Vybrid VF6 SOM Starter Kit Guide

Release 1.12.2.1

Table of Contents

1. Overview

This document is the Emcraft Systems Vybrid VF6 SOM Starter Kit Guide, Release 1.12.2.1.

The VF6 SOM starter kit provides a low-cost hardware platform enabling development of embedded applications using the Freescale Vybrid VF6 microcontroller devices and the Emcraft Systems VF6 System-On-Module (VF6 SOM). The kit includes the VF6 System-on-Module itself, a Freescale Tower-compatible Development Baseboard (TWR-VF6-SOM-BSB), and a mini-USB cable for USB-based power and serial console.

Emcraft supports Linux as an operating system for the Cortex-A5 processor core and MQX as an RTOS for the Cortex-M4 core. An inter-core communication API (Multi-Core Communication, or MCC) is provided for interactions between Linux applications/drivers running on the A5 core and MQX applications running on the M4 core.

The Linux and MQX BSPs and the software development environment targeting the VF6 SOM starter kit are available for free download from the Emcraft web site. Each starter kit comes preloaded with U-Boot and a sample Linux/MQX image.

2. Product Contents

This product includes the following components.

2.1. Shippable Hardware Items

The following hardware items are shipped to customers of this product:

1. VF6 SOM board;
2. TWR-VF6-SOM-BSB baseboard;
3. Mini-USB cable UART/power interface.

Note that this product does not include any JTAG programmer tools or associated hardware items, nor a Freescale Tower or additional Freescale or third-party Tower modules. Such equipment needs to be purchased directly from respective vendors.

2.2. Downloadable Hardware Materials

The following hardware materials are available for download from Emcraft's web site to customers of this product:

1. `TWR-VF6-SOM-BSB-2A-schem.pdf` - TWR-VF6-SOM-BSB schematics in PDF format;
2. `TWR-VF6-SOM-BSB-2A-bom.xls` - TWR-VF6-SOM-BSB Bill-Of-Materials (BOM) in Excel format;
3. `VF6-SOM.IntLib` - Altium Designer 9.4 integrated library for the VF6-SOM symbol and footprint.

2.3. Downloadable Software Materials

The following software materials are available for download from Emcraft's web site to customers of this product:

1. `u-boot-nand.flash` and `u-boot-qspi.flash` – Prebuilt U-Boot binary file ready for installation onto NAND or QSPI Flash (depending on chosen boot media) of the VF6 SOM. Each kit comes preloaded with this U-Boot firmware;
2. `networking.uImage` – Prebuilt bootable image of a sample Linux/MQX configuration ready for installation and loading onto the VF6 SOM. Each kit comes preloaded with this bootable image (refer to Section 6 for detailed description of the sample image);
3. `nand_fcb.bin` – NAND Firmware configuration block image ready for installation onto the VF6 SOM;

4. `u-boot-nand-migration.qspi` – Special pre-built U-Boot binary designated for switching to NAND boot for boards that boot from QSPI;
5. `linux-VF6-1.12.2.1.tar.bz2` - Linux Vybrid software development environment, including:
   a) U-Boot firmware;
   b) Linux kernel;
   c) `busybox` and other target components;
   d) Framework for developing multiple projects (embedded applications) from a single installation, including sample projects allowing to kick-start software development for the VF6 SOM.

2.4. Downloadable Documentation Materials

The following documentation materials are available for download from Emcraft's web site:

1. `vf6-som-ha.pdf` - Emcraft Systems VF6 SOM (System-On-Module) Hardware Architecture specification;
2. `twr-vf6-som-bsb-ha.pdf` - Emcraft Systems TWR-VF6-SOM-BSB Baseboard Hardware Architecture specification;
3. `vf6-som-skg-1.12.2.1.pdf` - Emcraft Systems VF6 SOM Starter Kit Guide (this document).

3. Software Functionality

3.1. Supported Features

The following list summarizes the features and capabilities of Linux Vybrid, Release 1.12.2.1:

- U-Boot firmware:
  - Runs on the Cortex-A5 core;
  - U-Boot v2011.12;
  - Target initialization from power-on / reset;
  - Loads from external Flash and runs from RAM;
  - Serial console;
  - Ethernet driver for loading images to the target from network;
  - Serial driver for loading images to the target over UART;
  - Device driver for Flash and self-upgrade capability;
  - Device driver for storing environment and Linux images in Flash;
  - Autoboot feature, allowing boot of OS images from Flash or other storage with no operator intervention;
  - Persistent environment in Flash for customization of target operation;
  - Sophisticated command interface for maintenance and development of the target;
  - Loads Linux kernel from a software-specified boot device, typically, QSPI or NAND Flash.
- Linux:
  - Runs on the Cortex-A5 core;
  - Linux kernel v3.0.15;
  - Serial device driver and Linux console;

  - Ethernet device driver and TCP/IP networking;
  - MTD-based Flash partitioning and persistent JFFS2 Flash file system in Flash;
  - Device drivers for all key I/O interfaces of the Vybrid;
  - `busybox` v1.17;
  - POSIX `pthreads`;
  - Loadable kernel modules;
  - Specifically optimized for fast boot-up;
  - Boot of the Cortex-M4 from a binary file in the Linux file system;
  - Low-level Linux-side inter-core MCC communications supported with appropriate device drivers/libraries;
  - Large pool of pre-built Linux packages ready for the Cortex-A5 core;
  - Yocto-based build and distribution model.
- MQX:
  - Runs on the Cortex-M4 core;
  - MQX RTOS v.4.0.1;
  - Support for select I/O interfaces of the Vybrid;
  - Low-level MCC communications supported with appropriate libraries.
- Multi-Core Communication (MCC):
  - Lightweight and fast inter-core API;
  - API calls are simple send / receive;
  - Uses shared SRAM and interrupts;
  - Received data can be passed by pointer or copied.
- Development environment:
  - Linux-hosted cross-development environment;
  - Separate GNU toolchains for Linux/A5 and MQX/M4;
  - Sample projects showcasing typical embedded configurations (Linux shell, networking, MCC-based interactions between Linux/A5 and MQX/M4; Qt GUI, etc);
  - Development of multiple projects (embedded applications) from a single installation.

3.2. New and Changed Features

This section lists new and changed features of this release:

1. Support Micrel KSZ8081RNB Ethernet PHY in U-Boot and Linux.
   ID: 103292.

3.3. Known Problems & Limitations

This section lists known problems and limitations of this release:

1. The JFFS2 filesystem works unreliably with QSPI MTD devices.
   ID: 98172.

4. Hardware Setup

This section explains how to set up the VF6 SOM Starter Kit hardware.

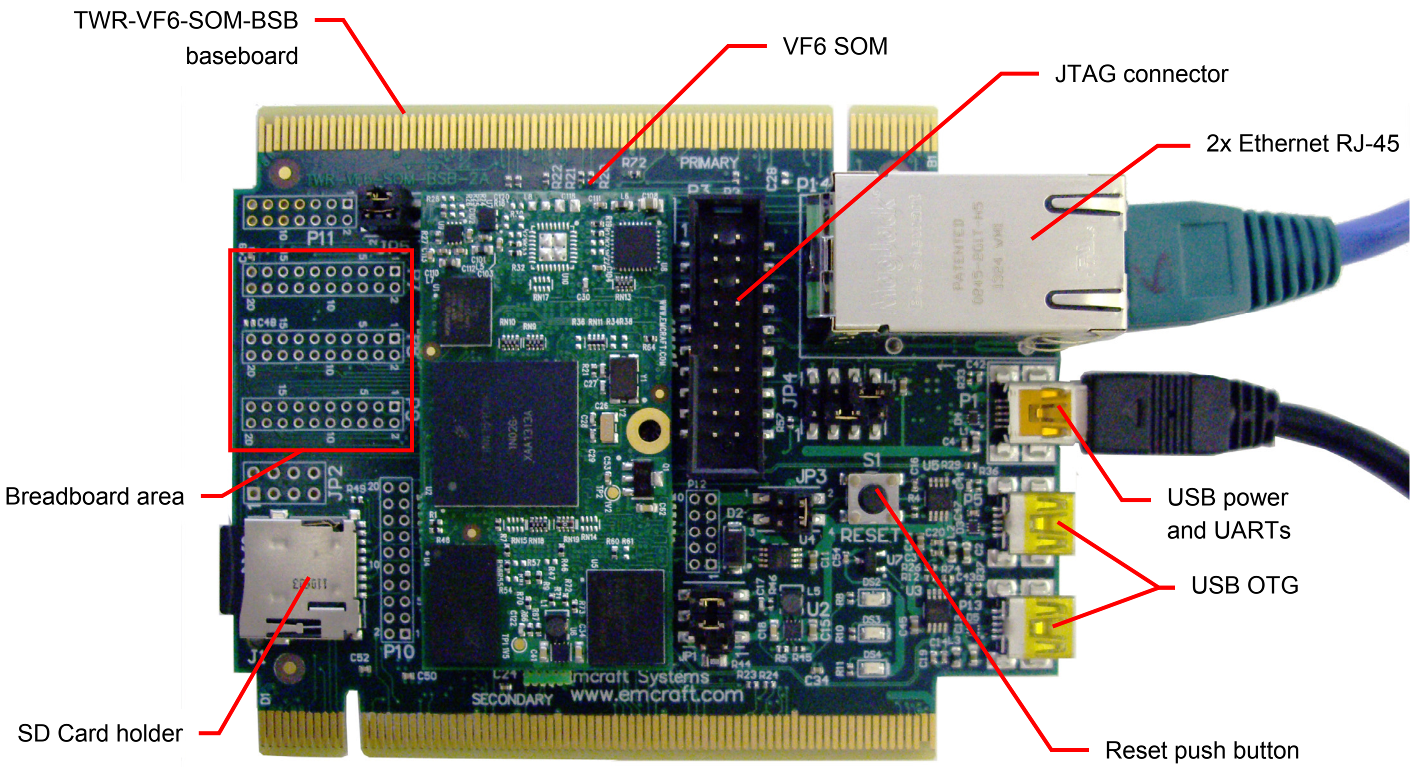
4.1. Standalone Operation

Each kit includes a Vybrid System-On-Module (VF6 SOM) and a TWR-VF6-SOM-BSB development baseboard. The VF6 SOM comes plugged into the TWR-VF6-SOM-BSB baseboard.

The TWR-VF6-SOM-BSB is a module compatible with the Freescale Tower system. However, as shipped by Emcraft, it can be used in standalone mode as well, without a Freescale Tower or any additional Tower modules.

4.1.1. Hardware Interfaces

The following picture shows the components and interfaces provided by the VF6 SOM Starter Kit in standalone mode.



4.1.2. TWR-VF6-SOM-BSB Jumpers

The following jumpers must be configured on the TWR-VF6-SOM-BSB board:

Jumper	Configuration	Notes
JP1	1-2 closed, 3-4 closed	To enable power on the VF6 SOM (VCC3) and save the battery life when the mini-USB is connected.
JP2	All pins open (SAI2 is not looped)	Used for local looping of the Vybrid SAI2 interface.
JP3	1-3 open, 2-4 closed	To use the mini-USB port as the power source.

Jumper	Configuration	Notes
JP4	1-2 open (USB1 connected to the P5 USB connector), 3-4 closed, 7-8 closed (UART2 connected to the USB-UART bridge (U1) and available on the P1 USB connector)	Controls routing of the Vybrid USB1 interface. Controls routing of the Vybrid UART2 interface.

4.1.3. Board Connections

To power the hardware platform up, simply connect it to a PC / notebook by plugging the mini-USB Y-cable into the P1 mini-USB connector on the TWR-VF6-SOM-BSB board. As soon as the connection to the PC has been made, the LED `DS2` should light up, indicating that the board is up and running.

On the PC host side, the U-Boot/Linux and MQX consoles are available via the P1 dual port UART/USB connector. The software installed on the VF6 SOM configures both serial consoles for a 115.2 Kbs terminal.

On some Linux distributions, connecting to the dual port UART/USB device causes the Modem Manager package to try opening the TTY device and sending modem commands to it, thus occupying the port. To avoid this effect, the `ModemManager` package must be disabled on the host with the following command:

```
sudo mv /usr/share/dbus-1/system-services/org.freedesktop.ModemManager.service
/usr/share/dbus-1/system-services/org.freedesktop.ModemManager.service.disabled
```

On the Linux host, the `dmesg` command can be used to figure out the TTY devices corresponding to the two serial consoles:

```
$ dmesg | tail
[495846.154337] cp210x 1-5.1.5:1.0: cp210x converter detected
[495846.216898] usb 1-5.1.5: reset full-speed USB device number 8 using ehci-pci
[495846.292179] usb 1-5.1.5: cp210x converter now attached to ttyUSB0
[495846.292643] usb 1-5.1.5: cp210x converter now attached to ttyUSB1
```

The U-Boot/Linux serial console is available on the second USB TTY device. For example:

```
$ picocom –l /dev/ttyUSB1 –b 115200
```

The MQX serial console is available on the first USB TTY device. For example:

```
$ picocom –l /dev/ttyUSB0 –b 115200
```

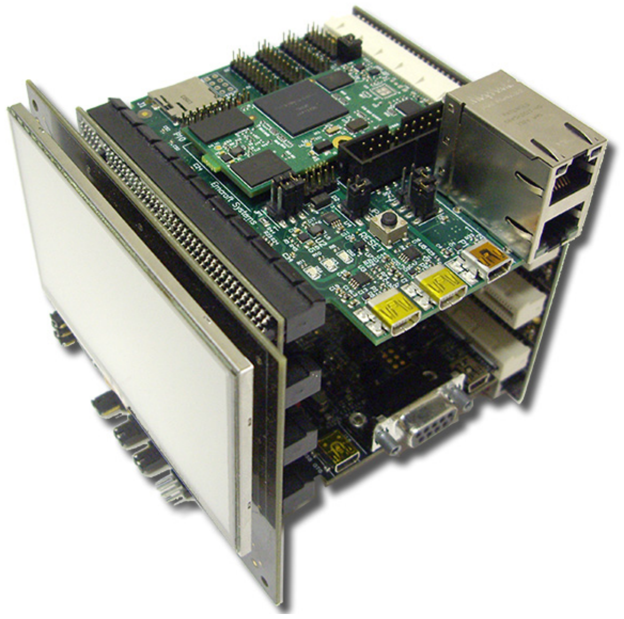
To provide network connectivity to the board, connect it into your LAN by plugging a standard Ethernet cable into the lower slot of the dual-port Ethernet connector. The board is pre-configured with an IP address of 192.168.0.2.

4.2. Operation in Freescale Tower

In addition to being able to operate in standalone mode, the TWR-VF6-SOM-BSB board can also be used as part of the modular Freescale Tower System development platform. The TWR-VF6-SOM-BSB can interoperate with all the standard peripheral modules of the Freescale Tower System, such as TWR-SER, TWR-SER2, TWR-LCD-RGB, and TWR-DOCK providing the various I/O interfaces unavailable in the standalone mode.

The following picture shows the TWR-VF6-SOM-BSB module with the VF6-SOM plugged in configured for operation in the Freescale Tower:

Note: *The VF6-SOM Starter kit does not include a Freescale Tower or any additional TWR modules or hardware.*



4.3. Extension Interfaces

For description of the extension interfaces provided by the VF6 SOM on the module connectors refer to *Emcraft Systems VF6 SOM (System-On-Module) Hardware Architecture*.

For description of the extension interfaces provided by the TWR-VF6-SOM-BSB baseboard refer to *Emcraft Systems TWR-VF6-SOM-BSB Baseboard Hardware Architecture*.

The above mentioned documents can be downloaded from the following page on the Emcraft web site:

http://www.emcraft.com/som/vf6#hardware

5. VF6 SOM Board Linux Software Set-up

5.1. Boot Device Configuration

Since release 1.12.2 the Starter Kit supports booting from the QSPI and NAND Flash devices available on the VF6-SOM rev 3a boards. Configuration of U-boot for proper boot device is chosen with the `CONFIG_BOOT_MEDIA_NAND` and `CONFIG_BOOT_MEDIA_QSPI` build-time configuration options.

VF6-SOM boards are supplied with the NAND Flash configured as the boot device. The next section describes the process of switching to NAND boot for older VF6-SOM boards that are configured for QSPI boot.

5.2. Migration of Boot Images to NAND Flash

This section describes the process of switching to NAND Flash as the boot device and storage of the Linux image for boards that boot from the QSPI Flash.

2. The following steps must be performed:
   Copy the pre-built NAND migration U-Boot image (`u-boot-nand-migration.qspi`) to the TFTP folder of the TFTP server.
3. On the target, download the U-Boot image from the TFTP server:

```
Vybrid U-Boot > tftp u-boot-nand-migration.qspi
Using FEC0 device
TFTP from server 172.17.0.1; our IP address is 172.17.44.46
Filename 'u-boot-nand-migration.qspi'.
Load address: 0x80007fc0
```

```
Loading: ###############
done
Bytes transferred = 219256 (35878 hex)
```

4. Program the migration U-Boot image to the QSPI Flash.

```
Vybrid U-Boot > qspi probe 1;qspi erase 0 +${filesize};qspi write ${loadaddr} 0
${filesize}
Vybrid U-Boot >
```

5. Print the current settings of the U-Boot environment:

```
Vybrid U-Boot > printenv
```

6. Reset the board to activate the migration version of U-Boot.
7. As the environment has been reset to the default values, it may be needed to restore some environment variables, such as `ipaddr`, `serverip`, and so on (use the values printed out by the `printenv` command above):

```
Vybrid U-Boot > setenv ipaddr 172.17.44.46
Vybrid U-Boot > setenv serverip 172.17.0.1
Vybrid U-Boot > setenv netmask 255.255.0.0
```

8. On the target, download the NAND firmware configuration block image from the TFTP server and program the image to NAND Flash:

```
Vybrid U-Boot > tftp nand fcb.bin
Using FEC0 device
TFTP from server 172.17.0.1; our IP address is 172.17.44.46
Filename 'nand fcb.bin'.
Load address: 0x80007fc0
Loading: ##
done
Bytes transferred = 16384 (4000 hex)
Vybrid U-Boot > nand erase 0 20000

NAND erase: device 0 offset 0x0, size 0x20000
Erasing at 0x0 -- 100% complete.
OK
Vybrid U-Boot > nand write ${loadaddr} 0 4000

NAND write: device 0 offset 0x0, size 0x4000
16384 bytes written: OK
```

9. On the target, download the U-Boot image from the TFTP server and program the image to the NAND Flash:

```
Vybrid U-Boot > tftp u-boot.flash
Using FEC0 device
TFTP from server 172.17.0.1; our IP address is 172.17.44.46
Filename 'u-boot.flash'.
Load address: 0x80007fc0
Loading: ###############
done
Bytes transferred = 219256 (35878 hex)
Vybrid U-Boot > nand erase 60000 80000;nand write ${loadaddr} 60000
${filesize}

NAND erase: device 0 offset 0x60000, size 0x80000
Erasing at 0xc0000 -- 100% complete.
OK

NAND write: device 0 offset 0x60000, size 0x35878
 219256 bytes written: OK
```

10. Set the Vybrid fuses to boot from the NAND Flash:

```
Vybrid U-Boot > set_boot_media
This will program Vybrid eFUSES to boot from NAND. This cannot be undone.
Continue? (Y/N):



Vybrid U-Boot >
```

11. Reboot the board and interrupt the boot process at the U-Boot command prompt.
12. Program the Linux images to the NAND Flash as described in sections 8.1.5.
13. Reset the board and observe the Linux image boot from the NAND Flash.

5.3. U-Boot Environment

As soon as the VF6 SOM kit is powered or reset, the bootstrap sequence proceeds to boot the U-Boot firmware from external Flash printing the following output to the Cortex-A5 serial console:

```
U-Boot 2011.12-vf6-1.12.2.1 (Oct 13 2014 - 16:11:32)

CPU:   Freescale VyBrid 600 family rev1.1 at 498 MHz
Board: VF6-SOM Rev 2.a, www.emcraft.com
DDR controller is initialized
DRAM:  512 MiB
NAND:  1024 MiB
MMC:   FSL SDHC: 0
Bad block table found at page 524224, version 0x01
Bad block table found at page 524160, version 0x01
nand read bbt: Bad block at 0x00000d340000
In:    serial
Out:   serial
Err:   serial
Net:   FEC0
Hit any key to stop autoboot:  0
Vybrid U-Boot >
```

Note: *Bad blocks are a normal thing for NAND Flash. A special ECC recovery procedure is implemented to recover the bad blocks data.*

U-boot makes use of the so-called environment variables to define various aspects of the target functionality. Parameters defined by the U-boot environment variables include: target IP address, target MAC address, address in RAM where a Linux bootable images will be loaded, and others.

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

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

```
Vybrid U-Boot > printenv
bootargs=console=ttyS0,115200 panic=10
bootcmd=run flashboot
bootdelay=3
baudrate=115200
...
Vybrid U-Boot >
```

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

```
Vybrid U-Boot > setenv image psl/networking.image
Vybrid U-Boot > printenv image
image=psl/networking.image
Vybrid U-Boot >
```

  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 external Flash. Running `saveenv` makes updates to the U-Boot environment persistent across power cycles and resets.

```
Vybrid U-Boot > saveenv
Saving Environment to Flash...
...
Vybrid U-Boot >
```

5.4. Ethernet MAC Address

The MAC address of the Ethernet interface is defined by the `ethaddr` U-Boot environment variable. The value of the MAC address can be examined from the U-Boot command line monitor as follows:

```
Vybrid U-Boot > printenv ethaddr
ethaddr=C0:B1:3C:88:88:88
Vybrid U-Boot >
```

Each VF6 SOM board comes with `ethaddr` set to a MAC address uniquely allocated by Emcraft for the specific module. Given that each VF6 SOM board has a unique MAC address allocated to it, there is no need to update the `ethaddr` variable (although it is possible to do so).

The MAC address can be changed by modifying the `ethaddr` variable as follows:

```
Vybrid U-Boot > setenv ethaddr C0:B1:3C:88:88:89
```

Don't forget to store your updates to Flash using `saveenv`.

5.5. Network Configuration

You will have to update the network configuration of your kit to match settings of your local environment.

Typically, all you have to do to allow loading images over network from a TFTP server is update the U-Boot environment variables `ipaddr` (the module IP address) and `serverip` (the IP address of the TFTP server). Here is how it is done.

Update `ipaddr` and `serverip (use IP addresses that make sense for your LAN)`:

```
Vybrid U-Boot > setenv ipaddr 172.77.44.46
Vybrid U-Boot > setenv serverip 172.17.0.1
```

and then save the updated environment to external Flash using `saveenv`.

5.6. Autoboot

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

- `bootcmd` – U-Boot 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. This will let you to enter the U-Boot command line interface.

5.7. Running Pre-installed Linux Image

The VF6 SOM kit comes pre-loaded with a sample Linux/MQX image installed into external Flash. To boot this sample image, just reset the board and let U-Boot perform the autoboot sequence.

Please refer to Section 6 for a detailed description of the sample Linux image.

6. Pre-loaded Linux Image

The pre-loaded Linux image provides a demonstration of the basic shell, networking and file system management capabilities supported by Linux running on the Cortex-A5 processor code. Additionally, the Linux `init` scripts are configured to load a sample MQX image to the Cortex-M4 core during Linux boot-up. Both Linux and MQX have the MCC (Multicore Communication Interface) API enabled, making possible for the two processor cores to communicate. The demo includes a sample Linux user-space application that makes use of the MCC API to offload certain computations to the MQX application running on the Cortex-M4.

The following is a detailed description of the functionality available from the pre-loaded Linux image.

On a power-on or reset, U-Boot is loaded from the external Flash to RAM. U-Boot loads the pre-loaded Linux image from external Flash to RAM and then passes control 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 U-Boot command line interface, however assuming no operator intervention, U-Boot proceeds to boot Linux as soon as `bootdelay` expires:

```
U-Boot 2011.12-vf6-1.12.2.1 (Oct 13 2014 - 16:11:32)

CPU:   Freescale VyBrid 600 family rev1.1 at 498 MHz
Board: VF6-SOM Rev 2.a, www.emcraft.com
DDR controller is initialized
DRAM:  512 MiB
NAND:  1024 MiB
MMC:   FSL SDHC: 0
Bad block table found at page 524224, version 0x01
Bad block table found at page 524160, version 0x01
nand read bbt: Bad block at 0x00000d340000
In:    serial
Out:   serial
Err:   serial
Net:   FEC0
Hit any key to stop autoboot:  0
## Booting kernel from Legacy Image at 80007fc0 ...
   Image Name:   Linux-3.0.15-linux-vf6-1.12.2.1
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    6077668 Bytes = 5.8 MiB
   Load Address: 80008000
   Entry Point:  80008000
   Verifying Checksum ... OK
   XIP Kernel Image ... OK
OK

Starting kernel ...
```

Linux proceeds to bootstrap and mount the root file system and then execute the Linux start-up scripts. The Linux kernel is configured to mount a root file system in the external RAM using the `initramfs` file system. `initramfs` is populated with required files and utilities at the kernel build time and then linked into the bootable Linux image. `initramfs` does not have hard limits on its size and is able to grow using all otherwise unused RAM memory.

```
Linux version 3.0.15-vf6-1.12.2.1 (psl@ocean.emcraft.com) (gcc version 4.7.2 (GCC) ) #10
Mon Oct 13 18:33:32 MSK 2014
CPU: ARMv7 Processor [410fc051] revision 1 (ARMv7), cr=10c53c7d
CPU: VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine: Emcraft Vybrid SOM Board
...
Freeing init memory: 3408K
init started: BusyBox v1.17.0 (2014-10-13 18:33:34 MSK)
eth0: Freescale FEC PHY driver [Micrel KS8051] (mii bus:phy addr=1:00, irq=-1)
~ #
```

As mentioned above, the Linux start-up scripts include commands to load an MQX binary image to the Cortex-M4 core. The MQX image is loaded from a specified file in the Linux file system. By the time the Linux start-up scripts are done and the first interactive shell is started, the MQX image is fully launched on the Cortex-M4 core:

```
Loading /cmsis example.bin to 0x3f400000 ...
Loaded 65624 bytes. Booting at 0x3f404081... done
```

As mentioned above, the Linux image installed on the kit provides a demonstration of basic shell and network capabilities. In addition to that, the demo provides support for Flash partitioning and persistent data storage using a JFFS2 journalled file system in external Flash.

Here is how you can test some of these capabilities.

From a local host, test that the target responds to `ping`. The Linux start-up scripts configure the Linux Ethernet interface with an IP address defined by the U-Boot environment variable `ipaddr`, so make sure you set this variable to an address that makes sense in your local network (refer to Section 5.5).

```
[psl@ocean linux-vf6-1.12.2.1]$ 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
```

From 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-vf6-1.12.2.1]$ 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:
~ #
```

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 Dec 09 17:06:34 UTC 2013
~ #
```

Mount an NFS share exported by a development host:

```
~ # 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 VF6 SOM.

Next step is to test the non-volatile file system on external Flash. 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:

```
~ # flash_eraseall -j /dev/mtd2
Erasing 64 Kibyte @ 500000 - 100% complete.Cleanmarker written at 4f0000.
~ # mkdir /m
~ # mount -t jffs2 /dev/mtdblock2 /m
```

Copy some files to the persistent JFFS2 file system storage:

```
~ # 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
File system           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
```

As a demonstration of how to make use of the Cortex-M4 processor core, the pre-loaded demo integrates an implementation of the Cortex Microcontroller Software Interface Standard (CMSIS) DSP/FP library with MQX on the Cortex-M4 and makes the library API available to the Cortex-A5 core via the MCC (Multi-Core Communication) interface. This allows Linux applications and device drivers running on the Cortex-A5 to offload math-intensive calculations to the Cortex-M4. Offloading computations to the Cortex-M4 removes the risk of starving user applications on Cortex-A5 and at the same time ensures that math calculations are completed in time to use them with soft real-time stacks and applications (eg. media players) running under Linux.

Run the following command to see how Cortex-M4 can be used to perform floating-point vector multiplications:

```
~ # /cmsis-demo
cmsis demo: arm mult f32() of 1000 elements...
 [000]: 0.000 x 9.990 = 0.000
 [001]: 0.010 x 9.980 = 0.100
 ...
 [332]: 3.320 x 6.670 = 22.144
 [333]: 3.330 x 6.660 = 22.178
 ...
 [664]: 6.640 x 3.350 = 22.244
 [665]: 6.650 x 3.340 = 22.211
 ...
 [996]: 9.960 x 0.030 = 0.299
 [997]: 9.970 x 0.020 = 0.199
 ...
```

Reboot the target:

```
~ # reboot -f
Restarting system.


U-Boot 2011.12-vf6-1.12.2.1 (Oct 13 2014 - 16:11:32)
```

7. Software Development Environment

7.1. Distribution and Installation

7.1.1. Distribution Image

The Linux VF6 software development environment is distributed as a `linux-VF6-<release>.tar.bz2` file available for download from the Emcraft site. 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-VF6-1.12.2.1.tar.bz2
linux-vf6-1.12.2.1/
linux-vf6-1.12.2.1/linux/
linux-vf6-1.12.2.1/linux/lib/
...
[psl@ocean SF]$ ls -l linux-vf6-1.12.2.1
total 24
drwxr-xr-x  3 psl users 4096 2014-06-06 17:06 target
-rwxr-xr-x  1 psl users  315 2014-06-06 16:26 ACTIVATE.sh
drwxr-xr-x 24 psl users 4096 2014-06-06 21:32 linux
drwxr-xr-x  5 psl users 4096 2014-06-06 20:16 projects
drwxr-xr-x 31 psl users 4096 2014-06-06 19:16 u-boot
```

You do not need to be the superuser (`root`) in order to install the Linux VF6 distribution. The installation can be performed from an arbitrary unprivileged user account.

7.1.2. ELDK Cortex-A5 Toolchain and SDK

The Linux Vybrid software development environment makes use of a modified ELDK software distribution developed by DENX Software Engineering (www.denx.de). The ELDK is a Yocto-based software distribution and development environment that includes the GNU cross development tools as well as a large number of pre-built target tools and libraries ready for immediate use with Linux running on Cortex-A5. ELDK is provided for free with full source code, including all patches, extensions, programs and scripts used to build the tools.

Basically, the idea is that the Emcraft provided part of the Vybrid software development environment (refer to Section 7.1.1) includes U-Boot, the Linux kernel, the Cortex-M4 MQX Board Software Package and some other Vybrid software components, all highly optimized for the VF6 System-On-Module (SOM). The ELDK is used as a second component of the Vybrid software development environment and provides the Cortex-A5 GNU toolchain as well as a Yocto-based management of various Linux user-space components ready for the Cortex-A5 processor core.

The ELDK release used in the Vybrid software development environment is version 5.3. This release is based on Yocto v1.3 (Danny-8.0).

Note that starting release 1.12.2, the ELDK distribution has been enhanced by Emcraft to add support for Java and other components. For that purpose, Emcraft updated the DENX ELDK configuration to add the new packages and built the entire distribution from scratch. The implication of the above is that the ELDK distribution has to be downloaded from the Emcraft secure web site, rather than from the DENX web site.

The Vybrid development software requires the following ELDK 5.3 capabilities be installed on the development host:

- ARMv7a toolchain (`eldk-eglibc-i686-arm-toolchain-qte-5.3.sh`);
- Java+Qt Embedded SDK (`core-image-qte-java-sdk-generic-armv7a.tar.gz`).

The following commands must be issued in an arbitrary writeable directory to install the specified ELDK capabilities on the Linux development host:

```
$ mkdir eldk-download
$ cd eldk-download
$ mkdir -p targets/armv7a
$ wget --no-check-certificate https://emcraft.com/Emcraft-ELDK/install.sh
$ cd targets/armv7a
$ wget --no-check-certificate https://emcraft.com/Emcraft-ELDK/target.conf
$ wget --no-check-certificate https://emcraft.com/Emcraft-ELDK/eldk-eglibc-i686-arm-
toolchain-qte-5.3.sh
$ wget --no-check-certificate https://emcraft.com/Emcraft-ELDK/core-image-java-qte-sdk-
generic-armv7a.tar.gz
$ cd ../..
$ sh ./install.sh -d /opt/eldk-5.3-vybrid -s qte -r java-qte-sdk armv7a
$ sudo chmod 0777 /opt/eldk-5.3-vybrid/armv7a/rootfs-java-qte-sdk/var/lib/opkg
$ sudo ln -s /opt/eldk-5.3-vybrid /opt/eldk-5.3
```

Note that some installation commands and commands executed by `install.sh` are invoked as the superuser (using `sudo`) and therefore these commands must be executed by `root` or a user that is added to the `sudo` configuration file.

Additionally, the `eldk-switch` script must be installed on the development host as described in http://www.denx.de/wiki/view/ELDK-5/WebHome#Section_1.8.3.

7.1.3. Cortex-M4 Toolchain

As a next step in the software installation procedure, you need to download the Sourcery Codebench toolchain for the Cortex-M4 processor. The toolchain can be downloaded from the following URL:

https://sourcery.mentor.com/GNUToolchain/package10387/public/arm-none-eabi/arm-2012.03-56-arm-none-eabi.bin

The toolchain must be installed with the superuser privileges as follows:

```
[psl@pvr linux-vf6-1.12.2.1]$ sudo /bin/sh ./arm-2012.03-56-arm-none-eabi.bin -i console
...
```

During the interactive toolchain installation, the user can choose some installation options. It is recommended to install the toolchain with the following options:

- Installation directory: `/usr/local/arm-2012`
- Don't create links

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 (refer to Section 7.1.5) to provide a correct path to the installed tools (specifically, `MQX_TOOLCHAIN_DIR` must include a correct path to the directory where you have installed the Cortex-M4 toolchain).

7.1.4. Installation Tree

Having been installed onto a Linux host, the Vybrid software development environment provides the following files and directories, relative to the top of the installation directory:

- `target/` - this is a directory with target components;
- `target/busybox/` - `busybox` source and development tree;
- `target/mqx-4.0` – Cortex-M4 MQX Board Support Package (BSP);
- `u-boot/` - U-Boot source and development tree;
- `linux/` - Linux kernel source and development tree;
- `projects/` - sample projects (embedded Linux configurations);
- `ACTIVATE.sh` - `shell` script you need to perform in order to activate the Linux VF6 development environment on your host.

Additionally, the following files and directories are installed to the specified absolute paths:

- `/opt/eldk-5.3-vybrid/armv7a/sysroots/i686-eldk-linux` – Cortex-A5 GNU toolchain;

- `/opt/eldk-5.3-vybrid/armv7a/rootfs-java-qte-sdk` – ELDK packages and SDK;
- `/usr/local/arm-2012` – Cortex-M4 GNU toolchain.

7.1.5. Activation

Whenever you want to activate a Vybrid software development session, go to the top of your installation and run:

```
[psl@pvr linux-vf6-1.12.2.1]$ . 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 Vybrid software development environment:

- `INSTALL_ROOT=`*<dir>* - root directory of the installation. This variable is used by the Vybrid `make` system as well as in *<project>*`.initramfs` files to provide a reference to the installation directory;
- `ELDK_ROOTFS=`*<path>* `– path to the ELDK toolchain and SDK;`
- `MCU=`*<arch>* - processor architecture supported by the Vybrid software development environment. This is set to `VF6`;
- `MQX_TOOLCHAIN_DIR=`*<path>* – path to the Cortex-M4 GNU toolchain;
- `HOST_PYTHON_EXE_PATH=`*<path>* – path to the host `python` exectable. This is set to `/usr/bin/python`.

Besides defining these environment variables, `ACTIVATE.sh` executes the `eldk-switch` script that defines environment variables required for by the Cortex-A5 GNU toolchain.

7.1.6. Dependency on Host Components

The Vybrid software development environment has the following dependencies on Linux-host software components:

- 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.
- Project build procedure requires the `python` programming language environment be installed on the host system. By default, the `/usr/bin/python` path is used to invoke the `python` interpreter. If the `python` executable resides in another location on the host, the `HOST_PYTHON_EXE_PATH` in `ACTIVATE.sh` must be adjusted accordingly.

7.2. Projects Framework

7.2.1. Multiple Target Projects

In the Vybrid software 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 the Vybrid software development. This directory has the following structure:

- `projects/Rules.make` - build rules common for all projects;
- `projects/rfs-builder.py` – `initramfs` parser script that extends the `initramfs` syntax to allow adding ELDK packages to the target file system (refer to Section 7.2.2);
- `project1/` - project1 source files and build tree;
- `project2/` - project2 source files and build tree;

- ...

The distribution provides a single sample project called `networking`. 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/networking`) contains the following configuration files:

- *<project>*`.kernel.VF6` - kernel configuration for the project;
- *<project>*`.busybox` - `busybox` configuration for the project;
- *<project>*`.initramfs` - specification defining content of the root file system for the 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` file system containing the newly built `busybox` binary as well as other target files defined by the `initramfs` file system specification file, and finally wraps it all up into a bootable Linux image.

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 file system 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 a bootable Linux image 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.VF6;`
- `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.VF6` 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) 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 sub-directory 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 Vybrid installation directory.

7.2.2. initramfs Specification File Syntax Extensions

As described above, the `initramfs` specification files define the content of the target root file system. Refer to `linux/Documentation/filesystems/ramfs-rootfs-initramfs.txt` for a description of the `initramfs` spec file syntax.

The Vybrid software development environment extends the `initramfs` syntax with the following statements:

- `opkg` – this statement includes all files belonging to a specified ELDK package to the target file system. For example:

```
opkg libc6
```

- `rm` – this statement removes a specified file; this comes in handy if a file from a previously included package is not needed in the target file system. For example:

```
rm /usr/share/udhcpc/default.script
```

- `rmdir` – this statement removes a specified directory with all its files and subdirectories. For example:

```
rmdir /usr/share/udhcpc
```

- `localdir` – this statement adds a specified directory with all its files and subdirectories to the target filesystem. For example:

```
localdir /home/app ${INSTALL_ROOT}/projects/${SAMPLE}/app
```

7.2.3. initramfs Specification File Helper Commands

The `rfs-builder.py` utility provides several commands that can be used to automate editting of an `initramfs` specification file:

- The `pkg-list` command prints a list of all packages available from the ELDK SDK. For example:

```
$ ${INSTALL ROOT}/projects/rfs-builder.py pkg-list
acl - 2.2.51-r3
acl-dev - 2.2.51-r3
...
```

- The `pkg-add` command adds a specified package and all its dependencies to the `initramfs` specification file. For example:

```
$ ${INSTALL ROOT}/projects/rfs-builder.py pkg-add networking.initramfs acl
Adding acl to networking.initramfs
Adding libacl1 to networking.initramfs as dependency of acl
libc6 (dependency of acl) already exists, skipping
```

- The `pkg-rm` command removes a specified package from the `initramfs` specification file For example:

```
$ ${INSTALL ROOT}/projects/rfs-builder.py pkg-add networking.initramfs acl
Adding acl to networking.initramfs
Adding libacl1 to networking.initramfs as dependency of acl
libc6 (dependency of acl) already exists, skipping
```

- The `file-add` command adds a specified file to the `initramfs` specification file. For example:

```
$ ${INSTALL ROOT}/projects/rfs-builder.py file-add networking.initramfs /sample file
~/sample_file 0755 0 0
Adding file /sample_file to networking.initramfs
```

- The `symlink-add` command adds a specified symbolic link to the `initramfs` specification file. For example:

```
$ ${INSTALL_ROOT}/projects/rfs-builder.py symlink-add networking.initramfs
/sample file link /sample file 0755 0 0
Adding symlink /sample_file_link to networking.initramfs
```

- The `nod-add` command adds a specified device node to the `initramfs` specification file. For example:

```
$ ${INSTALL ROOT}/projects/rfs-builder.py nod-add networking.initramfs /dev/sample nod
0755 0 0 c 255 255
Adding nod /dev/sample_nod to networking.initramfs
```

- The `dir-add` command adds a specified directory to the `initramfs` specification file. For example:

```
$ ${INSTALL ROOT}/projects/rfs-builder.py dir-add networking.initramfs /sample dir 0755
0 0
Adding directory /sample_dir to networking.initramfs
```

- The `file-rm` command removes a specified `file/symlink/device` node from the `initramfs` specification file. For example:

```
$ ${INSTALL ROOT}/projects/rfs-builder.py file-rm networking.initramfs /dev/sample nod
Adding removal of /dev/sample_nod to networking.initramfs
```

- The `dir-rm` command removes a specified directory from the `initramfs` specification file. For example:

```
$ ${INSTALL ROOT}/projects/rfs-builder.py dir-rm networking.initramfs /sample dir
Adding removal of /sample_dir to networking.initramfs
```

- The `localdir-add` command adds an instruction to include to the initramfs image a specified directory with all its files and subdirectories to the `initramfs` specification file. For example:

```
$ ../rfs-builder.py localdir-add qtdemo.initramfs /home/app
\$\{INSTALL ROOT\}/projects/\$\{SAMPLE\}/app
Adding local directory ${INSTALL ROOT}/projects/${SAMPLE}/app with its content to
/home/app
```

Note that the `file-rm` and `dir-rm` commands do not remove the specified file or directory from the `initramfs` specification file, but instead add an appropriate instruction so that the file or directory will be removed during the root filesytem generation. This allows removal of files and directories that are added by ELDK packages rather than created directly in the `initramfs` file.

7.2.4. Networking Demo Project

Resides in `projects/networking`.

Implements the functionality of the demo Linux/MQX configuration that comes pre-loaded on the VF6-SOM kit. Please refer to Section 6 for a detailed description of the demo.

8. Software Development Workflow

8.1. Linux Development

8.1.1. Sample Linux Development Session

This sample session illustrates a recommended software development workflow using the VF6 SOM kit and the Vybrid software development environment.

We will do the following:

1. Create a new project by cloning it off of the `networking` demo project.
2. Modify the new project to automatically NFS-mount a development tree from the Linux development host.
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.

8.1.2. 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 `networking` project and create a clone called `my_networking`:

```
-bash-3.2$ pwd
/home/vlad/test/linux-vf6-1.12.2.1/projects/networking
-bash-3.2$ make clone new=my networking
New project created in /home/vlad/test/linux-vf6-1.12.2.1/projects/my_networking
-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_networking /
-bash-3.2$ make
...
  Image arch/arm/boot/uImage is ready
...
-bash-3.2$ cp my_networking.uImage /tftpboot/
```

8.1.3. Set Up Target for 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
Vybrid U-Boot >
```

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

```
Vybrid U-Boot > setenv ipaddr 172.17.44.46
Vybrid U-Boot > setenv serverip 172.17.0.1
Vybrid U-Boot > setenv image my_networking.uImage
```

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

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

4. Save the updated environment in Flash:

```
Vybrid U-Boot > saveenv
Saving Environment to qspi...
Vybrid U-Boot >
```

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:

```
Vybrid U-Boot > run netboot
Using FEC0 device
TFTP from server 172.17.0.1; our IP address is 172.17.44.46
Filename 'my networking.uImage'.
Load address: 0x80007fc0
Loading: #################################################################
         #################################################################
         #################################################################
         ##############
done
Bytes transferred = 3057956 (2ea924 hex)
## Booting kernel from Legacy Image at 80007fc0 ...
   Image Name:   Linux-3.0.15-linux-vf6-1.12.2.1
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    3057892 Bytes = 2.9 MiB
   Load Address: 80008000
   Entry Point:  80008000
   XIP Kernel Image ... OK
OK

Starting kernel ...

Linux version 3.0.15-vf6-1.12.2.1 (psl@ocean.emcraft.com) (gcc version 4.7.2 (GCC) ) #10
Mon Oct 13 17:54:32 MSK 2014
CPU: ARMv7 Processor [410fc051] revision 1 (ARMv7), cr=10c53c7d
...

Freeing init memory: 3408K
init started: BusyBox v1.17.0 (2014-06-03 16:33:34 MSK)
eth0: Freescale FEC PHY driver [Micrel KS8051] (mii_bus:phy_addr=1:00, irq=-1)
Loading /cmsis example.bin to 0x3f400000 ...
Loaded 65624 bytes. Booting at 0x3f404081... done
```

8.1.4. 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-vf6-1.12.2.1/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_networking.uImage /tftpboot/
...
```

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
Vybrid U-Boot > run netboot
...
init started: BusyBox v1.17.0 (2014-06-06 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-vf6-1.12.2.1/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_networking
drwxr-xr-x    4 19270    19270         4096  Mar 25  2011 networking
-rw-r--r--    1 19270    19270         3581  Mar 25  2011 Rules.make
-rwxr-xr-x    1 19270    19270         14485 Mar 25  2011 rfs-builder.py
~ #
```

8.1.5. 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:

```
Vybrid U-Boot > print image
image=my networking.uImage
Vybrid U-Boot > run update
Using FEC0 device
TFTP from server 172.17.0.1; our IP address is 172.17.44.46
Filename 'psl/vy/networking.uImage'.
Load address: 0x80007fc0
Loading: #################################################################
          #################################################################
          #################################################################
          ##############
done
Bytes transferred = 3059448 (2eaef8 hex)
Saving Environment to qspi...
Vybrid U-Boot >
```

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

```
Vybrid U-Boot > reset
resetting ...
...
Hit any key to stop autoboot:  0
## Booting kernel from Legacy Image at 80007fc0 ...
...
init started: BusyBox v1.17.0 (2014-06-06 19:58:44 MSK)
~ # ls -lt /mnt
drwxr-xr-x    4 19270    19270         4096 Mar 25  2011 my networking
...
~ #
```

8.1.6. 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-vf6-1.12.2.1/projects/my_networking/app
```

2. The `networking` 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_networking/app # cd /mnt/my_networking/app/
```

5. Install the updated device driver to the kernel:

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

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

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

8. Unload the device driver:

```
/mnt/my_networking/app # rmmod sample
```

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

8.1.7. 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-vf6-1.12.2.1/projects/my_networking/app
```

2. The `networking` project you have cloned your new project off already provides a user-space application implemented in `app.c`. 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_networking/app # insmod sample.ko
/mnt/my_networking/app # ./app
./app: THIS IS CONTENT OF /dev/sample:
This is the simplest loadable kernel module
```

5. Unload the device driver:

```
/mnt/my_networking/app # rmmod sample
```

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

8.2. MQX Development

8.2.1. Cortex-M4 GNU Toolchain

The Cortex-M4 MQX BSP and application code is developed using the Linux-based GNU toolchain. This allows using a single Linux development host for development of both the Cortex-A5 and Cortex-M4 processor cores.

Refer to Section 7.1.3 for detailed information on how the Cortex-M4 GNU toolchain is installed to the development host.

8.2.2. MQX BSP Source Tree

In the Vybrid software development environment, the MQX BSP source tree resides in the `target/mqx-4.0/` sub-directory, relative to the installation. The same sub-directory provides the source files of the Cortex-M4 MCC (Mutli-Core Communication) component.

The MQX BSP configuration is defined by the following file: `target/mqx-4.0/config/emcraft_vf6som_m4/user_config.h`. Edit that file in case you need to change the MQX BSP configuration.

8.2.3. MQX Build

The `networking` demo project (refer to Section 7.2.4) illustrates how the MQX BSP and a sample MQX application are built on the Linux development host. Let's take a look at how this is done.

The `projects/networking/mcc` sub-directory contains all Cortex-M4 application code required in the demo. Take a look at the `Makefile` in that directory to see how the project build goes to the MQX source tree in `target/mqx-4.0/` to build the MQX BSP and the MQX MCC libraries.

The Cortex-M4 application code used by the demo resides in the `cmsis/` and `libcmsis/` sub-directories, relative to `projects/networking/mcc`. The above `Makefile` goes to these subdirectories to build the application components for Cortex-M4.

As the final step in the MQX build sequence, the `Makefile` links all Cortex-M4 components (MQX BSP library, MQX MCC library, demo application) into a Cortex-M4 executable binary (`cmsis_example.bin`). The demo root file system specification file (`projects/networking/networking.initramfs`) copies the binary to the Linux target file system.

8.2.4. Run-Time Load of MQX

On the target, an MQX executable binary is loaded and started on the Cortex-M4 processor core using the Linux `mqxboot` utility. For example, the following command loads onto the Cortex-M4 the `cmsis_example.bin` MQX binary:

```
~ # /mqxboot /cmsis_example.bin 0x3f000000 0x3f0043ad
Loading /cmsis example emcraft vf6som m4.bin to 0x3f000000 ...
Loaded 39772 bytes. Booting at 0x3f000485... done
```

Alternatively, the MQX image can be loaded and started from U-boot. For this, the MQX binary must be encapsulated into a `uImage` image (in the `networking` sample project this is automatically done during build process in `mcc/Makefile`). The `uImage` must then be loaded to RAM and booted using the `boot_cm4` U-boot command, e.g.:

```
Vybrid U-Boot > tftpboot mqx.uImage
Using FEC0 device
TFTP from server 172.17.0.1; our IP address is 172.17.44.46
Filename 'mqx.uImage'.
Load address: 0x80007fc0
Loading: #####
done
Bytes transferred = 66120 (10248 hex)
Vybrid U-Boot > boot_cm4 ${loadaddr}
## Booting MQX from Legacy Image at 80007fc0 ...
   Image Name:   MQX
   Image Type:   ARM U-Boot Firmware (uncompressed)
   Data Size:    66056 Bytes = 64.5 KiB
   Load Address: 3f400000
   Entry Point:  3f4043ad
Vybrid U-Boot >
```

8.3. U-Boot Development

8.3.1. U-Boot Build

The Vybrid software 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 on your Linux development host (refer to Section 7):

1. Go to the software installation and activate the cross development environment (unless you have activated it already):

```
[psl@ocean linux-vf6-1.12.2.1]$ . ACTIVATE.sh
Setup for armv7a (using ELDK 5.3-vybrid)
[psl@ocean linux-vf6-1.12.2.1]$
```

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

```
[psl@ocean linux-cortexm-1.12.2.1]$ cd u-boot/
[psl@ocean u-boot]$
```

3. Configure U-Boot for the VF6 SOM:

```
[psl@ocean u-boot]$ make vybrid som config
Generating include/autoconf.mk
Generating include/autoconf.mk.dep
```

```
Configuring for vybrid som - vybrid som 2a, Options:
SYS_TEXT_BASE=0x3F000800,IMX_CONFIG=board/emcraft/vybrid_som/vybridimage.cfg
[psl@ocean u-boot]
```

4. Build the U-Boot image and copy it to the TFTP server directory:

```
[psl@ocean u-boot]$ LDFLAGS="" make -j9 -s u-boot.flash



Image Type:   Freescale IMX Boot Image
Image Ver:    2 (i.MX53/6 compatible)
Data Size:    161128 Bytes = 157.35 kB = 0.15 MB
Load Address: 3f000420
Entry Point:  3f000800
[psl@ocean u-boot]$ cp u-boot.flash /tftpboot/
```

8.3.2. U-Boot Self-Upgrade

To upgrade U-Boot on the VF6-SOM, do the following on the target:

1. Download the U-Boot image from the TFTP server:

```
Vybrid U-Boot > tftp u-boot.flash
Using FEC0 device
TFTP from server 172.17.0.1; our IP address is 172.17.44.46
Filename 'u-boot.flash'.
Load address: 0x80007fc0
Loading: ###########
done
Bytes transferred = 160544 (27320 hex)
Vybrid U-Boot >
```

2. Program the new U-Boot image to the external Flash of the VF6 SOM.
   a) If the NAND Flash is used as the boot device, the following command must be used:

```
Vybrid U-Boot > nand erase 60000 80000 && nand write ${loadaddr} 60000 ${filesize}
Vybrid U-Boot >
```

   b) If the QSPI Flash is used as the boot device, the following command must be used:

```
Vybrid U-Boot > qspi erase 0 40000 && qspi write ${loadaddr} 0 ${filesize}
Vybrid U-Boot >
```

3. Reset the target and make sure that the new U-Boot comes up on the VF6-SOM:

```
Vybrid U-Boot > reset
resetting ...

CPU:   Freescale VyBrid 600 family rev1.1 at 498 MHz
Board: VF6-SOM Rev 2.a, www.emcraft.com
DDR controller is initialized
DRAM:  512 MiB
NAND:  1024 MiB
MMC:   FSL SDHC: 0
Bad block table found at page 524224, version 0x01
Bad block table found at page 524160, version 0x01
nand read bbt: Bad block at 0x00000d340000
In:    serial
Out:   serial
Err:   serial
Net:   FEC0
Vybrid U-Boot >
```

Note: *Be extra-careful when performing the upgrade sequence specified above. In case you program an incorrect U-Boot image to the external Flash, 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.*

9. Support

We appreciate your review of our product and welcome any and all feedback. Comments can be sent directly by email to:

a2f-linux-support@emcraft.com

The following level of support is included with your purchase of this product:

- Email support for installation, configuration and basic use scenarios of the product during 3 months since the product purchase;
- Free upgrade to new releases of the downloadable materials included in the product during 3 months since the product purchase.

If you require support beyond of what is described above, we will be happy to provide it using resources of our contract development team. Please contact us for details.