Linux Cortex-M
User's Manual

Release 1.13.0

Table of Contents

1. OVERVIEW 3

2. HARDWARE PLATFORM 3

3. PRE-INSTALLED DEMO 3

3.1. PRE-INSTALLED DEMO ON SUPPORTED HARDWARE PLATFORMS 3
3.2. DESCRIPTION OF THE PRE-INSTALLED DEMO 3

4. LINUX CORTEX-M SOFTWARE DEVELOPMENT ENVIRONMENT 7

4.1. DISTRIBUTION AND INSTALLATION 7
4.1.1. Distribution Image 7
4.1.2. Installation Tree 8
4.1.3. GNU Cross-Build Tools 8
4.1.4. Activation 9
4.1.5. Dependency on Host Components 9
4.2. PROJECTS FRAMEWORK 10
4.2.1. Multiple Target Projects 10
4.2.2. Hello World Demo Project 11
4.2.3. Networking Demo Project 11
4.2.4. Developer Demo Project 12

5. U-BOOT 12

5.1. POINTERS TO DETAILED INFORMATION ON U-BOOT 12
5.2. U-BOOT: THEORY OF OPERATION 12
5.3. SELECTED USE SCENARIOS 13
5.3.1. U-Boot Environment 13
5.3.2. Autoboot 13
5.3.3. Networking in U-Boot 14
5.3.4. Flash-related Commands 14
5.3.5. U-Boot Self-Upgrade 15

6. SAMPLE DEVELOPMENT SESSION 16

6.1. SAMPLE SESSION OUTLINE 16
6.2. CREATE A NEW PROJECT 16
6.3. SET UP TARGET FOR DEBUGGING OF THE NEW PROJECT 17
6.4. UPDATE THE NEW PROJECT 18
6.5. INSTALL THE NEW PROJECT TO FLASH 18
6.6. DEVELOP A CUSTOM LOADABLE DEVICE DRIVER OVER NFS 19
6.7. DEVELOP A CUSTOM USER-SPACE APPLICATION OVER NFS 20

7. SUPPORT AND FURTHER MATERIALS 21

1. Overview

This document is a User's Manual for Linux Cortex-M covering the following products:

- Linux STM32, supporting the STmicroelectronics Cortex-M3 based STM32F2, Cortex-M4 based STM32F4, and Cortex-M7 based STM32F7 microcontrollers;
- Linux LPC, supporting the NXP Cortex-M3 based LPC178X, LPX18XX and LPC43XX microcontrollers;
- Linux Kinetis, supporting the Freescale Cortex-M4 based Kinetis K70 microcontrollers;
- Linux SmartFusion, supporting the Microsemi Cortex-M3 based SmartFusion and SmartFusion2 configurable System-On-Chip (cSOC) microcontrollers.

Linux Cortex-M provides a platform and software development environment for evaluation and development of Linux on the Cortex-M CPU core of the MCU devices listed above.

2. Hardware Platform

Depending on a specific product option, Linux Cortex-M comes as a software distribution image supporting the corresponding Cortex-M microcontroller family.

For each supported microcontroller family, Linux Cortex-M provides support for one or more reference hardware boards. Board support comes as a Board Support Package (BSP) specific to a particular hardware board.

Please refer to an appropriate Starter Kit Guide (or BSP Guide) for detailed information on how to install and configure Linux Cortex-M on a specific hardware board.

This User's Manual document proceeds to outline information that is common for all supported MCU architectures and hardware boards. The sample sessions provided below emphasize a concrete microcontroller, however the functionality illustrated by those sessions is available on all supported architectures.

3. Pre-installed Demo

3.1. Pre-installed Demo on Supported Hardware Platforms

Those hardware boards that are shipped by Emcraft as part of a corresponding Linux Cortex-M product option come with the U-Boot firmware installed into the internal Flash of the microcontroller device and a Linux image installed into the external Flash memory.

For those product options that provide support for a third-party hardware board, you will have to install U-Boot and Linux onto your board as explained in a corresponding Starter Kit Guide (or BSP Guide).

3.2. Description of the Pre-installed Demo

The installed Linux image provides a demonstration of the basic shell, networking and file system capabilities supported by Linux Cortex-M. The specific functionality available in that demo is described in detail below.

On a power-on or reset, U-Boot runs from the embedded Flash and uses the embedded SRAM of the microcontroller as a storage for volatile data.

Having completed the basic initialization, U-Boot configures the memory controller to allow accesses to the external RAM and Flash memory and then copies the Linux image from Flash to RAM and jumps to the Linux kernel entry point in RAM. It is possible to interrupt the U-Boot autoboot sequence by hitting a key before U-Boot starts relocating the Linux image to RAM and enter the command line interface of U-Boot, however assuming no operator intervention, U-Boot proceeds to boot Linux up as soon as possible:

```
U-Boot 2010.03-linux-cortexm-1.13.0 (Feb 09 2015 - 19:43:45)

CPU: SmartFusion FPGA (Cortex-M3 Hard IP)
Board: A2F-LNX-EVB Rev 2.A, www.emcraft.com
DRAM:  16 MB
Flash:  8 MB
In:    serial
Out:   serial
Err:   serial
Net:   Core10/100
Hit any key to stop autoboot:  0
## Booting kernel from Legacy Image at 74020000 ...
   Image Name:   Linux-2.6.33-arm1
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    821816 Bytes = 802.6 kB
   Load Address: 70008000
   Entry Point:  70008001
   Verifying Checksum ... OK
   Loading Kernel Image ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
ï¿½Linux version 2.6.33-arm1 (psl@ocean.emcraft.com) (gcc version 4.4.1 (Sourcery G++
Lite 2010q1-188) ) #1 Mon Feb 09 19:59:56 MSK 2015
CPU: ARMv7-M Processor [411fc231] revision 1 (ARMv7M)
CPU: NO data cache, NO instruction cache
Machine: Actel A2F
...
Freeing init memory: 292K
init started: BusyBox v1.17.0 (2015-02-09 19:58:44 MSK)
~ #
```

The Linux kernel is configured to mount a root filesystem in the external RAM using the `initramfs` file system. `initramfs` is populated with required files and utilities at the kernel build time and then simply linked into the Linux image. `initramfs` doesn't have hard limits on its size and is able to grow using the otherwise unused RAM memory.

The Linux image installed on the board provides a demonstration of basic shell and network capabilities of Linux Cortex-M. In addition to that, the demo provides support for Flash partitioning and persistent data storage using the JFFS2 journalled file system for the external Flash memory.

Here is how you can test some of these capabilities.

From a local host, test that the target is accessible over network (assuming that the board is assigned with the `172.17.4.200` IP address):

```
[psl@ocean linux-cortexm-1.13.0]$ ping 172.17.4.200
PING 172.17.4.200 (172.17.4.200) 56(84) bytes of data.
64 bytes from 172.17.4.200: icmp_seq=1 ttl=64 time=3.11 ms
64 bytes from 172.17.4.200: icmp_seq=2 ttl=64 time=0.900 ms
64 bytes from 172.17.4.200: icmp_seq=3 ttl=64 time=0.854 ms
```

On the target, connect to a local host using `telnet`:

```
~ # telnet 172.17.0.212

Entering character mode
Escape character is '^]'.

Fedora release 12 (Constantine)
Kernel 2.6.32.26-175.fc12.i686 on an i686 (7)
login: psl
Password:
Last login: Mon Jan 31 17:38:57 from 172.17.4.199
[psl@pvr ~]$ logout
Connection closed by foreign host
~ #
```

Start the `telnet` daemon to allow connections to the target:

```
~ # telnetd
~ #
```

Connect to the target from a local host using `telnet` (hit `Enter` on the password prompt):

```
[psl@ocean linux-cortexm-1.13.0]$ telnet 172.17.4.200
Trying 172.17.4.200...
Connected to 172.17.4.200.
Escape character is '^]'.

a2f-lnx-evb login: root
Password:
~ #
```

Note: *For those targets that have 8 MBytes of RAM, it may be necessary to kill some of the processes that are no longer needed for subsequent steps of the demonstration sessions shown below. Unless some of those processes are killed, it may be not possible to run the follow-up steps in the session due to insufficient run-time RAM memory. For instance, to kill the* `telnetd` *daemon after the Telnet functionality has been tested use the following commands:*

```
~ # ps
  PID USER       VSZ STAT COMMAND
...
   23 root       324 S    telnetd
...
~ # kill -9 23
```

Start the `dropbear` SSH daemon to allow secure connections to the target:

```
~ # dropbear
~ #
```

Connect to the target from a local host using `ssh` (hit `Enter` on the password prompt):

```
[psl@ocean linux-cortexm-1.13.0]$ ssh root@172.17.4.200
The authenticity of host '172.17.4.200 (172.17.4.200)' can't be established.
DSA key fingerprint is d2:d1:5f:dd:84:65:1d:2f:ee:69:0c:85:d0:22:0c:87.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '172.17.4.200' (DSA) to the list of known hosts.
root@172.17.4.200's password:
~ #
```

On the target, configure a default gateway and the name resolver. Note how the sample configuration below makes use of the public name server provided by Google. Note also use of `vi` to edit target files under Linux:

```
~ # route add default gw 172.17.0.1
~ # vi /etc/resolv.conf
nameserver 8.8.8.8
~
```

Use `wget` to download a file from a remote server:

```
~ # wget ftp://ftp.gnu.org/README
Connecting to ftp.gnu.org (140.186.70.20:21)
README               100% |******************************|  1765  --:--:-- ETA
~ # cat README
This is ftp.gnu.org, the FTP server of the the GNU project.

Comments, suggestions, problems and complaints should be reported via
email to <gnu@gnu.org>.
...
```

Use `ntpd` to synchronize the time on the target with the time provided by a public server:

```
~ # date
```

```
Thu Jan  1 00:03:08 UTC 1970
~ # ntpd -p 0.fedora.pool.ntp.org
~ # sleep 5
~ # date
Mon Feb 09 17:06:34 UTC 2015
~ #
```

Mount a directory exported by a development host over NFS:

```
~ # mount -o nolock,rsize=1024 172.17.0.212:/opt/tmp /mnt
~ # mount
rootfs on / type rootfs (rw)
proc on /proc type proc (rw,relatime)
none on /dev/pts type devpts (rw,relatime,mode=600,ptmxmode=000)
172.17.0.212:/opt/tmp on /mnt type nfs
(rw,relatime,vers=3,rsize=1024,wsize=32768,namlen=255,hard,nolock,proto=udp,port=65535,t
imeo=7,retrans=3,sec=sys,mountport=65535,mountproto=,addr=172.17.0.212)
~ # ls mnt
CREDITS        busybox        hello.c        hello2.gdb     tests.tgz
address_cache  demo           hello.gdb      runtests.sh    uImage
bacserv        event          hello.good     test           window
bin            hello          hello2         test.log
~ #
```

Start the HTTP daemon:

```
~ # httpd -h /httpd/html
~ #
```

From a local host, open a Web browser to `http://172.17.4.200` and watch the demo web page provided by the Cortex-M target.

Next step is to test SNMP. Make sure that the daemon `snmpd` is not running on your host. Create the following configuration file `snmpd.conf`:

```
agentAddress udp:161
rocommunity public  default
rwcommunity private default
```

and run the daemon `snmpd` on the host with the above configuration:

```
[root@vladimir net-snmp]# snmpd -C -c snmpd.conf -Le -f
```

Then, run the `snmpget` application on the target and collect information about the host using the `snmpget` application:

```
~ # snmpget -c public 172.17.0.253 .1.3.6.1.2.1.1.1.0
.1.3.6.1.2.1.1.1.0 = STRING: "Linux vladimir.emcraft.com 2.6.32.26-175.fc12.i686.PAE #1
SMP Wed Aug 24 16:45:50 UTC 2011 i686"
```

Make sure that the daemon `snmptrapd` is not running on your host. Create the following configuration file `snmptrapd.conf`:

```
snmpTrapdAddr udp:162
authCommunity log public
```

and run the daemon `snmptrapd` on the host with the above configuration:

```
[root@vladimir net-snmp]# snmptrapd -C -c snmptrapd.conf -Le -f
```

Then, send a trap from the target to the host using the `snmptrap` application:

```
~ # snmptrap -v 1 -c public 172.17.0.253 '1.2.3.4.5.6' '192.193.194.195' 6 99 '55'
1.11.12.13.14.15 s "teststring"
```

On the host you should see a log message similar to the one below:

```
2011-08-23 13:45:34 192.193.194.195(via UDP: [172.17.4.201]:44563->[172.17.0.253]) TRAP,
SNMP v1, community public
        iso.2.3.4.5.6 Enterprise Specific Trap (99) Uptime: 0:00:00.55
        iso.11.12.13.14.15 = STRING: "teststring"
```

Erase the third partition of the external Flash, format it as a JFFS2 file system and then mount the newly created file system to a local directory. Copy some files to the persistent JFFS2 file system storage:

```
~ # flash_eraseall -j /dev/mtd2
Erasing 64 Kibyte @ 500000 - 100% complete.Cleanmarker written at 4f0000.
~ # mkdir /m
~ # mount -t jffs2 /dev/mtdblock2 /m
~ # cp /bin/busybox /m
~ # cp /etc/rc /m
~ # ls -lt /m
-rwxr-xr-x    1 root     root           389 Jan  1 00:19 rc
-rwxr-xr-x    1 root     root        238724 Jan  1 00:18 busybox
~ # df
Filesystem           1K-blocks      Used Available Use% Mounted on
...
/dev/mtdblock2            5120       436      4684   9% /m
~ # /m/busybox echo Hello from Flash
Hello from Flash
~ # umount /m
```

Reboot the target:

```
~ # reboot -f
Restarting system.


U-Boot 2010.03-a2f-lnx-evb-1.13.0 (Feb 09 2015 - 19:43:45)
...
```

4. Linux Cortex-M Software Development Environment

4.1. Distribution and Installation

4.1.1. Distribution Image

The Linux Cortex-M development software is distributed as a `linux-<mcu>-<release>.zip` file available for download from the Emcraft site.

When unpacked, this archive contains, along with the other materials, the software distribution file `linux-<MCU>-<release>.tar.bz2`. That file can be installed to an arbitrary directory on your Linux development host, as follows:

```
[psl@ocean SF]$ mkdir release
[psl@ocean SF]$ cd release
[psl@ocean release]$ tar xvjf ../linux-A2F-1.13.0.tar.bz2
linux-cortexm-1.13.0/
linux-cortexm-1.13.0/linux/
linux-cortexm-1.13.0/linux/lib/
...
[psl@ocean SF]$ ls -l linux-cortexm-1.13.0
total 24
drwxr-xr-x  3 psl users 4096 2015-02-09 17:06 A2F
-rwxr-xr-x  1 psl users  315 2015-02-09:26 ACTIVATE.sh
drwxr-xr-x 24 psl users 4096 2015-02-09 21:32 linux
drwxr-xr-x  5 psl users 4096 2015-02-09 20:16 projects
drwxr-xr-x  3 psl users 4096 2015-02-09 19:14 tools
drwxr-xr-x 31 psl users 4096 2015-02-09 19:16 u-boot
```

You do not need to be the superuser (`root`) in order to install the Linux Cortex-M distribution. The installation can be performed from an arbitrary unprivileged user account.

4.1.2. Installation Tree

Having been installed onto the Linux host, the Linux Cortex-M provides the following files and directories, relative to the top of the installation directory:

- `A2F/` - this is a directory with target components;
- `A2F/busybox/` - `busybox` source and development tree;
- `A2F/dropbear/` - `dropbear` SSH server source and development tree;
- `A2F/net-snmp/` - source and development tree of the `net-snmp` package;
- `A2F/uclibc` - source and development tree of the uClibc package;
- `A2F/gdb-2011.03` - source and development tree of the GNU Debugger package;
- `A2F/hostapd-1.0` - source and development tree of the HostAP Daemon package;
- `A2F/wireless_tools` - source and development tree of the Wireless Tools package;
- `A2F/libnl-3.2.11` - source and development tree of the Netlink Protocol Library package;
- `A2F/netperf-2.6.0` - source and development tree of the Netperf network benchmarking package;
- `A2F/root` - pre-built target binaries ready for use on the target;
- `u-boot/` - U-Boot source and development tree;
- `linux/` - Linux (uClinux) kernel source and development tree;
- `projects/` - sample projects (embedded applications);
- `tools/` - development tools;
- `tools/bin/mkimage` - utility used by the Linux Cortex-M kernel build process to create a bootable U-Boot kernel image such as `uImage`;
- `ACTIVATE.sh` - shell script you need to perform in order to activate the Linux Cortex-M development environment on your host.

4.1.3. GNU Cross-Build Tools

As a next step in the Linux Cortex-M installation procedure, you need to download the GNU cross-build tools for the Cortex-M target. The tools are available from CodeSourcery and can be downloaded from the following URL:

http://www.codesourcery.com/sgpp/lite/arm/portal/package6503/public/arm-uclinuxeabi/arm-2010q1-189-arm-uclinuxeabi-i686-pc-linux-gnu.tar.bz2

It is recommended that you install the GNU development tools to the `tools/` directory of the Linux Cortex-M installation, as follows:

```
[psl@pvr linux-cortex-m-1.13.0]$ cd tools/
[psl@pvr tools]$ wget
http://www.codesourcery.com/sgpp/lite/arm/portal/package6503/public/arm-uclinuxeabi/arm-
2010q1-189-arm-uclinuxeabi-i686-pc-linux-gnu.tar.bz2
...
Saving to: arm-2010q1-189-arm-uclinuxeabi-i686-pc-linux-gnu.tar.bz2
...
[psl@pvr tools]$ tar xvfj arm-2010q1-189-arm-uclinuxeabi-i686-pc-linux-gnu.tar.bz2
arm-2010q1/arm-uclinuxeabi/
arm-2010q1/arm-uclinuxeabi/libc/
...
```

It is possible to install the tools to an alternative location, however, should you do that, you will need to modify the `ACTIVATE.sh` script to provide a correct path to the installed tools

(specifically, `PATH` must include a correct path to the directory where you have installed the cross-tools).

The tools can be installed from an unprivileged account.

In case you already have the toolchain installed on your development host (it would be the case, for instance, if you have more than one installation of Linux Cortex-M on your host) and the underlying filesystem supports symbolic links (DOS FAT does not), you do not have to install the whole toolchain again. Instead, do the following:

```
[psl@pvr linux-cortexm-1.13.0]$ cd tools/
[psl@pvr tools]$ ln -s <path-to-tools>/arm-2010q1 .
```

Note: *The CodeSourcery tools are 32-bit applications and need a compatibility library to run on 64-bit Linux OS distributions. On the recent Fedora distributions, the compatibility package (e.g.* `glibc-2.17-14.fc19.i686`*) is installed by default. On Debian, you need to install it running* `sudo apt-get install ia32-libs`*.*

4.1.4. Activation

Whenever you want to activate a Linux Cortex-M development session, go to the top of your Linux Cortex-M installation and run:

```
[psl@pvr linux-cortexm-1.13.0]$ . ACTIVATE.sh
```

Note the space after the dot. This command has the same effect as "`source ACTIVATE.sh`". This command sets up (exports) the following environment variables required by the Linux Cortex-M development environment:

- `INSTALL_ROOT=`*<dir>* - root directory of the installation. This variable is used by the Linux Cortex-M `make` system as well as in the *project*`.initramfs` file to provide a reference to the Linux Cortex-M installation directory;
- `CROSS_COMPILE=arm-uclinuxeabi-` - Reference to the cross-tools used to build U-Boot and the Linux kernel;
- `CROSS_COMPILE_APPS=arm-uclinuxeabi-` - Reference to the cross-tools used to build user-space applications and tools;
- `PATH=$INSTALL_ROOT/tools/arm-2010q1/bin:$INSTALL_ROOT/tools/bin:`*<orig_path>* - Path to the cross-tools used by the shell;
- `MCU=`*<arch>* - MCU architecture supported by the Linux Cortex-M distribution.

4.1.5. Dependency on Host Components

The Linux Cortex-M distribution has the following dependencies on Linux-host software components. Please consult the notes below in case the Linux Cortex-M development environment does not function as documented in the sections that follow.

- The U-Boot, `busybox` and Linux kernel build systems require that certain host packages be installed on the development host to function correctly. These packages are: `make`, `gcc`, `perl` and some others. Please refer to `linux/Documentation/Changes` for a list of host tools required to build the Linux kernel. The same set of tools is required for the U-Boot and `busybox` build.
- The Linux Cortex-M GNU debugger (`tools/bin/arm-uclinuxeabi-gdb`) and the configuration scripts require that the following dynamically-linked libraries to be installed on the host:
  - `libncurses.so.5` and `libtinfo.so.5` (the `ncurses-libs` package);
  - `libexpat.so.1` (the `expat` package).
- The CodeSourcery cross-build tools. Please refer to the toolchain `README` at http://www.codesourcery.com/sgpp/lite/arm/portal/doc7632/getting-started.pdf.

- The `tools/mkimage` utility. It is expected that the utility will work as is on most of today's Linux distributions. If it fails to run on a certain host, you can build it for your specific Linux host from the sources included in the Linux Cortex-M distribution by running `make tools` from the top of the U-Boot tree.
- To exercise the sample projects included to the Linux Cortex-M distribution, you need to install the following host packages:
  - NFS server;
  - TFTP server;
  - Terminal emulator (picocom, minicom, kermit or PuTTY).

  If your target board uses USB for the serial console connection you might need to install the FTDI USB-UART driver from http://www.ftdichip.com/Drivers/VCP.htm for your host.

4.2. Projects Framework

4.2.1. Multiple Target Projects

In the Linux Cortex-M installation, there is a directory called `projects`, which provides a framework that you will be able to use to develop multiple projects (embedded applications) from a single installation of Linux Cortex-M. This directory has the following structure:

- `projects/Rules.make` - build rules common for all projects;
- *project1/* - *Project1* source files and build tree;
- *project2/* - *Project2* source files and build tree;
- ...

The installation provides three sample projects, called `hello` (A `Hello, world!` application), `networking` (a basic shell and networking demonstration) and `developer` (a template project demonstrating development of a custom Linux Cortex-M application). You will be able to add more projects to this directory and develop your own embedded applications using this framework.

Each project directory (such as `projects/hello`) contains the following configuration files:

- *project*`.kernel.${MCU}` - Kernel configuration for this project. Note that the kernel configuration is specific for the microcontroller architecture supported by the distribution;
- *project*`.busybox` - `busybox` configuration for this project;
- *project*`.initramfs` - File defining content of the `initramfs` filesystem for this project.

When you run `make linux` (or simply `make`) from the project directory, the build system builds project-specific versions of the Linux kernel and `busybox`, then creates an `initramfs` filesystem containing the newly built `busybox` binary as well as other target files defined by the `initramfs` filesystem specification file, and finally wraps it all up into a Linux image ready for download to the target by U-Boot.

Each project directory has a `Makefile` identifying the following `make` variables. You need to set these variables correctly for your project:

- `SAMPLE` - the name of the project. The downloadable Linux image has this name, with the `.uImage` extension. Additionally, `SAMPLE` is passed to the `initramfs` build subsystem as a reference to the directory where filesystem binaries reside.
- `CUSTOM_APPS` - this variable provides a list of the sub-directories containing custom applications for the project. Custom applications are built in the specified order, prior to building the Linux kernel and `initramfs` images. If a project doesn't have custom applications, the variable should be left empty.

The following `make` targets are implemented in `projects/Rules.make` common build rules file:

- `all` or `linux` - build an uncompressed Linux image ready for download to the target and place it to the project directory as *<project>*`.uImage`;
- `kclean` - clean up the Linux kernel source tree by running `make clean` in `$(INSTALL_ROOT)/linux`;
- `bclean` - clean up the `busybox` source tree by running `make clean` in `$(INSTALL_ROOT)/A2F/busybox`;
- `aclean` - clean up the custom application source trees by running `make clean` in each application source directory, as listed in `CUSTOM_APPS`;
- `clean` - clean up the entire project, by removing *<project>*`.uImage` and then cleaning up the Linux kernel and `busybox` trees, as described above. Additionally, if `CUSTOM_APPS` is not empty, `make clean` is performed in each custom application sub-directory;
- `kmenuconfig` - configure the Linux kernel by running `make menuconfig` in the Linux source tree. Copy the resultant configuration file to the project directory as *project*`.kernel.${MCU};`
- `bmenuconfig` - configure the `busybox` tool by running `make menuconfig` in the `busybox` source tree. Copy the resultant configuration file to the project directory as *project*`.busybox`;
- `clone new=`*newproject* - clone the current project into a new project with a specified name. The current project directory (with all sub-directories) is copied into the new project directory. The project kernel, `busybox` and `initramfs` configuration files are copied with the new name. In `Makefile SAMPLE` is set to the name of the new project.

The main idea behind this framework is that each project keeps its own configuration for the Linux kernel (in the *project*`.kernel.${MCU}` file), its own definition of the contents of the target file system (in the *project*`.initramfs` file) and its own configuration of `busybox` (in the *project*`.busybox` file). These files are enough to rebuild both the project kernel and the project file system (with a specific configuration of the `busybox` tool embedded) from scratch.

If a project makes use of a custom application specific to the project, such an application must be built from the sources located in an arbitrarily-named sub-directory local to the project directory. There could be several sub-directories in a project, one per a custom application. Each subdirectory will have its own `Makefile` defining rules for building the custom application from the corresponding sources.

Each custom application listed in `CUSTOM_APPS` must have a `Makefile` in the custom application sub-directory defining the following targets:

- `all` - build the application from the sources;
- `clean` - clean anything that has been previously built.

Content of the `initramfs` file system is defined for a project in the file named *project*`.initramfs`. Note that this file makes use of `${INSTALL_ROOT}` to provide a reference to the top of the Linux Cortex-M installation directory.

4.2.2. Hello World Demo Project

Resides in `projects/hello`.

A single-application Linux configuration demonstrating the minimal possible kernel/initramfs memory requirements. The application starts as an `init` process as soon as the kernel has mounted the root filesystem, mounts the `/proc` filesystem and prints out the content of `/proc/meminfo`. Then it proceeds to print a "Hello" message in an endless loop. There is no way to interrupt or stop the application (rather than resetting the system).

4.2.3. Networking Demo Project

Resides in `projects/networking`.

Implements the functionality of the demo system that comes pre-installed on the A2F-LNX-EVB board. Please refer to Section 3 for a detailed description of the demo.

The demo includes the `dropbear` SSH server, configured to accept only DSA connections. To configure the server to accept RSA connections edit `networking.initramfs` and rebuild the demo to add the `/etc/dropbear/dropbear_rsa_host_key` file to the target file system.

4.2.4. Developer Demo Project

Resides in `projects/developer`.

`developer` is intended as a template project that can be used to jump-start development of custom user-space applications and loadable kernel modules.

From the kernel configuration / start-up sequence perspective, `developer` is similar to `networking`. `developer` is ready for NFS-mount, although the user must explicitly enable the `mount` command in `local/rc` and edit it for his local environment.

From the `busybox` configuration perspective, `developer` enables `insmod`, `lsmod`, `rmmod` to allow management of loadable kernel modules.

`developer` has a custom-application directory called `app`. It contains:

- Loadable device driver called `sample.c`, which implements a device driver for a pseudo-device that returns a certain string on attempt to read the entire device; rejects any attempts to write the device;
- User-space application called `app.c`, which opens the `sample` device driver using a `/dev/sample` device node file, reads the device until `EOF` on a per-byte basis, and prints the read characters to the user terminal (`stdout`).

Please refer to Section 6 to find our more about how `developer` can be used to jump-start Linux Cortex-M development.

5. U-Boot

5.1. Pointers to Detailed Information on U-Boot

Linux Cortex-M makes use of the U-Boot firmware to bring the Cortex-M target up from a power-on / reset and to boot Linux onto the target.

U-Boot is a popular firmware monitor developed and maintained by DENX Software Engineering (www.denx.de). DENX publishes extensive U-Boot user documentation at their web site.

This section provides a brief introduction to U-Boot as relevant to the Linux Cortex-M software environment.

5.2. U-Boot: Theory of Operation

A typical Linux Cortex-M boot-up sequence is as follows:

- U-Boot firmware runs on the target from the integrated Flash (eNVM) / eSRAM (no external memory required) and performs all required initialization from power-on / reset, including setting up the memory controller for external memory accesses;
- U-Boot relocates Linux image from external Flash to external RAM and passes control to the kernel entry point in RAM. External Flash resides at the memory interface. Linux image can be compressed to save the Flash storage; in this case, U-Boot uncompresses the image when relocating it to RAM;
- Linux proceeds to boot up and mount a RAM-based filesystem (`initramfs`) as a root filesystem. `initramfs` is populated with required files and directories at build time and is then simply linked into the kernel.

U-Boot operation is controlled at run-time by the so-called "environment variables", which are stored in persistent storage such as external Flash. Each environment variable controls a certain aspect of U-Boot operation. For instance, there is an environment variable called `bootargs` that defines a kernel command string that U-Boot passes to the Linux kernel as the parameters.

5.3. Selected Use Scenarios

5.3.1. U-Boot Environment

To manipulate the U-Boot environment the following commands are used:

- `printenv <var>` - print the value of the variable `var`. Without arguments, print all environment variables:

```
A2F-LNX-EVB> printenv
bootargs=console=ttyS0,115200 panic=10
bootcmd=run flashboot
bootdelay=3
baudrate=115200
hostname=a2f-lnx-evb
loadaddr=70000000
addip=setenv bootargs ${bootargs}
ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:eth0:off
flashaddr=74020000
flashboot=run addip;bootm ${flashaddr}
netboot=tftp ${image};run addip;bootm
image=a2f/uImage
ethact=Core10/100
stdin=serial
stdout=serial
stderr=serial
ethaddr=00:22:44:66:88:AA

Environment size: 430/4092 bytes
A2F-LNX-EVB>
```

Note that the TTY device number (`ttyS0`) and other default options may differ on various boards.

- `setenv <var> <val>` - set the variable `var` to the value `val`:

```
A2F-LNX-EVB> setenv image psl/a2f/hello.image
A2F-LNX-EVB> printenv image
image=psl/a2f/hello.image
A2F-LNX-EVB>
```

  Running `setenv <var>` will un-set the U-Boot variable.

- `saveenv` - save the up-to-date U-Boot environment, possibly updated, using `setenv` commands, into persistent storage:

```
A2F-LNX-EVB> saveenv
Saving Environment to Flash...
Un-Protected 2 sectors
Erasing Flash...
.. done
Erased 2 sectors
Writing to Flash... done
Protected 2 sectors
A2F-LNX-EVB>
```

5.3.2. Autoboot

The autoboot sequence in U-Boot is controlled by the following environment variables:

- `bootcmd` - command to execute automatically after reset. To disable the autoboot, undefine this variable;

- `bootdelay` - delay, in seconds, before running the autoboot command. During the `bootdelay` countdown, you can interrupt the autobooting by pressing any key.

5.3.3. Networking in U-Boot

U-Boot supports downloading of images over network using the TFTP service.

To enable networking, the following U-Boot variables must be set to values matching the local network configuration:

- `ipaddr` - IP address of your A2F-LNX-EVB board;
- `serverip` - IP address of the TFTP server;
- `netmask` - local network mask (optional variable);
- `image` - filename of an image loaded from the TFTP server (optional variable).

After you have defined the above variables, you can use the `tftp` U-Boot command to download images to RAM. For example:

```
A2F-LNX-EVB> tftp psl/a2f/networking.uImage
Core10/100: link up (100/Full)
Using Core10/100 device
TFTP from server 172.17.0.1; our IP address is 172.17.10.200
Filename 'psl/a2f/networking.uImage'.
Load address: 0x70000000
Loading: #########################################################
done
Bytes transferred = 821888 (c8a80 hex)
A2F-LNX-EVB>
```

5.3.4. Flash-related Commands

To program a Linux image to the Flash, use the `update` U-Boot macro as follows:

```
A2F-LNX-EVB> print update
update=tftp ${image};prot off ${flashaddr} +${filesize};era ${flashaddr}
+${filesize};cp.b ${loadaddr} ${flashaddr} ${filesize}
A2F-LNX-EVB> setenv image psl/a2f/networking.uImage
A2F-LNX-EVB> run update
```

Alternatively, you can program a Linux image to the Flash using the following sequence of commands (the commands are only valid for the NOR type of the Flash):

- Download the image to RAM:

```
A2F-LNX-EVB> tftp psl/a2f/networking.uImage
Core10/100: link up (100/Full)
Using Core10/100 device
TFTP from server 172.17.0.1; our IP address is 172.17.10.200
Filename 'psl/a2f/networking.uImage'.
Load address: 0x70000000
Loading: #########################################################
done
Bytes transferred = 821888 (c8a80 hex)
A2F-LNX-EVB>
```

- Unprotect Flash:

```
A2F-LNX-EVB> protect off all
Un-Protect Flash Bank # 1
.................................................................................
........................................ done
A2F-LNX-EVB>
```

- Erase the destination Flash area:

```
A2F-LNX-EVB> erase ${flashaddr} +${filesize}
```

```
............ done
Erased 13 sectors
A2F-LNX-EVB>
```

- Copy the image from RAM to Flash:

```
A2F-LNX-EVB> cp.b ${loadaddr} ${flashaddr} ${filesize}
Copy to Flash... done
A2F-LNX-EVB>
```

5.3.5. U-Boot Self-Upgrade

The Linux Cortex-M distribution provides a full tree of the U-Boot source files. This allows you to configure or otherwise enhance U-Boot for your specific needs.

To build U-Boot, do the following from the Linux Cortex-M installation on your development host (refer to Section 4):

1. Go to the Linux Cortex-M installation and activate it (unless you have activated it already):

```
[psl@ocean linux-cortexm-1.13.0]$ . ACTIVATE.sh
[psl@ocean linux-cortexm-1.13.0]$
```

2. Go to the top of the U-Boot source tree:

```
[psl@ocean linux-cortexm-1.13.0]$ cd u-boot/
[psl@ocean u-boot]$
```

3. Configure U-Boot for your specific board. Please refer to the Starter Kit Guide (or BSP Guide) for your hardware board for the name of the corresponding U-Boot configuration target. For example, the following command configures U-Boot for the Emcraft A2F-LNX-EVB board:

```
[psl@ocean u-boot]$ make a2f-lnx-evb_config
Configuring for a2f-lnx-evb board...
[psl@ocean u-boot]$
```

4. Build the U-Boot image ready for download to the target over TFTP and copy it to the TFTP server directory:

```
[psl@ocean u-boot]$ make
[psl@ocean u-boot]$ cp u-boot.bin /tftpboot/psl/a2f
```

To upgrade the firmware, do the following on the target:

1. Configure the networking in U-Boot if necessary.
2. Download the U-Boot image from the TFTP server:

```
A2F-LNX-EVB> tftp psl/a2f/u-boot.bin
Core10/100: link up (100/Full)
Using Core10/100 device
TFTP from server 172.17.0.1; our IP address is 172.17.10.200
Filename 'psl/a2f/u-boot.bin'.
Load address: 0x70000000
Loading: #######
done
Bytes transferred = 96052 (17734 hex)
A2F-LNX-EVB>
```

3. Program the new U-Boot image to the embedded Flash of the MCU using the `cptf` command. Note that the `cptf` command automatically resets the target upon completion of the eVNM upgrade, which calls the newly programmed image from the reset vector in the eNVM. Note also that the first parameter passed to the `cptf` command specifies the base address of the embedded Flash area being updated by the command. Depending on your CPU, pass the following address as the first parameter to `cptf`:
   - STM32F = 0x08000000

- LPC17xx = 0x0
- SmartFusion and SmartFusion2 = 0x0
- Kinetis = 0x0

As an example, the following command performs the U-boot self-upgrade on SmartFusion:

```
A2F-LNX-EVB> cptf 0x0 ${loadaddr} ${filesize} 1
cptf: Updating eNVM. Please wait ...


U-Boot 2010.03 (Feb 09 2015 - 20:15:52)

CPU: SmartFusion FPGA (Cortex-M3 Hard IP)
Board: A2F-LNX-EVB Rev 2.A, www.emcraft.com
...
A2F-LNX-EVB>
```

Note: *Be extra-careful when performing the* `cptf` *command specified above. In case you program an incorrect U-Boot image to the eNVM, this will render the board non-bootable. The only resort in this scenario is to program the U-boot image to the board over the JTAG port.*

6. Sample Development Session

6.1. Sample Session Outline

This sample session illustrates a possible software development cycle in Linux Cortex-M.

We will do the following:

1. Create a new project by cloning it off of the `developer` demo project.
2. Modify the new project to automatically NFS-mount a development tree from the Linux host where we have installed Linux Cortex-M.
3. Program the new project image into Flash so that it autoboots on the target on each power-up / reset.
4. Develop a new custom application and a kernel module and debug them from the NFS-mounted host directory, without having to even reboot the target.

6.2. Create a New Project

This section creates a new project as a clone of the existing project and makes sure the new project builds:

1. Start off of the `developer` project and create a clone called `my_developer`:

```
-bash-3.2$ pwd
/home/vlad/test/linux-cortexm-1.13.0/projects/developer
-bash-3.2$ make clone new=my_developer
New project created in /home/vlad/test/linux-cortexm-1.13.0/projects/my_developer
-bash-3.2$
```

2. Go to the new project directory, build it and copy the downloadable Linux image to the TFTP server directory:

```
-bash-3.2$ cd ../my_developer/
-bash-3.2$ make
...
  Image arch/arm/boot/uImage is ready
...
-bash-3.2$ cp my_developer.uImage /tftpboot/vlad/v
```

6.3. Set Up Target for Debugging of the New Project

This section sets up U-Boot for debugging of the new project on the target:

1. Reset the target and enter the U-Boot command monitor, hitting any key to stop the autoboot:

```
...
Hit any key to stop autoboot:  0
A2F-LNX-EVB>
```

2. Define the IP addresses for the target and a TFTP server; define the name of the image to be downloaded by `tftpboot`:

```
A2F-LNX-EVB> setenv ipaddr 172.17.4.150
A2F-LNX-EVB> setenv serverip 172.17.0.1
A2F-LNX-EVB> setenv image vlad/v
```

3. Check that U-Boot defines appropriate commands (macros) for booting the Linux image from a TFTP server and running it on the target:

```
A2F-LNX-EVB> printenv netboot
netboot=tftp ${image};run addip;bootm
A2F-LNX-EVB> printenv addip
addip=setenv bootargs ${bootargs}
ip=${ipaddr}:${serverip}:${gatewayip}:${netmask}:${hostname}:eth0:off
```

4. Save the updated environment in Flash:

```
A2F-LNX-EVB> saveenv
Saving Environment to Flash...
Un-Protected 2 sectors
Erasing Flash...
.. done
Erased 2 sectors
Writing to Flash... done
Protected 2 sectors
A2F-LNX-EVB>
```

5. Boot the Linux image over the network and test that the networking is functional. Note that given a correct `ipaddr` setting in U-Boot, the kernel brings up the Ethernet interface in Linux automatically, without an explicit `ifconfig` command:

```
A2F-LNX-EVB> run netboot
Core10/100: link up (100/Full)
Using Core10/100 device
TFTP from server 172.17.0.1; our IP address is 172.17.4.150
Filename 'vlad/v'.
Load address: 0x70000000
Loading: ##################################################################
         ##################################
done
Bytes transferred = 1439232 (15f600 hex)
## Booting kernel from Legacy Image at 70000000 ...
   Image Name:   Linux-2.6.33-arm1
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    1439168 Bytes =  1.4 MB
   Load Address: 70008000
   Entry Point:  70008001
   Verifying Checksum ... OK
   Loading Kernel Image ... OK
OK

Starting kernel ...

Ðªlinux version 2.6.33-arm1 (vlad@ocean.emcraft.com) (gcc version 4.4.1 (Sourcery G++
Lite 2010q1-188) ) #8 Fri Mar 25 17:26:37 MSK 2011
...
IP-Config: Complete:
     device=eth0, addr=172.17.4.150, mask=255.255.0.0, gw=255.255.255.255,
     host=a2f-lnx-evb, domain=, nis-domain=(none),
     bootserver=172.17.0.1, rootserver=172.17.0.1, rootpath=
Freeing init memory: 328K
```

```
init started: BusyBox v1.17.0 (2015-02-09 19:58:44 MSK)

~ # ping 172.17.0.1
PING 172.17.0.1 (172.17.0.1): 56 data bytes
64 bytes from 172.17.0.1: seq=0 ttl=64 time=11.617 ms
64 bytes from 172.17.0.1: seq=1 ttl=64 time=2.016 ms
...
```

6.4. Update the New Project

This section updates the new project for the required functionality and validates it on the target:

1. In the new project, update the target start-up script so that it automatically mounts the `projects` directory from the Linux Cortex-M installation on the development host. This makes all projects immediately available on the target allowing you to edit, build and test your sample applications and loadable device drivers without having to re-Flash or even reboot the target:

```
-bash-3.2$ vi local/rc
#!/bin/sh
mount -t proc proc /proc
mount -t sysfs sysfs /sys
mount -t devpts none /dev/pts
mkdir /mnt
mount -o nolock,rsize=1024 172.17.0.1:/home/vlad/test/linux-cortexm-1.13.0/projects /mnt
ifconfig lo 127.0.0.1
```

2. Build the updated project and copy it to the TFTP server directory:

```
-bash-3.2$ make; cp my_developer.uImage /tftpboot/vlad/v
...
```

3. Reboot the target, load the updated image over the network and test it:

```
~ # reboot -f
Restarting system.

...
Hit any key to stop autoboot:  0
A2F-LNX-EVB> run netboot
...
init started: BusyBox v1.17.0 (2015-02-09 19:58:44 MSK)
~ # mount
rootfs on / type rootfs (rw)
proc on /proc type proc (rw,relatime)
sysfs on /sys type sysfs (rw,relatime)
none on /dev/pts type devpts (rw,relatime,mode=600,ptmxmode=000)
172.17.0.1:/home/vlad/test/linux-cortexm-1.13.0/projects on /mnt type nfs
(rw,relatime,vers=3,rsize=1024,wsize=32768,namlen=255,hard,nolock,proto=udp,port=65535,t
imeo=7,retrans=3,sec=sys,mountport=65535,mountproto=,addr=172.17.0.1)
~ # ls -lt /mnt
drwxr-xr-x    4 19270    19270         4096 Mar 25  2011 my_developer
drwxr-xr-x    4 19270    19270         4096 Mar 25  2011 developer
-rw-r--r--    1 19270    19270         3581 Mar 25  2011 Rules.make
drwxr-xr-x    3 19270    19270         4096 Mar 25  2011 hello
drwxr-xr-x    3 19270    19270         4096 Mar 25  2011 networking
~ #
```

6.5. Install the New Project to Flash

This section installs the new project to Flash so that it automatically boots up on the target on any power-up / reset:

1. At the U-Boot prompt, load the Linux image into the target RAM over TFTP and program it to Flash:

```
A2F-LNX-EVB> print image
image=vlad/v
```

```
A2F-LNX-EVB> run update
Core10/100: link up (100/Full)
Using Core10/100 device
TFTP from server 172.17.0.1; our IP address is 172.17.4.150
Filename 'vlad/v'.
Load address: 0x70000000
Loading: ##################################################################
         ##################################
done
Bytes transferred = 1439232 (15f600 hex)
Un-Protect Flash Bank # 1
.................................................................................
........................................ done
Erased 22 sectors
Copy to Flash... done
A2F-LNX-EVB>
```

2. Reset the board and make sure that the new project boots from Flash in the autoboot mode:

```
A2F-LNX-EVB> reset
resetting ...
...
Hit any key to stop autoboot:  0
## Booting kernel from Legacy Image at 74020000 ...
...
init started: BusyBox v1.17.0 (2015-02-09 19:58:44 MSK)
~ # ls -lt /mnt
drwxr-xr-x    4 19270    19270         4096 Mar 25  2011 my developer
...
~ #
```

6.6. Develop a Custom Loadable Device Driver over NFS

1. Go to the application sub-directory in your project's directory:

```
-bash-3.2$ cd app/
-bash-3.2$ pwd
/home/vlad/test/linux-cortexm-1.13.0/projects/my_developer/app
```

2. The `developer` project you have cloned your new project off already provides a loadable kernel device driver implemented in `sample.c`. The device driver allows reading "device data" from a pseudo-device, which keeps its data in a character array defined in `sample.c`. Let's enhance the device driver to allow changing the "device data" by writing into the character array. Note that, this being a simple example of a development session, the code below assumes that the user-supplied "device-data" does not exceed 1023 bytes.

```
-bash-3.2$ vi sample.c
...
/*
 * Device "data"
 */
static char sample_str[1024] = "This is the simplest loadable kernel module\n";
...
/*
 * Device write
 */
static ssize_t sample_write(struct file *filp, const char *buffer,
                            size_t length, loff_t * offset)
{
        int ret = 0;

        /*
         * Check that the user has supplied a valid buffer
         */
        if (! access_ok(0, buffer, length)) {
                ret = -EINVAL;
                goto Done;
        }
```

```
        /*
         * Write the user-supplied string into the sample "device string".
         */
        strncpy(sample_str, buffer, length);
        sample_str[length] = '\0';
        *offset += length + 1;
        ret = length;

Done:
        d_printk(3, "length=%d\n", length);
        return ret;
}
```

3. Build the updated device driver:

```
-bash-3.2$ make
```

4. On the target, go to the application directory:

```
/mnt/my_developer/app # cd /mnt/my_developer/app/
```

5. Install the updated device driver:

```
/mnt/my_developer/app # insmod sample.ko
```

6. Test that the read operation returns the content of the built-in "device data":

```
/mnt/my_developer/app # cat /dev/sample
This is the simplest loadable kernel module
```

7. Write into `/dev/sample` in order to change the "device data" and test the device returns the updated data on a next read:

```
/mnt/my_developer/app # cat > /dev/sample
This is the new content of /dev/sample
^D
/mnt/my_developer/app # cat /dev/sample
This is the new content of /dev/sample
```

8. Unload the device driver:

```
/mnt/my_developer/app # rmmod sample
```

9. Iterate to update and test your custom device driver from the NFS-mounted host directory.

6.7. Develop a Custom User-Space Application over NFS

1. You are in the application sub-directory in your project's directory:

```
-bash-3.2$ pwd
/home/vlad/test/linux-cortexm-1.13.0/projects/my_developer/app
```

2. The `developer` project you have cloned your new project off already provides a user-space application implemented in `app.c`. This is an example so let's add a simple print-out to the application code:

```
-bash-3.2$ vi app.c
...
        printf("%s: THIS IS CONTENT OF %s:\n", app_name, dev_name);

        /*
         * Read the sample device byte-by-byte
         */
        while (1) {
...
```

3. Build the updated application:

```
-bash-3.2$ make
```

4. On the target, install the device driver and test the updated application:

```
/mnt/my_developer/app # insmod sample.ko
/mnt/my_developer/app # ./app
./app: THIS IS CONTENT OF /dev/sample:
This is the simplest loadable kernel module
```

5. Unload the device driver:

```
/mnt/my_developer/app # rmmod sample
```

6. Iterate to update and test your custom application from the NFS-mounted host directory.

7. Support and Further Materials

Visit Emcraft's web site at www.emcraft.com to obtain additional materials related to Linux Cortex-M.