Självständigt arbete på grundnivå

*Independent degree project – first cycle*

*Electronics Design 15 credits*

*Evaluation of open source IP based Embedded System with Linux*

*Jiayi Wang*
Abstract

Embedded system plays an important role in various industry applications. An embedded system is consisting of software and hardware. The hardware platform of conventional embedded system is typically based on IC chips that have fixed resources. Besides, with the development of FPGA, an emerging approach for designing embedded system is implementing soft IP cores on FPGAs. Soft IP cores are synthesizable hardware blocks described in HDL language. Their source code can be either open or close to public. For example, OpenRISC 1200, is an open source 32-bit RISC microprocessor. In addition, the increasing complexity of embedded system forces software developers to consider operating system support to reduce their workload. Thus, in this thesis, a prototype of open source IP based embedded system with Linux is implemented on Atlys (Xilinx Spartan-6) FPGA board and the goal is to evaluate if the system is appropriate for industrial applications. The hardware platform is ORPSOC, which is a reference SoC design based on OpenRISC 1200 processor. For software, Linux operating system is installed. Furthermore, an application executes on Linux is developed that reads the output of an I2C compass sensor-LSM303DLM. With the success of the application and the investigation of license issues, the conclusion is drawn that open source IP based embedded system with Linux is usable for industry. Although comparing to conventional embedded system, the open source IP based embedded system with Linux has following cons, such as high product cost, basic-supported development environment and more difficult software development if Linux driver doesn’t support the hardware. However, its pros are high flexibility and scalability, high software portability, low software development difficulty and high reusability that make it more suitable for industry usage.

Keywords: embedded system, open source hardware, OpenRISC 1200, Linux
Acknowledgements

First, I would like to thank all of staffs in Mid-Sweden University and Beijing Information Science and Technology University that provide me the opportunity to become an exchange student in electronics design. Second, I am thankful to Xiaozhou Meng, with his patient guidance, this project is successful and I learned the knowledge of FPGA, open source IP cores and Linux operating system. Third, I wish to thank Benny Thörnberg for his careful examination of the thesis work.

Furthermore, I would like to thank Dingding Sun, my friend and also the classmate in my home university. We usually come together to discuss problems and exchange ideas of our study. And I am grateful to all the members in open source community for their warm assistance. I am inspired with their spirit for sharing ideas and experience to others.

At last, I’m greatly thankful to my parents and grandparents for their endless support of my study.
7.3. Pros and cons ................................................................. 24

8. Conclusion ............................................................................. 27

9. Future work ........................................................................... 29

10. Reference ............................................................................ 31

Appendix A: Detailed tutorial for implementing open source IP based embedded system with Linux ................................................................. 33

Appendix B: Configuration file for TFPT server ...................................................... 41

Appendix C: Device Tree .................................................................. 43

Appendix D: Source code of sensor reading application ................................. 45
List of Figures

Figure 1.1-1 Architecture of conventional embedded system ..................................................1
Figure 1.1-2 Architecture of conventional embedded system with Linux.................................2
Figure 3.2-1 OpenRISC 1200 CPU architecture .........................................................................8
Figure 3.3-1 user space and kernel space$^{[13]}$ ...........................................................................9
Figure 5.1-1 Prototype of open source IP based embedded system with Linux .....................13
Figure 5.2-1 PMOD and I2C signal assignment $^{[16]}$ .................................................................15
Figure 5.2-2 Block diagram of ORPSOC with I2C controller enabled ........................................15
Figure 5.2-3 Network is working .............................................................................................17
Figure 5.2-4 i2c-0 under "/dev" ...................................................................................................18
Figure 5.2-5 Flow chart of sensor reading application ............................................................19
Figure 5.2-6 Sensor reading result ..........................................................................................19
Figure 7.3-1 Schematic of high software portability .................................................................25
List of Tables

Table 5.2-1 Comparison of 3 different ways for installing GNU tool-chain ......................... 14
Table 5.2-2 Network parameters of u-boot and host computer ............................................. 16
Table 6.2-1 License investigation of each component ............................................................ 21
Table 6.3-1 A comparison between open source IP based embedded system with Linux and conventional embedded systems................................................................. 22
Terminology

EDK: Embedded development kit
FPGA: Field Programmable Gate Array
GDB: GNU Project Debugger
GPL: GNU General Public License
IC: Integrated Circuit
I2C: Inter-Integrated Circuit
IDE: Integrated development environment
IP: Intellectual property
JTAG: Joint Test Action Group
LAN: Local area network
LGPL: Lesser GNU General Public License
NFS: Network File System
ORPSOC: OpenRISC Reference Platform System-on-Chip
RTL: Register-transfer level
SOC: System-on-chip
SPI: Serial Peripheral Interface
TFTP: Trivial File Transfer Protocol
UART: Universal Asynchronous Receiver/Transmitter
UCF: User Constraint File
U-boot: Universal bootloader
1. Introduction

1.1. Background and motivation

Nowadays, embedded system plays a more and more important role in the world. It can be found in many applications for industrial use, such as process control and monitoring systems. The task for an embedded system is typically to control the machine to function correctly. It can be said that the industrial productivity is improved with introducing embedded system.

Diffrs from general purpose computer, an embedded system is designed for a dedicated task, which requires both hardware and software operate appropriately.

Normally, the embedded system designed in conventional way is consisting of 3 layers as shown in Figure 1.1-1. The hardware platform is a PCB board containing a microcontroller and other peripheral ICs such as on-board memory, Ethernet module and etc. The software is the application code of the desired task including hardware drivers.

![Microcontroller and peripherals on a PCB board](image)

Figure 1.1-1 Architecture of conventional embedded system

With the rapid development of semiconductor technology, the capability of embedded system is increasing over time to allow more complex task to be performed. Thus, the embedded operating system support becomes necessary. Figure 1.1-2 illustrates the architecture of a conventional embedded system with Linux. The benefits of introducing Linux are providing various hardware drivers, communication protocols and management of system resources that reduce the workload of software developer significantly.
In addition, FPGA (Field Programmable Gate Array) is an alternative for embedded design. It is well known as its hardware re-programmable feature. With this advantage, an emerging technology for achieving embedded system’s hardware platform is using soft IP cores. The term IP refers to intellectual property. Soft IP cores are the synthesizable hardware blocks for FPGAs and described in RTL level. A soft IP can be either a processor or other hardware modules such as UART, ETHERNET and I2C controller. Designing embedded system with soft IP cores is flexible and configurable. At present, there are two leading-edge commercial soft processors, Microblaze and Nios II, supported by Xilinx and ALTERA. Besides, there are other open source processors, such as OpenRISC and LEON3. Open IP cores inherit the advantages of soft IPs and are open source to public, which means they are free, and FPGA independent.

Therefore, it is meaningful to implement an open source IP based embedded system with Linux to evaluate if it is compatible for industrial applications.

In this thesis, the hardware platform is ORPSOC\textsuperscript{[1]} (OpenRISC Reference Platform System-On-Chip), which is a reference embedded system design based on OpenRISC soft processor. Then the Linux operating system is installed and an application running on Linux that reads the acceleration raw data from an I2C compass sensor, LSM303DLM.

1.2. Overall aim

The thesis’s overall aim is to prove usability of open source IP based embedded system with Linux for industry applications. Then check out if the implementation is valid by investigating license issues. Furthermore, comparing to conventional embedded system, pros and cons of open source IP based embedded system with Linux should be assessed to determine which one is more suitable for industrial usage.
1.3. Concrete goals

- Implement the desired embedded system as a development platform made up of ORPSOC (OpenRISC Reference Platform System-On-Chip) and Linux on Atlys FPGA board.
- Develop an I2C sensor reading application that runs on Linux to prove usability.
- Investigate license issues.
- Compare to conventional embedded system from following aspects:
  - Product cost,
  - Development environment,
  - Flexibility and scalability,
  - Software portability,
  - Software development difficulty,
  - Reusability

1.4. Outline

Chapter 2 provides a summary of related works.
Chapter 3 describes the related theory about open hardware, OpenRISC processor, Linux user space and kernel space.
Chapter 4 is methodology part, which is the approach to achieve goals of this thesis.
Chapter 5 describes implementation steps.
Chapter 6 presents the gained result covering usability, license issues, pros and cons.
Chapter 7 is the discussion of results.
Chapter 8 is the conclusion for this thesis.
Chapter 9 plans the future work.
Chapter 10 lists the references of this paper.
Appendix A shows a complete tutorial for how to implement open source IP based embedded system with Linux
Appendix B presents the configuration file for TFTP server
Appendix C is the device tree for Linux
Appendix D is the source code of sensor reading application
2. Related works

Jason G, Anderson and Khalid present a survey of soft-core processors that are used in embedded systems, including commercial cores: Nios II, Microblaze, PicoBlaze, Xtensa and open source cores: LEON3 and OpenRISC 1200\(^{(2)}\). For OpenRISC 1200 soft processor, there are several researches based on it. Younghoon Bin, Kwangmyong Kang, Hyungjun Kim, Hongkyun Jung and Kwangki Ryoo describe the development of synthesizable SoC platform using OpenRISC 1200 processor and WISHBONE On-chip bus\(^{(3)}\). Lihong Lian, Xiaochao Li, Fen Xiao and Donghui Guo developed a gdb-based debugging system for OpenRISC1200 Processor based on IEEE1149.1 JTAG interface\(^{(4)}\). Jiesheng Wei, Ling Wang, Feng Wu, Yibo Chen and Long Ju present a SoC architecture of wireless sensor node using OpenRISC 1200 microprocessor core\(^{(5)}\). Mehdizadeh, Shokrolah-Shirazi and Miremadi analyzed the effects and propagation of faults in open-core 32-bit OpenRISC 1200 processor\(^{(6)}\). FPGA features that is possible to reconfigure a section of the FPGA while the rest is working. Castillo, Huerta, Lopez and Martinez show how a secure self-reconfiguring architecture based on OpenRISC 1200 microprocessor takes advantage of this feature to perform the secure download of the firmware and the hardware needed to run an application\(^{(7)}\). Daroudja, A, Izeboudjen, N, Titri, S, Sahli, L, Louiz, F and Lazib, D describe the hardware/software development of a System on Chip (SoC) platform for Voice over Internet (VoIP) gateway, whose hardware part is based on OpenRISC 1200 processor\(^{(8)}\). Younjin Jung, Ok Kim, Byoungyup Lee, Hongkyun Jung and Kwangki Ryoo designed a SoC platform based on OpenRISC 1200 processor with multi-channel bus architecture reduces the bottleneck of on-chip communication by multi-channels\(^{(9)}\).
3. Theory

3.1. Open source hardware

The commercial soft IP cores such as Microblaze and Nios II are leading a revolution of embedded system design; they are high performance, well supported and flexible. However, they are not open source and FPGA dependent that means the designers only has the right to use them in their embedded design and the implementation of soft IP cores is only possible on specific vendor’s FPGAs.

In software world, open source software is becoming popular. The “open source” means “freedom” that everyone has the right to modify it and derived works are allowed under the condition that the developers should pass on the freedom to others. This causes a great success of open source software such as Linux operating system.

In hardware world, modern silicon chip is typically built from silicon “intellectual property” (IP), written in a hardware description language. Fabless design houses may never produce a chip themselves—one of the largest and best known is ARM in Cambridge, whose processor IP is built by other companies into one billion chips ever month. That IP costs the same amount to produce, whether it goes into one chip or one billion [10].

Due to this property, the hardware is possible to become open source by sharing the synthesizable hardware description language code of soft IP cores to public.

3.2. OpenRISC processor and ORPSOC

OpenRISC[11] CPU architecture, one of the flagship projects of opnecores.org[12], is a well-known open source processor. OpenRISC 1200 processor is an implementation of OpenRISC 1000 processor family. Figure 3.2-1 shows the architecture of OpenRISC 1200 CPU.
The OpenRISC 1200 CPU is a 32-bit scalar RISC with Harvard microarchitecture, 5 stage integer pipeline, virtual memory support (MMU) and basic DSP capabilities. The MMU enables the capability of running an operating system. Supplemental facilities include debug unit for real-time debugging, high-resolution tick timer, programmable interrupt controller and power management support \cite{11}. This processor can be synthesized and downloaded onto Altera and Xilinx FPGAs and supports embedded real time operating systems such as Linux, \mu\text{Linux} and OAR RTEMS real time operating system. For software development, tools are available that allow developers to compile programs written in C/C++, Java and Fortran to run on the OpenRISC processor \cite{2}.

ORPSOC (OpenRISC Reference Platform System-on-Chip) is a reference system-on-chip design that is primarily for OpenRISC based embedded system testing and development. And it is also the hardware platform of this thesis that contains an OpenRISC 1200 processor and several peripherals. The interconnections between CPU and peripherals are using Wishbone interface, which have been configured already. In ORPSOC’s source code, there are makefile scripts to generate FPGA configuration bit stream for particular FPGA boards, such as selected Atlys FPGA board in this thesis.

3.3. Linux user space and kernel space

Linux is a successful open source operating system that is widely used in embedded systems as a platform for executing applications. It manages the resources of an embedded system that the software developer can focus on application code on a high-level abstraction view without being involved in hardware driver development if the embedded system performs a complex task.

In Linux, memory is divided into two spaces, one is user space, and the other one is
kernel space. Figure 3.3-1 shows the relationship between user space and kernel space.

![Figure 3.3-1 user space and kernel space](image)

The top is user space where applications execute, while the kernel space is an exclusive space only for kernel running. The kernel has the highest authority to access all resources in an embedded system, such as memory and devices. And it should be as stable as possible to prevent any undesired errors happening and coordinates processes. By contrast, user space has less authority that it can't access the data in kernel space directly. The data transition between user space and kernel space is via system call interface. Once a user application invokes a system call to access a particular hardware, the kernel handles it with corresponding device driver.

A C library is responsible for implementing system calls. Usually it is GNU C Library (glibc). However, GNU C Library is not compatible for embedded Linux due to it is a large library. To quote from Ulrich Drepper, the maintainer of GNU libc: "...glibc is not the right thing for [an embedded OS]. It is designed as a native library (as opposed to embedded). Many functions (e.g., printf) contain functionality which is not wanted in embedded systems." 24 May 1999. Thus, another C library, µClibc is introduced as shown in Figure 3.3-1. µ means micro while C stands for controller. And µClibc is the short of “the microcontroller C library”. As the name it indicates, the µClibc is intended to support embedded systems that provide as much functionality as possible in a small amount of space [14].
4. Methodology

In this thesis, the usability, license issues, pros and cons of open source IP based embedded system with Linux are investigated via a case study.

The case is to develop an application runs on Linux that reads acceleration raw data from an I2C compass sensor-LSM303DL. The hardware platform of this case is ORPSOC with I2C controller enabled.

The work flow for implementing this case is shown as following:
1. Set up the development environment includes Xilinx ISE for hardware synthesis and GNU tool-chain for software compilation.
2. Generate FPGA configuration bit stream of ORPSOC with I2C controller enabled.
3. Install u-boot.
5. Build Linux image and install it through TFTP protocol.
6. Load the application software via NFS protocol to Linux to test the usability of whole system.

If the application is working, then the usability is proved. License issue is surveyed with interpreting license of each component involved in this thesis. And then check out if the usage of each component offends the rules. Pros and cons are obtained by comparing with the previous knowledge of conventional embedded system.
5. Implementation

5.1. Prototype of open source IP based embedded system with Linux

Figure 5.1-1 shows the setting up of whole system:

1) The whole system is implemented on Atlys FPGA board including a Xilinx Spartan-6 FPGA chip.
2) The USB to JTAG interface is used to program SPI FLASH. The SPI FLASH image contains ORPSOC bit stream for configuring FPGA and binary image of the program that will be executed. In this thesis, the program is u-boot, which is a bootloader for further Linux installation.
3) The interaction interface between user and the board is via UART console.
4) Another host computer acts as TFTP and NFS server. The TFTP server is for transferring Linux image to Atlys board and NFS server is for loading applications to Linux.
5) I2C slave device is a compass sensor, LSM303DLM.

5.2. Work flow

In this section, the steps of implementation are described summarily. A detailed
tutorial for implementing such an embedded system is presented in Appendix A.

5.2.1. Build the development environment

The development environment is under Ubuntu Linux operating system that should contain both GNU tool-chain \(^{[15]}\) for software cross-compilation and Xilinx ISE 14.5 for building the hardware platform.

There are 3 ways to install the GNU tool-chain as following:

- Build the GNU tool-chain from source code
- Install precompiled binaries
- Use a virtual box image

Table 5.2-1 compares the 3 different methods for installing GNU tool-chain. It can be concluded that using the virtual box image is the best way because it not only has a complete library for both bare metal and Linux software development support, but also it is the easiest approach since everything is pre-configured. Bare metal software stands for the software that accesses hardware in register-level directly without any help of operating system. By contrast, building from source code is the most difficult way because there are always unexpected errors happening. Precompiled binaries only support bare metal software development.

<table>
<thead>
<tr>
<th>Methods for installing GNU tool-chain</th>
<th>Newlib (for bare metal software development)</th>
<th>µClibc (for Linux software development)</th>
<th>Installation difficulty</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build from source code</td>
<td>✓</td>
<td>✓</td>
<td>High</td>
</tr>
<tr>
<td>Precompiled binaries</td>
<td>✓</td>
<td>×</td>
<td>Medium</td>
</tr>
<tr>
<td>Virtual box image</td>
<td>✓</td>
<td>✓</td>
<td>Low</td>
</tr>
</tbody>
</table>

Table 5.2-1 Comparison of 3 different ways for installing GNU tool-chain

5.2.2. Generate ORPSOC with I2C controller enabled

In the source code of ORPSOC, there is a specific folder contains RTL source code and makefile scripts for Atlys board.

The I2C controller is a soft IP core that should be added into ORPSOC by connecting it to Wishbone interface. However, it has been connected already and disabled as default in a top-define file “orpsoc-defines.v”, which is for configuring functions of ORPSOC. Thus, uncommenting the code “`define I2C0” will enable the I2C controller in ORPSOC.

Then the next step is assigning the signals of I2C controller to PMOD socket developed by Digilent for peripheral connection. The I2C interface requires 4 wires to communicate properly; they are Vcc, GND, clock line SCL and data line SDA. In
ORPSOC, the names of SCL and SDA signals are “i2c0_scl_io” and “i2c0_sda_io” gained from top-design file “orpsoc_top.v”. Figure 5.2-1 shows the PMOD interface on Atlys board and signal assignment.

![Figure 5.2-1 PMOD and I2C signal assignment](image)

SCL and SDA signals are assigned to Pin3 and Pin4 by editing user constraint file (ucf). The carrier board of I2C slave LSM303DLM has a level-shifter circuit that makes it compatible for PMOD 3.3V system.

At last, the hardware platform ORPSOC is generated with the help of makefile scripts. A bit stream for configuring FPGA can be achieved after the synthesis, mapping, place and route steps are finished.

Figure 5.2-2 shows the block diagram of ORPSOC. The blue blocks are default settings, and the red I2C controller is the one enabled in top-define file.

![Figure 5.2-2 Block diagram of ORPSOC with I2C controller enabled](image)
5.2.3. U-boot

Generally, GDB\textsuperscript{[15]} (GNU Project Debugger) is taking the responsibility of loading and debugging the software. However, the Xinlinx USB Platform Cable couldn’t be detected by GDB. Thus, an alternative should be found to load the software onto the platform.

The approach is u-boot\textsuperscript{[17]}. U-boot refers to universal bootloader, is a primary and powerful bootloader used in embedded systems that eases the procedure of loading Linux image or other application images. The u-boot can only fix the problem of loading software, whereas it can’t be a perfect replacement of GDB because software can hardly debug with u-boot. The installation of u-boot is to generate a SPI FLASH image that contains the FPGA configuration bit stream and the u-boot binary image.

In ORPSOC, there is a ROM module written in Verilog language, which acts like a read-only memory. The ROM includes a bootloader program to load the software that is located in SPI flash. Therefore, the ROM and the bootloader are combined to be a bootrom. After the Atlys board is powered on, the FPGA will be configured and the u-boot is copied from SPI flash to external DDR2 RAM to start execution with the help of bootrom. In addition, a block of 4 bytes should be added to the head of u-boot binary image that informs the bootloader in ROM how many bytes should be copied to RAM.

In order to load Linux image through TFTP (Trivial File Transfer Protocol), the network parameters of u-boot and host computer should be set properly to build a LAN (Local area network) as shown in Table 5.2-2. The Ethernet MAC (Media Access Control) address can be random as long as it is valid if the system is only for experimental use.

<table>
<thead>
<tr>
<th></th>
<th>U-boot</th>
<th>Host computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ethernet MAC address</td>
<td>00:12:34:56:78:9a</td>
<td>Assigned by manufacture</td>
</tr>
<tr>
<td>Local IP address</td>
<td>192.168.2.6</td>
<td>192.168.2.2</td>
</tr>
<tr>
<td>Network mask</td>
<td>255.255.255.0</td>
<td>255.255.255.0</td>
</tr>
<tr>
<td>Gateway IP address</td>
<td>192.168.2.1</td>
<td>192.168.2.1</td>
</tr>
<tr>
<td>Server IP address</td>
<td>192.168.2.2</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5.2-2 Network parameters of u-boot and host computer

Moreover, the network speed of host computer should be configured to 100Mbps or below it because 100Mbps is the maximum connection speed supported by Ethernet MAC IP core in ORPSOC.

If a ping command gets an active response as shown in Figure 5.2-3, then it proves u-boot is installed successfully and host computer is alive.
5.2.4. TFTP and NFS server

TFTP (Trivial File Transfer Protocol) is a simple file transfer protocol without any authentication. Because of its simplicity, it is implemented with less consumption of memory. TFTP server helps transfer Linux images from host computer to the RAM of Atlys board.

The setup procedure of TFTP server is shown below:

1) Install the required packages: tftp, xinetd, tftpd
2) Create a folder for storing images that will be transferred to u-boot. And more important, the folder and the files in it should be configured as everyone can access it.
3) Create a file named “tftp” under “/etc/xinetd.d” path, which is the configuration file for tftp server. The content of it is shown in Appendix B.
4) Start the service.

NFS is defined as Network File System. A NFS server can be mounted on any device supports NFS protocol. Using NFS server during testing avoids copying applications to target board, thus the debugging procedure becomes convenience.

The steps for configuring NFS server is shown below:

1) Install the necessary components: portmap, nfs-kernel-server
2) Edit the file “/etc/exports” to add the entry will be shared.
3) The same as TFTP server, the shared folder and the files it contains should be configured accessible for everyone.
4) Start NFS server and portmap.

5.2.5. Building Linux kernel

Before building Linux kernel, a DTS (device tree) file should be created at first. When Linux is booting up, it is necessary for Linux to know what hardware resources the board has. Thus, the device tree plays this role. Device tree is a data structure that describes the hardware and it is passed to Linux kernel by the bootloader. Then the Linux kernel can configure the hardware and work appropriately.

The device tree for Atlys board is derived from the one for OpenRISC simulator located in Linux kernel source code. Firstly, the clock frequency and memory size should be corrected to 50MHz and 128MB. And then the I2C controller node can be added according to an example of I2C controller description for device tree gained
from Linux’s opencores I2C driver. Appendix C shows a complete device tree for this case.

For Linux kernel, the I2C device interface support and I2C hardware driver for opencores are attached via “menuconfig”, which is a tool for selecting the features of Linux kernel. Then the binary image of Linux kernel can be generated by the GNU tool-chain for Linux development. More than that, the image should be modified to a bootable image for u-boot with the tool “mkimage” provided by u-boot.

Figure 5.2-4 shows the files under “/dev” path after Linux is transferred from TFTP server and boot up. The “i2c-0” file indicates the I2C support is added into Linux kernel successfully.

```
# cd /dev
# ls
console mem network_latency random
full network_throughput ttys0 ttys1
i2c-0 null ttys0 ttys1
kmsg urandom
```

Figure 5.2-4 i2c-0 under “/dev”

5.2.6. Sensor reading application

In Linux, any device is treated as a file. As a result, the I2C controller can be accessed by file operations such as open, write, read and close. Figure 5.2-5 is the flow chart of the sensor reading application, which is just a single access of I2C slave. Firstly, the application gets the exclusive access of I2C adapter with function “open()”. Secondly, the function “ioctl()” sets the address of I2C slave, then the following “write()” and “read()” function will attach the bit for indicating writing or reading operation automatically. Thirdly, using “write()” function to write control registers of I2C slave and “read()” function to get the sensor outputs. At last, the “close()” function will release the access of I2C adapter. The source code of this application can be found in Appendix D. The application should be compiled by GNU tool-chain for Linux development as well and stored in the folder of NFS server.
5.2.7. Stand-alone configuration

Loading the application from host computer to the RAM of development board via network protocols and booting it with u-boot are convenient when the application is under development. After the development of the application is finished, it is better to make the embedded system to be stand-alone. The solution is to store the application in SPI flash. Then u-boot can load the application from SPI flash and execute it. Thus, the embedded system becomes an independent device.

In u-boot, after the application is transferred to RAM by TFTP protocol, the application can be pushed into SPI FLASH within following steps:

- Probe the SPI flash in u-boot.
- Erase part of SPI flash that has the same size as application and the erased sector shouldn’t be the one that stores FPGA configuration bit-stream and u-boot.
- Copy the application located in RAM to the erased sector of SPI flash.

When the embedded system is reset, use the commands of u-boot to copy the application from SPI flash to RAM and execute from RAM or run it directly in SPI flash. A SPI flash sub-system in u-boot that can be invoked by command “sf” that is the key to perform this work.
6. Result

6.1. Usability

The application reads sensor output successfully that proves the Open source IP based embedded system with Linux is usable for industry.

6.2. License issues

Table 6.2-1 lists the license of each component.

<table>
<thead>
<tr>
<th>Components</th>
<th>Licenses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xilinx ISE</td>
<td>WebPACK</td>
</tr>
<tr>
<td>ORPSOC</td>
<td>LGPL</td>
</tr>
<tr>
<td>GNU tool-chain</td>
<td>GPL</td>
</tr>
<tr>
<td>U-boot</td>
<td>GPL</td>
</tr>
<tr>
<td>Linux kernel</td>
<td>GPL</td>
</tr>
<tr>
<td>uClibc Library</td>
<td>LGPL</td>
</tr>
<tr>
<td>Linux user space</td>
<td>Either proprietary or open source</td>
</tr>
</tbody>
</table>

Table 6.2-1 License investigation of each component

GPL: GNU General Public License
LGPL: Lesser GNU General Public License
As long as developers don’t offend the rules, there will be no license problem for applying such an embedded system in industry.

6.3. Pros and cons

Table 6.3-1 is a comparison between open source IP based embedded system with Linux and conventional embedded systems. The open source IP based embedded system with Linux refers to the one whose hardware platform is ORPSOC having Linux installed. The conventional embedded system refers to the one whose hardware platform is based on IC chips with fixed resources, such as AVR32 and ARM architecture. Developers can choose to install Linux on it or not.
<table>
<thead>
<tr>
<th></th>
<th>Open IP based embedded system with Linux</th>
<th>Conventional embedded system without Linux</th>
<th>Conventional embedded system with Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Product cost</strong></td>
<td>High</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td><strong>Development environment</strong></td>
<td>Basic-supported</td>
<td>Well-supported</td>
<td>Well-supported</td>
</tr>
<tr>
<td><strong>Flexibility and scalability</strong></td>
<td>High</td>
<td>-</td>
<td>Only Linux has scalability</td>
</tr>
<tr>
<td><strong>Software Portability</strong></td>
<td>High</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td><strong>Software development difficulty</strong></td>
<td>Low (except the case if hardware is not supported by Linux driver)</td>
<td>Normally is high, depends on the vendor's support</td>
<td>Low (except the case if hardware is not supported by Linux driver)</td>
</tr>
<tr>
<td><strong>Reusability</strong></td>
<td>High</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 6.3-1 A comparison between open source IP based embedded system with Linux and conventional embedded systems

Obviously, the pros of open source IP based embedded system with Linux are high flexibility and scalability, high software portability, low software development difficulty and high reusability. Whereas the cons are high product cost due to the higher price of FPGA chips, basic-supported development environment and more difficult software development if Linux driver doesn’t support the hardware.
7. Discussion

7.1. Usability

Although the application in this thesis is just to read output from an I2C sensor, but its success demonstrates that the open source IP based embedded system with Linux is usable. With the help of various open source IP cores and Linux drivers, it is possible to develop more complex embedded system for industry usage.

7.2. License issues

7.2.1. Summary of GPL and LGPL

GPL (GNU General Public License):
- Users have the right to run the program freely.
- Source code is open to public and free to be studied and modified.
- Developers can distribute derived works based on the source code, and they can decide if charge for the derived work.
- GPL is viral. It means as long as the original software is under GPL, the derived work should be under GPL as well. This grants the next users not only using the software freely, but also getting the source code to do whatever they want [18].

LGPL (Lesser GNU General Public License):
The most significant difference between LGPL and GPL is that the LGPL allows the libraries to be linked with proprietary software, which means the developers can use the libraries under LGPL freely in their software without opening source code to users [19].

7.2.2. License analysis of Open source IP based embedded system with Linux

Xilinx ISE is under Xilinx WebPACK license, which grants free usage of ISE basic functions. Thus, it is valid to use Xilinx ISE WebPACK to build the hardware platform.

GNU tool-chain and u-boot are both under GPL. However, there are no derived works based on them. GNU tool-chain and u-boot are just used as tools in this thesis.

The implemented embedded system is based on ORPSOC and the sensor reading application is taking advantage of “uClibc” library. Both of them are under LGPL.
Although the Linux kernel is under GPL, however, the copyright file in Linux kernel source code says that "Note! This copyright does *not* cover user programs that use kernel services by normal system calls – this is merely considered normal use of the kernel, and does *not* fall under the heading of derived work"[20]. In addition, it is also declared on OpenRISC project’s webpage “Third parties are free to create their own proprietary processor implementations. It is also possible to port proprietary software to the OpenRISC platform. Use and development of the implementations and software provided by the project is encouraged.”[21]

Therefore, it is valid to develop proprietary software on open source IP based embedded system with Linux and still allowing the industry company to keep their software’s source code as secret.

7.3. Pros and cons

The implemented open source IP based embedded system with Linux has some pros comparing to conventional embedded system as below:

1. High flexibility and scalability: the enabled I2C controller of ORPSOC indicates that the embedded system consists of open source IP cores. This is a flexible and scalable method of defining the hardware platform. Moreover, the Linux kernel is configurable such that the desired functions can be selected and unwanted functions are removable. On the opposite, the conventional embedded system is not flexible because the resources of hardware platform are fixed.

2. High software portability: With the help of Linux operating system, the software portability is increased because the software running on Linux user space is device independent. This grants the application to be ported among different architectures easily. Figure 7.3-1 is the schematic for explaining high software portability. Assume there are two embedded systems; one of the hardware platforms is ORPSOC, the other one is designed in conventional way. Both of them are having Linux installed. The two hardware platforms may have different components to perform identical function. As long as the application is running on the Linux user space, there will be no difficulty to port the application to each embedded system. For example, if the application is developed to access an I2C slave, both of the two different platforms should have an I2C controller. However, it doesn’t matter if the manufactures of I2C controllers are different. As for example, the I2C controller of ORPSOC is held by opencores community whereas the one in conventional embedded system is from a particular company. In Linux, the two I2C controllers are treated equally as an I2C adapter and the application is using the I2C adapter to access the I2C slave. If two embedded systems are without Linux installed, the device-specific drivers should be developed. Then
the software will have no guaranteed portability.

Figure 7.3-1 Schematic of high software portability

3. Lower software development difficulty: Linux provides hardware drivers for various devices. And any device is treated as a file in user space. The application can access the devices through system calls. Thus the software designer can concentrate on their application without being involved in complicated hardware driver development. Whereas the conventional embedded system typically has no hardware driver support that designer should firstly develop usable hardware drivers to support further development.

4. High reusability: Function change and migration are common issues in embedded design. Conventional embedded system with fixed resources will have a complete re-design when facing function change or migration. By contrast, the open source IP based embedded system with Linux will have less work when facing these situations. When it comes to function change, the open source IP based embedded system with Linux can modify its hardware platform on the current FPGA chip to fit new demands and Linux is scalable to add or remove supports for hardware changes. Moreover, Linux will reduce the workload of new software development due to its lower software development difficulty. Besides, function migration may be concerned when the embedded system becomes obsolescence, and open source IP based embedded system with Linux is a suitable platform being migrated to. The development of open source hardware makes it possible to find corresponding open source IP cores to replace the components of conventional embedded system. And it will be nice if the conventional embedded system has Linux installed then the software migration becomes simple due to high software portability. Even though the conventional embedded system has no operating system, developing software for Linux is still easier than developing a new device-specific software. Maybe one day the FPGA chip becomes obsolete and there will be no difficulty to port the whole system into another FPGA chip. Thus, the open source IP based embedded system with Linux is reusable to meet various requirements and situations.
The open source IP based embedded system with Linux is not perfect that has cons as well:

1. **High product cost:** Comparing to conventional embedded system chips, a FPGA chip is much more expensive which in turn leads to higher product cost. For instance, the cost of Spartan-6 FPGA chip is $67.65. By contrast, the prices of conventional embedded system chips such as AT32UC3A0128 microcontroller based on Atmel AVR32 architecture and AM3354ZCED50 processor from Texas Instruments based on ARM Cortex-A8 core are $6.33 and $16.8. All of the prices above are gained from AVNET\(^\text{[22]}\).

2. **Basic-supported development environment:** Differs from other embedded system’s development environment, the development environment for OpenRISC architecture only provides tools for compilation and debugging. There is no IDE or EDK. And the lack of documentation results in difficulty for being familiar with it. In addition, some documents are out of time making developers feel confusing.

3. **Software development becomes difficult when the hardware is not supported by Linux:** Linux has various device drivers covering a widely range of peripheral hardware. However, there are exceptions. The software development tends to be difficult if a peripheral has no driver support from Linux. In Linux, device drivers are distinct “black boxes” that make a particular piece of hardware respond to a well-defined internal programming interface. They hide completely the details of how the device works \(^\text{[23]}\). Thus, according to this principle, software developers only need to know the Linux operating system and the concept of a device. However, in case no device driver exists for a desired peripheral, a new driver needs to be developed. This is a more complex programming task involving system calls from user space as the mechanism to access the peripheral hardware.
8. Conclusion

In this thesis, a prototype of open source IP based embedded system with Linux is presented. A successful application reading the output of acceleration raw data from an I2C compass sensor-LSM303DLm is developed as well, which proves the usability of such an embedded system. Moreover, the investigation of licenses indicates the implementation doesn’t offend the rules of GPL and LGPL and developing proprietary software on the system is valid. Hence, companies are allowed to develop and sell the embedded systems without providing their source code to users or public.

Comparing to conventional embedded systems, although the open source IP based embedded system with Linux has some disadvantages such as: high product cost, only basic-supported development environment, and more difficult software development if Linux driver doesn’t support the hardware. However, it has: high flexibility and scalability, high software portability, low software development difficulty and high reusability.

Open source IP based embedded system with Linux is hard for beginners to start with and the application development for Linux takes time to learn. However, once the developer handles them well, the time to develop an embedded system will be decreased significantly because the lower software development difficulty of Linux user space. Moreover, when the function of such an embedded system is changed, a total re-design of the embedded system can be avoided due to it is flexible, scalable and reusable. In addition to function change, the work of function migration can be reduced because of the high software portability and reusability. Hence, the open source IP based embedded system with Linux is more suitable in industrial usage.
9. Future work

* Figure out how to enable GDB support. The GDB support will ease the procedure of debugging software. In this project, GDB is failed to access the target board by Xilinx USB platform cable. The probably reason may be that GDB and Xilinx USB platform cable are not configured properly. Thus, it is necessary to check out where the problem is in future.

* Add CAN (controller area network) controller to ORPSOC and develop Linux driver for it. CAN bus is widely used in industry. Adding the CAN controller IP core into ORPSOC will help developers learn how to customize the hardware platform of open source IP based embedded system. The mechanism of Wishbone interface will be studied as well. In addition, there is no Linux hardware driver that supports the open source CAN controller. Thus, the Linux device driver for the CAN controller is required to be developed.
10. Reference

[1] ORPSOC, http://opencores.org/or1k/ORPSoC


[20] Linux Torvalds, “Copyright of Linux”

[21] OR1K: Community portal, http://opencores.org/or1k/OR1K:Community_portal


Appendix A: Detailed tutorial for implementing open source IP based embedded system with Linux

1. Build development environment
   1) Download Virtual box from https://www.virtualbox.org/wiki/Downloads
   2) Download virtual box image from http://opencores.org/or1k/Ubuntu_VirtualBox-image_updates_and_information
   3) Add the image to virtual box and set parameters of it. Then all of the work will be done in this Ubuntu virtual machine.
   4) Download Xilinx ISE 14.5 and install it under WebPACK license.

2. Generate ORPSOC with I2C controller
   1) Open a new terminal in Ubuntu
   2) Download source code of ORPSOC with command “svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/orpsocv2”
   3) In ORPSOC’s source code, the path “/boards/Xilinx/atlys” is the folder for Atlys board.
   4) Modify the file “/rtl/verilog/include/orpsoc-defines.v”:

   ```
   `define XILINX_DDR2
   `define UART0
   `define UART0_EXPHEADER
   `define GPIO0
   `define SPI0
   `define I2C0
   `define I2C1
   `define ETH0
   `define ETH0_PHY_RST
   ```

   Uncomment this line to enable I2C controller.

   5) In the file “/rtl/verilog/orpsoc_top/orpsoc_top.v”, the signals for I2C controller can be found (i2c_sda_io, i2c0_scl_io):

   ```
   `ifdef I2C0
   `ifdef I2C1
   endif
   ```

   6) Assign the two signal to PMOD socket in ucf file under the path “./backend/par/bin/atlys.ucf”:

   ```
   # PMOD Connector
   # NET "JbD0> LOC = "$T3"; # Bank = 2, Pin name = IO_L62N_D0, Sch name = JA-D0_N
   # NET "JbD1> LOC = "$H3"; # Bank = 2, Pin name = IO_L64N_D9, Sch name = JA-D1_N
   ```

   And connect the I2C slave LSM303DLM as Figure 5.2-1 shows.

   7) Before start to build ORPSOC, the environment variables Xilinx ISE should be set with

8) Go to the folder “syn/xst/run” of Atlys board, and type “make all” to synthesize ORPSOC.

9) In Atlys board’s “backend/par/run” path, use command “make orpsoc.ncd” for mapping and place & route. And then the configuration bit stream can be generated with command “make orpsoc.bit”. The FPGA can be programmed with iMPACT tool from Xilinx.

3. Install U-boot

1) In terminal, download source code of u-boot with command: 
   "git clone git://openrisc.net/stefan/u-boot"

2) Set an environment variable named CROSS_COMPILE that assigns GNU tool-chain for bare metal development to compile u-boot:
   CROSS_COMPILE=or32-elf-
   export CROS_COMPILE

3) In u-boot’s folder, run command:
   make distclean
   make atlys
   make atlys
   Then the binary file “u-boot.bin” will be generated.

4) Go to the folder “/sw/utils” of ORPSOC, build the tools with makefile in this folder firstly if they are not compiled. And then add the size block to u-boot’s binary image with command “bin2binsizeword u-boot.bin u-boot_sz.bin”

5) Create a new folder “src” in Atlys board’s “/sw” path to store u-boot_sz.bin. And modify the makefile under Atlys board’s “/backend/par/bin”:

6) Use the command “make orpsoc.mcs” to generate SPI FLASH image that contains both bit stream for FPGA and u-boot binary image.

7) Write the generated image to SPI FLASH with iMPACT. It is recommended to use the iMPACT under Windows because iMPACT has a low speed when flashing SPI FLASH under Linux operating systems.

8) After the reset button is pressed, the FPGA will be configured and start running u-boot. Then the startup information of u-boot appears on an UART console:
NOTE: The Ethernet connection of u-boot may fall due to a bug in source code. The solution is to modify the code “#define ETHER_HDR_SIZE (sizeof(struct ethernet_hdr))” to “#define ETHER_HDR_SIZE 14” in the file “include/net.h”

4. Ethernet connection and TFTP, NFS server
   1) For u-boot, the network parameters should be set as environment variables with following commands:
      ● Set Ethernet MAC address: setenv ethaddr 00:12:34:56:78:9a
      ● Set gateway IP address: setenv gatewayip 192.168.2.1
      ● Set local IP address: setenv ipaddr 192.168.2.6
      ● Set net mask: setenv netmask 255.255.255.0
      ● Set server IP address: setenv serverip 192.168.2.2
      At last, save these parameters permanently with command “saveenv”
   2) In this thesis, the host computer is a Vmware virtual machine installs Ubuntu 12.04 distribution, its network should be configured as well:
      ● Choose an available Ethernet adapter as bridge-networking:
Set the network parameters:

- Press “Advanced” button, and go to “Hardware” label to configure the speed of Ethernet adapter:

- In the Ubuntu running in Vmware, the local IP address should be set to 192.168.2.2:
3) Set up TFTP server in host computer:
   - Install tftp, xinetd, tftpd with commands:
     sudo apt-get install tftp
     sudo apt-get install xinetd
     sudo apt-get install tftpd
   - Create a folder “/tftpboot” with command “mkdir /tftpboot”
   - Change the folder to be accessed by everyone “chmod –R 777 /tftpboot”
   - Put the configuration file “tftp” as shown in Appendix B to path “/etc/xinetd.d/”
   - Stop and then start TFTP server with command “/etc/init.d/xinetd stop” and “/etc/init.d/xinetd start”

4) Set up NFS server in host computer:
   - Install portmap and nfs-kernel-server with:
     sudo apt-get install portmap
     sudo apt-get install nfs-kernel-server
   - Add a line:
     “/nfs 192.168.2.2/255.255.255.0(rw,sync,all_squash,anonuid=500,anongid=500)”
     in file “/etc/exports”, “/nfs” is the folder will be shared.
     “192.168.2.2/255.255.255.0” are the local IP address and network mask of host computer.
   - The same as TFTP server, change the authority of “/nfs” folder as everyone is accessible: “chmod –R 777 /nfs”
   - Start NFS server:
     sudo /etc/init.d/portmap start
     sudo /etc/init.d/nfs-kernel-server start

5. Build Linux kernel
   1) Download source code with command “git clone git://git.openrisc.net/jonas/linux”
   2) Put the device tree “atlys.dts” in Appendix C to the path “/arch/openrisc/boot/dts/” in Linux source code.
   3) Set the environment variable CROSS_COMPILER:
      export CROSS_COMPILE=or32-linux-
4) In Linux source code’s folder, run “make menuconfig” to configure the functions of Linux kernel:

Select Processor type and features.

Change the “or1ksim” to “atlys” and then exit.

Go to general setup.

Select this one and then exit.
Select device drivers

Select I2C support and enter it.

Select I2C device interface

Then go into I2C Hardware Bus support
5) Save and exit the menuconfig. Then use command “make” to build kernel and file “vmlinux.bin” is the binary image of Linux.

6) Go back to the folder of u-boot, use the tool “mkimage” to build a bootable Linux image for u-boot with command “./tools/mkimage -n 'Linux for OpenRISC' -A or1k -O linux -T kernel -C none -a 0 -e 0x100 -d /the_folder_stores_vmlinux.bin/vmlinux.bin /the_destination_folder/uImage”. Then put the image into the folder of TFTP server.

6. Sensor reading application
1) Use the GNU tool-chain for Linux development to compile the application in Appendix D. The command is “or32-linux-gcc the_source_code_file.c the_desired_name_of_compiled_file”. And copy the compiled file to the folder of NFS server.

2) In u-boot, type “tftp uImage” to transfer the Linux image.

3) Boot Linux with command “bootm 100000”.

4) Set the embedded Linux’s IP address with command “ifconfig eth0 192.168.2.8”.

5) Mount NFS server using command “mount –t nfs 192.168.2.2:/nfs /mnt –o nolock”.

6) Then run the application from NFS server by command “./mnt/the_file_name_of_application”.

Select Opencores I2C controller
Appendix B: Configuration file for TFPT server

```plaintext
service tftp
{
    protocol     = udp
    port         = 69
    socket_type  = dgram
    wait         = yes
    user         = nobody
    server       = /usr/sbin/in.tftpd
    server_args  = /tftpboot
    disable      = no
}
```
Appendix C: Device Tree

//dts-v1/;
/
  compatible = "digilent,atlys";
  #address-cells = <1>;
  #size-cells = <1>;
  interrupt-parent = <&pic>;
/
chosen {
  bootargs = "console=uart,mmio,0x90000000,115200";
};

memory@0 {
  device_type = "memory";
  reg = <0x00000000 0x08000000>;
};
cpus {
  #address-cells = <1>;
  #size-cells = <0>;
  cpu@0 {
    compatible = "opencores,or1200-rtlsvn481";
    reg = <0>;
    clock-frequency = <50000000>;
  }
};

/*
 * OR1K PIC is built into CPU and accessed via special purpose
 * registers. It is not addressable and, hence, has no 'reg'
 * property.
 */

pic: pic {
  compatible = "opencores,or1k-pic";
  #interrupt-cells = <1>;
  interrupt-controller;
};

serial0: serial@90000000 {
  compatible = "opencores,uart16550-rtlsvn105", "ns16550a";
  reg = <0x90000000 0x100>;
}
interrupts = <2>;
clock-frequency = <50000000>;
};

enet0: ethoc@92000000 {
compatible = "opencores,ethmac-rtlsvn338";
reg = <0x92000000 0x100>;
interrupts = <4>;
};

i2c0: ocores@a0000000 {
compatible = "opencores,i2c-ocores";
reg = <0xa0000000 0x8>;
interrupts = <10>;

regstep = <1>;
clock-frequency = <50000000>;

#address-cells = <1>;
#size-cells = <0>;
};
Appendix D: Source code of sensor reading application

```c
#include <unistd.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <math.h>

// ACCELERATION Sensor Addresses (device address, sub register addresses and register value)
#define ACC_ADDRESS                     (0x30 >> 1)
#define CTRL_REG1_A_ADDRESS             (0x20)
#define CTRL_REG1_A_VALUE               (0x27)
#define OUT_X_L_A_ADDRESS Multi_READ    (0xA8)   // 0x28 is original value, for multi read,
// first bit should be set to 1
#define OUT_X_L_A_ADDRESS               (0x28)
#define test_address                    (0x29)   // an address for testing read and write
// function

int main()
{
    char buf[10]; // Define buffer
    int i, j;
    const char *devName = "/dev/i2c-0"; // device name displayed in embedded linux
    unsigned char acc_x_l, acc_x_h, acc_y_l, acc_y_h, acc_z_l, acc_z_h; // variables to store raw
    ACCELERATION Sensor outputs

    // open up the I2C bus
    int file = open(devName, O_RDWR);
    if (file == -1)
    {
        perror(devName);
        exit(1);
    }
    // init ACCELERATION Sensor
    // (1) Specify the address of the slave device: ACCELERATION
    if (ioctl(file, I2C_SLAVE, ACC_ADDRESS) < 0)
```
Evaluation of open source IP based embedded system with Linux

Appendix D

Jiayi Wang  2013-09-09

{  perror("Failed to acquire bus access and/or talk to slave");
  exit(1);
}

//Write to control register
buf[0]=CTRL_REG1_A_ADDRESS;
buf[1]=CTRL_REG1_A_VALUE;
if(write(file,buf,2)!=2)
{
  perror("Failed to write to the i2c bus");
  exit(1);
}

//Read ACCELERATION Sensor output
//Specify the address of the slave device:ACCELERATION
if (ioctl(file,I2C_SLAVE,ACC_ADDRESS)<0)
{
  perror("Failed to acquire bus access and/or talk to slave");
  exit(1);
}

//Write the address of the first byte
buf[0]=OUT_X_L_A_ADDRESS_MULTI_READ;
if(write(file,buf,1)!=1)
{
  perror("Failed to write to the i2c bus");
  exit(1);
}

//Read output
if(read(file,buf,6)!=6)
{
  perror("Failed to read from the i2c bus");
  exit(1);
}

//Assign ACCELERATION raw outputs to variables
acc_x_l=(unsigned char)buf[0];
acc_x_h=(unsigned char)buf[1];
acc_y_l=(unsigned char)buf[2];
acc_y_h=(unsigned char)buf[3];
acc_z_l=(unsigned char)buf[4];
acc_z_h=(unsigned char)buf[5];
printf("ACCELERATION raw data: 0x%x, 0x%x, 0x%x, 0x%x, 0x%x, 0x%x\n",acc_x_l,acc_x_h,acc_y_l,acc_y_h,acc_z_l,acc_z_h);
close(file);//close port
return 0;