

## Lab session 6

### Introduction to Embedded Linux

In previous labs we investigated Bare-metal C coding and JTAG debugging. However, many embedded systems use an operating system to provide some standard functionality such as a user interface and a file system as well as simpler software interfaces to control I/O devices. For this lab we will investigate Linux and some of the built in features and programming languages.

**Step 1:** Power up the Beaglebone board but this time do not put it into Bare-metal mode (do not press S2 while powering it up). Also, do not connect the SEGGER JTAG debugger unit. You should see the blue LEDs flash. Now connect the TTL/USB serial cable and launch a terminal program with the baud rate set to 115200. Press the enter key of your keyboard and you should see a login prompt. Here we are using the “console” interface of the Linux operating system. This is a simple low-level interface that always exists in a Linux system but is not always made available. We usually experience a GUI like Android, which an extension to the Linux system and can be thought off as sitting on top of it.

**Step 2:** Login as username **root** and use the password **root** if required. This is the default administrator login and password and should give you full access to all the capabilities. You will be surprised how many systems out there still have the default username and password. At the prompt type “cd /” and then enter. Then “ls” enter. You should now see the contents of the top level of the file system. Enter “pwd + enter” to see the working directory. “ls -l” shows the contents and permissions of the directory. Look up some basic file system commands and try them.

Question 1: Explain what “cd”, “ls”, “mkdir” and “rm” do.

**Step 3:** Type “more /etc/issue” and then return. This displays the version of Linux you have. The “more” command is useful for displaying files and the output of commands eg “issue”. Enter the command “env” to see the system Environment Variables”.

**Step 4:** Use the command “touch” to create “hello.txt” in the “tmp” directory. Then enter the command “nano -c hello.txt”. Welcome to the world of simple text editors!

Enter some text, save it and then exit. Use the “more” command to see the contents of your text file.

**Step 5:** Type “sudo reboot” and then enter. The system will reboot and you may have to renew your terminal connection. Check the “tmp” directory and you will notice that your text file has disappeared. Our Linux system is using a RAMdisk for the file system, that is the file system is stored in RAM. To make things non-volatile, you would use a SD card and “mount” it.

**Step 6:** Time for a light show! The Linux Kernel runs in an area of memory called the “Kernel Space” and likewise the user applications run in “user space”. The memory management unit prevents the user from accessing the Kernel space, but the Linux Kernel has access to all memory. Within the Kernel space there are a set of modules known as device drivers which are specific to the hardware and are basically functions that provide a bridge between the Kernel and the low-level hardware registers. The device drivers themselves are set up in a way so that they mimic file type I/O using readfile/writefile type commands. The “sys” directory is a virtual file system that provides access to drivers that are normally only allowed to be accessed by the Kernel. In our case we are going to access the LED driver. Type “cd /sys/class/leds” then return. Use “ls” to show the directory contents. You will see four “green” user led “files”. Our LEDs are blue! Enter “cd beaglebone\:\green\:\usr3” and then “ls”. You will see a list of functions that can be used to change settings or access information. Enter “echo 1 > brightness”. This should turn on LED3. Enter “echo 0 > brightness” to turn it off. The “echo” command is usually used to send things to the standard console, but using “>” redirects the item.

**Step 7:** To make the LED flash we will use the “trigger” and “delay” entries. Enter “echo timer > trigger”. We now have a flashing LED! Enter “echo 50 > delay\_on” and then “echo 50 > delay\_off”. The LED should now be flashing, 50ms on and 50ms off. Enter “echo none > trigger” to go back to the default settings.

For the remainder of the lab we will look at three built in language options that we can use to perform useful stuff on the Beaglebone board. Three more ways to flash LEDs!

**Step 8:** The first language option is the standard Linux scripting language called “Bash”. It is a “batch” type scripting language that uses the console commands. Download the file “bashLED.txt” from the Wiki and open it in a text editor to view the contents. On the Beaglebone type “nano bashLED” and then enter. This will launch the nano editor. Then in Teraterm select “File > Send file” then select “bashLED.txt” and then “open”. This will download the text into your nano editor. You will then need to edit the file to fix up a few formatting issues. Save when complete and check it using the “more” command. The first line of the file “#!/bin/bash” indicates the location and name of the interpreter to be used. The “#” indicates that the file is an executable.

**Step 9:** Enter “ls -l” to see the permissions. You will notice that your bash file does not yet have executable status. Type “chmod ugo+x bashLED” and then return. Check the permissions to ensure you now have executable status.

**Step 10:** To run the script type “./bashLED on” and then return. This will turn on the LED. Then try “./bashLED flash” and finally “./bashLED off”. If you are not logged on as “root” then you will need to prefix the command with “sudo”.

**Step 11:** The next language to try is “Python”. This is also an interpreted language. Type “nano pythonLED.py” and then download the text file “pythonLED.py.txt” from the wiki. Fix up the text, save it and give the file executable permission.

**Step 12:** To run the script type “./pythonLED on” and then return. Try “flash” and then “off”.

**Step 13:** The final language we will try is “C”. This is a compiled language and the Beaglebone, like most Linux systems, has a “C” compiler onboard. Type “nano helloworld.c” and then download the text file “helloworld.c from the wiki. Fix up the text and save it. To compile the file and create a binary executable type “gcc helloworld.c -o helloworldc” and then return. Type “./helloworldc” to run it.

**Step 14:** Back to flashing LEDs! Type “nano makeLED.c” and download the text file “makeLED.c” from the wiki. Fix it and save it and then to compile it type “gcc makeLED.c -o makeLEDc”. Type “./makeLEDc on” to run it. Try the “flash” and “stop” options.

**Step 15:** Have a good look at the function “writeLED” in the C file. You should notice that file I/O type functions are used. The LED3\_PATH can be considered as a “handle” to the device and it specifies the device driver.

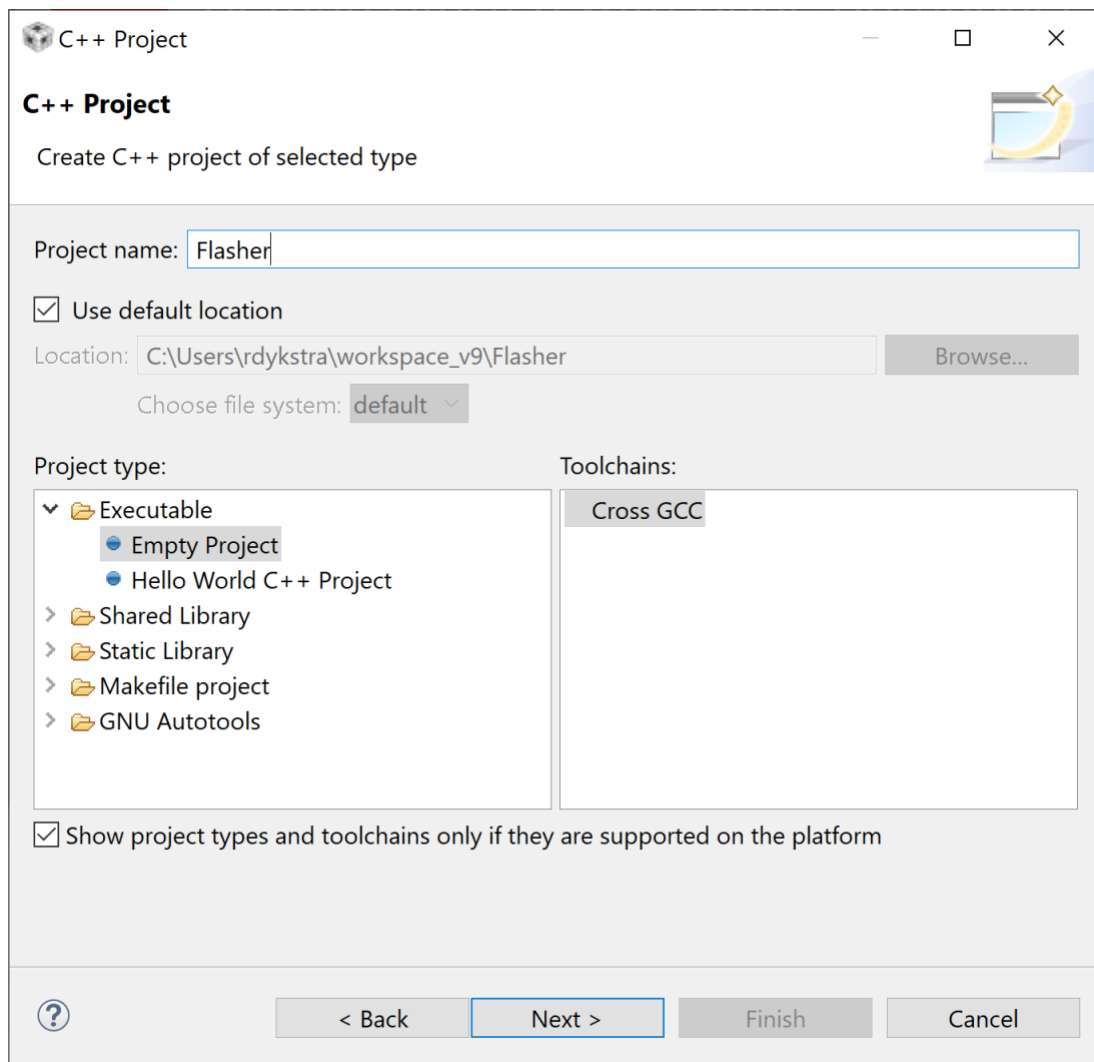
**Step 16:** For those of you who like a challenge, write a “C” program that inputs a character from the terminal and then displays the lower 4 bits on the 4 LEDs.

# Lab Session 7

## Cross compiler IDE and GNU debug

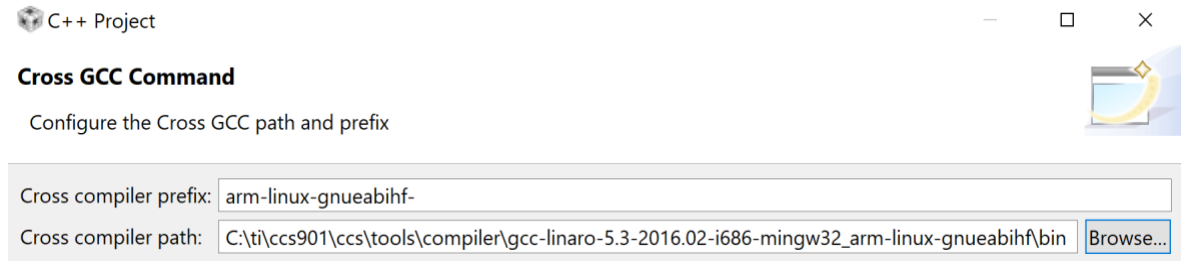
For serious code development using an editor like “nano” and onboard compilation is not suitable. Also, we would like to have “JTAG” like debugging of our applications running on the Linux system. This lab will explore both of these by showing you how to set up a “professional” development environment.

**Step 1:** Launch Code Composer Studio and start a New > Project. Select “C++” project then next. Select “Empty Project” and “Cross GCC” and give your project a name. Then Next.

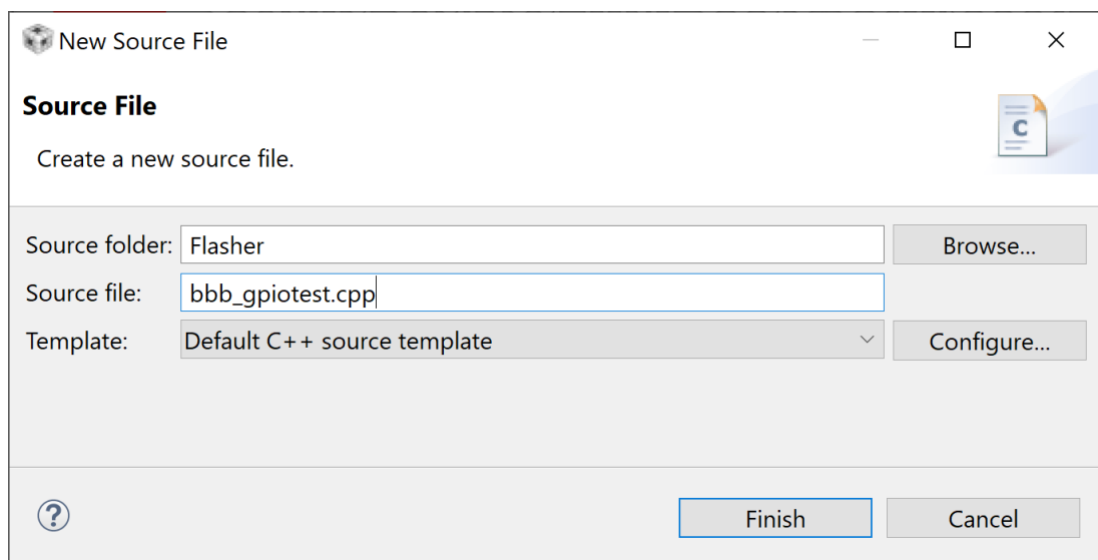


Select all configurations on the next page and then next. Now we have to set up the toolchain.

Set the Cross compiler prefix to: arm-linux-gnueabihf-, and the Cross compiler path so that it points to the \bin folder of the Linux GCC tool: gcc-linaro-5.3-2016.02-i686-mingw32\_arm-linux-gnueabihf. Note: your version may be different from ccs901.

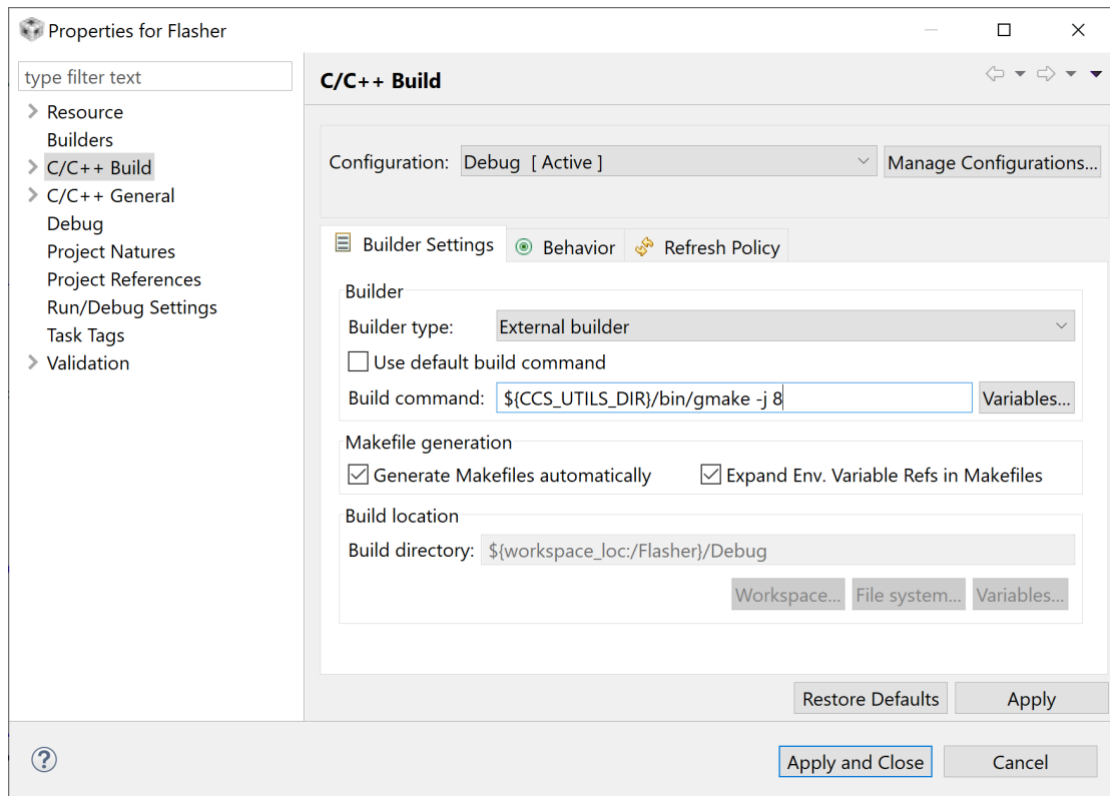


**Step 2:** Create a new C++ file and give it a name with a .cpp extension. File > New > Source File.

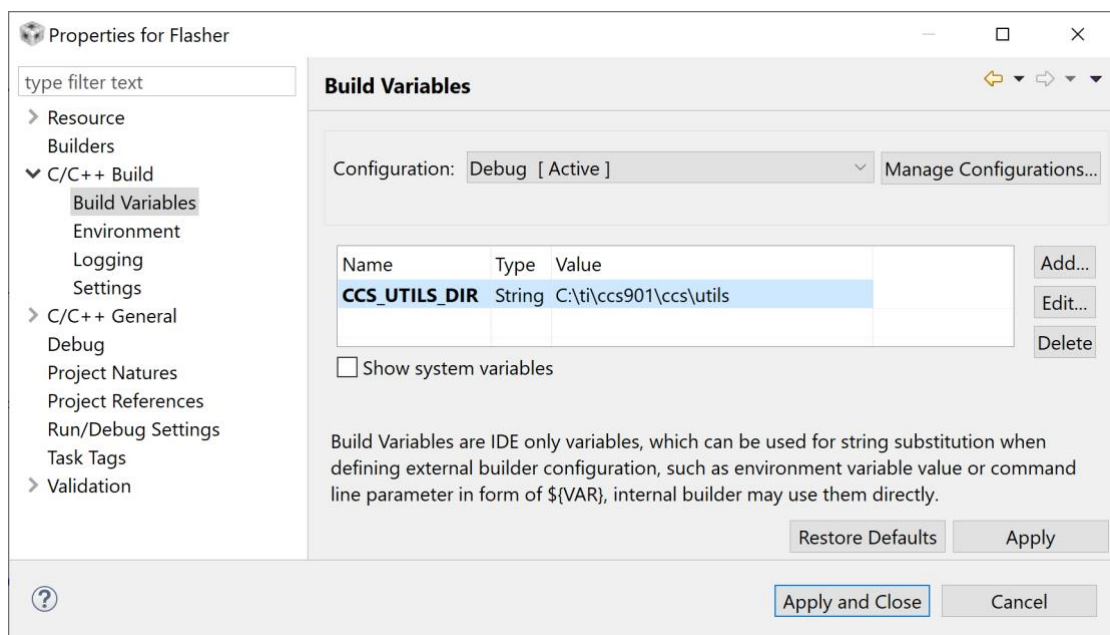


**Step 3:** Download “bb\_gpio\_test.cpp” from the wiki and copy the contents into your C++ source file and then save it.

**Step 4:** Go to Project > Properties > C++ Build > Builder settings. Deselect “use default build command” and then insert “\${CCS\_UTILS\_DIR}/bin/gmake -j 8” into the “Build command box.”



**Step 5:** Expand the C/C++ tab and then select “Build Variables”. Add the variable “CCS\_UTILS\_DIR and paste in the directory of CCS\utils. Then “Apply and Close”. Again, your version of CCS may be not be ccs901.



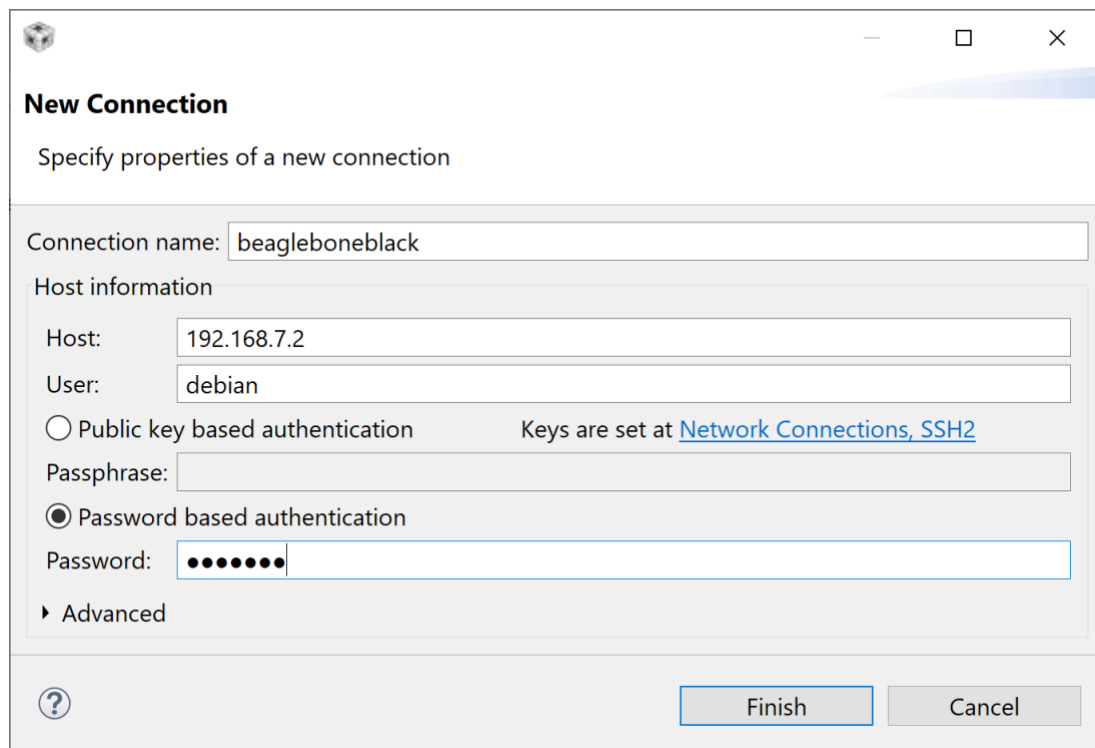
**Step 6:** Build the project and then you will see a Linux native executable.

**Step 7:** Connect the Beaglebone board to your computer via the USB cable only!

**Step 8:** Give it some time to boot and then check it is OK by looking for the new drive to appear. Then launch a web browser and go to the IP address location: 192.168.7.2. You should see a page load up.

**Step 9:** Launch Putty, select SSH and enter the IP address. Login as debian:temppwd. You should have a Linux console. This shows that all the connections are working. The USB connection is mimicking an Ethernet network connection. This is provided by the Remote-NDIS driver.

**Step 10:** Now we will run your application on the Beaglebone board in debug mode. Select Run > Debug Configurations and then right click on “C/C++ remote application” and then select “New Configuration”. Click on the “New” button in the “Connection” area, choose SSH and OK. Enter the values as per below (password = temppwd):



**New Connection**  
Specify properties of a new connection

Connection name:

Host information

Host:

User:

Public key based authentication      Keys are set at [Network Connections, SSH2](#)

Passphrase:

Password based authentication

Password:

▸ Advanced

**Step 11:** Click on the “Browse” button and select the path /home/Debian/bin. The tool will add your project name after the path name. Here you are actually browsing the file system of your board using SSH.



**Step 11:** Click on the “Arguments” tab and type “flash”. That is the command we want passed to our program when it is executed. Later on try “on” and “off”.

**Step 12:** Click on the “Debugger” tab. We now need to define the debugger client that installed on the host computer. Click on the “Browse” button inline with “GDB debugger” and select the path to the file “arm-linux-gnueabi-hf-gdb.exe”. It will probably be:  
C:\ti\ccs901\ccs\tools\compiler\gcc-linaro-5.3-2016.02-i686-mingw32\_arm-linux-gnueabi-hf\bin. Click “Apply”.

GDB debugger:	C:\ti\ccs901\ccs\tools\compiler\gcc-linaro-5.3-2016.02-i686-mingw32_arm-linux-gnueabi-hf\bin\arm-linux-gnueabi-hf-gdb.exe	Browse...
GDB command file:	.gdbinit	Browse...

**Step 13:** Click on the “Debug” button. Some things will happen and you will see a message in the “console” window.

```
Last login: Mon Oct 7 08:15:24 2019 from 192.168.7.1
gdbserver :2345 /home/debian/bin/Flasher flash;exit

debian@beaglebone:~$ gdbserver :2345 /home/debian/bin/Flasher flash;exit
Process /home/debian/bin/Flasher created; pid = 1657
Listening on port 2345
Remote debugging from host 192.168.7.1
```

You will also notice that the GUI will change to a debug perspective very similar to what we had for JTAG debugging. Select the green “Resume” button to run your program. Go back to the C++ perspective to launch another debug session or to also make changes to your code.

**Step 14:** Use the “Step Over” button to step through your code. Also launch debug with different commands. Try setting some breakpoints in the source code by double clicking on the line numbers.

**Step 15:** Now that you have a nice and professional developer environment you should modify the code so that the LEDs make a “Cylon eye”.

## **Labs 6 and 7 reports**

Lab sessions should be written up as a single report that discusses the key concepts as well as answering the questions. More specifically, for labs 6 and 7, please address the following:

Lab 6: Briefly describe the purpose of an operating system and some of the simple programming tools you would expect an operating system to have.

Lab 7: Explain what a 'cross compiler' is and the advantages it has over using the simple inbuilt compiler tools.