w` stands for words; `.l` stands for long words; `.q` double long word (64b);
- `address`, is address in the RAM;
- `# of objects`, "if another parameter, it is the length to display. Length is the number of objects, not number of bytes."

5.4 usb

The `usb` command is used to perform R/W accesses to an USB device.

For a write operation, the following arguments are used:

```
$usb write addr blk# cnt
```

- `addr` is a RAM pointing to the data to be written;
- `blk#` is the starting block to write on the usb device;
- `cnt` is the number of blocks to write.

Basically, the command copies data from the RAM address into the specified block in the USB. For this, the `do_usb()` function is called.

For a read operation, the following arguments are used:

```
$usb write addr blk# cnt
```

- `addr` is a RAM pointing to the position where data read from the device will be stored;
- `blk#` is the starting block to read from on the device;
- `cnt` is the number of blocks to read.

Basically, the command copies the desired amount of blocks in the USB into the RAM address. This command calls the `do_usb()` function. The latter has the following arguments:

- `cmd_tbl_t *cmdtp,`
- `int flag,`
- `int argc,`
- `char * const argv[]`

Depending on the input, the `do_usb()` function will read or write as can be interpreted from the following snippet of `<common/cmd_usb.c>`:

```
[...]  
if (strcmp(argv[1], "read") == 0) {  
    if (usb_stor_curr_dev < 0) {  
        printf("no current device selected\n");  
        return 1;  
    }  
    if (argc == 5) {  
        unsigned long addr = simple_strtoul(argv[2], NULL, 16);  
        unsigned long blk = simple_strtoul(argv[3], NULL, 16);  
        unsigned long cnt = simple_strtoul(argv[4], NULL, 16);  
        unsigned long n;  
        printf("\nUSB read: device %d block # %ld, count %ld"  
            " ... ", usb_stor_curr_dev, blk, cnt);  
        stor_dev = usb_stor_get_dev(usb_stor_curr_dev);  
        n = stor_dev->block_read(usb_stor_curr_dev, blk, cnt,  
            (ulong *)addr);  
        printf("%ld blocks read: %s\n", n,  
            (n == cnt) ? "OK" : "ERROR");  
        if (n == cnt)  
            return 0;  
        return 1;  
    }  
}  
[...]  
if (strcmp(argv[1], "write") == 0) {  
    if (usb_stor_curr_dev < 0) {  
        printf("no current device selected\n");  
        return 1;  
    }  
    if (argc == 5) {  
        unsigned long addr = simple_strtoul(argv[2], NULL, 16);  
        unsigned long blk = simple_strtoul(argv[3], NULL, 16);  
        unsigned long cnt = simple_strtoul(argv[4], NULL, 16);  
        unsigned long n;  
        printf("\nUSB write: device %d block # %ld, count %ld"  
            " ... ", usb_stor_curr_dev, blk, cnt);  
        stor_dev = usb_stor_get_dev(usb_stor_curr_dev);  
        n = stor_dev->block_write(usb_stor_curr_dev, blk, cnt,  
            (ulong *)addr);  
        printf("%ld blocks write: %s\n", n,  
            (n == cnt) ? "OK" : "ERROR");  
        if (n == cnt)  
            return 0;  
        return 1;  
    }  
}
```

5.5 mmc

The mmc command is used to perform read/write accesses to an SD card.



For a write operation, the following arguments are used:

```
$mmc write addr blk# cnt
```

- Addr is the address to the RAM
- Blk# is the starting block to read
- Number of “cnt” blocks

Similarly to the usb command, it copies data from the RAM address into the specified block in the MMC. This command calls `do_mmcops()` which then will call `do_mmc_write()` function.

For a read operation, the following arguments are used:

```
$mmc read addr blk# cnt
```

- addr is a RAM pointing to the position where data read from the device will be stored;
- blk# is the starting block to read from on the device;
- cnt is the number of blocks to read.

Similarly to usb, the mmc command copies the desired amount of blocks in the USB into the RAM address. For this, the commands calls `do_mmcops()` which then calls the `do_mmc_read()` function. The former has the following arguments:

- `cmd_tbl_t *cmdtp,`
- `int flag,`
- `int argc,`
- `char * const argv[]`

If the input command is “mmc read”, then `do_mmc_read()` is called; the same goes for “mmc write” and `do_mmc_write()`.

5.6 Usage Examples

5.6.1 USB R/W Access

In this subsection we will use the Beagle setup shown in section 3.5 with a uSD and a USB card installed.

Before starting to read and write to the USB device, we need to know the RAM addressable range. For this, we can call `bdinfo` from the U-Boot terminal:

```
$bdinfo
```

The output of this command is shown in Figure 19.

```
U-Boot# bdfinfo
arch_number = 0x00000E05
boot_params = 0x80000100
DRAM bank   = 0x00000000
-> start     = 0x80000000
-> size      = 0x20000000
eth0name    = cpsw
ethaddr     = 78:a5:04:ef:9a:85
eth1name    = usb_ether
eth1addr    = 78:a5:04:ef:9a:87
current eth = cpsw
ip_addr     = <NULL>
baudrate    = 115200 bps
TLB addr    = 0x9FFF0000
relocaddr   = 0x9FF4D000
reloc off   = 0x1F74D000
irq_sp      = 0x9EF2CEC0
sp_start    = 0x9EF2CEB0
U-Boot#
```

Figure 19: bdfinfo command output

The “-> start” indicator refers to the initial addressable RAM address, which we will use hereafter. Before writing anything to the USB, we will check the USB device contents. To do so, and every time the USB needs to be accessed, it is necessary to start the device by typing:

```
$usb start
```

which will prompt the output shown in Figure 20:

```
U-Boot# usb start
starting USB...
USB0: scanning bus 0 for devices... 1 USB Device(s) found
      scanning usb for storage devices... 1 Storage Device(s) found
```

Figure 20: USB start instruction output

Now we are ready to perform read/write accesses. The memory contents at 0x80000000 can be displayed with md:

```
$ md 0x80000000
```

which prompts us with the output shown in Figure 21.

```

U-Boot# md 0x80000000
80000000: 41615252 00000000 00000000 00000000  RRaA.....
80000010: 00000000 00000000 00000000 00000000  .....
80000020: 00000000 00000000 00000000 00000000  .....
80000030: 00000000 00000000 00000000 00000000  .....
80000040: 00000000 00000000 00000000 00000000  .....
80000050: 00000000 00000000 00000000 00000000  .....
80000060: 00000000 00000000 00000000 00000000  .....
80000070: 00000000 00000000 00000000 00000000  .....
80000080: 00000000 00000000 00000000 00000000  .....
80000090: 00000000 00000000 00000000 00000000  .....
800000a0: 00000000 00000000 00000000 00000000  .....
800000b0: 00000000 00000000 00000000 00000000  .....
800000c0: 00000000 00000000 00000000 00000000  .....
800000d0: 00000000 00000000 00000000 00000000  .....
800000e0: 00000000 00000000 00000000 00000000  .....
800000f0: 00000000 00000000 00000000 00000000  .....

```

Figure 21: md output

As explained in sub-section 5.4, to write to a USB device it is necessary to initialize a RAM block with the data to be written. This can be done with the mw command explained in section 5.3:

```
$mw 0x80000000 0x1234ABCD
```

```
$md 0x80000000
```

Which prompts us with the output shown in Figure 22.

```

U-Boot# mw 0x80000000 0x1234ABCD
U-Boot# md 0x80000000
80000000: 1234abcd 00000000 00000000 00000000  ..4.....
80000010: 00000000 00000000 00000000 00000000  ++++++
80000020: 00000000 00000000 00000000 00000000  ++++++
80000030: 00000000 00000000 00000000 00000000  ++++++
80000040: 00000000 00000000 00000000 00000000  ++++++
80000050: 00000000 00000000 00000000 00000000  ++++++
80000060: 00000000 00000000 00000000 00000000  ++++++
80000070: 00000000 00000000 00000000 00000000  ++++++
80000080: 00000000 00000000 00000000 00000000  ++++++
80000090: 00000000 00000000 00000000 00000000  ++++++
800000a0: 00000000 00000000 00000000 00000000  ++++++
800000b0: 00000000 00000000 00000000 00000000  ++++++
800000c0: 00000000 00000000 00000000 00000000  ++++++
800000d0: 00000000 00000000 00000000 00000000  ++++++
800000e0: 00000000 00000000 00000000 00000000  ++++++
800000f0: 00000000 00000000 00000000 00000000  ++++++
U-Boot# █

```

Figure 22: write/read operation on RAM

A USB write can now be performed with:

```
$usb write 0x80000000 1 1
```

The information previously stored in the RAM is then copied to the USB and a confirmation message is presented, as shown in Figure 23.

```
U-Boot# usb write 80000000 1 1
USB write: device 0 block # 1, count 1 ... 1 blocks write: OK
```

Figure 23: Output of usb write command

To check that the USB was written properly we empty the RAM contents at 0x80000000:

```
$mw 0x8000000 00000000
$md 0x8000000
```

Which prompts us with the output shown in Figure 24.

```
U-Boot# mw 80000000 00000000
U-Boot# md 80000000
80000000: 00000000 00000000 00000000 00000000 .....
80000010: 00000000 00000000 00000000 00000000 .....
80000020: 00000000 00000000 00000000 00000000 .....
80000030: 00000000 00000000 00000000 00000000 .....
80000040: 00000000 00000000 00000000 00000000 .....
80000050: 00000000 00000000 00000000 00000000 .....
80000060: 00000000 00000000 00000000 00000000 .....
80000070: 00000000 00000000 00000000 00000000 .....
80000080: 00000000 00000000 00000000 00000000 .....
80000090: 00000000 00000000 00000000 00000000 .....
800000a0: 00000000 00000000 00000000 00000000 .....
800000b0: 00000000 00000000 00000000 00000000 .....
800000c0: 00000000 00000000 00000000 00000000 .....
800000d0: 00000000 00000000 00000000 00000000 .....
800000e0: 00000000 00000000 00000000 00000000 .....
800000f0: 00000000 00000000 00000000 00000000 .....
```

Figure 24: Memory contents reset

```
$usb read 80000000 1 1
```

Which prompts us with the output shown in Figure 25.

```
U-Boot# usb read 80000000 1 1
USB read: device 0 block # 1, count 1 ... 1 blocks read: OK
```

Figure 25: Output of USB read command

Finally, we can check the read data in the destination address:

```
$md 80000000
```


which prompts us with the updated memory contents:

```

USB read: device 0 block # 1, count 1 .. 1 blocks read: OK
U-Boot# md 0x80000000
80000000: 1234abcd 00000000 00000000 00000000  ..4.....
80000010: 00000000 00000000 00000000 00000000  ++++++
80000020: 00000000 00000000 00000000 00000000  ++++++
80000030: 00000000 00000000 00000000 00000000  ++++++
80000040: 00000000 00000000 00000000 00000000  ++++++
80000050: 00000000 00000000 00000000 00000000  ++++++
80000060: 00000000 00000000 00000000 00000000  ++++++
80000070: 00000000 00000000 00000000 00000000  ++++++
80000080: 00000000 00000000 00000000 00000000  ++++++
80000090: 00000000 00000000 00000000 00000000  ++++++
800000a0: 00000000 00000000 00000000 00000000  ++++++
800000b0: 00000000 00000000 00000000 00000000  ++++++
800000c0: 00000000 00000000 00000000 00000000  ++++++
800000d0: 00000000 00000000 00000000 00000000  ++++++
800000e0: 00000000 00000000 00000000 00000000  ++++++
800000f0: 00000000 00000000 00000000 00000000  ++++++
U-Boot#
  
```

Figure 26: Check back read data from USB

5.6.2 SD R/W Access

The procedure to write to the MMC is the same as that of the USB, meaning, the input arguments to the `mmc read` and `mmc write` commands are exactly the same. However in this case, we do not need to initialize the device with a 'start' command.

For example we can use the `mw` command to fill the outgoing RAM position and issue a device write/read:

```

# mw 80000050 10101010 10
# mmc write 80000050 1 1
# mw 80000050 00000000 10
# md 80000000
# mmc read 80000050 1 1
# md 80000000
  
```

And the buffer RAM memory zone will be filled with the read data as shown in Figure 27.

```

80000020: 00000000 00000000 00000000 00000000  .....
80000030: 00000000 00000000 00000000 00000000  .....
80000040: 00000000 00000000 00000000 00000000  .....
80000050: 10101010 10101010 10101010 10101010  .....
80000060: 10101010 10101010 10101010 10101010  .....
80000070: 10101010 10101010 10101010 10101010  .....
80000080: 10101010 10101010 10101010 10101010  .....
80000090: 00000000 00000000 00000000 00000000  .....
800000a0: 00000000 00000000 00000000 00000000  .....
  
```

Figure 27: Read data from SD card with "mmc" command

6 References

- [1] Brad Stewart, *Building U-Boot in Eclipse Helios*, Jan 31 2011, <https://community.nxp.com/servlet/JiveServlet/download/222791-1-172009/844-BuildingUBootinEclipse.pdf>
- [2] Texas Instruments, *Processor SDK Building The SDK*, http://processors.wiki.ti.com/index.php/Processor_SDK_Building_The_SDK
- [3] elinux.org, *Virtual Development Board*, http://www.elinux.org/Virtual_Development_Board
- [4] Sathya, *Can't install Guest Additions using VirtualBox, Ubuntu guest OS, Win7 host OS*, March 2012, <https://superuser.com/questions/261643/cant-install-guest-additions-using-virtualbox-ubuntu-guest-os-win7-host-os>
- [5] qemu.org, *Hosts/Linux*, November 2015, <http://wiki.qemu.org/Hosts/Linux>
- [6] shibely, *What is different between u-boot.bin and u-boot.img*, <https://stackoverflow.com/questions/29494321/what-is-different-between-u-boot-bin-and-u-boot-img>
- [7] XILLYBUS, *U-Boot programming: A tutorial*, <http://xillybus.com/tutorials/uboot-hacking-howto-1>
- [8] Everspin Technologies, *MR25H40 datasheet*, Ver. 5, November 2011
- [9] Otto Linnemann, *How to create a Boot SD Card for the BeagleBone black*, Jan 2016, <https://github.com/linneman/planck/wiki/How-to-create-a-Boot-SD-Card-for-the-BeagleBone-black>
- [10] Müller T., *U-Boot on BeagleBone Black*, August 2014, <https://www.twam.info/hardware/beaglebone-black/u-boot-on-beaglebone-black>
- [11] Gentoo, *Format the MicroSD card the way BeagleBoard wants it*, June 2017, https://wiki.gentoo.org/wiki/BeagleBone_Black#Format_the_MicroSD_card_the_way_BeagleBoard_wants_it
- [12] Charles Cazabon, *Memtester*, 1999-2017, <http://pyropus.ca/software/memtester/>
- [13] U-Boot-2015.07/ project README docs (README.standalone)
- [14] DENX Software Engineering, *The DENX U-Boot and Linux Guide (DULG) for canyonlands*, 2011, <https://www.denx.de/wiki/view/DULG/Manual>
- [15] ARM, *Procedure Call Standard for the ARM Architecture*, November 2015, http://infocenter.arm.com/help/topic/com.arm.doc.ihl0042f/IHL0042F_aapcs.pdf
- [16] Boot-2015.07/ project README docs (README.commands)
- [17] ARM, *ARM ELF File Format*, 1998, http://infocenter.arm.com/help/topic/com.arm.doc.dui0101a/DUI0101A_Elf.pdf
- [18] ARM, *ARM ELF(SWS ESPC 0003 A-06)*, 1998 <http://netwinder.osuosl.org/pub/netwinder/docs/arm/ARMELFA06.pdf>
- [19] AM335x ARM® Cortex™-A8 Microprocessors (MPUs) Technical Reference Manual, October 2011, <http://elinux.org/images/6/65/Spruh73c.pdf>