Introduction

Often embedded systems are designed to be small, low weight, low power, low energy, low cost, real-time, distributed, reliable, durable, safe, and secure. In general this means the simplest systems are usually preferred and whatever hardware and software that is unnecessary is taken away from the system and its development process. Therefore, embedded engineers frequently face limited resources and deal with low-level programming as well as bit-operations on small processors and microcontrollers. Many embedded systems are bare-machine, meaning there is no operating system. Such systems are often programmed in assembly and C.

The concepts that do not need to be practiced on a bare-machine, are performed on the host-machine.

In the following we start with a hello-world program on the bare-machine in chapter 1. We will see how a simple C code is cross compiled to run on a machine without OS. We use Qemu virtualization for the labs, but the overall process is similar to working with a physical machine. Chapter 2 focuses on bit operations in C on the host machine (without cross compilation and Qemu). Chapter 3 focuses on peripherals. At first, we work with timers on the bare-machine. Then we work with input preprocessing on the host machine.

Chapter 4 explains how to mix C code with assembly. This may be needed to design a superior system by writing the crucial segments of the code directly in assembly. Chapter 5 explains polling- and interrupt-based I/O operations as well as serial communication between devices (virtually on Qemu). Chapter 6 explains non-preemptive multi-threading on the bare-machine. Chapter 7 explains preemptive multi-threading as well as handling mechanisms for shared resources, on the bare-machine. Chapter 8 focuses on Finite-State Machine (FSM) representation of systems on the host machine.

Feedback Questionnaire

Please include answers to the following questions, at the end of each chapter (along with the other deliverables):

- Which program are you in (DI, EL, …)?
- What are the positive aspects of this chapter?
- What can be improved?
- How much time did it take to complete (in hours)?
- Any other comments?

This questionnaire can be submitted anonymously. You may write the lab assistant's name and send it through mail-slot (brevinkast) in the IDA mailroom (postrum). Alternatively, feel free to hand them directly to the lad assistant or email them.

Notes

The distribution of the deadlines does not exactly correspond to the time required for each chapter to complete. The deadlines are displayed on the lab's web page.

Please read the related chapter in the lab manual and the required resources, early on, so that you can estimate the required time and plan ahead of the deadlines.

Please do not jump over the chapters. The chapters are arranged in such a way that the earlier ones provide information and training that is required for the later chapters.

Please read each chapter completely to the end, before you start working on the assignment. Different chapters might be organized differently.

The number of pages in a chapter does not correspond to the time required for completing the assignments. Some small chapters may require you to study a lot from other resources.

Please arrive on-time in the lab sessions.

The lab assistants' time (for questions and demo) is prioritized for the students that are working in sync with the deadlines, as well as the students that arrive on-time in the lab session.

You may skip questions/assignments marked as "optional". But once you decided to do them, you must complete them correctly. We may ask you to correct them in order to pass. This is to prevent learning wrong information.

Qemu: Use Ctrl+Alt to release the mouse (pointing device). Qemu grabs the mouse when one clicks inside its window. This may not work in ThinLinc.

Working Remotely: ThinLinc might not work properly. Please use ssh as:

```
ssh -X  userID@remote-und.ida.liu.se
```

Authors' and Assistants' history:

2016, Nima Aghaee, Ahmed Rezine

2015, Ke Jiang, Adrian Horga

2014, Ke Jiang, Adrian Horga

…

Chapter 1 Bare-Machine Hello World

1.1 Host and Target Machine

Typical embedded devices do not provide software development friendly environments such as editors, compilers, or debuggers. They might not even have a console or an operating system. Therefore, most of the time, the software development process is carried out on ordinary desktop workstations with full-fledged integrated development environments. The development workstations are called host machines, while the embedded device is called the target machine.

1.2 Cross-Compilers

The host machines are equipped with cross-compilers, i.e. compilers that do not produce code for the machine/architecture they are running on but for a different machine/architecture. For example, if the host computer is an Intel x86 with Windows, a cross-compiler would be a compiler that creates code for an Intel x86 with Linux. A different example would be the case when the host is an Intel x86 with Linux and the target is a Sun Ultra SPARC with Linux. In the first example, the hardware architecture is the same but the interfaces to the hardware (the operating systems running on the host and target respectively) differ. In the second example, the hardware architecture differ but the operating systems are the same. A third example illustrates the case in which both hardware architecture and operating system differ from host to target: the host is an Intel x86 with Windows, while the target is an ARM processor with no operating system.

1.3 Emulators

Code developed on the host for the target cannot be run natively on the host for testing purposes. An alternative would be to upload the code on the target computer/board and to test it there. This approach may have several limitations. For example, the target could not be present at the developing site, or the target (or the uploading process) is very slow which would render the testing process very time consuming. Therefore, in many cases, the software is debugged and tested on the host machine by making use of an emulator, i.e. a program that mimics the behavior of the target machine.

Depending on the host and target platforms, the emulation can be slower or faster than the actual execution on the embedded platform. The emulated execution can be slower due to the emulator overheads. On the other hand, the emulator may be running on a much faster host machine. For example, the host can be an Intel x64 at 2.6GHz while the target is an 8-bit microcontroller at 4MHz.

1.4 Target Machine in this Lab

The target machine is an Intel x86 PC with no operating system (bare-machine). The embedded software will run directly on the hardware (Qemu virtualization) without any help from an operating system. The software will be loaded to the target from a standard boot device (floppy disk).

1.5 Work Flow

Copy “hello world” from “skeleton” directory to your local directory (we assume: userID/TDDI11/ hello_world)

```
cp -r /home/TDDI11/lab/skel/hello_world /home/userID/TDDI11/hello_world
```

Please note that userID is the user name that you use to login to the lab computers. Now let us change to the new directory and check it:

```
cd /home/userID/TDDI11/hello_world

ls
```

Check to see if “hello_world” directory contains the following:

- main.c
- Makefile
- floppy.img
- mtools.conf
- makeNrun.sh

The source code is in “main.c”. Compiler and linker commands are listed in “Makefile”. The binary file that will be generated by this process must be placed in the floppy image file “floppy.img”. Placing the binary file into the floppy image must be done with “mcopy” command. The necessary settings for “mcopy” are defined in the configuration file “mtools.conf”. The floppy image is used to start the virtual machine “Qemu”. The overall process is described in the script “makeNrun.sh”. The details are discussed in the following.

1.5.1 The Source Code

Open the “main.c” file (for example with “cat” if you only want to check the content. We can use for example “nano” or “gedit” if we want to edit the content):

```
cat main.c
```

Or

```
gedit main.c &
```

The code is as follows:

```
#include <libepc.h>

int main(int argc, char *argv[])

{

ClearScreen(0x07);

SetCursorPosition(0, 0);

PutString(">>>>>>> Empty … Skeleton <<<<<<<\r\n");

return 0;

}
```

This application will run without using an OS. A library is used to provide some basic functionality. In most C programs strings are displayed by means of the “printf” function. “Printf” is part of the standard C library “libc”. “Printf” typically is implemented by the operating system call “write”. As there is no OS on the target platform here, we cannot use “write” and “printf”. Therefore, we use the function “PutString” from the “libepc” library, which communicates uses functions such as “inport” and “outport”. These will directly communicate with the bare-machine.

1.5.2 Cross Compilation

The target machine architecture is “i386-elf”. The “Makefile” includes instructions for cross compiling the main.c file and linking it with the libepc. The compiler is installed in the following directory:

```
/sw/i386-elf-gcc-4.9.0/bin/i386-elf-gcc-4.9.0
```

The linker is installed in the following directory:

```
/sw/i386-elf-gcc-4.9.0/bin/i386-elf-ld
```

Usual compilers/linkers will generate an executable program suitable for a specific operating system. In such an implementation, the library codes are not integrated in the generated program file (shared libraries are typically linked in by the OS when the program is running). However, for this bare-machine implementation, a “raw” application with all needed codes integrated in one binary application file must be generated. The object files and libraries are combined in a single binary file, “embedded.bin”. In order to generate this file, type:

```
make
```

Then, do

```
ls
```

Check if “embedded.bin” file is generated.

1.5.3 Copying to Floppy

The binary file must be placed in the floppy (we use a virtual image). A prepared floppy image, “floppy.img”, is provided in the skeleton that we copied at the beginning. There is a skeleton binary file in the floppy image that must be replaced. Since the desired binary file must be placed to the floppy under specific considerations “mcopy” command is used. The configuration file “mtools.conf” defines “floppy.img” as drive “a:”. To pint the “mcopy” to the proper configuration file, do:

```
export MTOOLSRC=/home/userID/TDDI11/hello_world/mtools.conf
```

Place the newly generated “embedded.bin” into drive “a:”

```
mcopy embedded.bin a:
```

A question will be asked by “mcopy” about what to do with the existing binary file. This is the skeleton binary that you must overwrite with your newly generated binary file. Select “o”.

1.5.4 Starting Qemu

The final step is to load the application in the emulator (Qemu). The file that represents the floppy disk must be specified for Qemu to boot the system:

```
qemu-system-i386 -fda floppy.img
```

A new window for Qemu will open. The application will be executed printing:

```
>>>>>>> Empty Floppy Used as Skeleton <<<<<<<
```

Now you can close Qemu window.

If you click inside Qemu window, it grabs the mouse. Use Ctrl+Alt to release the mouse. This may not work in ThinLinc.

1.5.5 The Script to Make and Run

There is a script with required commands to simplify the process that we did manually in the previous sections (1.5.2 to 1.5.4). In order to run the script type:

```
makeNrun.sh
```

1.6 Evaluation

1.6.1 Assignments

Modify the program (main.c) to write the date, your names, and your student IDs instead of "Skeleton" message. Start from 1.5.1 and edit "main.c". Then you can either take step by step from 1.5.2 to 1.5.4, or use the script as described in 1.5.5.

1.6.2 Demonstrations

Unlike the following chapters, you do not need to demonstrate your work to the lab assistant for this chapter.

1.6.3 Deliverables

- Feel free to give feedback according to the questionnaire

Email your deliverables to your lab assistant. Write in the subject: TDDI11 Chapter 1.