Linux on the ARM Integrator Compact Platform (CP)

Alan Casey

August 9, 2006

Contents

0.1 Introduction

This document will give an introduction of why Linux is being used on the ARM Integrator/CP, how to configure and cross-compile the Linux Kernel, how to build a Linux Filesystem, how to build a suitable Linux Bootloader and how to start/setup Linux on the ARM Integrator/CP.

The Centre for Digital Video Processing (CDVP)'s Hardware Group required a standalone and network enabled demonstration of their Low Power Hardware Accelerators on the ARM Integrator/CP (where the Hardware Accelerator would be burnt onto an FPGA on the ARM Integrator platform). The existing method of running semihosted[1] ARM Developer Suite (ADS) software applications on the ARM Integrator/CP (which would interface with the Hardware Accelerator on the FPGA) via a host PC running ARM Debugger Software is unsuitable as program execution and demonstration output (both in textual or visual formats) are very slow. Hence it was decided to run an actual Operating System (OS) on the ARM Integrator/CP to remove the semihosting bottleneck, enabling much faster program execution and demonstration output.

An OS was also needed which supported the ARM920T Processor and also the peripherals on the ARM Integrator/CP such as the the keyboard and mouse, the Ethernet Chip/interface, and the VGA or LCD display interface (for textual or visual demonstration output). For both of these purposes Linux was chosen, as the Linux 2.6.x+ kernel officially supports the ARM Integrator/CP as well as it's peripherals. In addition, the Linux kernel also contains a TCP/IP stack which enables network connectivity across the Ethernet interface on the ARM Integrator/CP. Other general reasons for using Linux include:

- Linux kernel source code and build tools are freeware,
- Quality and Reliability of Code,
- Hardware Support - Linux supports many different types of hardware platforms and devices,
- Broad communication protocol and software standards support,
- Open source community - mailing lists for support,[2]

[1]mechanism whereby any software program input/output is sent or received across a JTAG connection between a PC and the ARM Integrator/CP

[2]http://www.arm.linux.org.uk/mailinglists - main ARM Linux website

- Increasing usage on mobile phones and PDAs[3]

The terms Linux and Linux kernel are used interchangeably throughout this document and refer to the actual Linux kernel as maintained by Linus Torvalds, as opposed to a Linux distribution provided by an organisation such as Red Hat, Montavista etc. (where in this case the Linux distribution includes a preconfigured and precompiled version of the Linux kernel plus applications that run on the kernel such as a GUI envirnoment like KDE for example). The same Linux kernel will be used on the ARM Integrator/CP but configured for the ARM Integrator/CP and it's perihperals. The main purpose of the kernel is to manage hardware in a coherent manner while providing familiar high-level abstractions to user-level software/applications. It drives devices, manages I/O accesses, controls process scheduling, enforces memory sharing, handles the distribution of signals, manages filesystems and network protocols, and tends to other administrative tasks [1]. Note that the term Embedded Linux simply refers to a version of the Linux kernel (i.e. with a particular configuration) that is used in an embedded system/platform (such as for example on the Sharp Zaurus PDA or on the ARM Integrator/CP).

A Cross-Compiler will be used on a Linux workstation (the host) to cross-compile code for execution on the ARM Integrator/CP (the target). The Cross-Compiler will be used to cross-compile the Linux kernel, the Linux filesystem/utilities and the Linux bootloader on the host, after which each cross-compiled image will be burned onto the ARM Integrator/CP's baseboard FLASH. On power-up of the ARM Integrator/CP, the bootloader will do some low level hardware initialization on the ARM Integrator/CP and launch the Linux kernel. Once launched, the Linux kernel will then mount the Linux filesystem, after which the user can login and launch a GUI environment such as X Windows. Each of the next sections details each of these steps.

[3]http://www.linuxdevices.com/articles/AT3908389811.html

0.2 Workspace Setup

The Cross-Compilation/building of the Linux kernel, filesystem and Bootloader will be done on a Linux machine. For these purposes you need to download a Linux based ARM Cross-Compiler from:
//poppintree.eeng.dcu.ie/volume2/vmpg/HARDWARE_VMPG/Alan/
ARM-Cross-Compiler/Linux/arm-cross-compiler.tar.gz.

The Linux based ARM Cross-Compiler allows you to compile software code on a standard Linux workstation for execution on the ARM Integrator/CP platform (with an ARM920T Processor). A Linux based ARM Cross-Compiler was originally developed using similar procedures to that as described at [2] but it eventually became apparent (approximately one month later) that ARM had already developed a prebuilt Linux based ARM Cross-Compiler at [3] and this is the same version that you can download from poppintree.eeng.dcu.ie as mentioned previously.[4].

Also download: //poppintree.eeng.dcu.ie/volume2/vmpg/
HARDWARE_VMPG/Alan/embedded-linux-project.tar.gz to your home directory or similar - this contains the following directories:

bootldr:	The bootloader for Linux on the ARM Integrator/CP
images:	The binary images of the bootloader, Linux kernel and filesystem
kernel:	The Linux kernel location and directory for cross-compiling
project:	Any demo source code for your project
rootfs:	The root filesystem location and directory for compiling
sysapps:	Any system applications required by your demo (e.g. YUV player)
tmp:	Directory to store temporary files
tools:	Miscellaneous build utilities

It is suggested that you add something similar to the following to your .bashrc or .bash_profile in your Linux home directory and download and unzip/untar the Linux based ARM Cross-Compiler to the ${PRJROOT} directory:

[4]Since the writing of this document, ARM (via a company called CodeSourcery) have updated their Cross-Compiler to only support ARM Processors with the ARM5vT architecture or above, so use the above version on poppintree (ARM920T is ARMv4T architecture)

```
export PRJROOT=/home/username/embedded-linux-project
export TARGET=arm-linux
export PREFIX=${PRJROOT}/arm-cross-compiler
export TARGET_PREFIX=${PREFIX}/${TARGET}
export PATH=${PREFIX}/bin:${PATH}
PATH=./:${PATH}
```

These environment variables are needed to ease the build process. The TARGET variable defines the type of target for the cross-compiler. The PREFIX variable provides a pointer to the directory where a Linux based ARM Cross-Compiler is installed. The TARGET_PREFIX variable points to a directory that stores the installation of target-dependent header files and libraries. Note that if your username contains spaces then the build process is more than likely to fail.

0.3 Configuring and Cross-Compiling the Linux Kernel

Download from http://www.kernel.org/pub/linux/kernel/v2.6/ the latest Linux kernel version known to be working without any problems on the ARM Integrator/CP, which is linux-2.6.9.tar.gz at the time of writing, to your ${PRJROOT}/kernel directory (later kernel versions have problems with driving the VGA/LCD displays properly). The last two numbers of a linux kernel release designates the version number, odd version numbers indicate development releases while even version numbers indicate stable releases. Then perform the following:

```
$ cd ${PRJROOT}/kernel
$ tar xvzf linux-2.6.9.tar.gz
$ cp linux_kernel_config.txt linux-2.6.9/.config
```

The kernel configuration file specifies information about the desired kernel configuration, such as for example that the kernel will be executing on the ARM Integrator/CP platform, that the processor type is ARM920T, information about supported peripherals on the platform and miscellaneous kernel configuration parameters etc. Some important configuration parameters in this file are:

Some Linux Kernel Config Parameters	
CONFIG_ARCH_INTEGRATOR_CP=y	configure the kernel for the ARM Integrator Compact Platform
CONFIG_CPU_32=y	configure the kernel for 32-bit CPU
CONFIG_CPU_ARM920T=y	configure the kernel for the ARM920T
CONFIG_ARM_AMBA=y	configure the kernel to support the ARM AMBA Bus
CONFIG_BLK_DEV_RAM=y	configure the kernel to support a filesystem in RAM
CONFIG_BLK_DEV_RAM_SIZE=32768	configure the kernel to support a filesystem of max size 32768KB
CONFIG_BLK_DEV_INITRD=y	configure the kernel to support initial RAM Disks for filesystems
CONFIG_NET=y	configure the kernel for network support
CONFIG_NET_ETHERNET=y	configure the kernel for Ethernet support
CONFIG_SMC91X=y	configure the kernel to support the Ethernet chip on the Integrator/CP
CONFIG_SERIO_AMBAKMI=y	configure the kernel to support keyboard and mouse peripherals on Integrator/CP
CONFIG_EXT2_FS=y	configure the kernel to support EXT2 type filesystems
CONFIG_FB_ARMCLCD=y	configure the kernel to support the ARM CLCD/VGA Video Driver
CONFIG_CPU_FREQ_INTEGRATOR=y	configure the kernel to allow user to change CPU Freq on the fly

The following kernel configuration parameter specifies kernel parameters on bootup:

```
CONFIG_CMDLINE="root=/dev/ram0 rw mem=128M
video=vc:1-2clcdfb console=ttyAMA0,38400 ramdisk_size=32768
initrd=0x00400000,5836340 ether=04:79:F7:00:00:02"
```

This specifies that the root filesystem (which is writable/readable) is in RAM, there is 128M of memory (SDRAM), that the framebuffer/video driver is the ARM CLCD/VGA driver, that the serial port/console is of type ttyAMA0 (i.e the Integrator/CP's UART peripheral), that the maximum size of the filesystem (which is a ramdisk) cannot exceed 32768KB, that the actual compressed filesystem binary image will be located at address 0x00400000 in memory and that it's compressed size is 5836340 bytes (note that when uncompressed it will be less than the maximum filesystem size so that the user can add/create his own files on the filesystem etc.), and that the Ethernet MAC address of the Ethernet Chip is 04:79:F7:00:00:02. Note that the *CONFIG_CMDLINE* kernel parameter can be overridden when booting the kernel using the Bootloader (as we will see later), which will save having to re-cross-compile the Linux kernel everytime the compressed filesystem binary image's size changes. As will be seen later, the Bootloader when launched, will copy the filesystem image which is initially stored in FLASH to address 0x00400000 in SDRAM - when the kernel is launched one of it's last tasks will be to uncompress the filesystem at address 0x00400000 in SDRAM and mount it.

Using this configuration file (.config) configure the kernel using the following command (the *CROSS-COMPILE* variable specifies to use the Cross-Compiler that we downloaded earlier - this will force the configure script to prepend arm-linux- to any gcc references):

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- menuconfig
```

You will be presented with a configuration utility displaying the kernel configuration items that were picked up in the .config file. You can browse through each item using the arrow keys (but do not change anything). Use the Escape key to quit a specific menu item and scroll down to the Exit item to quit the configuration menu - if prompted by the utility to save the kernel's configuration click Yes (this will overwrite the existing .config file which is Ok). At this stage a few Linux kernel header files are created, some symbolic links and a Makefile. Now Cross-Compile the Linux Kernel:

```
$ make ARCH=arm CROSS_COMPILE=arm-linux- zImage
```

The *zImage* target instructs the Makefile to build a kernel image that is compressed using the gzip algorithm. It will also create a *vmlinux* target which is an uncompressed kernel image version. The above Cross-Compile process will take approximately 10 minutes. Next use the *arm-linux-objcopy* utility to strip the *vmlinux* target of all notes, comments and symbols which we do not need and to output the stripped binary image:

```
$ arm-linux-objcopy -O binary -R.note -R.comment -S vmlinux
linux-2.6.9.bin
```

Since the FLASH on the ARM Integrator/CP's baseboard is not structured like a filesystem and does not contain any sort of file headers, the Linux Kernel cross-compiled binary image downloaded to the FLASH must carry headers for the Linux Bootloader (U-Boot, which will be described later) to recognise it's content and understand how to load it. The *mkimage* utility is used for this purpose, it adds the information to the binary image that the Linux Bootloader needs, while attaching a checksum for verification purposes. Use this utility as follows (where -*n* specifies the image name, -*A* specifies the architecture, -*O* specifies the operating system, -*T* specifies the image type, -*C* specifies the compression type, -*a* specifies the address in memory to load the Linux Kernel image, -*e* specifies entry/start point of the Linux Kernel image, -*d* specifies the input binary file and linux-2.6.9.img is the output binary image file):

```
$ ${PRJROOT}/tools/mkimage -n 'ARM Linux 2.6.9' -A arm -O
linux -T kernel -C none -a 0x7fc0 -e 0x8000 -d linux-2.6.9.bin
linux-2.6.9.img
```

Finally copy this output binary image file to the appropriate storage directory:

```
$ cp linux-2.6.9.img ${PRJROOT}/images
```

0.4 Building the Linux Filesystem and Utilities

The top-level directories in a typical root filesystem each have a specific use and purpose:

Linux root filesystem structure	
bin	essential user command binaries
boot	static files used by a bootloader (optional)
dev	device files for interfacing with disks and hardware
etc	system configuration files, including startup files
home	user home directories, including setup entries for services such as FTP
lib	essential libraries, such as the C Libary glibc
mnt	mount point for temporarily mounted filesystems/disks
opt	add-on software packages (optional)
proc	virtual filesystem for kernel and process information
root	root user's home directory
sbin	essential system administration binaries
tmp	temporary storage for programs and people to use
usr	secondary hierarchy containing most applications and documents useful to most users, including the X server
var	variable data stored by daemons and utilities

It is possible to build such a filesystem from scratch following the procedures as detailed in Chapter 6 of [1]. However, this proved to be a lengthy and error-prone process and it eventually became apparent that a Linux filesystem generation package could be downloaded from ARM's website [4] and this was the one that was used in generating the Linux filesystem (but modified to add additional filesystem utilities) for use on the ARM Integrator/CP.[5]

Two possible filesystem image types can be generated, JFFS2 (filesystem format for FLASH) and Ext2 over RAM disk (filesystem format for RAM). It was decided to use Ext2 over RAM disk due to the limited memory of the FLASH (16MB FLASH on the ARM Integrator/CP's baseboard) and

[5]Since the writing of this document ARM have completely updated/rewritten their Linux filesystem generatorion package to ARM_Embedded_Linux-2.0.tar.gz - however the first version (1.1b) appears to be just as good as the latest version and contains all the script(s) functionality needed to build the filesystem

also due to write problems when trying to create new directories and files on the JFFS2 filesystem (JFFS2 documentation states that it is possible to do writes on a JFFS2 filesystem but there seem to be problems with this on the Integrator/CP).

The disadvantage of Ext2 over RAM Disk is that when powered down, any new files and directories you create on the filesystem will be lost. Apparently large Ext2 RAM Disks can also fail to work with the Linux kernel (im not exactly sure why but it has been stated on some mailing lists [5]) so i have limited the maximum size of the filesystem so far to 65536KB (64 MB) which has proved to be sufficient so far for demonstation purposes - however you can try increasing this if necessary.

Extract the modified Linux Filesystem generation package from ARM:

```
$ cd ${PRJROOT}/rootfs
$ tar xvzf ARM_Embedded_Linux-1.1b-mod.tar.gz
```

The *scripts/* directory contains Perl and shell scripts to build a Linux filesystem, the generated filesystem will be created (including all directories and files) in the *build/* directory and will have the structure similar to that as illustrated in the table above. The Perl and shell scripts will operate on the Debian packages (.deb) in the *packages/* directory to create the filesystem. Debian [6] is an organistion that creates precompiled and archived Linux filesystem components (in many cases with stripped down functionality for embedded systems) for different architectures (such as ARM, i386, PPC etc.) and also provides distributions of Linux. Example miscellaneous Debian packages:

Busybox	:	provides stripped down versions of most Linux commands
libc-2.3.2	:	version 2.3.2 of the GNU C Library glibc
ftp, ncftp, telnet	:	network file transfer utilities
XServer	:	standard GUI Environment utility (but reduced size)
armelinux-base	:	startup configuration scripts for /etc directory
zlib1g	:	compression library
login,passwd	:	login and password utilities

Blackbox	: X Windows Manager
nedit	: text editor

The *.config* file in the ${PRJROOT}/rootfs/ARM_Embedded_Linux-1.1b-mod/ directory contains configuration parameters for the base utilities of the filesystem such as *CONFIG_OS_X=y* (support X windows environment), *CONFIG_OS_NET_ EXTRA=y* (support fpt, telnet), *CONFIG_OS_HOSTNAME=localhost* (sets the hostname) for example. The main perl script is located at: ${PRJROOT}/rootfs/ARM_Embedded_Linux-1.1b-mod/scripts/ARMLINUX_CONFIG.pm and fortunately you do not need much Perl experience to hack this script to make any necessary changes etc. (as in my case). The main packages array in this file is *ARM_LINUX_COMPONENTS* which lists the Debian packages which need to be extracted to create the filesystem. You can add further utilities to the filesystem by adding further *package* elements to *ARM_LINUX_COMPONENTS* such as for example:

```
{
    package => "xloadimage",
    source  => "libjpeg62,libpng3,libtiff3g,xloadimage",
},
```

In the above case, *xloadimage* is the package/utility name, while *libjpeg62*, *libpng3*, *libtiff3g* and *xloadimage* are the source Debian packages that are needed to add *xloadimage* (image viewer) support on the filesystem. A search on the Debian website [6] for "xloadimage" will give you a list of the source packages that you need to download to build *xloadimage* (i.e. the above). There are a number of Perl functions in the *ARMLINUX_CONFIG.pm* file that operate on the *.config* file and on the *ARM_LINUX_COMPONENTS* array to extract the precompiled Debian archive packages to the *build/* directory and a description of the main ones are listed below:

Main ARMLINUX_CONFIG.pm Perl Script Functions	
arm_config_load	Reads the .config file and updates the ARM_LINUX_CONFIG section of ARMLINUX_CONFIG.pm. This sets the default filesystem utilities to install.
find_package	Checks a specific directory (i.e. *packages/*) for the specified Debian package to see if it is there and if so returns it's name
install_package	Uses the Linux *ar* command to extract the directories and files in the specified Debian package to the *build/* directory
arm_generate_system	Iteratively searches through the *ARM_LINUX_COMPONENTS* array and uses the Perl install_package function to install each Debian package name found. It first installs those packages with it's "config" variable set to "y" followed by any other packages without any "config" variable such as *xloadimage* above for example.

The Makefile in ${PRJROOT}/rootfs/ARM_Embedded_Linux-1.1b-mod calls the ${PRJROOT}/rootfs/ARM_Embedded_Linux-1.1b-mod/scripts/ generate.pl Perl script (which in turn calls the above Perl functions defined in ARMLINUX_CONFIG.pm) to generate the filesystem. After generating the filesystem (i.e. *build/*), the Makefile calls the ${PRJROOT}/rootfs/ARM _Embedded_Linux-1.1b-mod/scripts/build_add.sh shell script which adds some extra pre-cross-compiled utilites and files in the *build_add* directory to the *build/* directory (e.g. YUV Player, skin detection demo, network config files etc.). The Makefile subsequently calls the ${PRJROOT}/rootfs/ARM _Embedded_Linux-1.1b-mod/scripts/make_ramdisk.sh shell script which will convert the *build/* directory (i.e. the filesystem) into a single compressed Ext2 RAM Disk filesystem image that can be programmed into the FLASH on the ARM Integrator/CP's baseboard (as will be seen later). The main lines in the *make_ramdisk.sh* shell script are:

```
mkdir disk
mkdir disk/ramdisk
```

```
dd if=/dev/zero of=ramdisk.img bs=1k count=65536

/sbin/mke2fs -F -v -m0 ramdisk.img
mount -o loop ramdisk.img disk/ramdisk
cp -av build/* disk/ramdisk
umount disk/ramdisk

gzip -9 ramdisk.img
mv ramdisk.img.gz ramdisk.bin
```

The *dd* command creates a 65536KB filesystem image (*ramdisk.img*) and initializes it using */dev/zero*. The */sbin/mke2fs* command creates a Ext2 type filesystem on the initialized filesystem image. The *-F* option is used to force *mke2fs* to run on a file as opposed to a block device. The *-v* option specifies that the command should be verbose and the *-m0* option specifies that no blocks should be reserved for the super user on the filesystem. The filesystem image is then mounted to the *disk/ramdisk* location (i.e. the actual RAM Disk) where the contents of the *build/* directory (i.e. the actual filesystem that was generated earlier) are copied to the RAM Disk (*disk/ramdisk*). The RAM disk in then unmounted and compressed to create the final RAM Disk image (*ramdisk.bin*).

To generate the filesystem correctly you have to be logged in as root because the build process creates devices (in /dev) and softlinks:

$ cd ${PRJROOT}/rootfs/ARM_Embedded_Linux-1.1b-mod/
$ su root
$ make

The output of the filesystem generation process is *ramdisk.bin* (i.e. Ext2 Filesystem image) which will be downloaded to FLASH on the Integrator/CP's baseboard as will be seen later. Copy this to the *images/* directory for now:

$ cp ramdisk.bin ${PRJROOT}/images

0.5 Cross-Compiling the Linux Bootloader

The bootloader that will be cross-compiled, for eventual booting of the Linux kernel, is officially called *DAS U-Boot* [7]. It is one of the richest, most flexible and most actively developed open source bootloader available and supports hundreds of different platforms (100+ PowerPC based platforms, 12+ ARM based platforms). It is currently maintained by Wolfgang Denk of Denx Software Engineering [8] and is contributed to by a wide range of developers (with the disadvantage that many patches are submitted each week, there is currently a backlog of 100+ patches to be applied to the latest CVS tree). U-Boot development is stable for the ARM Integrator/CP platform with the latest patch supplied to [7] by ARM (which is not integrated into the latest CVS tree as at the time of writing). Extract the U-Boot source (this version has the necessary patch from ARM already applied to it):

```
$ cd ${PRJROOT}/bootldr
$ tar xvzf u-boot-1.1.2-mod.tar.gz
```

When launched (usually after power-up of the system), U-Boot will setup/ start the ARM processor via the code in *u-boot-1.1.2-mod/cpu/arm920t/*, initialize memory and map memory according to the ARM Integrator/CP memory address space using the code in *u-boot-1.1.2-mod/board/integratorcp/*, copy Linux Kernel and filesystem/RAMDisk images from FLASH to RAM and boot the Linux Kernel.

The *README* file included with the package discusses the features of U-Boot, the source code layout, the available build options, U-Boot's command set and the typical environment variables used in U-Boot. The main configuration file of interest is *include/configs/integratorcp.h* which contains configuration variables for booting of the Linux Kernel on the ARM Integrator/CP. There are two main classes of configuration variables: *CONFIG_* which are selectable by the user and *CFG_* which are dependent on the hardware. The table below describes some miscellaneous configuration variables:

Example integratorcp.h configuration variables	
CONFIG_DRIVER_SMC91111	Support for SMCS's Ethernet LAN91C111 Chip
CONFIG_SMC91111_BASE	Physical Base Address of the Ethernet Chip
CFG_PL011_SERIAL	Support for Amba PrimeCell PL011 UARTs
CONFIG_PL011_CLOCK	Used to specify the clock speed of the UARTs
CONFIG_NR_DRAM_BANKS	Number of banks of SDRAM
CFG_FLASH_BASE	Physical start address of FLASH
CFG_MAX_FLASH_SECT	Max. number of FLASH sectors

The two most important U-Boot configuration variables are *CONFIG_BOOTARGS* which is used to pass parameters to the kernel (when U-Boot launches the kernel) and *CONFIG_BOOTCOMMAND* which defines the last action to take before booting the kernel. An example of the *CONFIG_BOOTARGS* variable is as follows:

#define CONFIG_BOOTARGS "root=/dev/ram0 rw mem=128M video=vc:1-2clcdfb console=ttyAMA0,38400 ramdisk_size=65536 initrd=0x00400000,7959088 ether=04:79:F7:00:00:02"

This will tell the kernel that the root filesystem will be in RAM and that it is readable/writable (root=/dev/ram0 rw), that there is 128MB of RAM available (mem=128M), that the video driver to use is the AMBA CLCD FrameBuffer driver (video=vc:1-2clcdfb - this also supports VGA displays), that the serial port console/is of type AMBA (UART PL011) and that the baud rate is 38400bits/sec (console=ttyAMA0,38400 - this is useful to setup so as to allow viewing of u-boot and kernel messages during booting via the serial port and a communication program such as HyperTerminal), that the maximum size of the root filesystem will be 65536KB (ramdisk_size = 65536), that the compressed filesystem image (ramdisk.bin that was generated earlier) will be located at address 0x00400000 in RAM and that it's compressed size is 7959088 bytes (initrd=0x00400000,7959088 - the kernel will, after launching, subsequently uncompress the filesystem image and mount it) and that the Ethernet Hardware/MAC Address of the Ethernet Chip is 04:79:F7:00:00:02 (i.e. the MAC address of the Ethernet Chip on the ARM

Integrator/CP). An example of the *CONFIG_BOOTCOMMAND* variable is as follows:

#define CONFIG_BOOTCOMMAND "cp 0x24080000 0x7fc0 82634; cp 0x24300000 0x00400000 1E5C8C; bootm"

This will tell U-Boot to (i) copy the Linux Kernel uncompressed image at address 0x24080000 (i.e. the image at address 0x24080000 in FLASH on the ARM Integrator/CP's baseboard after FLASH programming - see the next section) to address 0x7fc0 in RAM and that the uncompressed image size is 82634(in Hexadecimal) words (ii) copy the RAMDisk image (ramdisk.bin) at address 0x24300000 (which will again be in FLASH on the Integrator/CP's baseboard after programming) to address 0x00400000 in RAM and that the compressed image size is 1E5C8C (Hex) words (which is equal to 7959088 bytes as defined previously by *CONFIG_BOOTARGS*) - address 0x0040000 is the location in RAM that the Linux Kernel will expect to find the RAMDisk image as defined previously using *CONFIG_BOOTARGS* (iii) boot the Linux Kernel from memory, which will automatically look for a Linux Kernel image at 0x7fc0 in RAM and boot it using the kernel parameters that are passed via *CONFIG_BOOTARGS* as described previously.

Note 1: if the Linux Kernel or RAMDisk image size is not a multiple of words, e.g. 7959090 bytes, then you must use the U-Boot *cp.b* command in the above *CONFIG_BOOTCOMMAND* definition (i.e. copy byte by byte, by default U-Boot *cp* copies word by word), e.g. **cp.b 0x24300000 0x00400000 797232** (797232 Hex = 7959090 Dec).

Note 2: the *CONFIG_BOOTARGS* U-Boot configuration variable parameters overrides the *CONFIG_CMDLINE* Linux Kernel configuration parameters (defined in ${PRJROOT}/kernel/linux-2.6.9/.config) that are used when cross-compiling the Linux Kernel - i.e. the U-Boot Linux Kernel boot parameters override the default ones that are defined when cross-compiling the Linux Kernel.

Note 3: everytime the Linux Kernel uncompressed image size changes or the RAMDisk compressed image size changes you must edit the *CONFIG_BOOTARGS* and *CONFIG_BOOTCOMMAND* configuration variables appropriately and re-cross-compile U-Boot. One of the advantages of *CONFIG_BOOTARGS* is that if only the RAMDisk image size changes it only suffices to modify *CONFIG_BOOTARGS* and *CONFIG_BOOTCOMMAND* and re-cross-compile U-Boot (i.e. you do not need to re-cross-compile the

Linux Kernel if the root filesystem/RAMDisk changes).

After making any changes to *include/configs/integratorcp.h* cross-compile U-Boot (will take only a few minutes to complete):

```
$ cd ${PRJROOT}/bootldr/u-boot-1.1.2-mod
$ make distclean
$ make clean
$ make integratorcp_config
$ make CROSS_COMPILE=arm-linux-
```

The main output(s) of the U-Boot cross-compilation are *u-boot* (U-Boot in ELF binary format), *u-boot.bin* (U-Boot raw binary image) and *u-boot.srec* (U-Boot image in Motorola's S-Record format). Copy the *u-boot* ELF binary format executable (ARM format) to the *images* project storage directory:

```
$ cp u-boot ${PRJROOT}/images
```

0.6 Programming the FLASH on the Integrator/CP

Now we want to program the FLASH on the ARM Integrator/CP's baseboard with the U-Boot image, the Linux Kernel image and the RAMDisk image, in that order (U-Boot must be the first image in the user area of FLASH so that on power-up of the Integrator/CP it will be run and launch the Linux Kernel etc.). Transfer the Linux kernel, RAMDisk and U-Boot images from ${PRJROOT}/images to an appropriate directory on a PC (which has Multi-ICE 2.2, ADS 1.2 and AFS 1.2.1 or similar installed), e.g. C:\linux_images.

The ARM Integrator/CP's baseboard contains 16MB of FLASH. The start of FLASH is memory mapped to address 0x24000000. The top 256KB of FLASH is reserved for system boot code (ARM Boot Monitor) while the remaining FLASH (62 blocks of size 256KB each) is available for our own code requirements. The ARM Boot Monitor is a μHAL (pronounced Micro-HAL - ARM Hardware Abstraction Layer that is the basis of AFS) application. It uses the μHAL library to initialize the ARM Integrator/CP platform when it runs.

Before programming, the ARM Integrator/CP's switch settings must be set to first run the ARM Boot Monitor only (which is in the system boot area of FLASH, the Boot Monitor will initialize the ARM Integrator/CP platform including processor and memory). To achieve this the switch settings beside the Ethernet connection on the Integrator/CP's baseboard must be set to DOWN-UP-UP-DOWN (from left to right) or as according to the ARM Integrator/CP documentation [10] S2[1], S2[2], S2[3], S2[4] must be set to ON-X-X-ON (where S2[4] is the left-most switch closest to the Ethernet connection, X is dont care, i.e. S2[1]=DOWN, S2[2]=UP, S2[3]=UP, S2[4]=DOWN).

Launch Multi-ICE 2.2 (ensuring the Multi-ICE cable is connected to the ARM Integrator/CP) and do File−>Auto-Configure to get the ARM Processor in the scan chain. Next Launch the AXD Debugger (All Programs −>ARM Developer Suite−>AXD Debugger), do File−>Load Image and navigate to and select C:\Program Files\ARM\AFSv1_4_1\Images\Integrator\afu.axf. Ensure the console window is active within AXD, if not select View−>Console. Run the afu.axf image by pressing *F5*, or by typing *go* in the command window, or by clicking on the *GO* icon.

The ARM Flash Utility (AFU - afu.axf) [9] is an application for manipulating and storing data within the FLASH on the Integrator/CP's baseboard only (it will not work for the FLASH on the ARM Integrator/CP's Logic Module). It will be used for programming the FLASH with the U-Boot, Linux Kernel and RAMDisk images. Do the following:

AFU>list

You should be presented with a list of the images that are already stored in FLASH (U-Boot, Linux Kernel, RAMDisk in that order), where each image has a seperate image number (U-Boot has image number 1, Linux Kernel has image number 2, RAMDisk has image number 3). You will also be shown the start block number and the end block number of each image in FLASH, for example the Linux Kernel image takes up blocks 2 (block 2 corresponds to address 0x24080000 which is where we have told U-Boot to expect the Linux Kernel image as described in the previous section) to block 10 (where each block contains 256KB of FLASH memory). To delete an image from FLASH you can use the AFU *Delete* command and to program the FLASH with a new image you can use the AFU *Program* command. For example if you want to program the FLASH with new U-Boot and RAMDisk images do the following:

AFU>Delete 1
AFU>Program 1 u-boot-cm920t C:\linux_images\u-boot B0
AFU>Delete 3
AFU>Program 3 ramdisk C:\linux_images\ramdisk.bin B12

In the above *u-boot-cm920t* and *ramdisk* are the respective image names, while *B0* and *B12* specifiy the start block locations in FLASH to program the respective images (note that B12 which corresponds to address 0x2430000 is the location in FLASH that u-boot expects to find the compressed filesystem image as described in the previous section). It has occurred previously that some FLASH sectors/blocks can get corrupted due to incorrect programming/crashing of PC (e.g. due to power failure or system error). If this occurs (you will get an error message when you try to program corrupted blocks) delete the offending blocks in FLASH using the AFU *DeleteBlock* command and reprogram the blocks, e.g. if block 2 to block 10 are corrupted:

AFU>DeleteBlock B2
AFU>DeleteBlock B3
AFU>DeleteBlock B4

```
AFU>DeleteBlock B5
AFU>DeleteBlock B6
AFU>DeleteBlock B7
AFU>DeleteBlock B8
AFU>DeleteBlock B9
AFU>DeleteBlock B10
AFU>Program 2 linux_2_6_12 C:\linux_images\linux-2.6.9.img B2
```

When finished with programming the FLASH with the relevant images, quit from the *AFU* utility and exit the AXD Debugger:

```
AFU>quit
```

0.7 Starting Linux on the Integrator/CP

We require the ARM Boot Monitor to run first as this program initializes the ARM Integrator/CP's system including processor and memory. The Linux Bootloader (U-Boot) will also do this when it runs but it does not seem to run correctly if this is the first image (in the user area of FLASH) that is called on power-up of the Integrator/CP. Hence the ARM Integrator/CP's switch settings must be set to first run the ARM Boot Monitor (which is in the system boot area of FLASH) followed by the Linux Bootloader (first image in the user area of FLASH). To achieve this the switch settings beside the Ethernet connection on the Integrator/CP's baseboard must be set to UP-UP-UP-DOWN (from left to right) or as according to the ARM Integrator/CP documentation [10] S2[1], S2[2], S2[3], S2[4] must be set to ON-X-X-OFF (where S2[4] is the left-most switch closest to the Ethernet connection, X is dont care, i.e. S2[1]=DOWN, S2[2]=UP, S2[3]=UP, S2[4]=UP).

With these switch settings and on power-up of the Integrator/CP, the ARM Boot Monitor will run and control will subsequently be passed to the first image in the user area of FLASH, i.e. the Linux Bootloader (U-Boot). The Bootloader will (i) again initialize the ARM Integrator/CP system in preparation of the launching of the Linux kernel (ii) copy the Linux kernel and RAMDisk/filesystem images from FLASH to the correct locations in RAM (iii) boot the Linux kernel, which will then uncompress the filesystem image in RAM and mount it.

Depending on the size of the filesystem, it can take U-Boot approximately 30-45 seconds to copy the compressed filesystem image from FLASH to RAM - during this stage the VGA display will be blank. When the Linux penguin icon appears on the VGA display and the left most LED above the alpha-numberic display on the baseboard flashes this indicates that the Linux kernel is now booting etc. On successful booting of the Linux kernel you will be presented with a **login:** prompt - login as **guest** or **root** with no password required. Note that the user *guest* also has root permissions - this is to allow users with software applications that need to access the address space of the ARM Integrator/CP's Logic Module (i.e. the FPGA and SRAM) the necessary permissions to open the */dev/mem* device file for memory access/mapping (ie. for the purpose of data transfers to/from the Logic Module). After logging in, type **startx** to launch the X Windows application (Blackbox [11] is used as the Window Manager), after which an *xterm* will be launched where you can run your application programs etc.

The *startx* application also sources the **.xinitrc** script in the users home area (e.g. /home/guest/.xinitrc) which tells X Windows the initial applications to launch (i.e. Blackbox Window Manager, xterm etc.) for a particular user. The *.xinitrc* script also sources the **/bin/setup_cdvp_net** shell script on the filesystem which will configure the ARM Integrator/CP system with the CDVP hardware group's network parameters for the purposes of ftp/telnet connections and transfers to other machines (the Ethernet LED on the baseboard will flash when the Ethernet hardware on the platform is setup correctly).

You can view the U-Boot and Linux Kernel booting messages/info by connecting the serial cable of the Integrator/CP to a PC and using a communication progam such as Hyperterminal with the following settings: Baud rate = 38400, Data bits = 8, Parity = None, Stop bits = 1, Flow control = Xon/Xoff. Launch Hyperterminal with the above settings and make a connection call to the ARM Integrator/CP (Call−>Call) followed by switching on the ARM Integrator/CP to boot Linux. Viewing these boot messages can be useful to aid debugging if the Linux Kernel fails to boot for any reason.

Amongst the pre-installed applications on the fileystem are the skin detection demos (/bin/skinDet_demo, /bin/skinDet_demo_nomenu), a YUV player (/bin/yuv_play), Busybox (utility that provides stripped down version of most Linux commands), ftp/ncftp/telnet (network file transfer utilities - can telnet to Integrator/CP with user names of guest/root with no password required), XServer/X Windows, nedit (text editor), GNU ftpd (ftp server - login to Integrator/CP from another machine as and with password:), gdb, xli (image viewer), scrot (screen grab utility), madplay (MP3,WAV player - however it does not appear to work properly, either a problem with the program itself or the audio interface/hardware on the Integrator/CP platform) as well as many standard Linux utilities.

Please refer to [12] for any general Linux system adminstration tasks (e.g. adding new users, changing passwords, network services configuration etc.) on the ARM Integrator/CP.

0.8 Issues to be aware of

0.8.1 ARM Boot Monitor

It is possible for the system boot area (i.e. the ARM Boot Monitor) of the FLASH on the ARM Integrator/CP to get erased due to static electricity (even just by touching the outer case holding the Integrator/CP boards!), as has happened a few times. If this happens then the Linux kernel will not boot properly (since the ARM Boot Monitor must run first before the Linux Bootloader). This occurence will be noticeable if Linux has previously run ok on the Integrator/CP but a subsequent boot of the system does not launch the kernel etc. If this happens you will need to reprogram the system boot area of the FLASH with the ARM Boot Monitor. Download 8554.zip from http://www.arm.com/support/downloads/info/8554.html (i.e. at http://www.arm.com/support/downloads/integrator.html: Boardfiles: Baseboards, Integrator/CP), extract it and it's internal *cp_bootmonitor.zip* file. This zip file is also stored on the local network at:
//poppintree.eeng.dcu.ie/HARDWARE_GROUP/Alan/ARM-Support/
8554.zip.

Launch Multi-ICE 2.2 (ensuring the Multi-ICE cable is connected to the ARM Integrator/CP) and do File−>Auto-Configure to get the ARM Processor in the scan chain. Next from a DOS prompt change to the *cp_bootmonitor* directory and run *progcards*. You will be presented with a choice of different versions of the ARM Boot Monitor - select the latest and *progcards* will subsequently program the system boot area of FLASH with the ARM Boot Monitor. Change the ARM Integrator/CP's switch settings (i.e. beside the Ethernet connection) to UP-UP-UP-DOWN (from left to right) and power-up the Integrator/CP again to subsequently run Linux again. If it again fails try an older version of the ARM Boot Monitor during the progcards menu selection stage (one of the versions of the ARM Boot Monitor should eventually work if the latest version does not).

0.8.2 Ethernet configuration EEPROM

The Integrator/CP incorporates an RJ-45 Ethernet socket which is driven by an SMSC LAN91C111 Ethernet Controller IC. The default setup parameters for this device, including the controller base address and the interface MAC address are stored in a serial EEPROM, which is connected directly to the Ethernet Controller via it's own dedicated interface. The Ethernet Controller reads these setup parameters from the EEPROM at reset and optionally when commanded to do so under software control. The ARM processor has indirect access to the EEPROM via some registers in the Ethernet Controller IC. This is the mechanism used to program the setup values at the time of board manufacture. Unfortunately, there is no protection mechanism to prevent inadvertent writes to the EEPROM. Therefore, it is relatively easy for the EEPROM to become corrupted by faulty software running on the ARM processor (e.g. if you have some code where you want to write to a memory mapped register in the Logic Module's FPGA but by error you write to a register in the Ethernet Controller IC that affects the EEPROM). This Ethernet EEPROM configuration corruption has previously happened once (a check for this is to run the selftest LAN program supplied with the Integrator/CP which will report that it cannot find the board's Ethernet interface or similar if this corruption occurs). If this corruption occurs you will not be able to do any network file transfers to/from the ARM Integrator/CP.

To fix any corruption of the Ethernet EEPROM there is a utility that you can download from ARM's website:
http://www.arm.com/support/downloads/info/9042.html (i.e. at http://www.arm.com/support/downloads/integrator.html Utility: Fix EEPROM). The *Fix_EEPROM* utility attempts to identify a corrupted Ethernet configuration EEPROM on the Integrator/CP - if the EEPROM is found to be corrupt this utility reprograms it with the factory default values (it also allows the user to program any Ethernet MAC address into the board if desired). This zip file is also stored on the local network at: //poppintree.eeng.dcu.ie/HARDWARE_GROUP/Alan/ARM-Support/9042.zip

Download the above utility and extract it to a local directory on your PC. Launch Multi-ICE 2.2 (ensuring that the Multi-ICE cable is connected to the ARM Integrator/CP) and do File−>Auto-Configure to get the ARM processor in the scan chain. Next Launch the AXD Debugger (All Programs −>ARM Developer Suite−>AXD Debugger), do Load−>Image and navigate to and select the *Fix_EEPROM.axf* image from the extracted directory. Ensure the console window is active within AXD, if not select View−>
Console. Run the *Fix_EEPROM.axf* image by pressing *F5*, or by typing *go* in the command window, or by clicking on the *GO* icon. If you are asked to specifiy a new MAC address enter 0 (zero) to skip reprogramming the MAC address (assuming that the one reported is correct: 04:79:F7:00:00:02).

0.9 Process Profiling

A single process in Linux is usually profiled using a combination of special compiler options and the GNU *gprof* utility. Basically, source files are compiled with a compiler option that results in profiling data to be collected at runtime and written to file upon the applications exit. The data generated is then analyzed by *gprof*, which displays the profiling data. *gprof* is not installed on the Linux filesystem due to the fact that the Debian package which it is part of (binutils) is quite large and it was thus considered not worthwhile installing on the filesystem itself. However, applications can still be cross-compiled with the special *gprof* options and subsequently run on the ARM Integrator/CP to produce the *gprof* output file *gmon.out* that contains the profiling data. Then *gmon.out* can be transferred to your PC for your host *gprof* executable to analyze the profiled data from the ARM Integrator/CP.

To cross-compile your applications for the purposes of profiling ensure that your *Makefile* includes the following cross-compiling and linker options:

```
CFLAGS = -Wall -pg
...
LDFLAGS = -pg
```

Note that the *-pg* option is used both for the cross-compiler flags and for the linker flags. The *-pg* option tells the cross-compiler to include the code for generating the performance data. The *-pg* linker option tells the linker to link the binary with *gcrt1.o* instead of *crt1.o*. The former is a special version of the latter that is necessary for profiling. Note also that we are not using the *-O2* cross-compiler optimization option. This is to make sure that the application generated executes in exactly the same way as we specified in the source file. You can then measure the performance of your algorithm(s) instead of measuring those optimised by the cross-compiler.

Once your application has been cross-compiled, transfer it to the ARM Integrator/CP and run it under Linux. Upon the applications exit, a *gmon.out* output file is generated with the profiling data. This file, as indicated previously, is cross-platform readable and you can use your PC host's *gprof* to analyze it. After having copied the *gmon.out* file to the local directory where you cross-compiled your application code on your PC, use *gprof* to retrieve the profiling data:

$ gprof exe_name gmon.out

Where in the above, *exe_name* is the name of the cross-compiled application executable in your local directory that you transfer to and run on the ARM Integrator/CP. The above command prints the profiling data to the standard output, you can redirect the output to a file using the > operator if you like. For more information regarding the use of *gprof* see the GNU *gprof* manual [13].

0.10 Miscellaneous Useful Websites

http://www.arm.com/products/DevTools/IntegratorCP.html
http://cmp.imag.fr - *Distributor of ARM products to Universities*
http://www.arm.com/linux/linux_download.html - *precompiled Linux sources*[6]
http://www.arm.linux.org.uk - *Main ARM Linux project site*
http://www.arm.linux.org.uk/armlinux/common/kernparams.html
http://www.simtec.co.uk/appnotes/AN0014/ - *Accessing I/O under Linux*
http://www.kernel.org - *Linux Kernel sources and utilities*
http://www.tldp.org/LDP/tlk/tlk.html - *Info on Linux Kernel*
http://www.aleph1.co.uk/armlinux/thebook.html - *ARM Linux Guide*
http://www.debian.org - *Debian distribution of Linux packages*
http://www.denx.de/twiki/bin/view/DULG/ - *Main U-Boot info site*
http://sourceforge.net/projects/u-boot - *Main project site for U-Boot sources*
http://blackboxwm.sourceforge.net - *Blackbox X Windows Manager*
http://www.cs.utah.edu/dept/old/texinfo/as/gprof.html - *Gprof info*
http://www.oreilly.com/catalog/runux4/ - *Running Linux*
http://www.oreilly.com/catalog/linag2/book/ - *Linux Net. Admin Guide*
http://www.xml.com/ldd/chapter/book/ - *Linux Device Drivers*
http://dsl.ee.unsw.edu.au - *Microprocessors and Interfacing course*
http://www.codesourcery.com - *Provides GNU Compilers for ARM*
http://www.kegel.com/crosstool/ - *Scripts that auto. build a Cross-Compiler*[7]
http://www.kaffe.org - *Embedded Linux Java VM*
http://www.lart.tudelft.nl - *StrongARM+Linux development board*
http://www.embeddedtux.org - *Book website for [1]*
http://www.uclinux.org - *Linux for MMU-less ARM Processors*
http://opensrc.sec.samsung.com - *Linux for MMU-less ARM Processors*
http://www.alsa-project.org/documentation.php - *ALSA Docs*
http://www.linuxdevices.com - *Info/articles on Linux Devices*
http://www.siliconpenguin.com - *Links to sw/hw embedded Linux resources*
http://www.xbox-linux.org/ - *Linux on the X-Box*

[6]currently only support for ARM Processors with ARMv5T architecture (not ARM920T which is ARMv4T)

[7]Not tested, automated cross-compiler build scripts can be dodgy (scripts can often require changes to build properly) but this website has been referenced quite often

Bibliography

[1] Karim Yaghmour, *Building Embedded Linux Systems*, 2003 (O'Reilly)

[2] //poppintree.eeng.dcu.ie/HARDWARE_GROUP/Alan/ARM-Cross-Compiler/docs/cygwin-cross-compiler.pdf

[3] http://www.arm.com/linux/arm-linux-gcc-2004-q3.tar.gz

[4] http://www.arm.com/linux/ARM_Embedded_Linux-1.1b.tar.gz

[5] http://www.arm.linux.org.uk/mailinglists - main ARM Linux Mailing Lists

[6] http://www.debian.org

[7] http://sourceforge.net/projects/u-boot

[8] http://www.denx.de/twiki/bin/view/DULG/

[9] AFS Reference Guide - Chapter 7 Using the ARM Flash Utilities

[10] ARM Integrator/CP User Guide - Chapter 2 Getting Started - page 2-3

[11] http://blackboxwm.sourceforge.net

[12] Welsh, Dalheimer, Kaufman, *Running Linux*, 2002 (O'Reilly)

[13] http://www.gnu.org/software/binutils/manual/gprof-2.9.1/gprof.html