Android Port to Zedboard

Electrical Engineering

Indian Institute Of Technlogy, Bombay

Balraj Parmar - 14D070001
Meet Udeshi - 14D070007
Prathu Baronia - 14D070046
Sudipto Banerjee - 14D070028

May 9, 2018

Contents

Abstract

To build a bare-bone IoT core system running Android which can be interfaced with a number of sensors and used for a wide variety of applications. IoT sensor interfacing is done by the FPGA, and all data processing and communication can be done by the Android OS running on the integrated ARM-A9 core(s).

Keywords: Android, FPGA, IoT, Zedboard.

0.1 Motivation

Motivation behind using Android OS is to expose Android SDK on an IoT platform so that android apps can be developed for data processing. Currently there are lot of apps in the market which utilize multiple sensors in phones to use for health monitoring, location tracking, gaming applications using a common API. Exposing this API to a much wider range of sensors can help develop better IoT applications.

Choice between Android and Generic Linux Distributions: This choice obviously depends on certain technical factors such as :-

- Larger Community support for Android since it is the most widely used OS now leaving behind Windows.
- Wider inbuilt driver support in Android in the context of IoT applications such as touch screen support, battery driver support whereas linux lacks a touch screen support and uses third party softwares such as APM or ACPI.
- Android also offers better UI support for user applications.
- Android already has a built in telephony stack that includes GSM, CDMA, LTE in addition to providing VoIP calling integrated into its suite.
- Linux supports many more processor architectures than Android.
- Since prebuilt libraries and stacks are available for Android the Time to Market for Android is expected to be way less than Linux.

FPGA along with being an interface to the sensors can also perform preprocessing on data through dedicated codecs if required.

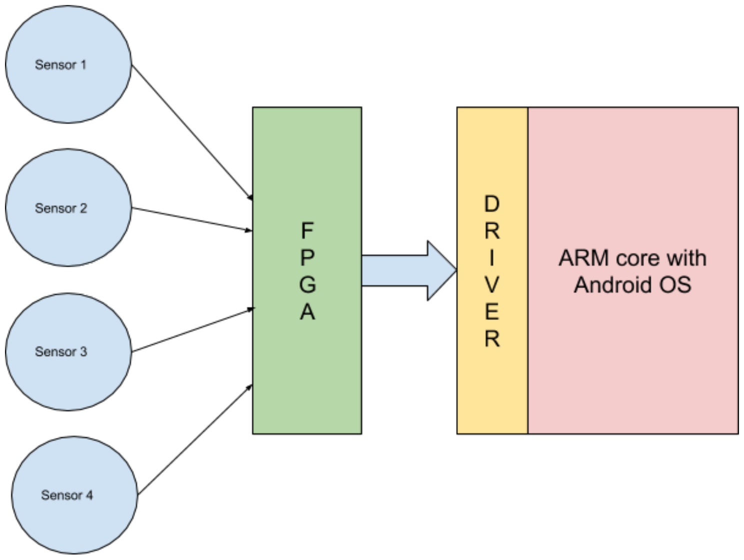


Figure 1: Block diagram

Chapter 1

FSBL and Uboot

Table of contents

The process of loading an OS on Zynq starts of with the internal boot ROM being loaded. It setups the system and executes the so-called First Stage Boot Loader (FSBL) dependent on the jumper configuration. The FSBL initializes the hardware as configured by the developer (ps7_init), performs sanity checks on all system peripherals and executes any provided program. In our case, this is U-Boot in the role of a primary bootloader, which is responsible for loading and booting the Linux kernel.

Fetching Sources

All the below git repos need to be cloned in order to build uboot and dtc.

- The u-boot bootloader with Xilinx patches and drivers
- Device Tree generator plugin for xsdk
- Device Tree compiler (required to build U-Boot

FSBL Building

For building FSBL we have two methods :

- Using Xilinx SDK GUI
- Using hsi command line tool

I followed the first one since it is more descriptive for first time users. The built FSBL needed to be compiled to .bif format and was then needed to be compiled along with Uboot executables into a boot.bin file with the bootgen cmd tool provided in Xilinx SDK, from this boot.bin zedboard was able to boot.

Uboot Building

Instructions from Xilinx's Uboot build guide can be roughly followed step by step to build uboot with some extra things were needed to be taken care of:

- The ARM Cross Compiler Toolchain needs to be installed to overcome the asm-offset error faced during the buid.

- gcc6 has been made a prerequisite by the makers which is actually a serious issue since it is in the experimental stage. The build fails if we try to compile it with any gcc below version 6 linked to the path. Even after installing gcc6 and soft linking it to gcc, the checkgcc6 error didn't get resolved. Hence Iedited the config.mk file in /arch/arm direcotry within the cloned uboot git directory and changed the checkgcc6 function to check gcc for version 4 and was able to build the uboot executable.
- Link to original solution post

Device Tree Compiler Building

- Step by step guide on Xilinx's dtc build guide can be followed with a small edit.
- Instead of device-tree-overlay package we need to install device-tree-compiler package which is different from steps mentioned in the link .
- dtc compile process can only be done after uboot compilation since dtc complation requires the output of the first one.

Arch Linux on Zedboard

- Booted ARM Arch Linux on zedboard succesfully
- Edited the sdboot parameter with correct boot addresses to boot it successfully after initial boot failed.
- Link to the project Zedboard Arch Linux Project

1.1 References

FSBL Build Guide from Xilinx
Great Tutorial from ReconOS
Digilent Uboot Git
User Uboot git

Chapter 2

Andoid 5 Build

Zedroid OS installation steps

Note: the installation procedure uses a lot of custom scripts, which have been written for an Ubuntu host system. The scripts are no more than a list of shell commands, and can be manually executed accordingly on any other host OS.

- All scripts have been added to the `scripts/` folder. Make sure you are in correct folder before running (`~/zedroid` by default).
- Whenever a file has to be edited, there is a final version of that file in `scripts/` folder, which can be referred to in case of doubt. Please manually edit all source files, and don't copy the provided files.
- If you want to follow whole procedure in another folder, copy `scripts/` folder there and follow installation from that folder only.

1. Install dependencies

Run:

```
$ scripts/install-deps.sh
```

Installs required libraries and tools, JDK8, repo. You can use it on another host OS by changing `apt-get` to the relevant tool. Some package names may not match, and will have to be manually checked.

2. Fetch sources

Run:

```
$ scripts/fetch-sources.sh
```

Fetches all sources for Android 5. It will take a long time, around 25-30GB of source is fetched. The sources are stored in `android_sources/` folder.

If this command fails, run:

```
cd android_source
repo sync
```

3. Modify source files

Change dir to `android_source` before editing any file. All file paths are relative to that folder.

3.1 Engineering build

- Open `device/xilinx/zedboard/vendorsetup.sh'
- Add these lines at start of file (after comment block)

```
# Engineering build config
add_lunch_combo zedboard-eng
```

3.2 Enable ethernet in init.rc

- Open file `device/xilinx/zedboard/init.xilinxzynqplatform.rc`
- Add these lines at the very end of the file

```
service ethernet /eth0init.sh
    class main
    user root
    group root
    oneshot
```

3.3 Add eth0init.sh to sources

- You will find `eth0init.sh` in `scripts/` folder
- Copy `eth0init.sh` to `device/xilinx/zedboard/`
- Open `device/xilinx/zedboard/device.mk`
- Add following line to the end of the file

```
PRODUCT_COPY_FILES += device/xilinx/zedboard/eth0init.sh:root/eth0init.sh
```

4. Make Android 5 OS

Make sure you are in `android_source` directory.

Run:

```
$ source ./build/envsetup.sh
$ lunch zedboard-eng
$ make -k 2> make-errors.log | tee make.log
```

Some components of the OS may fail to build. But in that case too, we can obtain a working version of the OS. For that `-k` flag is used to continue making even with errors.

To see what failed, see `make-errors.log` file.

Troubleshoot: if errors are regarding syntax errors in python scripts, make sure the `python` *executable in your PATH is* `python2` *and not* `python3`*. Check output of* `python -V` *to make sure, and relink* `/usr/bin/python` *to correct version of python. Then run* `make` *again.*

4.1 Verify changes propagated properly

Change to `android_source` directory

- In directory `out/target/product/zedboard/`, you should see 2 files
  - uImage
  - ramdisk.img
- In same directory, there should be `system/` and `data/` directories containing android root.
- There should also be 'root/' directory which contains ramdisk root. Verify that the changes you made to `init.xilinxzynqplatform.rc` are there and `eth0init.sh` exists.
- If it doesn't exist you have to discard ramdisk.image and repack manually. Else, skip the next section

4.2 Repack ramdisk manually

Change to `android_source/out/target/product/zedboard/' directory

Make the changes said in (3.2) and (3.3) in the `root/` directory directly

Run: from `zedboard/` directory

```
$ cd root && find . | cpio -o -H newc | gzip > ../ramdisk.img && cd ..
```

4.3 Adding uboot header to ramdisk

After obtaining proper `ramdisk.image`

Run:

```
$ mkimage -A arm -O linux -T ramdisk -C none \
    -a 0x02000000 \
    -n 'Zedroid ramdisk' \
    -d ramdisk.img uramdisk.image.gz
```

5. Preparing boot medium

You will need an SD-Card of atleast 2GB, 4GB is preferable.

You need to create 4 partitions as listed in the same order with same partition type. Using GParted is most suitable.

- partition 1: FAT32 - 500MB - Label="boot"
- partition 2: EXT4 - 500MB - Label="system"
- partition 3: EXT4 - use remaining space on card, minimum 500MB - Label="data"
- partition 4: SWAP - 500MB - Label="swap"

It is important to keep order and type of partitions as mentioned above. The mounting of these is hardcoded into u-boot and boot will fail otherwise.

After successfully creating partitions on SD-Card, we will copy over the OS files.

Mount on host system

Make sure you are in `zedroid/` folder (parent of `android_source/`)

Also check which device is SD-Card by running `lsblk`. In instructions `/dev/sdX` would stand for SD-Card. It is generally `/dev/sdb`

Run:

```
mkdir -vp mnt/{boot,system,data}
sudo mount /dev/sdX1 mnt/boot
sudo mount /dev/sdX2 mnt/system
sudo mount /dev/sdX3 mnt/data
```

Copy boot binaries

In `scripts/` folder you will find `boot.bin` and `devicetree.dtb`.

For all these operations you will have to use `sudo cp` command to copy

- In `mnt/boot/`, copy
  - boot.bin
  - devicetree.dtb
  - uImage
  - uramdisk.image.gz
- In `mnt/system/`, copy
  - All contents of `android_source/out/target/product/zedboard/system/` directory
- In `mnt/data/`, copy
  - All contents of `android_source/out/target/product/zedboard/data/` directory

Then run:

```
sudo umount mnt/{boot,system,data}
```

It may take a while to unmount all the partitions.

You now have a ready-to-boot SD card with Android 5 OS.

6. Booting up

- Insert SD-Card into Zedboard and power-up.

- Connect to USB-UART port on the zedboard.
- Start serial terminal like `minicom` using 115200 baud 8N1 configuration (generally default). Running `minicom -D /dev/ttyACM0 -w` or respective device name should work. `-w` flag is needed for line-wrap to work.

You should either boot directly into Android and see linux terminal, or `zynq-uboot>` terminal of u-boot.

6.1 Properly configuring U-Boot

If you are stuck at u-boot terminal, then follow these steps to boot into Android.

```
zynq-uboot> env print sdboot
zynq-uboot> env set sdboot "mmcinfo && fatload mmc 0 0x3000000 ${kernel_image} &&
fatload mmc 0 0x2000000 ${ramdisk_image} && fatload mmc 0 0x2A00000
${devicetree_image} && bootm 0x3000000 0x2000000 0x2A00000"
zynq-uboot> env save
```

Now close minicom, and restart the Zedboard. You should boot into Android.

6.2 Connecting to ADB

Once you have boot into android and have access to the shell via minicom, connect ethernet and run `ip addr`.

- If the status if `eth0` is `DOWN` then ethernet wasn't correctly enabled.
- run `cat /init.xilinxzynqplatform.rc` and see if the lines added are present.
- Also verify that `eth0init.sh` is there and correct.
- run `dmesg | grep eth` to see if ethernet service was started, and if there were any errors.

You will have to recheck the ramdisk image if new changes are not present.

Once you have a proper ramdisk image, you will see that `eth0` is `UP` at boot and also be able to know IP. For easier use, you can also scan IP from your host using `nmap -p5555 192.168.1.1/24` or equivalent IP subnet. Port 5555 is used by ADB so whichever host shows it "open" is zedboard.

You can now connect to zedboard root shell via

```
$ adb connect <ip-of-zedboard>
$ adb shell
```

Chapter 3

HDMI Core

Display of Android 5 screen :

The audio and the video interface for Android 5 display has been done using the ADV7511 which is a part of the Zynq evaluation board. The ADV7511 is a 225 MHz High-Definition Interface transmitter. The video uses a 16bit YCbCr interface (except VC707 which uses 36bit 444 RGB interface) and the audio uses a single bit SPDIF interface.

It is DVI 1.0-compatible. Since SPDIF is supported, it can carry compressed audio including Dolby Digital, DTA and THX. There is an independent DPDIF input and output.

The HDMI transmitter has 25 connections to Bank 35 (3.3V) of the Zynq-7000 AP SoC.

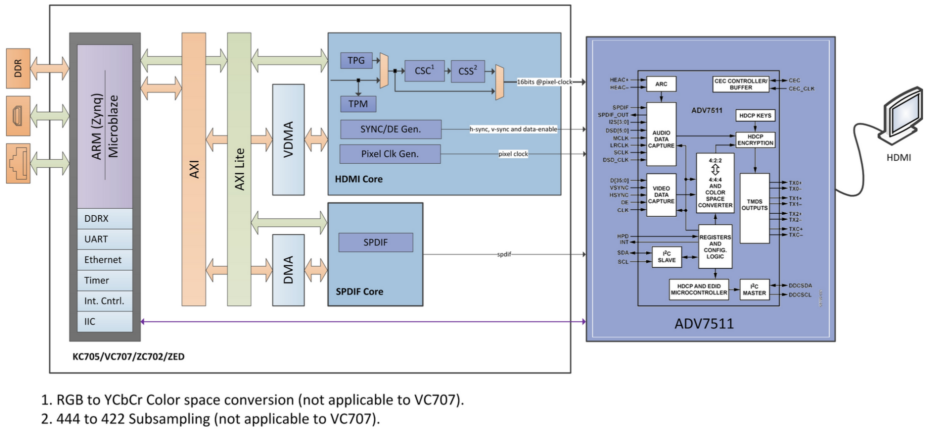
The needed software is Xilinx and a UART terminal (Tera Term/Hyperterminal) of baud rate 115200 and the required hardware is an HDMI Monitor and the appropraite Zedboard (AC701/KC705/VC707/ZC702/ZC706/Zed board).
Now, we go in depth with the explanation of how the video and audio interfacing is done.

1. The video part consists of a Xilinx VDMA interface and the ADV7511 video interface. The ADV7511 interface consists of a 16bit YCbCr 422 with separate synchorinzation signals. The VDMA streams frame data to this core. The internal buffers of this pcore are small (1k) and do NOT buffer any frames as such. Additional resources may cause loss of synchronization due to DDR bandwidth requirements. The video core is capable of supporting any formats through a set of parameter registers (given below). The pixel clock is generated internal to the device and must be configured for the correct pixel frequency. It also allows a programmable color pattern for debug purposes. A zero to one transition on the enable bits trigger the corresponding action for HDMI enable and color pattern enable.

2. The audio part consists of a Xilinx DMA interface and the ADV7511 spdif audio interface. The audio clock is derived from the bus clock.

A programmable register (see below) controls the division factor. The audio data is read from the DDR as two 16bit words for the left and right channels. It is then transmitted on the SPDIF frame. The sample frequency and format may be controlled using the registers below. The reference design defaults to 48KHz.

The interfacing is shown in detailed labeled block diagram as given below :



1. RGB to YCbCr Color space conversion (not applicable to VC707).
2. 444 to 422 Subsampling (not applicable to VC707).

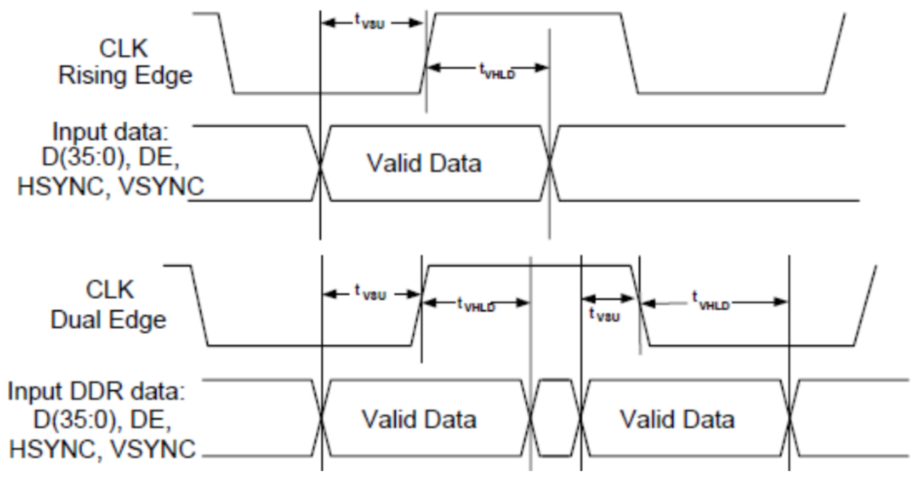
Figure 3.1: Block Diagram of HDMI transmitter ADV7511

The video mode is 1080p by default (the pixel frequency for 1080p is 148.5MHz) and can be changed by the user by programming the following registers :

1. HSYNC count: is the total horizontal pixel clocks of the video, for 1080p this is 2200.
2. HSYNC width: is the pulse width in pixel clocks, for 1080p this is 44.
3. HSYNC DE Minimum: is the number of pixel clocks for the start of active video and is the sum of horizontal sync width and back porch, for 1080p this is 192 (44 + 148).
4. HSYNC DE Maximum: is the number of pixel clocks for the end of active video and is the sum of horizontal sync width, back porch and the active video count, for 1080p this is 2112 (44 + 148 + 1920).
5. VSYNC count: is the total vertical pixel clocks of the video, for 1080p this is 1125.
6. VSYNC width: is the pulse width in pixel clocks, for 1080p this is 5.
7. VSYNC DE Minimum: is the number of pixel clocks for the start of active video and is the sum of vertical sync width and back porch, for 1080p this is 41 (5 + 36).

8. VSYNC DE Maximum: is the number of pixel clocks for the end of active video and is the sum of vertical sync width, back porch and the active video count, for 1080p this is 1121 (5 + 36 + 1080).

The HDMI Video Interface Timing is given in the below diagram.



The reference design reads 24bits of RGB data from DDR and performs color space conversion (RGB to YCbCr) and down sampling (444 to 422). If bypassed, the lower 16bits of DDR data is passed to the HDMI interface as it is. A color pattern register provides a quick check of any RGB values on the monitor. If enabled, the register data is used as the pixel data for the entire frame.

To run program in the HDMI monitor, we connect an HDMI cable between the board HDMI out and the HDMI in of the monitor.

Chapter 4

GPIO Core

Tutorial:
Importing a custom IP into Vivado and getting started in Xilinx SDK.
At the end of this tutorial we will have:

- Imported and implemented a custom DigiLEDs IP block into the design.
- Created .C Project in Xilinx Vivado SDK to interface with the Zedboard.

Prerequisites:

Hardware:
1. Digilent Zedboard Board
2. Micro USB Cable: Used for UART communication and JTAG programming
3. Programmable RGB LEDs(WS2812, Neopixels): Data-in Signal wire connected to Zedboard's JB1

Software:
1. Xilinx Vivado 2016.2 with the SDK package(Follow this to how to install and activate Vivado)
2. Zedboard Support Files
    - These files will describe GPIO interfaces on your board and make it easier to select yout FPGA board and add GPIO IP blocks.
    - Follow this guide on how to install Board Support Files for Vivado.

3. Project Files:
    DigiLED Custom IP
    main.c file

Summary of steps:
I. Vivado

- Open Vivado
- Create a new block design
- Add the Zynq core IP and automate it
- Add the DigiLEDs custom IP to the project's IP repository
- Add the DigiLEDs IP to the design and configure it.
- Validate and save block design
- Create HDL system wrapper
- Run design Synthesis and Implementation
- Generate Bit File
- Export Hardware Design including the generated bit stream file to SDK tool
- Launch SDK

Now the Hardware design is exported to the SDK tool. The Vivado to SDK hand-off is done internally through Vivado. We will use SDK to create a Software application that will use the customized board interface data and FPGA hardware configuration by importing the hardware design information from Vivado.

II. SDK

- Create new application project and select Empty Application template
- Import main.c
- Program FPGA

Follow this for detailed step