Porting Android to Altera Cyclone V SoC

Muhammad Obaidullah and Gul N. Khan

***Abstract*- In this paper we analyze the android ecosystem and adapt it to fit onto an FPGA-based SoC (Cyclone V). The end hardware-software system is capable of controlling the FPGA hardware from the Android application. Android is predominantly used in general purpose embedded systems but has been recently been adapted to work for hard real-time embedded system due to a real-time patch available for the Linux kernel. This allows the kernel tasks to be preempted. Porting Android to any embedded system comes with huge advantages such as vastly available applications and ease of GUI design (using XML layouts). Porting android stack to a HPS-FPGA system allows apps to take advantage of FPGA fabric and perform partial reconfiguration to build and tear-down application specific accelerators on the remaining FPGA fabric. This opens up a whole new domain of download and installable virtual hardware components.**

I. INTRODUCTION

Android has grown to be the number one smartphone operating system. While Android continues to be the most popular Operating System (OS) for smartphones and tablets, it is also being adopted for other embedded devices and industrial equipment. Applications which are soft real-time to hard real-time can take advantage of the GUI rich OS, advanced APIs and Inter-Process Communication (IPC) framework. Since the introduction preemptive kernel mode in Linux version 2.4 and later, Linux OS can now be used for hard real-time applications.

There are a number of free software libraries provided with Android and taking advantage of these will reduce the development cost of the application. Provided libraries include Web browser, File System, WebKit, SQLite, OpenGL/ES, Graph/chart, and Network Stack. Until now, powerful embedded GUIs were rarely available and very expensive. GUI for Android can be designed easily using android XML layout files and tuned according to customer's needs. GUI by default is compatible with multi-touch, giving the familiar easy-to-use feeling of a smartphone operating system.

A. Why android on DE1 SoC ?

Porting Android to Cyclone V will allow much higher abstraction level than working with plain linux C/C++ native code. Applications within the OS can take advantage of FPGA acceleration and the well-implemented and well-tested Android software stack. Porting android to DE1 SoC allows use of over 2.4 million applications currently available in the Google Play store. It also opens new boundaries for software developers to explore hardware accelerations for compute intensive applications. It has been found that real-time operating systems designed for Symmetric Multi-Processing (SMP) will generally provide similar or better performance and lower latency than bare-metal applications (no operating system). Besides having an FPGA fabric within the Cyclone V, it contains dual ARMv7 cores within the HPS which are well-capable of running Android.

From research point of view, it is beneficial to explore the advantages and drawbacks of an embedded system which makes use of hardware acceleration rather than increased CPU clocks. Use of such hybrid system may reduce overall system power consumptions.

B. Linux Kernel

Linux is an operating system found in a wide variety of computing devices including personal computers, servers, smartphones and other embedded systems. Among other features, most important feature provided by any operating system is multitasking. This is so that the hardware resources (processing unit, memory, etc.) available on the chip can be shared among requesting tasks and allow the processor to give illusion of performing multiple tasks at once (concurrency).

Although sometimes used interchangeably, there is a difference between a kernel and an operating system. The kernel is part of operating system. It is responsible for memory management, network management, device driver, file management, and process management. An operating system is composed of the kernel and applications (i.e. compiler, text editor, window manager, GUI) which enables users to perform useful tasks. In other words, the kernel is the 'brain' of the operating system.

Any task which is to be performed by the kernel has higher priority than the task performed by the user. Therefore, modern CPUs have two modes in which a task can be run, kernel mode (system mode) or user mode. In kernel mode, the CPU executes instructions which are related to kernel (kernel code) and provides unlimited access to the code. Any instruction can be executed in kernel mode and any memory address can be referenced. All other codes are executed in user mode where some CPU instructions cannot be directly initiated and some memory locations are off-limits. Hence, if the user code wants to execute some instructions which are not available in user mode, it has to make a '*system call'* in order to perform privileged instructions. Such privileged instructions include process creation and input/output operations.

In non-preemptive type kernels, while a process is in kernel mode, it cannot be arbitrarily suspended and replaced by another process (i.e., preempted) for the duration of its time slice (i.e., allocated interval of time in the CPU), in contrast to user mode, except when it voluntarily relinquishes control of the CPU. Although the Linux kernel was of non-preemptive nature, which means that the kernel mode tasks were not interruptible, the version 2.4 and later are preemptive. This makes Linux kernel strong candidate for embedded devices and control applications where timing and interrupt handling are crucial. Processes in kernel mode can be interrupted by an interrupt or an exception.

C. Android

Contrary to popular belief, Android is not an operating system. Android is an open source software stack for a wide range of mobile and embedded devices and corresponding open source project led by Google. It is emerging as developer's choice in order to program, develop, and design or IoT, mobile computing, and other embedded devices because of its highly abstracted and modularized components which eases software development. Android provides the freedom to the developer to implement own device specifications and drivers. The hardware Abstraction Layer (HAL) provides a standard method for creating hooks between the Android platform stack and custom hardware.

A. Application Framework

Application framework provides software developers with APIs to use in order to develop android applications. This is the final layer on which the app runs. Some developer APIs map directly to underlying HAL interfaces too.

B. Binder IPC

Binder Inter-Process Communication (IPC) is a mechanism which facilitates inter-processes communications. It allows application framework to cross boundaries and call into the Android system services code. This allows high-level APIs to interact with Android systems services. Normally, under Linux environment, the processes communicate by using named FIFOs, signals, sockets, semaphores, message queues, or shared memories. A higher abstraction is done of this inter-process communication technique and provided as a feature in Android stack. It is a lightweight RPC (Remote Procedure Communication). One Android process can call a routine in another Android process, using binder to identify the method to invoke and pass the arguments between processes.

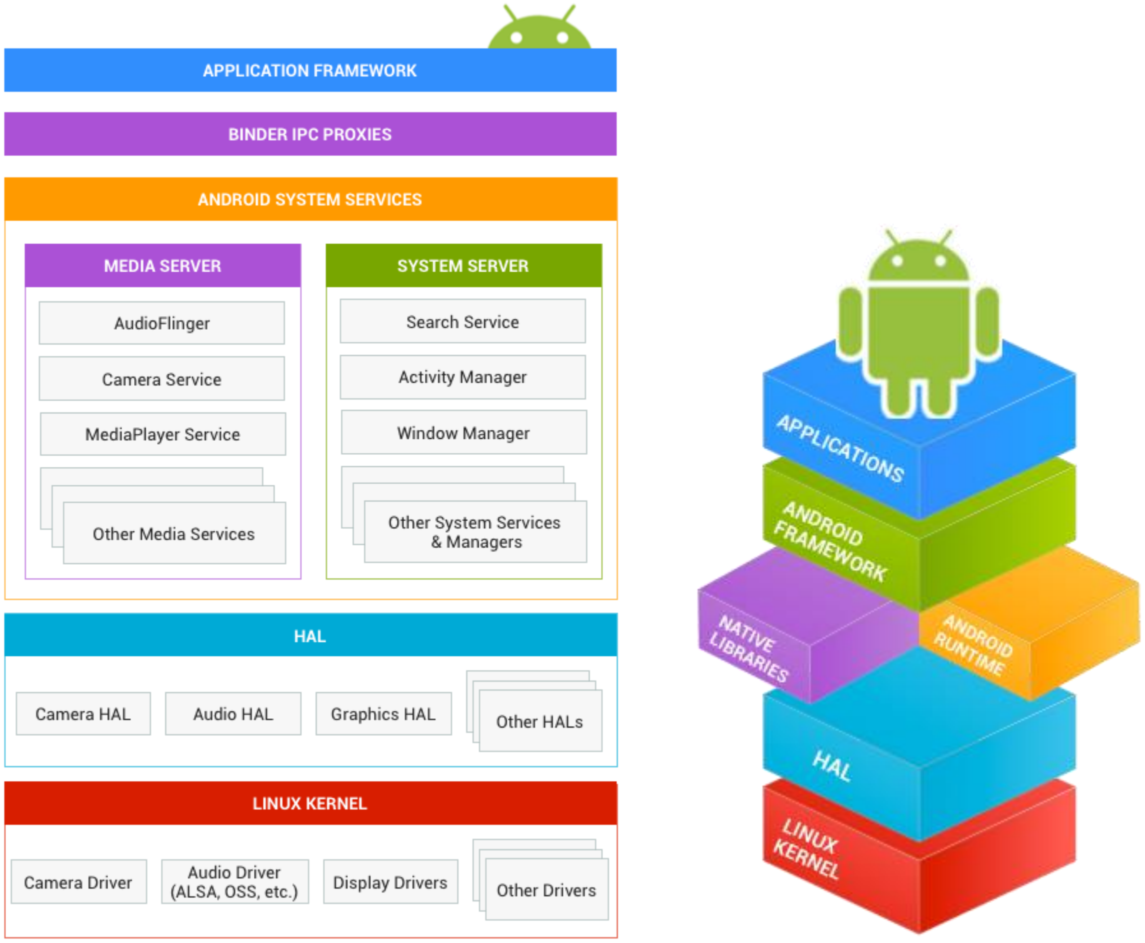


Figure 1. Android software stack and layers.

C. System Services

System services lie in the middle of HAL and application framework and allow APIs to communicate with the underlying hardware. They are focused and modular components which provide focused helping hand to the software developer. For example Notification Manager manages all notifications and can be asked by any process to view/edit/delete notifications. Similarly, Camera Service allows processes to view camera stream, trigger image capture etc.

D. Hardware Abstraction Layer (HAL)

The hardware abstraction layer defines a standard interface for hardware vendors to implement and allows Android to be agnostic about lower-level driver implementations. HAL implementations are packaged into modules (.so file) and loaded by Android system at appropriate time (dynamic linking). Each specific hardware added into the custom system needs to have a corresponding HAL and driver. Android does not specify a standard interaction between HAL implementation and device drivers, so developers are free to do what is best for the situation. However, in order for Android to correctly interact with custom hardware, contract defined in each hardware-specific HAL interface should be followed.

Each hardware-specific HAL interface has properties that are defined in source code provided by Google. This guarantees that HALs have a predictable structure. This interface allows the Android system to load the correct versions of custom HAL modules in a consistent way. There are two general components that a HAL interface consists of: a module and a device.

A module represents a custom packaged HAL implementation, which is stored as a shared library (.so file). It contains metadata such as the version, name, and author of the module, which helps Android find and load it correctly. The hardware.h header file in the source code defines a struct, hw_module_t, that represents a module and contains information such as the module version, author, and name.

In addition, the hw_module_t struct contains a pointer to another struct, hw_module_methods_t, that contains a pointer to an "open" function for the module. This open function is used to initiate communication with the hardware that the HAL is serving as an abstraction for. Each hardware-specific HAL usually extends the generic hw_module_t struct with additional information for that specific piece of hardware.

E. Linux Kernel

Developing custom device drivers is similar to developing a typical Linux device driver. Android uses a version of the Linux kernel with a few special additions such as wake locks (a memory management system that is more aggressive in preserving memory), the Binder IPC driver, and other features important for a mobile embedded platform. These additions are primarily for system functionality and do not affect driver development.

Developer can use any version of the kernel as long as it supports the required features (such as the binder driver). However, it is recommended to use the latest version of the Android kernel provided by Google.

II. PAST WORK

So far three entities have managed to port Android successfully to Altera Cyclone V SoC.

TABLE I

ANDROID PORTED TO CYCLONE V

Entity Name	Missing Features	OS Image Available	Source Code Available	Link
MRA Digital [2]	-	No	No	https://youtu.be/zHqS_yWiMNI
Fujisoft [3]	-	No	No	https://www.fsi-embedded.jp/e/_emb/gaforandroid_e/
University of Toronto [4]	Audio CODEC	Yes	No	https://rocketboards.org/foswiki/view/Projects/AndroidForDE1SoCBoard

One implementation is from University of Toronto's 4th year capstone project at Department of Electrical & Computer Engineering, with group members Kevin Nam, David Xie, and Steven Nesmith, under supervision of Prof. Stephen Brown.

An Android app called "DE1SOC" comes preinstalled in the image. This Android app was created to demonstrate Android app to FPGA communication. Shown in Figure 2, the app has an interface that allows users to toggle the LEDs on the board, as well as read the value of the switches. These LEDs and Switches are connected to the FPGA's I/O pins, which are in turn connected to PIO IP cores that have been instantiated in the FPGA. These cores provide memory-mapped register interfaces which are made available to the ARM processor through the lightweight HPS-to-FPGA bridge. The Linux kernel's GPIO drivers are used to expose the pins as GPIO devices (in /dev/), providing a file-based I/O mechanism. The Android app is then able to read and write these files to read and write the values of the LEDs and switches. The FPGA is programmed automatically as part of the boot sequence.

Users can interact with the GUI by using the touch screen of the Terasic MTL. If an MTL is not available, users can use a USB mouse, connected to one of the two USB ports. When a mouse is connected, a mouse cursor will appear on the screen. Users can provide internet connectivity by plugging in an ethernet cable. Support for the board's audio CODEC has not been implemented at this time. They used Chris Rauer's modified Linux kernel with added support for the Altera frame buffer.

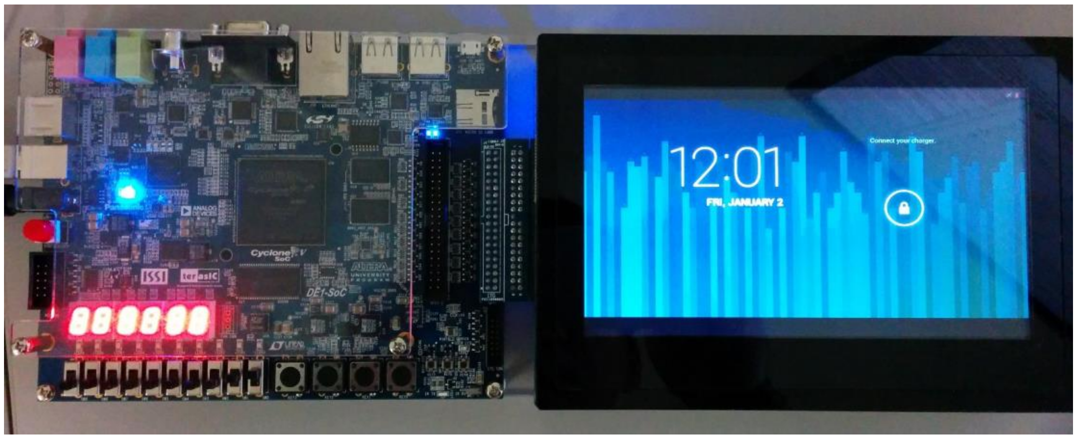


Figure 2. Android running on DE1 SoC Altera Cyclone V by University of Toronto.

A. Difference between Linux and Android Boot Sequence

As android is forked from Linux source code, the boot sequence are similar until init() program is called. In Android, after init() is called, it calls a base service called Zygote which is responsible for running the Android Runtime (ART) or Dalvik Virtual Machine (DVM). We will explain the significance of ART or DVM later in this section. This is depicted in figure 3.

In desktop PC Linux, the memory location of the BIOS software is hardwired on the silicon. As the power button is pressed, the CPU loads the BIOS software into RAM and starts to execute it. This BIOS software is vendor specific and usually provided by the company who manufactures motherboards (eg. Asus, Gigabyte, etc.). Something similar happens in embedded devices but instead of running BIOS, the embedded CPU runs instructions in the boot ROM. Typical boot flow of the Cyclone V SoC is shown in figure 4.

Typically there are two bootloaders in a typical boot flow. In PCs, they are simply named as bootloader 1 and bootloader 2. In Embedded devices, they are named as Pre-loader and bootloader. Pre-loader is a compiled binary form of code compiled specifically for the CPU architecture and is put into a memory location where there is no file system or formatting. This is because at pre-loader stage, CPU has not yet set up memory hierarchy and SDRAM.

Pre-loader is a compiled binary code which can be directly executed by the CPU without compiling or interpreter. Pre-loader is supposed to pave the path for the boot loader to run on CPU. It sets up the CPU frequency, bus settings, bring up SDRAM, and load the next stage bootloader from flash to SDRAM and jump to it.

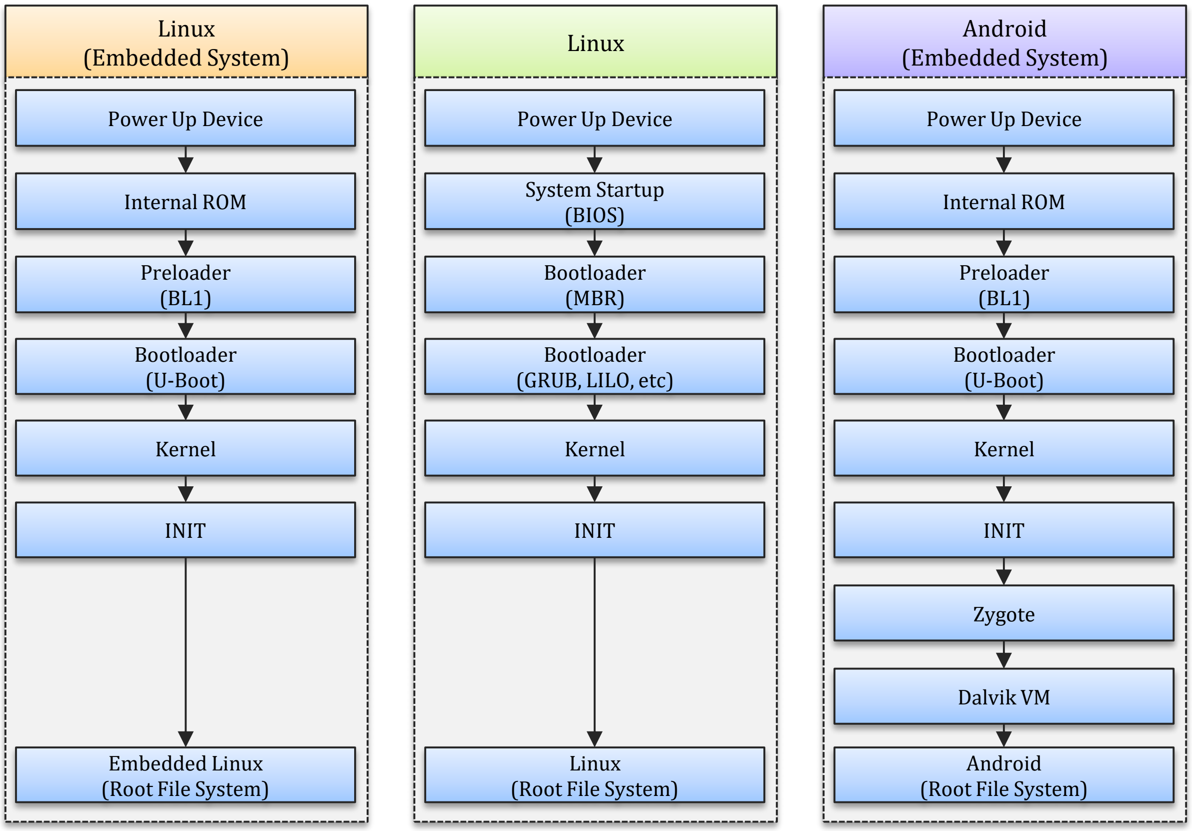


Figure 3. Boot sequence of Linux on an embedded system (left), Linux on PC (center), and Android on an embedded system (right).

In the DE1 SoC development board, the switches at the bottom of the board can change the hardwiring of the CPU to load the preloader from either SD-Card or EPCQ. Preloader is generated from the Board Support Package (BSP) provided by the hardware manufacturer (in this case Altera). Since Pre-loader is supposed to load the bootloader, it needs to know the bootloader name. This is configured from the Preloader generator provided by Altera (mkpimage).

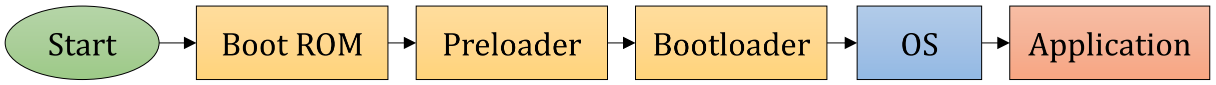


Figure 4. Altera Cyclone V SoC typical boot flow.

The most popular bootloader among PC users is GRUB (short for GNU GRand Unified Bootloader) because of its multi-boot feature and ease of use. It allows detection of multiple bootable operating systems and provides a user interface at boot time to choose the OS to load and boot. It is popular among Linux and Windows users who like to have dual-boot options although it is capable of detecting and booting Mac OS too.

In embedded domain, most popular bootloader is U-boot. It is simple, easy, ported to most architectures and open source. Altera has forked the U-boot code and added support for SoC FPGAs. The source code for it is available at https://github.com/altera-opensource/u-boot-socfpga. The bootloader is written in C and comes with multiple MakeFiles and board configurations. Stages from preloader to init() are similar for embedded Linux devices and embedded Android devices.

Similar to pre-loader, bootloader's job is to pave the path for the next stage which is the kernel. So it sets up essential CPU parameters, memory addresses, provide functionality to read file systems, provide drivers for reading from memory card file system, set up serial baud rate for debugging, and then load the kernel image from the FAT32 partition of SD card into RAM and run it. As pre-loader, it also needs to know the name of the file which has the kernel image. But instead of changing the source code and recompiling every time for the new kernel image, the u-boot bootloader looks for a u-boot script to run which informs it of the kernel image name to load and perform any other commands before loading the kernel. This u-boot script (usually with extension ".scr") is present in the U-Boot partition as shown in figure 5.

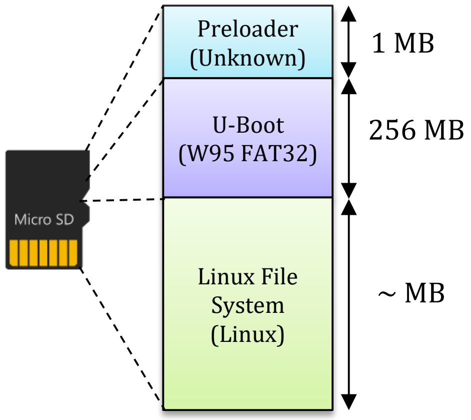


Figure 5. Bootable SD card contents.

When the kernel has been loaded into the memory, the CPU starts running the init() function and that leads to kernel and user space being setup. After this, the kernel loads the file system from the SD card and mounts it with proper permissions (ie. "chown 777” etc.). This file system is in another partition of the memory card as shown in figure 5.

In Android 4.4 (KitKat), ART (Android Runtime) was introduced which replaced the old DVM (Dalvik Virtual Machine). ART is an application runtime environment used by the Android OS which is the process virtual machine. ART translates the application’s bytecode into native instructions which the particular processor in the embedded system can understand. These native instructions are then later executed by the device’s runtime environment.

We need ART because each Android application runs on its own virtual machine and a process. ART handles creation of virtual machine and providing dynamically linked libraries to the application. ART introduces ahead-of-time (AOT) compilation, which can improve app performance. ART also has tighter install-time verification than Dalvik. Launching Android applications was really slow before Android KitKat. Ahead-of-time compilation compiles the java bytecode into native machine code. When the android boots up, it creates a virtual machine ready to accept an application to run. This is being ready for execution ahead of time. When the user presses on the application to launch, the application code is compiled using the already running virtual machine and the application is assigned to that virtual machine. And then the ART launches another VM ready to receive another application. This way number of VMs started are always 1 more than the number of applications running on the embedded system.

B. Raw Binary Files (.rbf)

These are raw binary files with extension .rbf which are compiled from the VHDL source code. The purpose of these files is to represent a design entity which can be loaded onto the FPGA fabric. This is shown in figure 6. These files are popularly used by Altera to do partial reconfiguration on the FPGA.

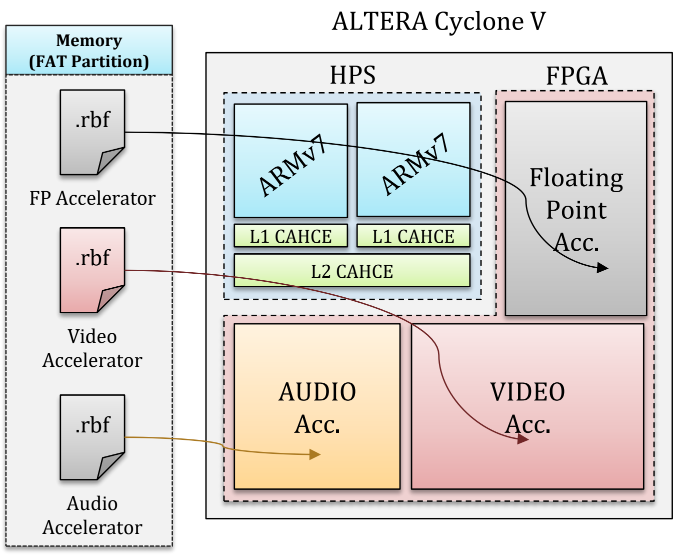


Figure 6. Purpose of .rbf files.

The steps to generate these are as following:

1) Write hardware code in VHDL/Verilog/System Verilog
2) Compile and verify the design using Quartus to get SRAM Object File (.sof)
3) Convert .sof file to .rbf

4) Copy the generated .rbf file to SD card's FAT partition
5) Decide on how to load the configuration bit stream (Choose from following)
    a. Pre-loader script
    b. U-Boot source code
    c. U-Boot script
    d. Linux init
    e. Linux application (runtime)

As one might see that loading these .rbf files onto the FPGA fabric is not that straight forward. The design entities should not short two pins together and should not use the same Logic Elements (LEs). Altera provides a hardware device known as FPGA manager to control, manage, and program the FPGA fabric.

C. *Device Tree*

In order to not compile the kernel every time, the hardware changes, a system hardware description in a tree format can be provided to the kernel to inform it of the addresses and the hardware attached to the CPU. The format of the tree is simple and is kind of like JavaScript Object Notation (JSON). The device drivers area usually written in C and embedded into the source code of the kernel. When the kernel boots up, the device tree is loaded and the driver is connected to hardware by use of major and minor number. Device trees are especially important in our case since the FPGA hardware changes very frequently and the kernel should not be built every time a new FPGA configuration is loaded. The connection between application and hardware is performed by use of device tree as shown in figure 7.

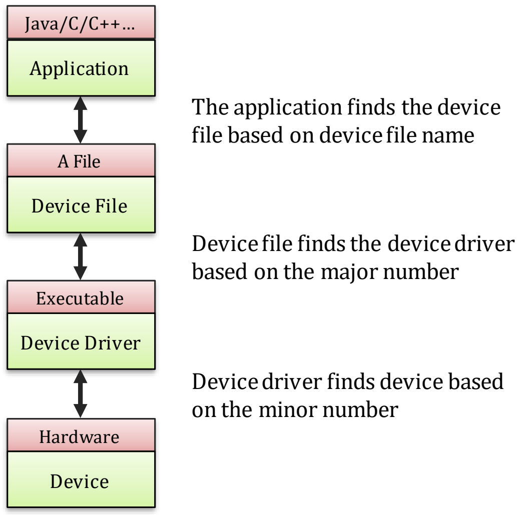

Java/C/C++...
Application
The application finds the device file based on device file name
A File
Device File
Device file finds the device driver based on the major number
Executable
Device Driver
Device driver finds device based on the minor number
Hardware
Device

Figure 7. Connecting hardware with software application using device trees.

FPGA Regions are introduced as a way to solve the problem of how to program an FPGA under an operating system and have the new hardware show up in the device tree. By adding these bindings to the Device Tree, a system can have the information needed to program the FPGA and add the desired hardware, and also the information about the devices to be added to the Device Tree once the programming has succeeded. The FPGA manager has a driver provided by Altera and also has an entry in the device tree as shown in figure 8.

An FPGA Region specifies the devices (FPGA Manager and FPGA Bridges) needed to reconfigure a FPGA device. In the live Device Tree, an FPGA Region reflects the current configuration of the device. If the live tree shows a "firmware-name" property under a FPGA Region, the FPGA already has been programmed with that firmware.

A device tree overlay that targets a FPGA Region and adds the "firmware-name" property and child nodes is a request to reprogram the FPGA and, if successful, add the child nodes. If reprogramming is not successful, the overlay must be rejected and not added to the live tree.

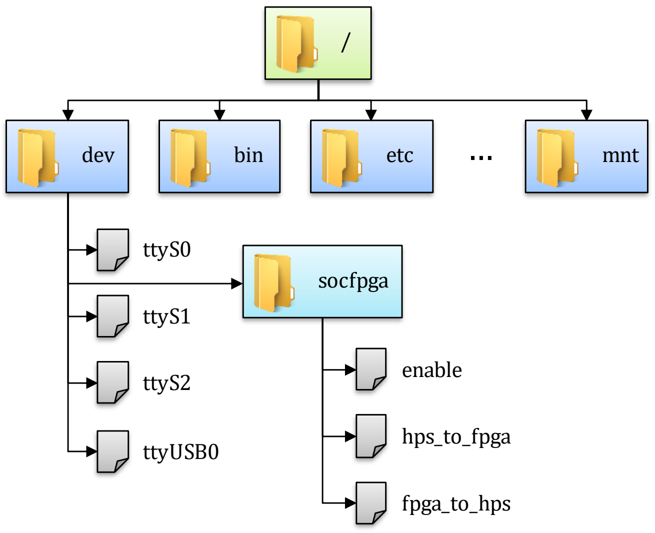


Figure 8. Linux Device Files Structure for SoC FPGA.

SNIPPET 1: Sample Device tree overlay for LED PIOs (GPIO)

```
fragment@0 {
        target-path = "/soc/base_fpga_region";
        #address-cells = <1>;
        #size-cells = <1>;
        __overlay__ {
            #address-cells = <1>;
            #size-cells = <1>;
            firmware-name = "soc_system.rbf";
            jtag_uart: serial@20000 {
                compatible = "altr,juart-1.0";
                reg = <0x20000 0x8>;
                interrupt-parent = <&intc>;
                interrupts = <0 42 4>;
            };
            led_pio: gpio@10040 {
                compatible = "altr,pio-1.0";
                reg = <0x10040 0x20>;
                altr,gpio-bank-width = <4>;
                #gpio-cells = <2>;
                gpio-controller;
            };
        };
};
```

III. METHODOLOGY

A. *Studying Cyclone V HPS Memory Map*

Before designing the software or modifying the kernel, one needs to study the internal architecture of the device or board at hand. The most useful information for porting an operating system is the memory map. Because it contains all the virtual bindings (connections) from memory to hardware modules which the CPU can control. This is done through memory mapping. As seen in figure 9, SD MMC (SD card reader/writer module) is mapped to the memory locations 0xFF704000 – 0xFF7043FF. Whenever the CPU writes to or reads from these memory locations, the CPU is actually talking to the SDMMC module.

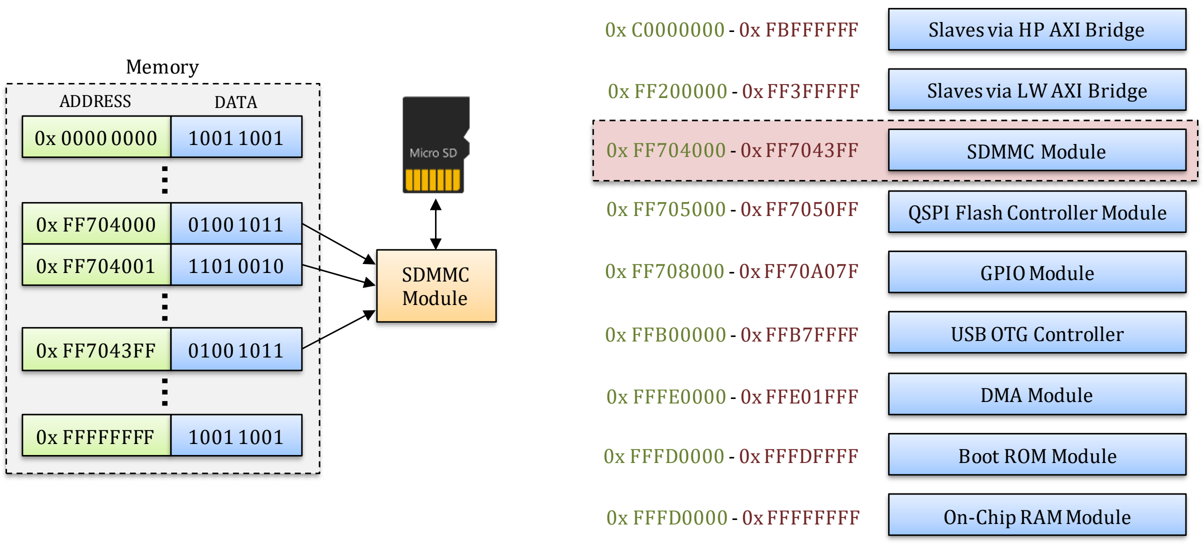


Figure 9. Cyclone V brief memory map.

B. *Finding Sources*

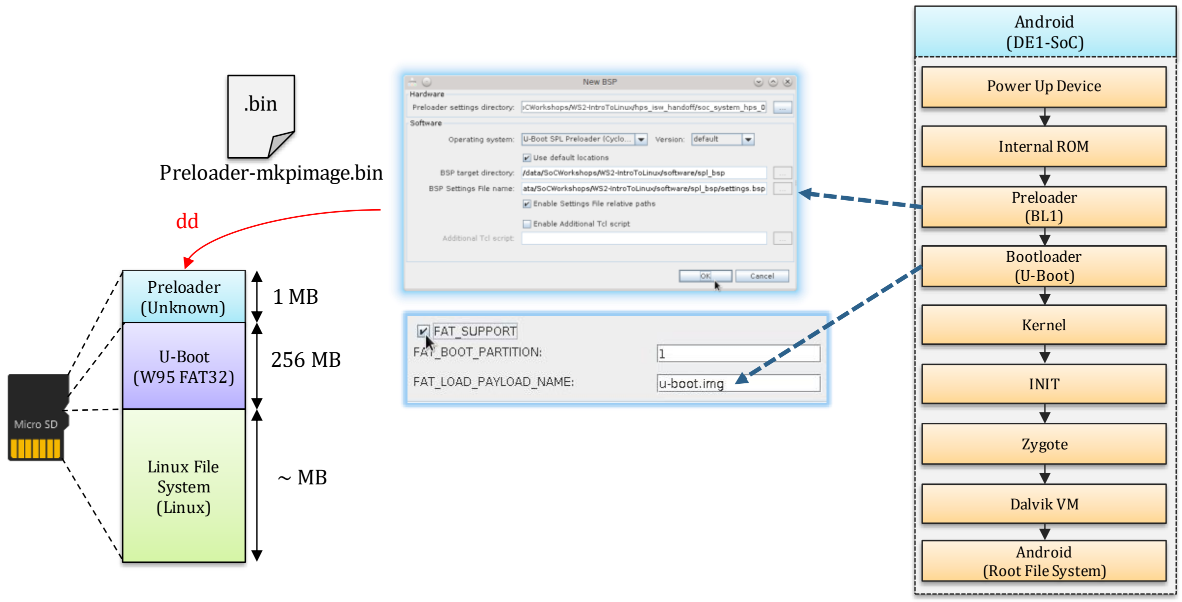
Most of the open source code is provided by Altera and is maintained and contributed in the close eyes of Altera and Linus Torvalds (creator of Linux).

S.No.	Component	Repository Name	GitHub URL
1.	Angstrom Scripts	angstrom-socfpga	https://github.com/altera-opensource/angstrom-socfpga
2.	Boot Loader	u-Boot-socfpga	https://github.com/altera-opensource/u-boot-socfpga
3.	Device Tree Generator	sopc2dts	https://github.com/altera-opensource/sopc2dts
4.	Linux Kernel	linux-socfpga	https://github.com/altera-opensource/linux-socfpga
5.	Reference Designs	linux-refdesigns	https://github.com/altera-opensource/linux-refdesigns
6.	Yocoto Layer	meta-altera	https://github.com/altera-opensource/meta-altera

C. *Internal ROM Configuration*

MSEL[4:0]	Configuration Scheme	Description
10010	AS	FPGA configured from EPCQ (default)
01010	FPPx32	FPGA configured from HPS software: Linux
00000	FPPx16	FPGA configured from HPS software: U-Boot, with image stored on the SD card, like LXDE Desktop or console Linux with frame buffer edition

D. Generating Preloader



Preloader settings directory: :CWorkshops/WS2-IntroToLinux/hps_isw_handoff/soc_system_hps_0

Operating system: U-Boot SPL Preloader (Cyclo... Version: default

[✓] Use default locations

BSP target directory: /data/SoCWorkshops/WS2-IntroToLinux/software/spl_bsp

BSP Settings File name: ata/SoCWorkshops/WS2-IntroToLinux/software/spl_bsp/settings.bsp

[✓] Enable Settings File relative paths

[ ] Enable Additional Tcl script

Additional Tcl script:

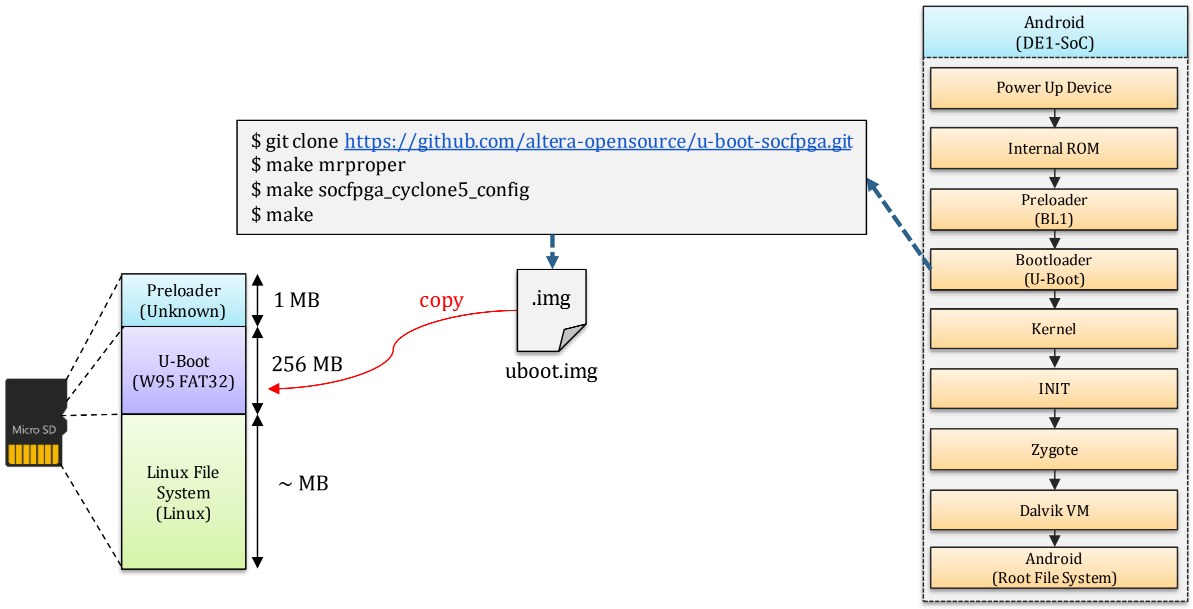
[✓] FAT_SUPPORT

FAT_BOOT_PARTITION: 1

FAT_LOAD_PAYLOAD_NAME: u-boot.img

Figure 10. Steps involved in generating a preloader.

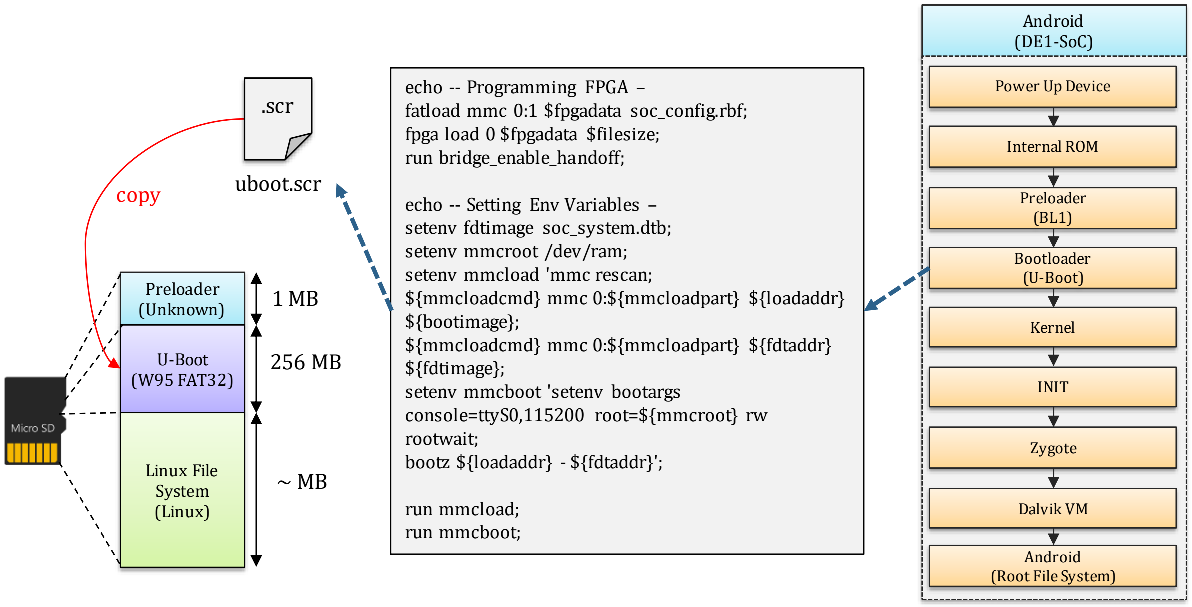
E. Generating U-Boot



```
$ git clone https://github.com/altera-opensource/u-boot-socfpga.git
$ make mrproper
$ make socfpga_cyclone5_config
$ make
```

Figure 11. Steps involved in generating U-Boot.

F. *Generating U-Boot Script*



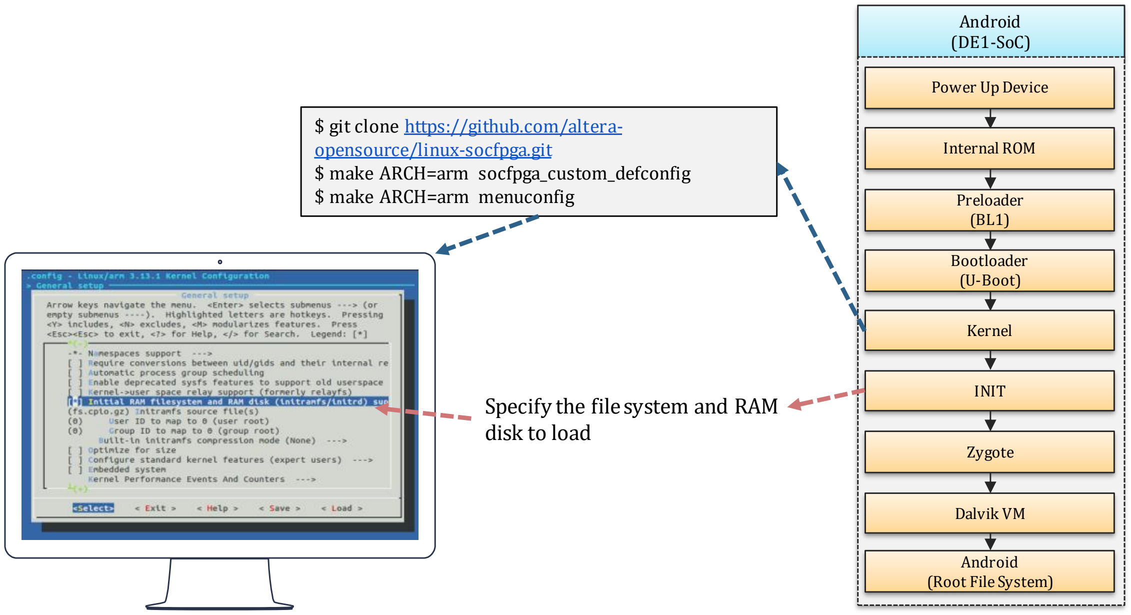
```
echo -- Programming FPGA –
fatload mmc 0:1 $fpgadata soc_config.rbf;
fpga load 0 $fpgadata $filesize;
run bridge_enable_handoff;

echo -- Setting Env Variables –
setenv fdtimage soc_system.dtb;
setenv mmcroot /dev/ram;
setenv mmcload 'mmc rescan;
${mmcloadcmd} mmc 0:${mmcloadpart} ${loadaddr}
${bootimage};
${mmcloadcmd} mmc 0:${mmcloadpart} ${fdtaddr}
${fdtimage};
setenv mmcboot 'setenv bootargs
console=ttyS0,115200 root=${mmcroot} rw
rootwait;
bootz ${loadaddr} - ${fdtaddr}';

run mmcload;
run mmcboot;
```

Figure 12. Steps involved in generating U-Boot Script.

G. *Configuring Linux*



```
$ git clone https://github.com/altera-opensource/linux-socfpga.git
$ make ARCH=arm socfpga_custom_defconfig
$ make ARCH=arm menuconfig
```

```
.config - Linux/arm 3.13.1 Kernel Configuration
> General setup
                              General setup
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or
  empty submenus ----).  Highlighted letters are hotkeys.  Pressing
  <Y> includes, <N> excludes, <M> modularizes features.  Press
  <Esc><Esc> to exit, <?> for Help, </> for Search.  Legend: [*]
      ^(-)
      -*- Namespaces support  --->
      [ ] Require conversions between uid/gids and their internal re
      [ ] Automatic process group scheduling
      [ ] Enable deprecated sysfs features to support old userspace
      [ ] Kernel->user space relay support (formerly relayfs)
      [*] Initial RAM filesystem and RAM disk (initramfs/initrd) sup
      (fs.cpio.gz) Initramfs source file(s)
      (0)     User ID to map to 0 (user root)
      (0)     Group ID to map to 0 (group root)
              Built-in initramfs compression mode (None)  --->
      [ ] Optimize for size
      [ ] Configure standard kernel features (expert users)  --->
      [ ] Embedded system
          Kernel Performance Events And Counters  --->
      ↓(+)
        <Select>    < Exit >    < Help >    < Save >    < Load >
```

Figure 13. Steps involved in configuring Linux build.

H. Compiling Linux

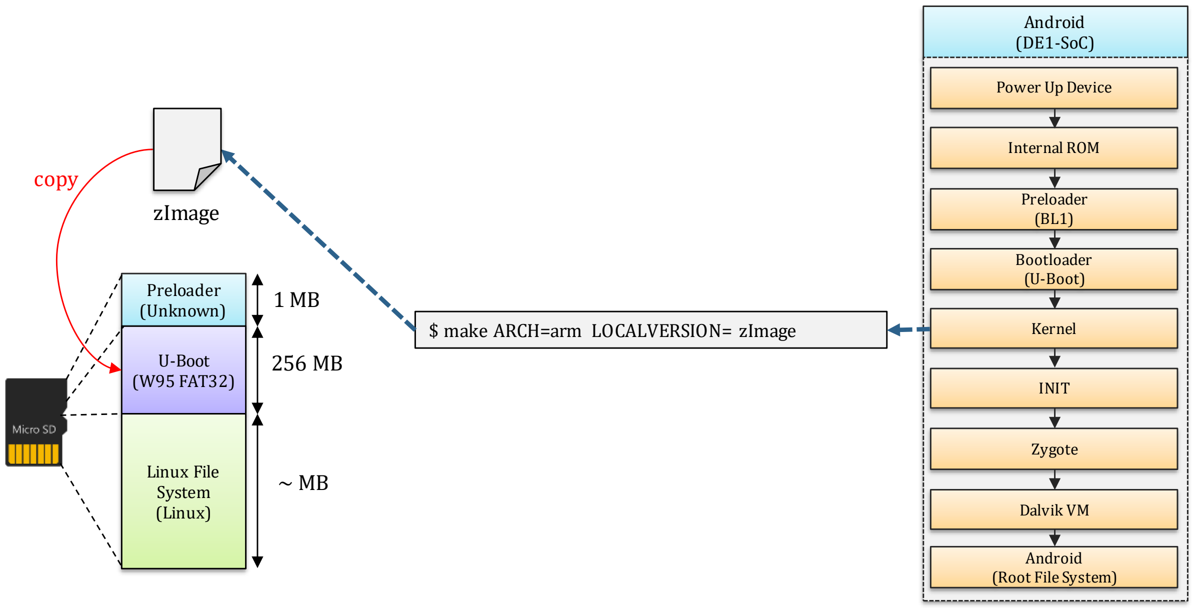


Figure 14. Steps involved in compiling Linux (takes about 15 minutes).

I. Generating Device Tree Blob

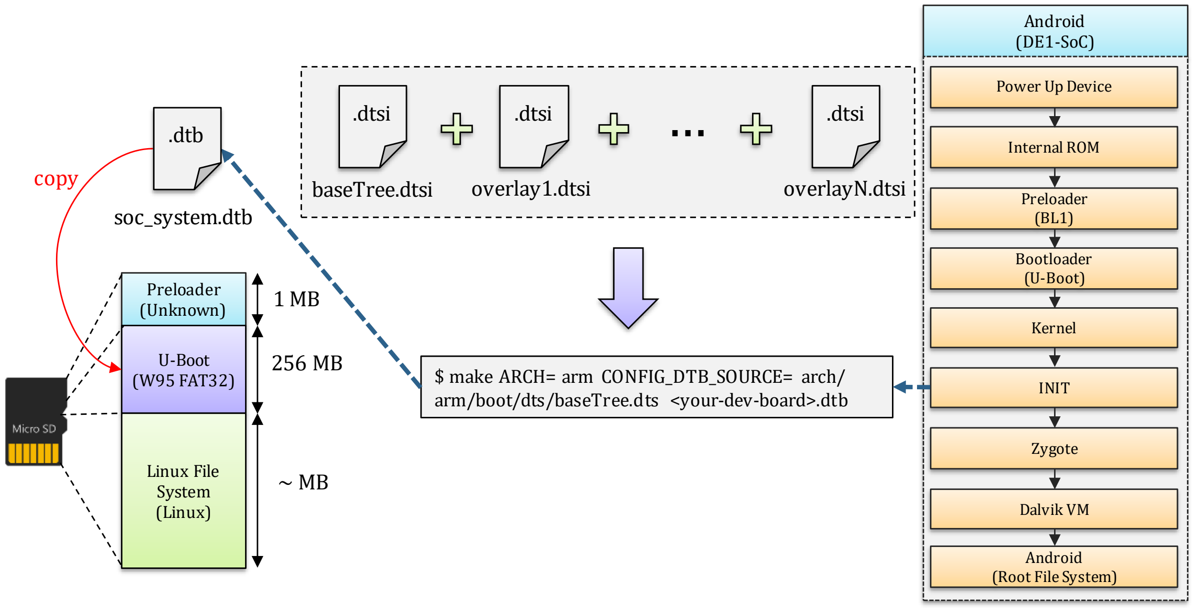


Figure 15. Steps involved in generating device tree blob.

IV. DESIGN

A. Hardware to software hand off

The purpose of the Device Tree is to move a significant part of the hardware description into a data structure that is no longer part of the kernel binary itself. This data structure, the Device Tree Source is compiled into a binary Device Tree Blob. The Device Tree Blob is loaded into memory by the bootloader, and passed to the kernel. It replaces all the board-*.c files, and removes all the manual registration of platform_device. Also, no longer needed to have Kconfig options for each board. Usage of the Device Tree is mandatory for all new ARM SoCs.

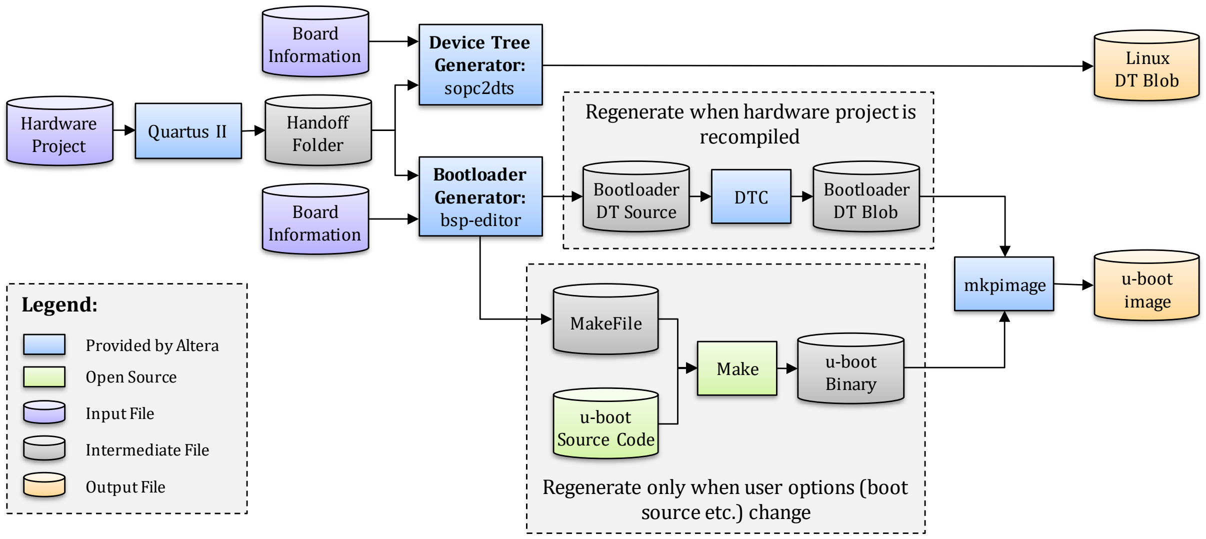


Figure 9. Linux Device Files Structure for SoC FPGA. [5]

B. Building Altera Linux OS

I. Host Setup

We are using 64 bit Ubuntu MATE (16.04.1 LTS) as a host machine to compile the Altera Linux OS. There are few packages required to be installed before building the Linux OS image for Altera Cyclone V FPGA.

1. First we need to update the package manager using the following command:

```
$ sudo apt-get update
```

2. Update the package manager using the following command:

```
$ sudo apt-get uprade
```

3. Install the required packages using the following command:

```
$ sudo apt-get install sed wget cvs subversion git-core coreutils unzip texi2html texinfo libsdl1.2-
dev docbook-utils gawk python-pysqlite2 diffstat help2man make gcc build-essential g++
desktop-file-utils chrpath libgl1-mesa-dev libglu1-mesa-dev mercurial autoconf automake groff
libtool xterm
```

4. Install software to make bootable image using the command given below:

```
$ sudo apt-get install uboot-mkimage
```

5. Also, since the host machine runs 64 bit version of OS, we also need to install:

```
$ sudo apt-get install ia32-libs
```

II. *Linux Setup*

The source code package (linux-socfpga-13.02-src.bsx) was downloaded from Altera's website and the following steps were taken.First we need to update the package manager using the following command:

1. The default install location is /opt/altera-linux. To install the package, enter the following command:

```
$ sudo linux-socfpga-13.02-src.bsx /opt/altera-linux
```

2. Update the package manager using the following command:

```
$ sudo /opt/altera-linux/bin/install_altera_socfpga_src.sh ~/yocto- 13.02
```

3. Run the script provided by Altera to set up the build variables needed to compile the Linux kernel:

```
$ sudo source altera-init ~/yocto-13.02/build
```

4. Build u-boot:

```
$ sudo bitbake u-boot
```

5. Build Linux kernel:

```
$ sudo bitbake linux-altera
```

6. Build root filesystem:

```
$ sudo bitbake altera-image
```

7. The images are generated in ~/build/tmp/deploy/images

III. *Linux Testing*

To boot the Linux images on SoC FPGA development kit, we can write the images we just built with Yocto into one of the three Flash devices: SDMMC, NAND and QSPI. We will use SDMMC due to its easy detachability. For SDMMC boot, all boot images will be located inside SDMMC card. A script is provided with the release that will create an SD card image, ready to be deployed.

The provided tool, named make_sdimage.sh, will create a 2GB SD card image, with three partitions:

Partition 1 (FAT)	**Partition 2 (ext3)**	**Partition 3 (NONAME)**
Linux kernel and device tree	The Linux root file system	• Partition used by the SoCFPGA to load the preloader. • Also contains u-boot image

Following command is entered to make the image:

```
$ sudo    make_sdimage.sh \
        -k uImage,socfpga.dtb \
        -p u-boot-spl-socfpga_cyclone5.bin \
        -b u-boot-socfpga_cyclone5.img \
        -r fs \
        -o sd_image.bin
```

Where:

- -k accepts a comma separated list of files. Here, we show the kernel and the device tree blob.
- -p the preloader raw binary, as generated by Yocto or the U-Boot Makefile
- -b the bootloader image, as generated by Yocto or the U-Boot Makefile
- -r the directory where the file system is located
- -o the image name

C. Building Android

This is the next step which involves placing the already build kernel into the working directory of the android source code and then building the Android OS image. Android Open Source Project (AOSP) has several branches, repositories (repos) and contributors. This is the reason they use a script called Repo to manage all git repositories in context of Android. [1]

I. Installing Repo

1. Create a directory called bin in home folder by entering the following command:

```
$ sudo mkdir ~/bin
```

2. Add the newly created directory to PATH using the following command:

```
$ sudo PATH=~/bin:$PATH
```

3. Download curl. Curl downloads files from servers using the http link provided. This is done by using the following command:

```
$ sudo apt-get install libcurl3 php5-curl
```

4. Download the Repo tool:

```
$ sudo curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
```

5. Change the downloaded script to executable:

```
$ sudo chmod a+x ~/bin/repo
```

II. Downloading Source

1. Create an empty directory called WORKING_DIRECTORY in home folder by entering the following command:

```
$ sudo mkdir WORKING_DIRECTORY
```

2. Install git (version control tool) by using the following command:

```
$ sudo apt-get install git-all
```

3. Configure git by putting user name and email:

```
$ git config --global user.name "Muhammad Obaidullah"
$ git config --global user.email "mobaidullah@ryerson.ca"
```

4. Initialize repo in the WORKING_DIRECTORY by:

```
$ sudo repo init -u https://android.googlesource.com/platform/manifest
```

5. Checkout, sync, and download the source from Google's repository by using the following command:

```
$ sudo repo sync
```

III. Building Android

1. Synchronize:

```
$ repo sync -j24
```

2. Setup environment:

```
$ . ./build/envsetup.sh
```

3. Setup new device:

```
$ lunch addcombo ninja-userdebug
```

4. Choose device by using lunch:

```
$ lunch ninja-userdebug
```

5. Compile:

```
$ make -j32
```

IV. Final SD Card Image

Partition 1 (IMAGE)	**Partition 2 (ROOTFS)**	**Partition 3 (NONAME)**	**Partition 4 (DATA)**
Linux kernel	Android OS APKs eg. apps from Google Play	Uboot	Flexible Space Pictures (.jpg, .png)

V. RESULTS & DISCUSSIONS

I. Dynamic Reconfiguration

Dynamic reconfiguration can be done by using the following steps:

- Load .rbf file on activity create event onto the fpga for acceleration
- Remove .rbf file on activity close event
- Android Init() should configure the fpga with base configuration file and several PRRs (Partial Reconfiguration Regions).

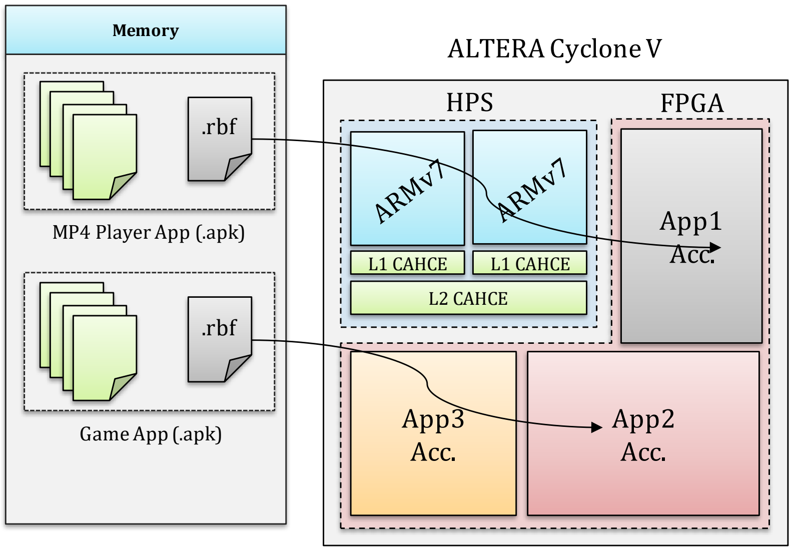


Figure 10. The apk package includes application resources

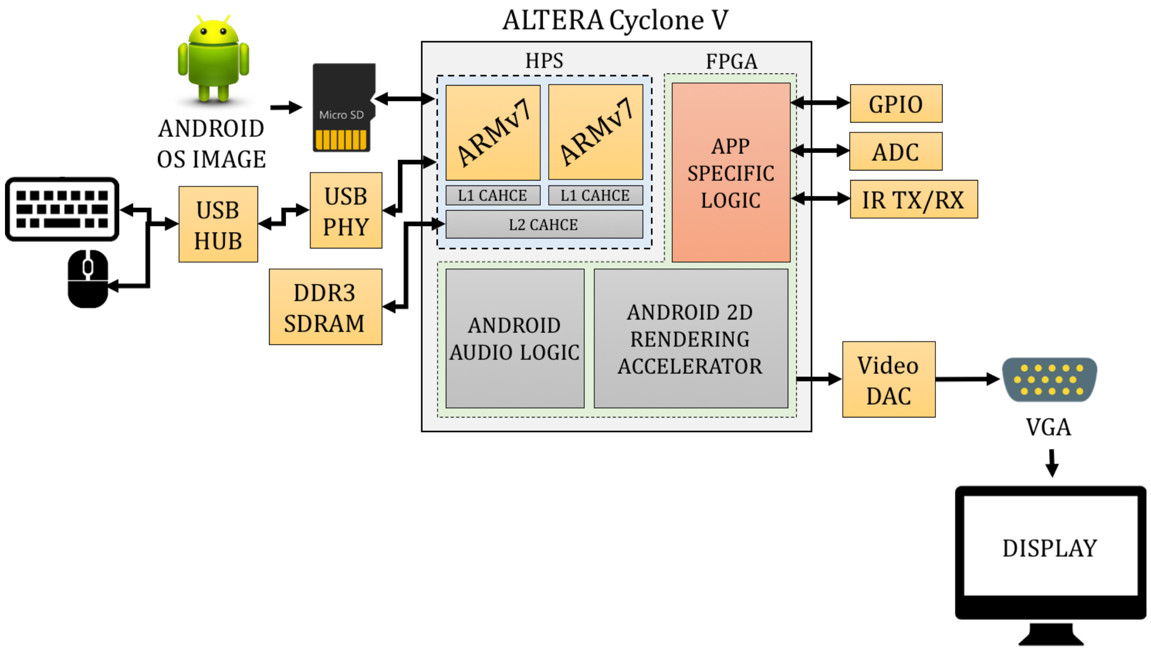
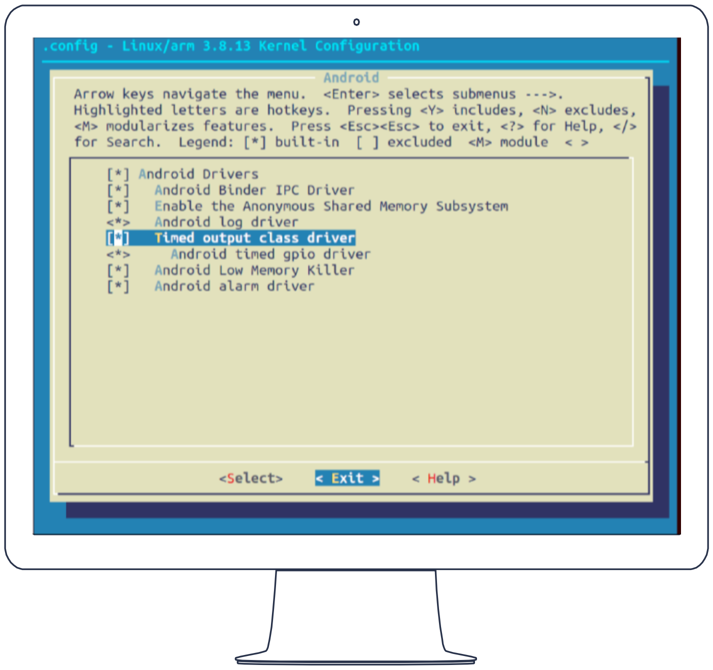


Figure 11. App-based hardware acceleration

II. Android Bindings

Developing your device drivers is similar to developing a typical Linux device driver. Android uses a version of the Linux kernel with a few special additions such as wake locks (a memory management system that is more aggressive in preserving memory), the Binder IPC driver, and other features important for a mobile embedded platform. These additions are primarily for system functionality and do not affect driver development.

We can use any version of the kernel as long as it supports the required features (such as the binder driver).



```
.config - Linux/arm 3.8.13 Kernel Configuration
                              Android
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >

      [*] Android Drivers
      [*]   Android Binder IPC Driver
      [*]   Enable the Anonymous Shared Memory Subsystem
      <*>   Android log driver
      [*]   Timed output class driver
      <*>     Android timed gpio driver
      [*]   Android Low Memory Killer
      [*]   Android alarm driver

                  <Select>    < Exit >    < Help >
```

Figure 11. Configuring Linux kernel to have android drivers.

III. Setting up Android Boot Sequence

When the variable PRE_BUILT_KERNEL is provided to the android build source, the android is image is generated which is based on that particular kernel. This can be seen by going into Settings > About and checking for kernel version.

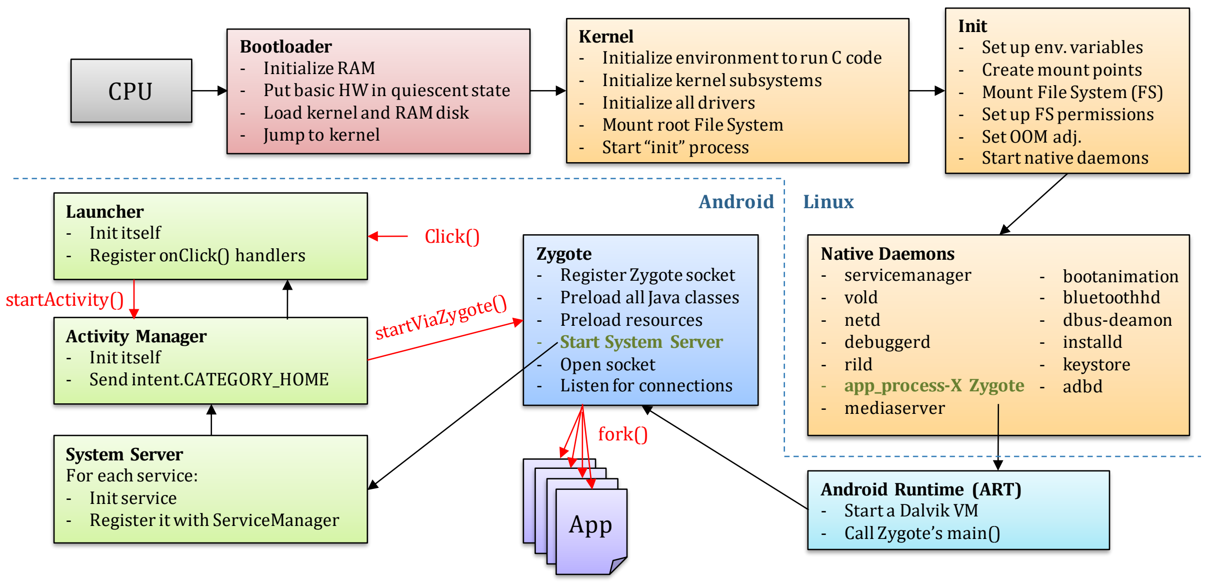


Figure 11. Android boot sequence.

I. *Developing Android Application*

A simple android application was made which wrote to the device files and turned LEDs ON or OFF. It also implements **multi-threading and mutex-es** in order to access device files. Since multiple access to device files at the same time by multiple threads can lead to wrong values and the app to crash, a mutexes were also used where appropriate to block critical areas of threads.

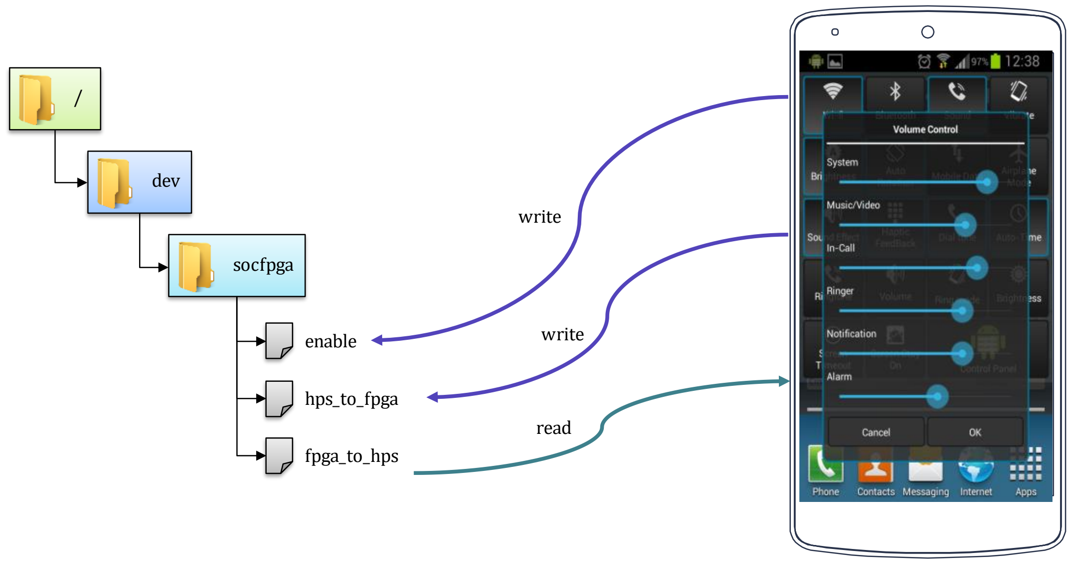
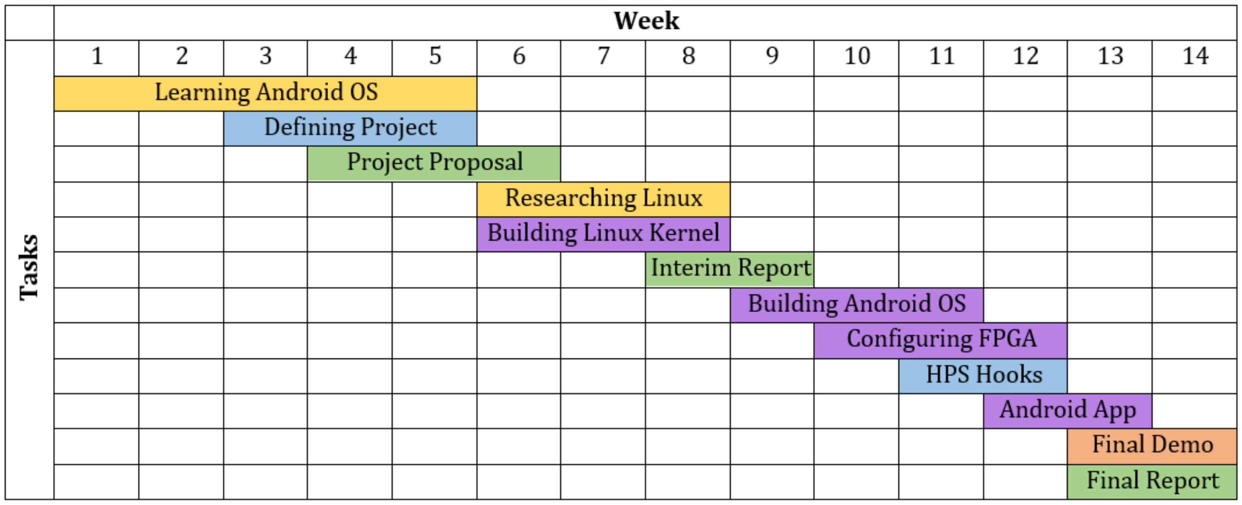


Figure 11. Android application accessing FPGA hardware by use of device files (/dev).

VI. PRJOECT TIMELINE



Legend:

🟨 Study or Research
🟩 Submission
🟦 Writing or Design
🟪 Coding
🟧 Demo

VII. CONCLUSION

Porting android stack to a HPS-FPGA system allows apps to take advantage of FPGA fabric and perform partial reconfiguration to build and tear-down application specific accelerators on the remaining FPGA fabric. Since Cyclone V is not designed to run Android by default, graphic rendering engine and audio logic needs to be implemented in the FPGA. The left-over LEs can be used to implement any app-specific logic accelerator.

Since, audio and video logic should not be touched while the Android OS is running, partial reconfiguration of the FPGA fabric needs to be done. Cyclone V can be dynamically partially reconfigured using Partial Masked SRAM Object File (.pmsf) and Raw Binary File for Partial Reconfiguration (.rbf). Each app can load its own .rbf file from SD card onto FPGA. However, there should be certain checks in place in order to catch and avoid short-circuits and other common I/O errors used by loaded logic and logic to be loaded.

"People who are really serious about software should make their own hardware."

~ Alan Kay (Computer Scientist)

ACKNOWLEDGMENTS

The authors of this work would like to acknowledge the support and funding provided by the Ontario Graduate Scholarship (OGS) program and Ryerson University FEAS.

REFERENCES

[1] Google Inc., "Android Open Source Project," Google, 23 August 2016. [Online]. Available: https://source.android.com. [Accessed 9 November 2016].

[2] Intel, "Implementation of an Android™ Operating System on an Altera SoC," Intel, 8 January 2014. [Online]. Available: https://youtu.be/zHqS_yWiMNI. [Accessed 9 November 2016].

[3] "Graphics Accelerator for Android," FUJISOFT INCORPORATED, 15 November 2013. [Online]. Available: https://www.fsi-embedded.jp/e/_emb/gaforandroid_e/. [Accessed 9 November 2016].

[4] M. Daum, "Android for DE1-SoC Board," RocketBoards.org, 6 October 2016. [Online]. Available: https://rocketboards.org/foswiki/view/Projects/AndroidForDE1SoCBoard. [Accessed 9 November 2016].

[5] Altera, "Altera SoC Linux Intro Workshop," 2016. [Online]. Available: https://rocketboards.org/foswiki/pub/Documentation/WS2LinuxKernelIntroductionForAlteraSoCDevices/WS_2_Linux_Kernel_Introduction_Workshop.pdf. [Accessed 28 November 2016].

APPENDICES