Embedded Software Development

Host and Target Machine

Typical embedded devices do not provide software development friendly environments such as editors, compilers, 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.

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.

Emulators

Obviously, code developed on the host for the target cannot be run natively on the host for testing purposes. One 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 behaviour of the target machine.

A common criticism of emulated execution is that it is much slower than native execution on the target. While definitely the emulator brings significant overhead to the execution of the embedded software, the emulator runs on the host machine which is typically much faster than the target machine. Compare a host Intel x86 at 2GHz with a target 8-bit controller at 4MHz!

Lab Environment

Target Machine

The target machine is an Intel x86 PC with no operating system. The developed embedded software will run directly on the hardware without the intercession of the operating system. The software will be loaded to the target from a standard boot device (floppy disk, memory-stick, etc.).

Machines matching the target are available in PC lab rooms, and in principle we could test our embedded software directly on them by inserting a boot device and reboot the computer. Unfortunately such approach have several issues. First, the administrator of those would hardly allow us to boot arbitrary software that could mess up the installed OS. Second, such approach would be very cumbersome. Copy the software to a boot device, connecting it to the target, and rebooting is a very slow process. We will instead automate the process of preparing a boot device, load it to the target and boot the target by using software, files and an emulator for the target machine.

Host Machine

The host machine will be a Sparc running SunOS 5.10 (Solaris 10). Available on the host are a cross-compilation tools for the i386 architecture, and software emulating a standard x86 PC (target machine) with some common devices.

Work process to build a Hello World application (long version)

I will here describe the long version of how to create and test an embedded application. The primary purpose is for documentation, useful if some issue occur, and to outline the steps needed for those who want to set up an environment at home. The process is automated using the following important files, containing all details:

```
/home/TDDI11/sw/lib/bootload.bin
/home/TDDI11/sw/lib/link.cmd
/home/TDDI11/sw/lib/Makefile.common
/home/TDDI11/sw/lib/Makefile.default
/home/TDDI11/sw/lib/mtools.conf
```

1) Write the source code for the embedded application in your favorite program code editor. The recommended tool is emacs. Other available tools are vim, gedit and eclipse. Our application will not have any OS, but a small library will provide some basic hardware abstraction.

   ```
   emacs main.c &
   ```

   The following application implements “Hello World”.

   ```
   #include <libepc.h>
   int main(int argc, char *argv[])
   {
       ClearScreen(0x07);
       SetCursorPosition(0, 0);
       PutString("Hello World\r\n");
       return 0;
   }
   ```

   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 on our target platform, there is no operating system, we cannot rely on the write system call. Moreover, libc is also not present for the target machine. Therefore, we use the function `PutString` from the libepc library, which communicates with the hardware directly by means of inport and outport operations, without relying on the services of an operating system.

2) Compile your source files. We will use the C compiler from the GNU Compiler Collection configured to cross-compile for our target (i386-elf-gcc). For assembly code the Netwide Assebler (nasm) will be used.

   ```
   i386-elf-gcc -Wall -c main.c
   ```

3) Link the object files and libraries to one single file. By default the compiler/linker will generate a executable program suitable for a specific operating system, and will NOT include library code in the generated program file (shared libraries are typically linked in by the OS when the program is started). We do not want that, instead we want a “raw” application with all needed code available in one binary application file. To achieve this we must specify rules for the linker to follow. The rules specify what libraries the linker should include in the application by default and how the application will be placed in memory.

```
i386-elf-ld main.o -T link.cmd -static -ustart -Map link.map
```

4) Create a file that represent an empty, unformatted floppy drive.

```
dd if=/dev/zero of=floppy.img count=2880
```

5) Create a standard FAT12 or FAT16 DOS filesystem on the floppy (file). We use mtools configured to work on a file instead of a real floppy. On a linux system you can use mkdosfs.

```
mformat a:
```

6) Populate the bootsector with a small program that reads the embedded application to memory and starts it.

```
copyboot bootsect.bin floppy.img
```

7) Copy the embedded application to the floppy (file). We use mtools again. If you are on your own Linux system you can loop mount the file instead.

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

8) The final step is to load the application in the emulator. We use QEMU. The only thing we have to do is to tell QEMU which file that represent the floppy disk, and that is should be used to boot the system:

```
qemu -fda floppy.img -boot a
```

When we later want to use the serial port in our embedded application it must be connected to something. You can easily connect it to the terminal where you start QEMU:

```
qemu -fda floppy.img -boot a -serial stdio
```

To connect the serial port to the serial port of another computer for communication between two applications you must start two emulated machines, and connect the ports. Let one of the machines act as server:

```
qemu -fda floppy.img -boot a -serial tcp::8888,server,nowait
qemu -fda floppy.img -boot a -serial tcp::8888
```

Running on a real target machine

If you or your parents have an old PC with a working floppy drive in your closet, and some stray floppy disk, you can try your embedded application on a real target machine. Load the application to the floppy according to the steps described previously (but now you use a real floppy instead of a file). You can also take the final floppy.img file and dump it to a floppy using some tool the write raw sectors (dd, rawrite). If your intended target machine is the only one with a floppy drive you can probably boot some old OS on it and use it to get the application to the disk.

Then put the floppy in the floppy drive of the target machine. Make sure that the first boot device is the floppy and then start up the target machine. The boot loader from the floppy will put embedded.bin in memory and execute it.

Work process (student version)

The work process described above is tedious. To simplify your work the process are automated.

Setting up the lab system

To get access to the compilation tools and the prepared lab files you must execute a few simple commands. First you want to grab the lab files prepared for you (unless you already did in `lab1`):

```
cp -r /home/TDDI11/lab/skel ~/TDDI11
```

Then you must make the cross-compiler, assembler, and emulator available:

```
module add /home/TDDI11/sw/modulefiles/tddi11
module list
```

If the last of the above commands display tddi11 in the list it worked. If it worked you probably want the first of the two commands above to happen automatically the next time you log on. It is possible:

```
module initadd /home/TDDI11/sw/modulefiles/tddi11
```

Compile and run Hello World

To get started and run a simple application on the emulated target you should do the following. First enter the folder with the application (I assume you copied the prepared files to ~/TDDI11):

```
cd ~/TDDI11/hello_world
```

Then you must compile the application and prepare the floppy:

```
gmake
```

Finally you can test the application by starting one single emulator:

```
gmake run
```

To start two connected emulators (relevant in later labs) you do:

```
gmake link
```

To stop the emulation you just have to press enter in the terminal that started it. As the emulation require considerable processing even when not doing anything you should not leave it running for extended periods. You should also avoid clicking in the emulated window (it is safe to click on the titlebar), as that will destroy all subsequent input in that emulator (we need working input in lab 3, 4, and 5).

Run emulation on remote Linux server

N.B. ***Unfortunately we do not seem to get reliable results even from*** `crabbofix` ***anymore*****.**

Unfortunately `CPU_Clock_Cycles` does not work in our Solaris version of the emulator. To work around this we will run the labs that require use of the `CPU_Clock_Cycles` function remotely on a Linux server. One server is available: `crabbofix`. To start a simulation on it (do not leave the simulation running for extended periods of time):

```
gmake crabbofix
```

Before the simulation starts some information about the actual CPU the emulator runs on will be printed for your reference.

Working at Home

If you want to work at home you have several options:

- Use Linux to connect to astmatix.ida.liu.se and run remotely
- Use putty and some X-server in Windows to connect to astmatix.ida.liu.se and run remotely
- Connect to and work on astmatix, but download the final floppy.img and run QEMU locally
- Setup a complete environment in Windows and work locally
- Setup a complete environment in Linux and work locally

Remember that none of the option are supported. You are on your own. The last two options require much setup and configuration. The software you need is distributed with the book Fundamentals of Embedded Systems, where C and Assembler meet.

Related software

This is where you can find reference information:

- DJGPP, http://www.delorie.com/djgpp/
- NASM, http://www.nasm.us/
- QEMU, http://www.qemu.org/
- GCC, http://gcc.gnu.org/
- MTOOLS, http://www.gnu.org/software/mtools/
- uC/OS-II, http://micrium.com/page/products/rtos/os-ii/
- MULTI-C, http://www.mixsoftware.com/product/multic.htm