Very Small Information Systems
Lab Exercises

Martin Schoeberl
mschoebe@mail.tuwien.ac.at

Rasmus Pedersen
rup.inf@cbs.dk

February 10, 2006

Introduction

These exercises provide an introduction to programming for the embedded Java processor (JOP). The exercises are intended to get you started with embedded systems and as a preparation for your project.

The exercises will focus on the build process for JOP, real-time threads, interaction with the physical world (IO), and networking with the available tiny implementation of the TCP/IP protocols (ejip). The main advantage of this system is that all sources are available and all systems code (to which a TCP/IP stack usually belongs) will be compiled together with the application. That means you can dig into the inner workings of the Java virtual machine and the Internet protocol suite.

We hope that you will share our fascination on programming of small devices and wish you a lot of fun!

Rasmus & Martin

1 Getting Started with JOP

The first mandatory exercise gives an introduction into the design flow of JOP. JOP will be built from the sources and a simple *Hello World* program will run on it.

To understand the build process you have to run the build manually. This understanding will help you to find the correct files for changes in JOP and to adjust the `Makefile` for your needs.

A detailed document about the build process can be found at `http://www.jopdesign.com/doc/build.pdf`.

1.1 Manual build

Manual build does not mean entering all commands, but calling the correct batch files (`.bat`) with the required arguments (if any) in the correct order. The idea of this exercise is to get knowledge of the directory structure and the dependency of various design units.

View all batch files and the one that are called from it before running them.

1. Create your working directory under `D:\xxx\yyy`
2. Download the sources from the opencores CVS server (There is a batch file for the CVS string under `D:\xxx`)
3. Connect the FPGA board to the PC and the power supply
4. Perform the build as described in the *Getting Started* section from the *An Introduction to the Design Flow for JOP* document.

As result you should see a message in your command prompt and a blinking LED on the FPGA board.

1.2 Using make

In the root directory (`jop`) there is a `Makefile`. Open it with an editor and try to find the corresponding lines of code for the steps you did in first exercise. Reset the FPGA by unplugging and plugging the power and run the build with a simple

```
make
```

The whole process should run without errors and the result should be identical to the previous exercise.

1.3 Change the Java program

The whole build process is not necessary when changing the Java application. Once the processor is built a Java application can be built and downloaded with the following make target:

```
make japp
```

Change `Hello.java` and run it on JOP.

1.4 Change the Microcode

The JVM is written in microcode and several `.vhdl` files are generated during assembly. For a test change only the version string in `jvm.asm` and run a full make.

```
version     = 20050827
```

change to:

```
version     = 20060215
```

The start message should reflect your changes.

1.5 Compile a different Java application

The class that contains the main method is described by three arguments:

1. The first directory relative to `java/target/src` (e.g. `app` or `test`)
2. The package name (e.g. `test`)
3. The main class (e.g. `Hello`)

These three arguments are used with `doit.bat` in the directory `java/target` and are set in the variables `P1`, `P2`, and `P3` in the `Makefile`.

Change the `Makefile` to compile the Java application `ejip.Main`. Open a web browser and locate the address `http://192.168.0.123/`[1]. This web page is served by a very tiny web server running on JOP.

Hint: You can invoke `make` with parameters to compile a different application without changing the `Makefile`:

```
make japp -e P1=common P2=ejip P3=Main
```

should result in the same application compiled and downloaded to JOP.

1.6 Simulation - Optional

There are two ways to simulate JOP: A simple debugging JVM written in Java that can execute *jopized* applications and VHDL level simulation with ModelSim. The make targets are `jsim` and `sim`.

Try to run the *Hello World* example and other Java programs on JopSim, the JOP simulation in Java. If you are interested how a simple implementation of the JVM works look into the source (`JopSim.java` in directors `java/tools/...`).

1.7 Eclipse

Using Eclipse as the editor for JOP applications is really helpful. Setup the Eclipse workspace as described in *An Introduction to the Design Flow for JOP*.

2 Threads on JOP and IO

This exercise will give you an introduction to thread programming on JOP and will show you how the IO ports can be accessed.

[1] The chances are high that this address does not work in your subnet. Select a proper address and set it in `ejip/Net.java`.

2.1 Real-Time Threads

Standard Java threads (`java.lang.Thread`) are not suitable for real-time programming (see JOP thesis Section 4 and Section 6.1). Therefore, JOP contains it's own real-time profile with `RtThread` threads. The main difference between `j.l.Thread` and `RtThread` is that the real-time threads are strictly periodic. That means you decide at thread creation the period of the thread. The thread application code is basically a *forever* loop with one `waitForNextPeriod()` invocation to block the thread until the next period.

```
// First parameter is the priority that should be
// assigned rate monotonic - meaning threads with
// shorter periods get a higher priority
//
// The second parameter is the period in us.
//
new RtThread(5, 10000) {
    public void run() {
        for (;;) {
            // do your periodic work here
            work();
            // and wait for the next period
            waitForNextPeriod();
        }
    }
};
```

Write a simple `RtThread` application that prints out a message every second. Extend this program with a second thread with a different period that also prints out the message. When you play around with the periods and priorities can you see sometimes weird messages? Does a `synchronized` block[2] help?

The scheduler in JOP (as any real-time scheduler) uses strict priority order. That means when a high-priority thread is ready to run than it will interrupt a lower priority thread. But a lower priority thread will *never* interrupt a higher priority thread. That is very different to standard OS scheduler which try to be fair. Fair is not a real-time category.

Question: What happens when you have two threads with two different priorities and the higher priority thread does not invoke `waitForNextPeriod()`?

2.2 Input and Output

Embedded system programming without an interface to the physical world is no fun. Therefore, the JOP board contains 10 digital inputs, 4 digital outputs and 2 analog inputs. These IO ports are connected by industrial style headers.

Run `test.Baseio` to check your IO ports and as an example how those ports are accessed. Write a simple counter program that increments a counter with each press of a button connected

[2] In the current version of JOP `synchronized` on methods is ignored. You have to use `synchronized` blocks.

to one input port and print the counter value out. Switches chatter when opened or closed – do you observe any strange effects?

Questions: How can this be compensated for (think about a periodic thread and a small state machine)?

We can attach a temperature sensor to the analog input and a relay to one output that switches a heater. Now it's easy (or not?) to implement a temperature control.

Questions: Would you like a web server running on JOP where you can switch on or off the heating? Then run `ejip.Main`. Can we use this tiny web server to set the desired temperature for the temperature control?

3 Network Programming with EJIP

In this session we will learn network programming on JOP. The library provided does not contain the standard `java.net` network API (again this library is not real-time compliant and too complex for JOP). JOP comes with it's own implementation of an Embedded Java IP (EJIP). This (TCP)/UDP/IP stack is intended for small Java embedded systems and does not need a garbage collector (which is often avoided in real-time systems and even sometimes not available on small Java systems).

3.1 A First IP Packet

In this example you will capture an IP packet on JOP and display it's contents on the serial line. All `System.out` prints are written to the Windows command box.

To print out the IP information (e.g. source and destination address, protocol) you have to add `print` statements into the TCP/IP stack. The TCP/IP stack for JOP is in package `ejip` (which stands for Embedded Java IP) and the relevant class is `TcpIp.java`.

The task list:

1. Prepare for the exercise by reading the relevant official IP document (RFC 791)
2. Use ping on the PC to send IP packets
3. Capture the packages with Ethereal
4. Change the method `TcpIp.receive()` to print the IP header information

Hint: to print out an IP address in the usual dot notation you can use the following method:

```
static void printIp(int ip) {

    System.out.print(ip>>>24);
    System.out.print(".");
    System.out.print((ip>>>16)&0xff);
    System.out.print(".");
    System.out.print((ip>>>8)&0xff);
    System.out.print(".");
```

```
        System.out.print((ip)&0xff);
    }
```

3.2 JOP Pings the PC

After receiving packets on JOP it is time to make JOP active and send a packet to the PC. Implement an ICMP echo request (the famous Ping) on JOP. Ping your PC every 5 seconds and watch the packets with Ethereal.

As a preparation for this exercise find the related RFC document and scan the section *Echo or Echo Reply Message*. You can also take a look into the source `TcpIp.java` to get an idea how an IP packet is constructed with `ejip`. You have to build the whole ICMP packet from scratch (except checksum calculation).

When your ICMP echo request is correct (and the firewall on the PC is correctly configured) the PC will send a reply to this echo request. JOP will print out the character `P` and the *type* and *code* from the reply (see `doICMP` in `TcpIp.java`).

Hint: If you do not want to start from scratch you can use the example `Pinger.java`. You have to change the destination address (which is 192.168.0.5 in the example) to match your PC's IP address. Furthermore change the data that is part of the ICMP echo message so it contains your group number and watch for your packet in Ethereal.

3.3 Simple UDP Communication

Implement a UDP server on JOP. Select a port number and listen to this port for UDP messages. Take the message, build an answer of your choice and send it back to a different port where the client, who sent the original message, is listening.

This exercise contains two parts: The JOP UDP server and the client on the PC side. Take a look into the following files for an idea of the implementation:

- `ejip/UDP.java`
- `ejip/UDPHandler.java`
- `app/oebb/Comm.java`
- `udp/UDPDbg.java`

Questions: Is the role of the client and server so clear? Does UDP itself make distinctions between client and server?

Hint: Create a new class `UDPServer.java` that extends `UDPHandler`. Then look into `Comm.java` and `UDPDbg.java` for ideas of the implementation.

Questions: Can we use this UDP client/server to send commands for the output ports and receive the status of the input ports? How about a Java Swing application where you can control those ports and visualize the input values? It could be a start for a simple SCADA[3] system.

[3] Google for the meaning of SCADA