COMP1010
4. Compiling code for the robots

Mark Handley
M.Handley@cs.ucl.ac.uk

Contents

1 Introduction

You've written some C code and you want to run it on a robot. How do you do that? In this brief note, we'll explain.

2 Logging in to the robots

You will be using Linux/MacOS command-line tools to login to the robots, copy code on to them, compile it, and run it. Most servers on the Internet run Linux or a variant of Unix, and this is how they are usually managed too, so if you're not used to Linux command line, now is the time to learn.

When the robot starts up it will display its name and IP address on the LCD display. You should be able login to the robots either by name or by address, but the name will only work if the robot is connected to the network the DNS server expects the robot to connect to. In this document we'll use the IP address.

Suppose the robot displays the IP address "128.16.79.1" on its screen. To login to the robot, you type:

```
bash-3.2$ ssh localuser@128.16.79.1
```

You will be prompted for a password - the password is also "localuser" (the same as the netbooks).

After logging in, you will be presented with a Linux command prompt listing the name of the robot you've logged in to.

3 Copying code on to the robots

We recommend using *subversion* to copy code to and from the robots, but you can also manually copy code using the `scp` copy program.

3.1 Copying code with scp

If you're on one of the netbooks and you wish to copy the file `foo.c` to the robot at IP address `128.16.79.1` you would use the following command:

```
bash-3.2$ scp foo.c localuser@128.16.79.1:
```

Don't forget the trailing colon, or you'll create a file called "guest@128.16.79.1" on the netbook instead of copying to the robot!

If you edit the code on the robot, when you've finished you'll need to copy the code back to the netbook:

```
bash-3.2$ scp localuser@128.16.79.1:foo.c .
```

And then manually store the code somewhere safe so you've still got the changes after you return the netbook.

3.2 Copying code with subversion

If you've only one source code file, it's not too bad to copy files manually with scp. But you should have at least one source file and a `Makefile`, and as time progresses you should accumulate multiple files that make up your code. Copying them all manually, one by one, is sure to end up with you forgetting to preserve some changes at the end of a lab session.

It is better to copy changes all in one go using subversion. If you've set up a subversion repository on the computer `newgate`, in the directory `/cs/students/foo/ucacxxx/comp1010`, then you can check out all your code in one go using the command:

```
bash-3.2$  svn co svn+ssh://ucacxxx@newgate/cs/students/foo/ucacxxx/comp1010
```

`co` is short for *checkout*, but that's still a fairly long incantation the first time you type it.

However, once you've checked out your respository, you can download any updates from the server using:

```
bash-3.2$ svn update
```

and upload changes using:

```
bash-3.2$ svn commit
```

You can use `up` as an abbreviation for `update` and `ci` as an abbreviation for `commit`.

If you prefer to edit files on your laptop rather than directly on the robot, you can use `svn ci` to upload changed files from your laptop to the server, then use `svn up` to download those changes to the robot for compilation. In this way you always keep your files backed up on the server.

If you prefer to edit files on the robot, it's probably a good idea to do `svn ci` periodically to back them up to the server.

4 Compiling your code

4.1 Compiling a single file

If you have a program as a single source code file, you can compile it on the robot using `gcc`:

```
bash-3.2$  gcc -o program program.c
```

This compiles the source code file program.c and puts the compiled executable in `program`. You can then run it by typing:

```
bash-3.2$  ./program
```

Normally though, we want to specifiy some additional compiler options:

```
bash-3.2$  gcc -g -Wall -o program program.c
```

This specifies some flags for the C compiler to use.

- `-g` This includes debugging symbols in the executable, so that when it crashes we can debug it using the debugger *gdb*.
- `-Wall`. Nothing to do with wall following - this specifies that the compiler should switch on the generation of `all` `W`arnings. This is always a good idea when compiling C code, as without it the compiler won't warn you about many silly errors.

You should **always** use these flags when compiling your code - they'll greatly help with debugging problems.

4.2 Compiling Multiple Files

It's a bad idea to put all your code in one file (though you may start off that way). You typically want to put all the functions related to one feature into one file and specify it's interface using a header file. Splitting code up like this helps you work together on the program, and helps avoid your software growing into incomprehensible spaghetti code.

If you've got functionality related to sensor parsing in sensors.c, functionality related to driving motors and reading motor encoders in motors.c, and the main loop of your wall-follower code in wall.c, you can compile the program using:

```
bash-3.2$  gcc -g -Wall -o program sensors.c motors.c wall.c
```

But usually you don't want to do that (you'll get confused between any warnings generated by each of the separate C files). Normally you compile the source files separately to object files (ending in `.o`), and then link them together:

```
bash-3.2$  gcc -g -Wall -c sensors.c
bash-3.2$  gcc -g -Wall -c motors.c
bash-3.2$  gcc -g -Wall -c wall.c
bash-3.2$  gcc -g -Wall -o program sensors.o motors.o wall.o
```

But that's a lot of typing, so we usually use a utility called `make` to manage all this (and dependency tracking) for you.

4.3 Using make for compilation

It's easy to forget to recompile a file you've copied or changed, or to miss out a flag you need. It's a good idea to use a utility called *make* to control the compilation process, so you don't get confused. Here's an example:

```
CC=gcc
CFLAGS=-g -Wall
all:    program
program: wall-follower.o motors.o sensors.o
        $(CC) $(CFLAGS) -o program wall.o motors.o sensors.o
wall.o: wall.c wall.h
        $(CC) $(CFLAGS) -c wall.c
motors.o: motors.c motors.h
        $(CC) $(CFLAGS) -c motors.c
sensors.o: sensors.c sensors.h
        $(CC) $(CFLAGS) -c sensors.c
```

This code compiles the three source files wall.c, sensors.c and motors.c separately to .o files, then links them together to create the wall-follower executable. If you change any of the .c or .h files, all the correct files will be recompiled just by taking `make`.

One gotcha with Makefiles is that the *make* utility treats the TAB character as being meaningful. In the 3rd and 4th lines, the whitespace after the "`:`" must be a TAB. Similarly the leading whitespace on the 5th and 7h lines must be a TAB, not space characters.

In fact `make` has built in rules for how to generate .o files from .c files, so if you stick to the conventional variable names `CC` and `CFLAGS`, you can usually just use:

```
CC=gcc
CFLAGS=-g -Wall
all: program
program: wall-follower.o motors.o sensors.o
	$(CC) $(CFLAGS) -o program wall.o motors.o sensors.o
```

And then just type `make` to compile anything that needs recompiling.