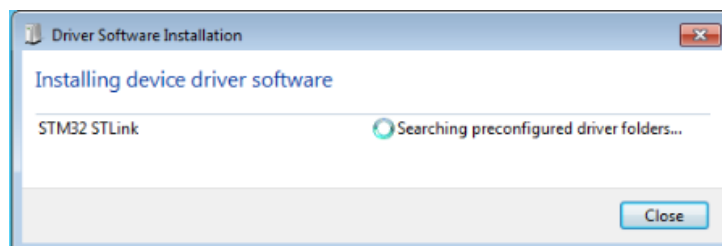# Chapter 1

Setting Up Eclipse

1.1 Configuring Eclipse on the UCT Computers

Eclipse is fully configured on the UCT campus computers but there are some steps that you need to remember before getting started. First plug in your STM32 board, Windows should start searching for a pre-configured driver once it's plugged in.
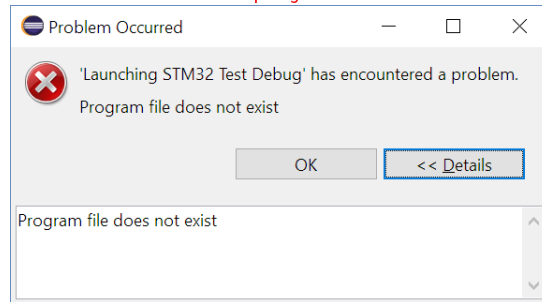

Windows should eventually find the drivers and you're ready to go. Launch up Eclipse, it is important you leave the default workspace as **C:\Workspace**.
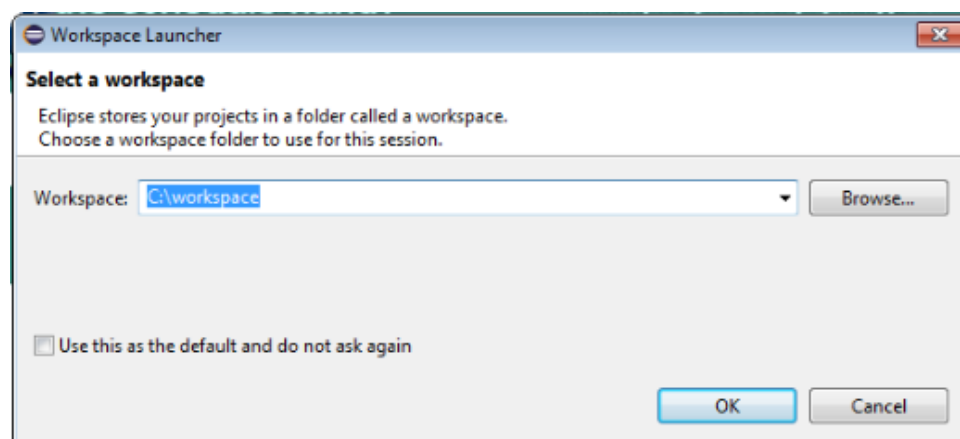
Then wait for **Eclipse** to load up, then create a new **STM32F0xx C\C++ Project**. The compiler will load up all the relevant files and put out a pretty empty generic template. First off, build the project so you have a set of binaries (.ELF files) to work with. You Build a project by following **Project → Build All** ,or selecting **Build Project** by right-clicking on the project file in the *File Explorer window*.

To run the code on your STM32 board, simply **Debug** the project using the **GDB OpenOCD Debugging** configurations. The UCT computers will auto detect the project you are working on (only if you've already built it). Conveniently, the config for **OpenOCD** is auto filled out already so you can debug your code from the **Debug Perspective**.

> If you receive an error saying **"Program file does not exist"** then you haven't pre-built your program. For the Eclipse debugger to correctly fill out the default configurations for debugging you need to have built the project first.
>
>
>
> **Build** your project and then in the debug configurations menu, under the C\C++ Application select to **Search Project** and chose the .ELF file. You should be able to debug now

1.2   TL;DR

You might have already used your computer with your STM32 board in previous courses and you just want to get eclipse working. This is the itemized process, if you run into any problems you can use the rest of the guide as reference.

1. Download Eclipse MARS for C/C++:
   http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/mars2

2. In Eclipse, add the repository for GNU ARM ECLIPSE Tools:
   Help → Install New Software → Add
   Then in <u>name</u>, fill in **GNU ARM ECLIPSE** and paste the P2P repository URL for the toolset in the <u>location</u>; `http://gnuarmeclipse.sourceforge.net/updates`
   Finally,tick and install **GNU ARM C/C++ Cross Development Tools**

3. From the same **Install New Software** window, add the P2P repo for the MARS CDT:
   `http://download.eclipse.org/tools/cdt/releases/8.8.1` and install:
   Optional Features → C/C++ GDB Hardware Debugging

4. Create a new C project;
   File → New → Project → C/C++ → C Project
   Click **Next**, choose a name for the project and make sure to select the **STM32F0XX C/C++ Project** in project type and click **Next** all the way though.

5. Once you've made the project, build it using the **Internal Builder** which is selected by **Right-Clicking** on the project;
   Project → Properties → C/C++ Build
   Then select **All Configurations** and choose the **Internal Builder**. Finally **Right-Click** again on the project and choose **Build**.

6. Once the project has been built, configure the Debugger by **Right-Clicking** on the project;
   Debug As → Debug Configurations...
   Double-Click on **GBD OpenOCD Debugging** to create a new config, then in the <u>Debugger</u> tab, fill in the location of your OpenOCD version (the 64-bit one), mine is:
   C:\Program Files\OpenOCD\bin-x64\openocd.exe. For the OpenOCD config options use:
   -f interface/stlink-v2.cfg -f target/stm32f0x_stlink.cfg
   And for the <u>GDB executable</u> just use:
   arm-none-eabi-gdb
   If you have problems with this, see Section: 1.4

7. All thats left is to **Apply** those settings and **Debug**. Each time you alter the **main.c** file of the project you'll need to rebuild it and then debug it.

1.3   Configuring Eclipse for the STM32

The following is a guide to configuring the **Eclipse IDE** for Windows to debug the STM32F051 UCT Development board; I understand it is quite lengthy but it covers many details that are important when developing for the STM32. If your **Eclipse** is already set up, skip ahead to **Section 1.5 :Perspectives in Eclipse** to see how to navigate around the Eclipse environment.

After installing and configuring Eclipse there is a section explaining Eclipses' functionality and how to get the most out of your STM32 development board. This guide was written using Eclipse Mars 2 CDT 8.8.1, if you follow the steps EXACTLY it will work.

1.3.1   Installation

What you need to install and the order in which you ought to do it.

gcc-arm-none-eabi

As with regular assembly development for the STM32F051 (hence forth the STM32), the standard arm development tools are needed to assemble our written source code (main.s) into an Assembly coded object file (main.o) and finally to link that object file to the particular memory addresses of our target processor (main.elf). So download and install the **GCC-ARM-NONE-EABI toolset** from Vula.Otherwise get it from the official site [1].

Remember that this is a 32-bit version of the development tools consisting of 32-bit executables. Even if you are running a 64-bit system you will still be using these tools. Once the files are extracted you will be prompted to **"Add path to environment variable"**
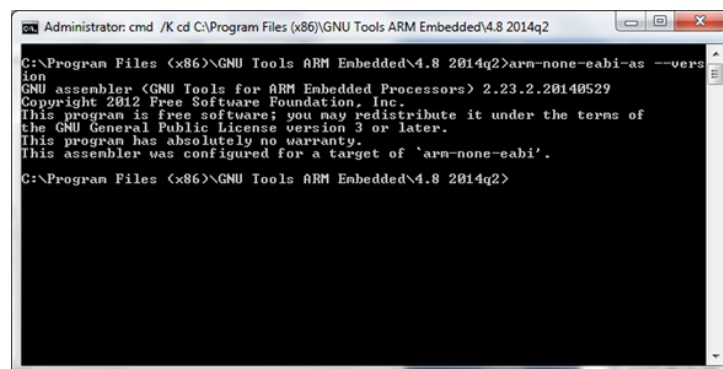

---

[1]https://launchpad.net/gcc-arm-embedded
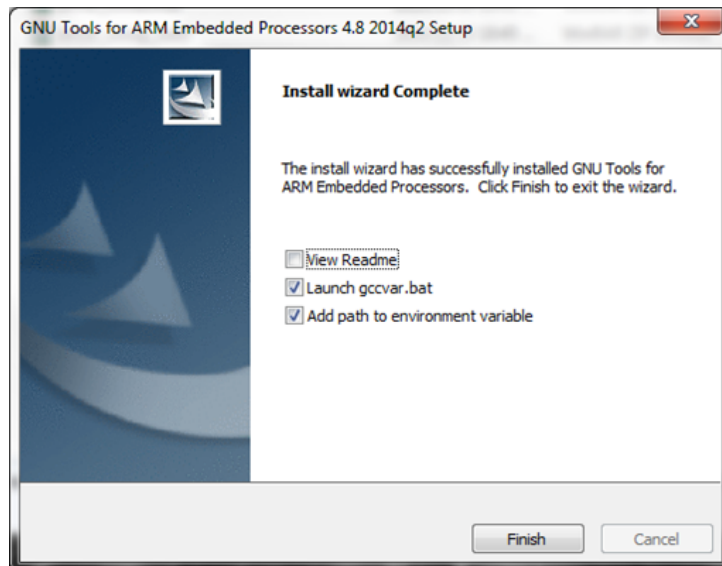
You'll want to select this parameter, it adds the location of all the .exe's to your Windows Path Variables, in advanced system settings, so they can be run from any location without having to navigate to **C:\Program Files (x86)\Gnu Tools Arm Embedded\...**

You don't need to worry about the Readme or GCCVAR.bat, so deselect both and finish the installation. Included in the **GNU Tools** we've just installed are debugging, compiling and linker tools provided specifically for Arm development, its' an open source tool chain which you can read about on the official GCC-Arm-Embedded website[2]. Just to ensure that it is installed correctly (and the system path has been added to your environment variable correctly), from your command line run;

$ arm-none-eabi-as -- version

It should produce some details about the current GNU assembler version (this test is applicable for any of the GCC Arm tools; **ls**, **gdb** etc ...)

A useful trick to remember is that, within any file in windows, pressing the **SHIFT key** and **right clicking** inside the folder gives you the ability to open a command window (CMD) at the files current location. This isn't really important but it might come in handy.

---

[2]https://launchpad.net/gcc-arm-embedded
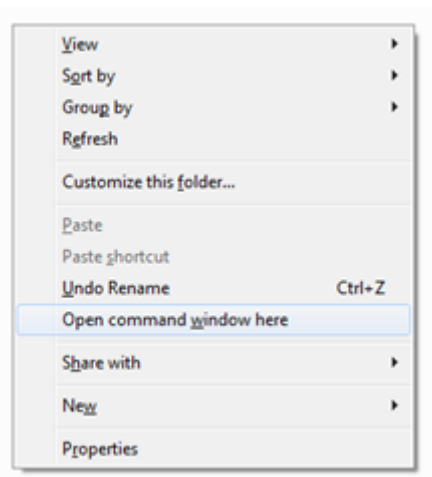
If you receive an error in which the command is unrecognized it means you either have not installed the tools correctly or the environment variable path has not been appended.

To add the path variable you must navigate to:
   Control Panel → Advanced System Settings → Environment Variables
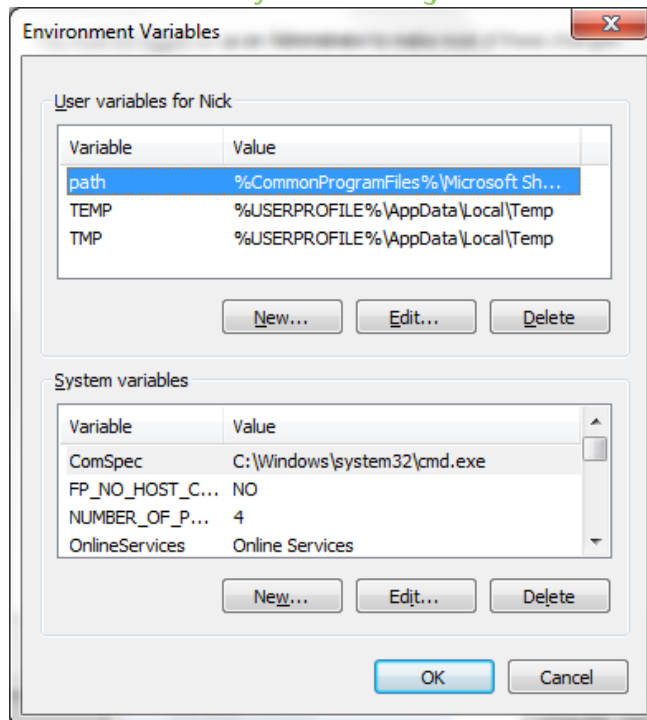
We want to edit the **path variable** , in the variable value paste (the semi-colon is VERY IMPORTANT):
   C:\Program Files (x86)\GNU Tools ARM Embedded\4.8 2014q2\bin;
and then accept this by pressing **OK** until you are out of the system properties tab. Depending on where you installed your Tool Chain to be, this location may change so find the **bin** folder within that location and add that value to the path variable

Install OpenOCD

Next we need to install **OpenOCD**, the **on chip debugging utility** that lets us upload our code onto our target STM32. The debugging process takes place between the ST-Link device and the target STM32. What you may not realize is that the ST-Link is actually another specially programmed STM32 microcontroller. The ST-Link receives code from the computer, pretty much line by line, it then prepares our target STM32 to receive new code to be flashed to its memory by pulling some pins high/low in a particular order and finally it feeds our code to the target micro via a communication standard called JTAG.

Download <u>OpenOCD-0.8.0.zip</u> from Vula or the <u>OpenOCD Website</u>[3] and extract its' contents somewhere useful like in **C:\Program Files\OpenOCD**

*Remember this location (**file name**) to which you extracted OpenOCD to, mine is called OpenOCD and is in the location listed above, and yours might be different. We will need this for Debugging with Eclipse: 1.4*

Then navigate to **C:\Program Files\OpenOCD\bin-x64** and then **rename executables**:
openocd-x64-0.8.0.exe to **openocd.exe**
*Windows 10 automatically includes executables in the file names so you would just change **openocd-x64-0.8.0** to **openocd***
This just makes things a bit easier for us. We now want to add OpenOCD's path to the Environment Variables so once again, navigate to:

Control Panel → System → Advanced System Settings → Environment Variables...

Editing the path variable, we will see there is already a value for **arm-none-eabi's** tools which was added earlier in 1.3.1, so after this we <u>append</u> or paste in the location of OpenOCD:

C:\Program Files\OpenOCD\bin-x64\openocd.exe;
(NOTE: the semi-colon is VERY IMPORTANT)

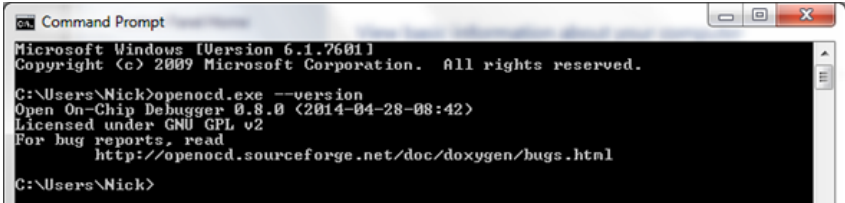> Make sure there are <u>no spaces</u> between each subsequent path variable, only semi-colons. Mine looks like:
> **C:\ MinGW\bin;C:\ Program Files (x86)\ GNU Tools ARM Embedded\ Q2\ bin;C:\ Program Files\ OpenOCD\ bin-x64;**
> If there are spaces after the semi-colons, windows won't recognize the paths as system variables and it won't work!

Finally, test the path has been added correctly by running the following in the command line:
$ openocd.exe - -version
And you should be rewarded with some more version details...


---

[3]http://openocd.org/

Install ST-Link Utility

As mentioned above, the on chip debugger makes use of the ST-Link device (we use the ST-Link V2 in particular) so we obviously need some drivers and programs for all this to work. Download the ST-LINK Utility from Vula or ST Electronics' Site [4] and install the utility. This will also install the ST-Link USB drivers as well as the device utility (choose to **Trust/Install** the STMicroelectronics Ports (COM & LPT) device software)
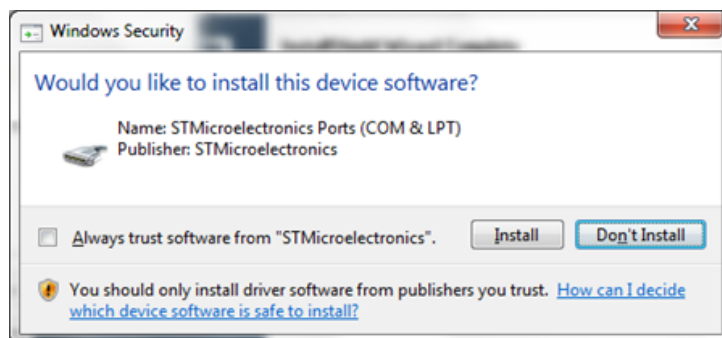
For some reason the installation attempts to install the device drivers twice, just click **Next** and it will take you through to the end of the installation. Now if you connect your STM32 board to your computer it should automatically detect the device (it will search for the drivers for a while and eventually give you a ready to use message)
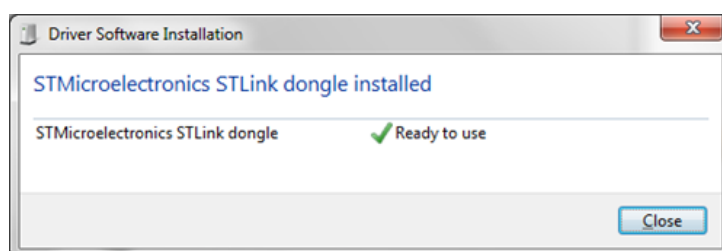
Whilst the USB drivers are crucial for us, the actual **ST-Link utility** isn't necessary but it is very useful. If you really mess up the program running on the micro or incompletely flash a program to it, it will let you erase the memory on board your STM32 micro.

Run the STM32 ST-Link Utility from wherever you installed it to and press the **Connect to the Target** button . There should now be a flashing green and red LED on your board and the table in the program will be populated with memory addresses (from 0x0800000 onwards) and hexadecimal numbers.

> If you receive an error saying **"No ST-Link detected"** just unplug and plug back in the USB cable. When the USB driver first installs it resets the COM port which the ST-Link has previously been allocated and windows may not detect it.

---

[4]http://www.st.com/content/st_com/en/products/embedded-software/development-tool-software/stsw-link004.html

A console will display some info about the current state of the device as well as the devices details. This is a powerful too which lets you manually flash programs to your micro or erase individual memory sectors but we don't need to worry about this. All you need to know is that you can erase your chip by selecting:

Target → Erase Chip
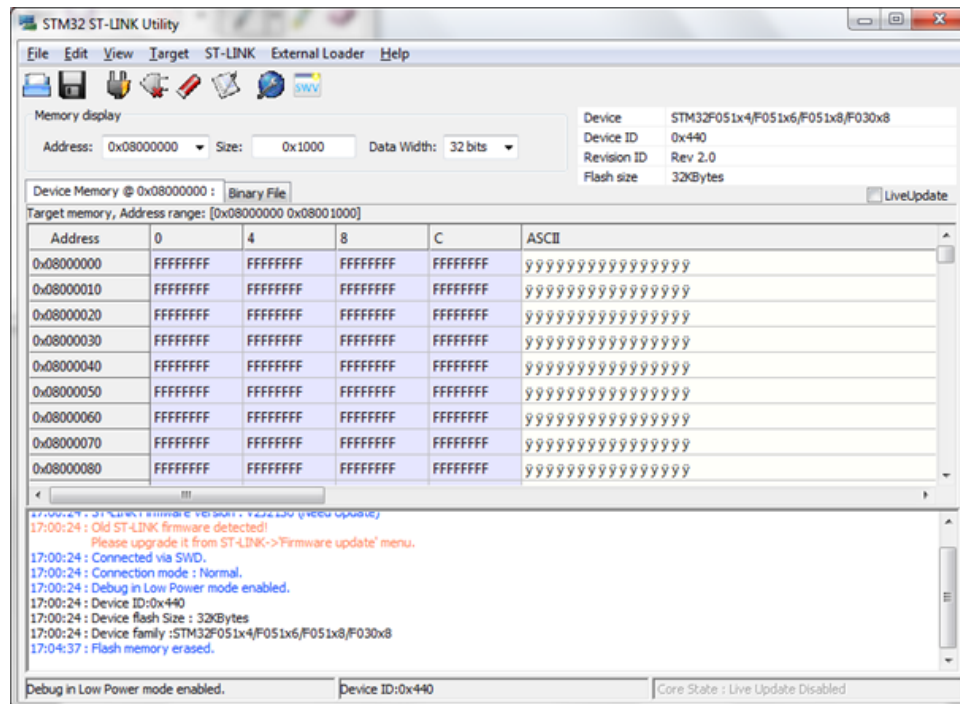
This cleans off any code you've flashed onto the micro and resets all memory vectors to **0xFFFFFFFF**. The first (and only the first) time you plug in your STM32 board you should update the firmware of the ST-Link, this just means we flash new (better) code onto the STM32 which operates as the ST-Link. To do this, you will want to navigate to:
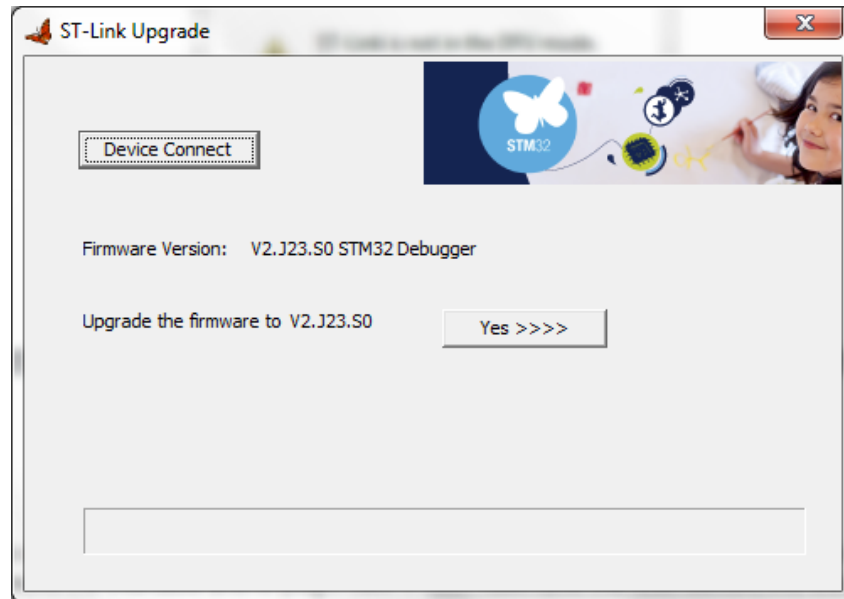
ST-Link → Firmware Update

Which opens the ST-Link Update window, first select **Device Connect**. This will give you what appears to be an error about the device not being in DFU (Device Firmware Update) mode.
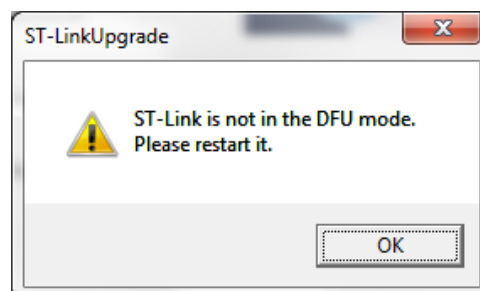
This simply means the ST-Link isn't ready to have new code flashed to it, so we have to restart it. Remove the USB cable and re-plug it back in (don't just cycle the reset button, this won't work). The LED on the ST-Link should be a constant red now.

Press the **OK button** and then select **Device Connect** again, the update utility should connect to your ST-Link now and it should display some information about what the current ST-Link firmware version is.

Then select **Yes »»⟩** to update the firmware to the newest version. The Red LED on the ST-Link will flash a few times until you receive an update complete notification.

Note: you'll only ever need to update the firmware once when you connect the STM32 for the first time. My ST-Link has already been upgraded so my current firmware version and the version of firmware it'll update to are both V2.J23.S0 but yours will be different. Even if it's not just re-flash the software again anyway.

Before continuing make sure you have disconnected the ST-Link utility from your development board. The ST-Link works through the telnet port 3333 which can only have one program interfacing with it at a time, so if ST-Link utility was connected to your board then OpenOCD would not be able to connect to it. All communication ports can only be used by one program at a time, if you tried to connect OpenOCD whilst the ST-Link Utility was connect you'd receive an error that looks like:

in procedure 'transport'
in procedure 'init'

A 'transport' error is almost always because you have another process already using the debugging port, we'll talk more about this later in Section 1.4. So if you ever receive this sort of error, make sure all previous connections have been **terminated**.

Through the ST-Link Utility you can flash compiled programs directly to the micro controller, assuming the program has been compiled correctly. To do this, select:

Target → Program

And then navigate to a corresponding .hex or .bin file in your Eclipse Project (we will discuss this later) , its usually:

C:\<Eclipse Workspace> \<Eclipse Project Name> \Debug \<Eclipse Project Name>.hex

Finally press **Start** to begin the flashing procedure. You'll notice that, if the program is compiled correctly, the flasher will know the entry point to the STM32s' memory (0x8000000)
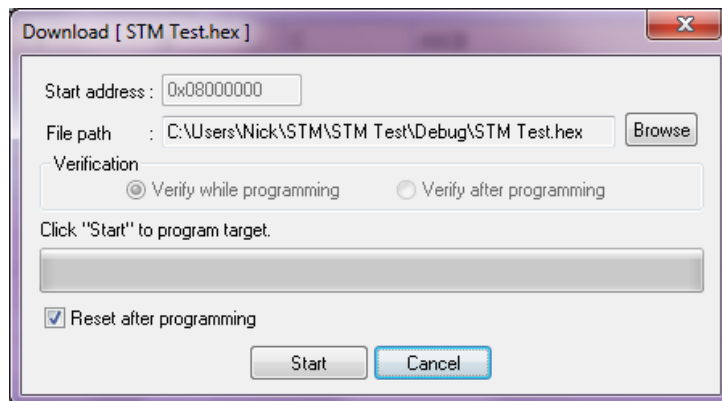
Install Java

So, first things first, we need to set up Java. Even though we will be using C/C++ to program our micro, Eclipse is an inherent Java IDE so download the latest Java Runtime Edition[5] (JRE) from the official Java website. Make sure to download the correct 64-bit or 32-bit version (depending on your operating system). Alternatively if you already have Java installed, you can update it through the control panel;

Control Panel → Java → Update → Update Now

Get Eclipse

Now this is where things get a bit tricky, Eclipse is a very versatile IDE which can be configured to do almost anything but it can get pretty confusing to configure/use. You want to download the latestEclipse C development tool package[6] (Eclipse CDT) at the time of writing; the latest build of Eclipse is Mars 8.8.1.

Eclipse is distributed as an empty package which requires a lot configuring before it does what you want it to. Extract the Eclipse CDT zip to an easily accessible folder, I recommend extracting to your **C:\file.**

> Seeing as some of you might be doing computer programming and you use Eclipse for Java development, rename this Eclipse folder to **Eclipse-arm-dev**, to differentiate it from your Java IDE.

---

[5]https://www.java.com/en/download/manual.jsp

[6]https://eclipse.org/cdt/downloads.php

Once everything is extracted, pin the Eclipse.exe to your Taskbar for easy access and start it up for the first time. On your first launch you will be requested to choose a workspace location where all your projects will be stored.

It doesn't matter where you select your workspace to be but I like to keep it within my Eclipse folder, be sure to select **Use this as the default and do not ask again** otherwise you will always be asked this.
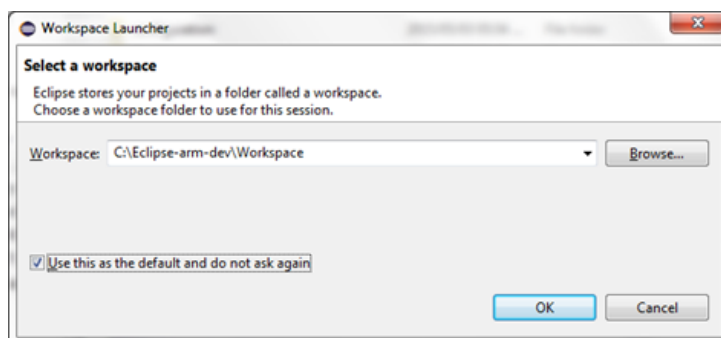
Click **OK** and wait whilst Eclipse loads up. Usually your latest project will always open up but seeing as this is our first time running the program we are greeted with a welcome message.

Note that when you are using the UCT lab computers you must not change the work space directory.

If Java is not correctly installed you will receive an error that Eclipse cannot open a Java DLL or a Config file, if this happens re-download and install Java, make sure that the Eclipse and Java versions you are using are both 32-bit or 64 bit. THEY MUST BE THE SAME!
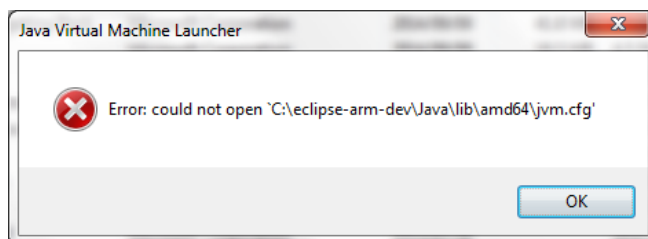
If Eclipse is already installed in your machine you don't need to do the above, just skip ahead to Configuring Eclipse, 1.3.1. If you are configuring a computer that already has Eclipse, make sure you are working with the C/C++ version of eclipse. To check this go to:

   **Help → Install New Software → What is already installed?**

In the presented list, you should see Eclipse IDE for C/C++ developers or EPP CPP Feature. These are very important as it means the Eclipse build inherently supports CDT (C Development Tools). If it lists Eclipse IDE for Java developers then the build of Eclipse is a Java version, you then need to install C/C++ Development tools if it is not listed.

This guide will work for a Java environment with the CDT package installed, you can locate it at:
`http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/mars2`

This guide is based around a Mars build which inherently supports C/C++ development, using a Java build and adding CDT to the environment isn't recommended as I can't guarantee it will work.

Configuring Eclipse

Seeing that we have already installed the CDT build of Eclipse the IDE will naturally support C/C++ compiling. We don't need to re-install the CDT package but there are some parts of it we don't already have. If you are working with eclipse and you need to add a package to the environment (not necessarily the ones we are working with now, the process is the same), you can follow these steps. You can add additional software to your eclipse build under:
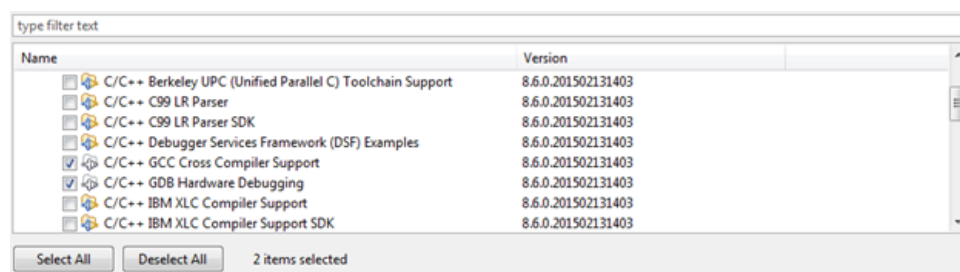
> **Help → Install New Software**
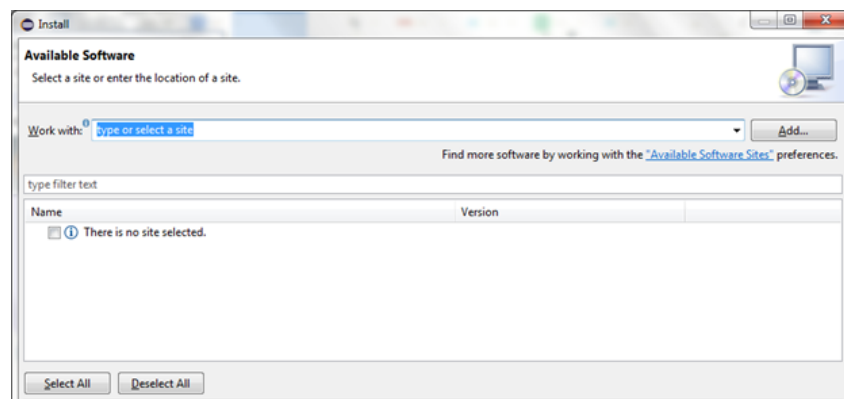
Then **Add** (adjacent to the "<u>Work with:</u>" field) a new repository to work with, call it **Mars CDT** and choose its location to be;
**http://download.eclipse.org/tools/cdt/releases/8.8.1**
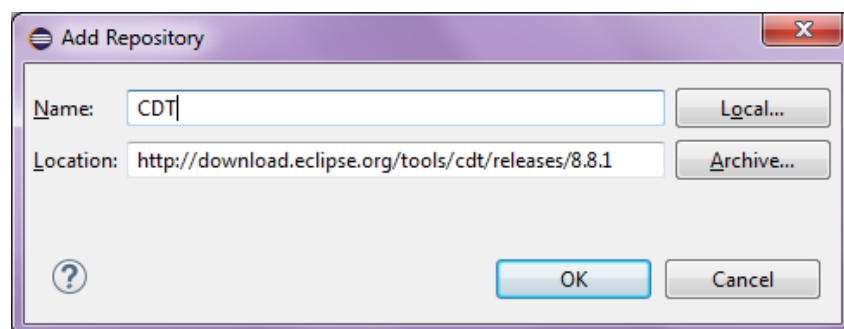
This is where you would **install/update** your CDT package if you needed to, expand the **CDT Optional Features** and select both <u>C/C++ GDB Hardware Debugging</u> and <u>C/C++ GCC Cross Compiler Support</u> then click next and wait while it calculates the items which are to be installed.

Click **Next** and **Agree** with the Eclipse Terms and Conditions (you can read them if you really want to) and then wait while Eclipse installs the new software.
 Eclipse will then download the **GDB Hardware Debug** package which allows us to use OpenOCD
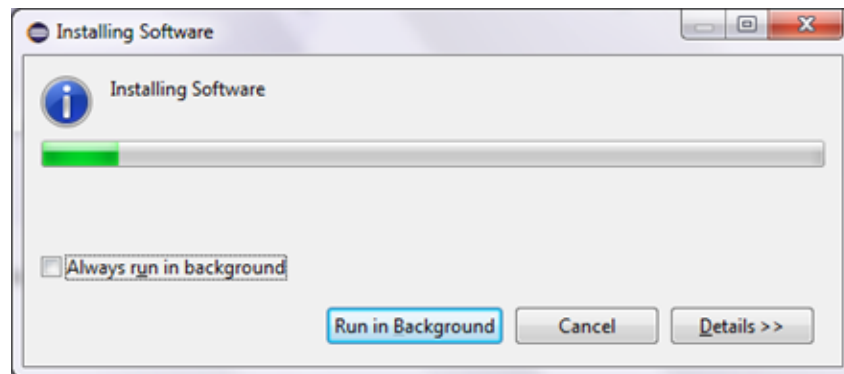
from Eclipse. At the end of this installation you will be prompted to restart Eclipse. The next step is to configure Eclipse to follow the process of compiling, making, linking, assembling and then flashing our code which would otherwise be a huge, complicated task. Thankfully Eclipse is open source and Arm processors (especially the STM32 family) are very popular for embedded systems development, so people have created packages which set up eclipse for us.

Once again navigate to the install new software window and add another repository, call it **GNU ARM Eclipse plug-ins** and use the location: **http://gnuarmeclipse.sourceforge.net/updates**. Then select all the packages and install them. This will take quite a while to download/install so take this time to think about life and reflect on the decisions you've made which have led you here to Mechatronics, or think about some of the silly things you've said or done which you regret. If you feel inclined, you can read up on the Eclipse arm tools[7].

---

[7]http://gnuarmeclipse.livius.net/blog/

Recently the Eclipse Arm Tools repository has appeared to have been inaccessible so, if for whatever reason you cannot install the ARM plugin via adding the above URL to eclipse, you can download a zip file from SourceForge[a] or alternatively the same files are on Vula[b] . Extract the zip file to any easily accessible location. I extracted it to a folder on my desktop; it should contain some .JAR files.
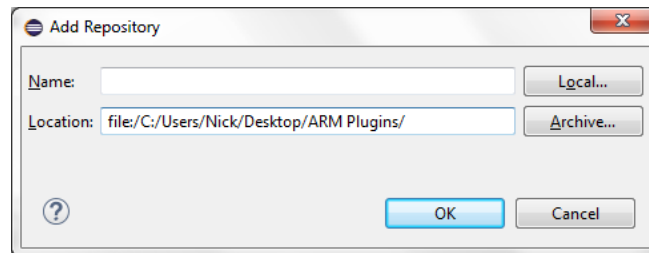
Then in Eclipse, navigate to the **Install New Software** window and click on **Add** a new repository, instead of filling in the URL to the update, select **Local** and navigate to the location of the folder you extracted the zip to.

Click **Ok** and progress through the installation. Remember if you ever want to install any additional software for the Eclipse environment and you canâĂŹt find it in the official **Eclipse Marketplace** you may have to follow this method.

[a]http://sourceforge.net/projects/gnuarmeclipse/
[b]https://vula.uct.ac.za/x/0qauui

For whatever reason the SourceForge website may be down/inaccessible from your particular line, the files can be downloaded from Vula. It is far better to use the official download through their supplied URL as the plugin will automatically update through eclipse natively.

After installing this package and restarting Eclipse (again) you'll now see, if we choose to create a new project:

   File → New → C Project

YouâĂŹll be presented with multiple project templates for all manner of STM32 micros. Select the **STM32F0xx C/C++ Project** and choose the Cross ARM GCC toolchain (you donâĂŹt have any other options) and name your project (STM32 Test), then click next.
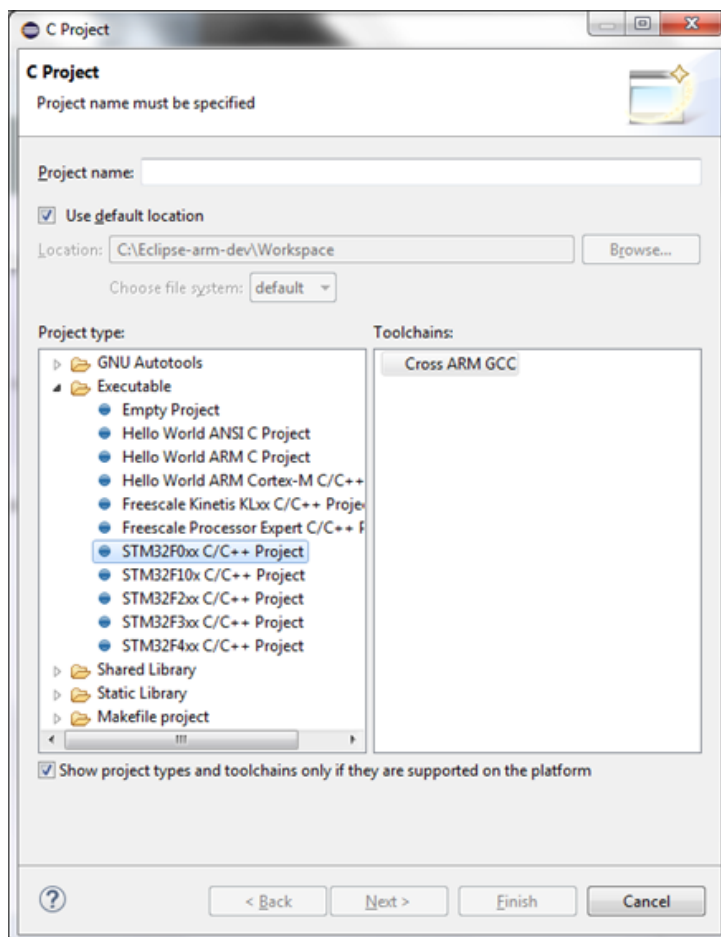
Obviously we choose the STM32F0xx C/C++ Project because our micro-processor falls into this category (STM32F051C) but if you were working with, say the STM32F10, you would choose a project catered for that. Those of you who've worked with the GT16 (the old micro used for this course, may she rest in peace) would recognize the name Freescale, you could even use Eclipse to debug programs written for the GT16!

The Discovery boards would use the same templates (but may require additional libraries to be included) depending on what peripherals are used. In this course we use the UCT STM32 Development board, which is effectively a "Bare Metal" micro-controller breakout board so we don't have to worry about additional libraries.

Following this you are presented with another window, select the Chip family to be **STM32F051** (which is our micro; obviously if we were working with a different chip we would choose the appropriate one). Leave the flash, ram and clock as default. Choose content to be **"Empty (add your own content)"**. The default is Blinky(Led) which we don't want because this is designed with the STM32F0 Discovery board in mind.

Leave the system calls as **"Freestanding (no POSIX system calls)"** which is selected by default and finally choose the Trace output to be **"None (no trace output)"**.
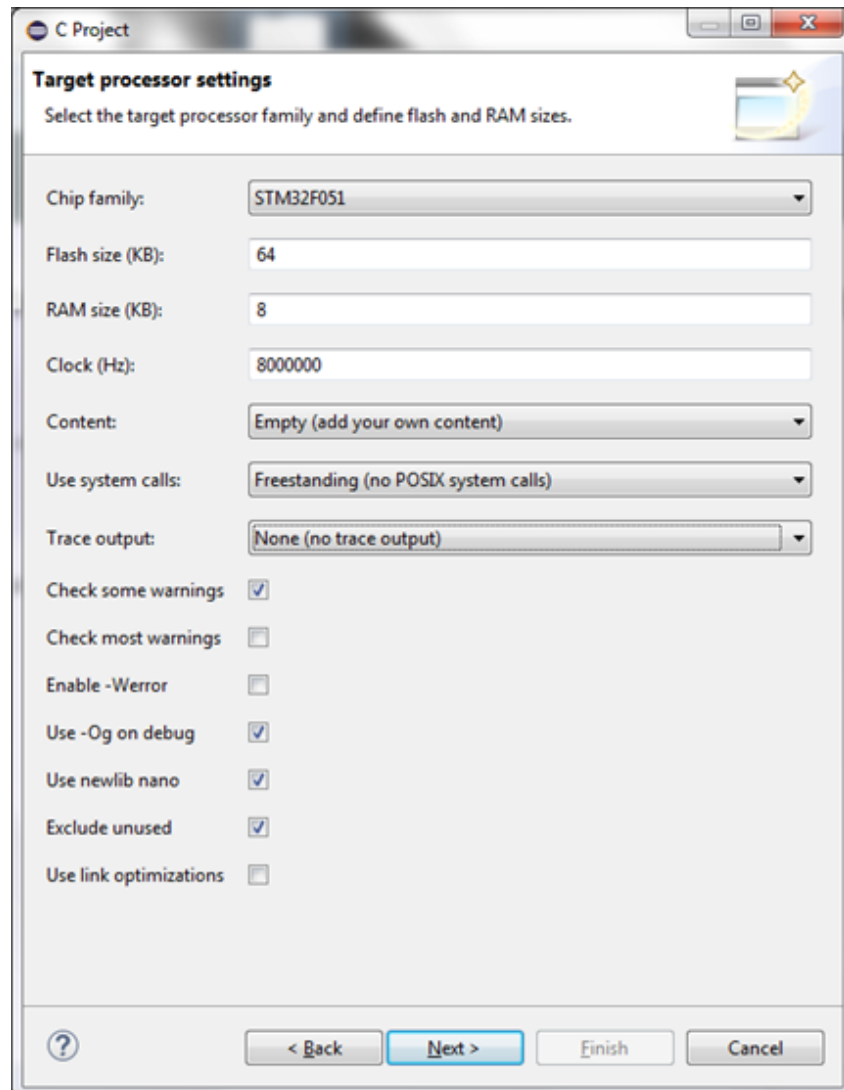

Click **Next**, leave the subsequent page as **default**, and click **Next** again. Some of these project settings may be familiar if you've compiled your C projects with a makerfile before, the Eclipse defaults are fine and I don't recommend changing them. At the configurations page there are some settings we will need to configure by default for all our projects, so click on **Advanced Settings**.

All the default settings are sufficient for our needs but we should discuss some of the settings involved as working on more complicated projects may produce problems that need to be addressed through more advanced feature configurations

This opens up a properties window with a whole bunch of settings we can change. **CLICK on C/C++ Build** (don't expand the tab, just select it) and choose to alter **"[All Configurations]"** and not just **"Debug [Active]"**. On the Builder Settings tab you can chose a Builder Type, in previous Eclipse versions an external builder was needed however this was fixed in a Luna update, ensure the **Builder Type** is set to **Internal Builder**. In **C/C++ Build → Settings**

In the Tool Settings tab there are many different subsections we can alter. Most of the default settings are sufficient for our purposes but some need to be changed. From the top down, let's discuss them briefly because they may become important later on in the course. The Target Processor subsection will allow you to change what ARM family you are developing for as well as a few other parameters, leave all of these as default as they are set when configuring a new project. This is important because some processors (like the F4 and F10 families) support multi-threaded processing, but we donâĂŹt have to worry about this.
Just to note, if your processor supported different operating systems or instruction sets you could select an alternative other than âĂŞmthumb), by default the Endianness are configured correctly.
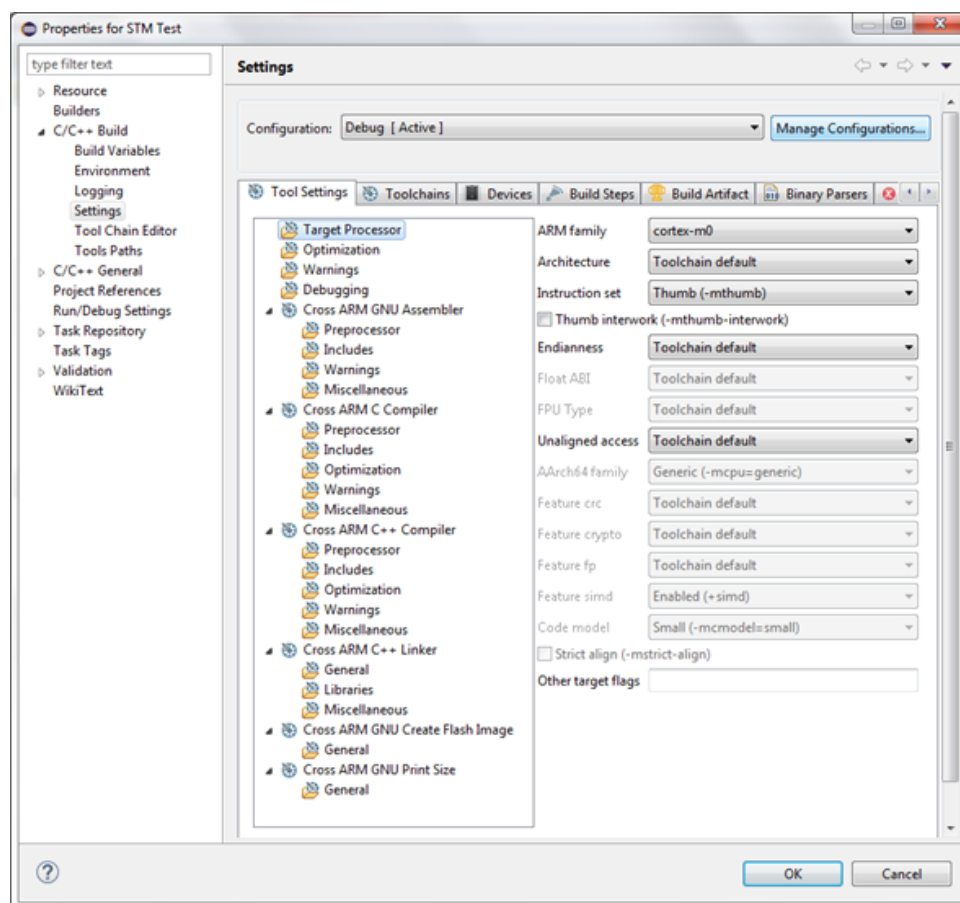
For Optimization, select an **Optimization Level** of **"None (-O0)"**. We don't want any level of code optimization just yet. Ignore Warnings and Debugging; all other tool settings can be left as their default. We don't really need to concern ourselves with their details but as you get more experienced with STM32 development in Eclipse you might want to look them up. Select **OK** and **create the new project**, this project will now be used as a template for other created projects, you can alter the settings for different processors but for now the IDE is set up to only Build your programs from a source file written in C.

In your Project Explorer Window (*more on this in citesec:Navigating Eclipse*, Eclipse provides you with a view of the Resources included in the project. We are mainly concerned with the **main.c** file in which we will write our code, it's in the SRC folder.

Right now the Eclipse IDE will happily recognize any STM32 Standard Peripheral Library command, but we will be working with <u>Hardware Registry Names</u>, such as **GPIOA -> IDR** for example. These are hard coded labels which represent the memory locations for the corresponding registers. At the compilation stage, Eclipse will recognize this label to be the **memory address** $0x48000010$. The registry labels are constructed by base memory location and the offset in the form:

BASE -> OFFSET

So here, GPIOAs' base is $0x48000000$ and the offset for GPIOx_ IDR is $0x10$. You can look up these details about the STM32F0 in the reference manual.

The compiler will throw **Symbol Unresolved errors** if you tried to include a line of code like that, so one final tool is needed, embregsysview. From:

     Help → **Eclipse Marketplace**

Search for **embsysregview** and install that plugin. This will allow the compiler to recognize the abstracted hardware names we will be using. We'll talk about how this plugin actually makes your life easier later in Section [**?**]
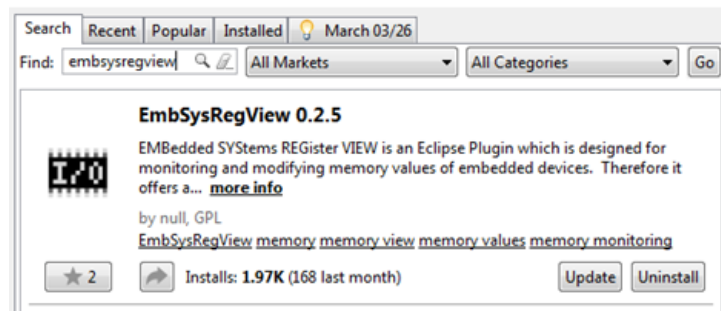

Once again, if the **Eclipse Market Place** cannot link you to the Embedded Systems Registry Viewer application or it won't install correctly then download the archived files from <u>Vula</u> or from the <u>Embsysregview Source Forge Page</u>[a] and extract them into the **dropins** folder within your Eclipse installation file **C:\Eclipse\dropins"** or something like that.

This is this standard approach to installing unofficial Eclipse packages

To load up Eclipse with these additional plugins, exit and restart Eclipse completely. To check if the plugins have been detected, select: → **Help** → **Installation Details** Then choose the **Plug-ins** tab and search for embsysregview. The search should produce two, unsigned plugins, another reason why it is better to install any plugins through the correct URL. This method is purely as a last resort!


[a] https://sourceforge.net/projects/embsysregview/files/
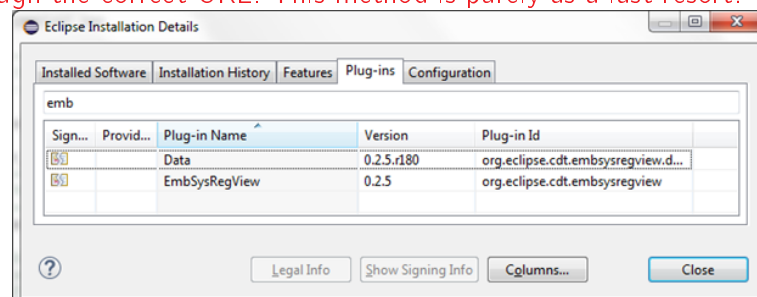
Finally we need to tell the compiler to look at this tool and use it for recognizing some of our names weâĂŹve given to the registers. Select: → **Window** → **Show View** → **OtherâĂę**

Then under **Debug**, double click **Embsysregview** (alternatively you can just type it into the search bar). Doing this loads up the plugin into our perspective, you should have a new tab open in your perspective.

[EmbSys Registers ⊠]
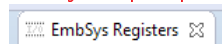
However there isn't anything there yet because the plugin can't tell what type of ÂţC we are looking at, so click on the settings box [icon] which opens up the preferences page for the **Embsysreg plugin**. Fill in all the details for our STM32 as follows:

Architecture:	Vendor:	Chip:	Board:
cortex-m0	STMicro	stm32f051x	--- none ---
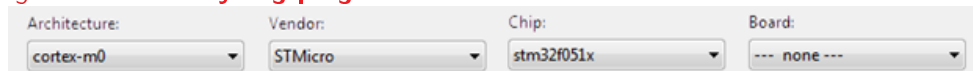
Then press **Ok** and you should be good to go, you might have to close and reopen your already open projects (if you have any) before the compiler starts to recognize the registry names.

If you were using a discovery board, you'd select that in the reg view settings. This is a different installation procedure than that followed when installing the Eclipse Arm Tools plugin locally. The dropins folder allows you to implement plugins directly without any installation. This method is slightly riskier than installing the tools through the market place because we are assuming the archived .JAR files actually work. In this case they do ...

The hardware names we using are from pointer offset names defined in the particular header file used for our micro which defines the **Type_ Defs** used, the general format is:

Peripheral -> Register

And mostly require a 3 stage **Read/Modify/Write** operation to alter its' contents. This makes life a little easier than referencing direction memory addresses.

The standard operations are:

- **Peripheral -> Register** |= Register_ BIT to toggle a particular bit (logical or

- **Peripheral -> Register** & = Register_ VALUE to rewrite the registers value (logical and)

- **Peripheral -> Register** & Register_ BIT to check for a bit being set (bitwise and, mostly inputs)

If you aren't sure about the naming, look up the particular Register or Bit names in **STM32F0xx.h**[8] and familiarize yourself with the names we will be using when referring to particular peripheral registries. Alternatively you can explore the various registers you have available through the **Embsysregview plugin**.

Eclipse should now be happy building projects which then produces a **proj.elf** (in the Binaries sub folder) file which can be flashed to the STM32 via the terminal, but that's not a particularly elegant solution. Eclipse can be further configured to allow for debugging. The default settings for debugging won't work without some additional configuration.

---

[8]System → include → CMSIS → stm32f0xx.h

To compile your project, right-click on your project and select **"Build Project"**; this should invoke the build process which, if no errors occur, should produce an .elf binary file with the same name as your project (see below).

At this stage we've made no changes to the default template created by the Eclipse Arm Plugins, it is an empty project and will do nothing if loaded onto a micro.

<div style="color:red; border:1px solid black; padding:5px;">
If you get any errors at this stage make sure you have selected the External Builder and configured it to use the installed GNU Make tool.
</div>
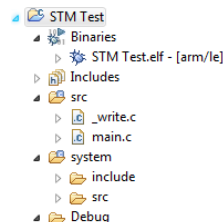
The Binaries (like the .elf and sometimes .hex or .bin files) are created during the build process, so before you build a project you've been working on, right-click on the project file and select **"Clean Project"**. This removes all pre-built binaries and allows for a fresh build. To build the project, select **"Build Project"** directly above the Clean Project instruction. Alternatively you can select a build configuration from the drop-down menu aside the Build Icon

When building a project, in the Eclipse Console, you will see outputs from the assembler programs which you would otherwise see in the terminal. All Eclipse is doing is executing the **arm-none-eabi** tools on our behalf with a whole bunch of pre-configured parameters which are defined by which processor we are developing for.

```
CDT Build Console [STM Test]
15:02:27 **** Incremental Build of configuration Debug for project STM Test ****
"C:\\Program" Files "(x86)\\GnuWin32\\bin\\make.exe" all
'Invoking: Cross ARM GNU Print Size'
arm-none-eabi-size --format=berkeley "STM Test.elf"
   text    data     bss     dec     hex filename
   1812      36     268    2116     844 STM Test.elf
'Finished building: STM Test.siz'
' '

15:02:28 Build Finished (took 253ms)
```

If you don't see this, don't worry, I'll explain how to navigate through Eclipse later.

1.4   Debugging with Eclipse

The process of debugging is first to establish a connection with the debugging device. In this case it is the ST-Link V2, which connects to our computers via a full duplex USB connection. The ST-Link is made by STMicroElectronics and supports debugging on all of their micros. The benefit of using a dedicated Debug chip will be discussed later in section [**?**]. Following a correction connection, the Debugger puts the target microchip (an STM32F051C in our case) into programming mode, then our debugging program starts sending the compiled code, in ASSM, to the debugger which in turn writes the code to the required registries and flash memory via J-TAG. Previously you've learnt how to debug the STM32 board using a few terminal applications, we are now going to encompass that into a single process using Eclipse.

1.4.1   GDB OpenOCD In Eclipse

To configure your debug settings (or in our case create a new debug configuration), right click on your project and select: **Debug As → Debug Configurations**
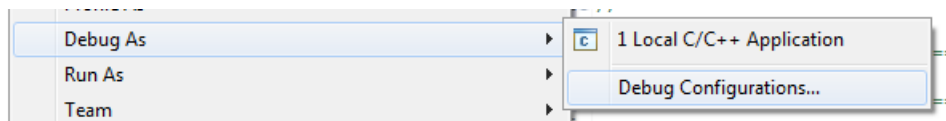
This then brings up the **Debug Configurations** window, where you can select and edit your various debug configs. Obviously we don't have any yet, so right click on **GDB OpenOCD Debugging** and select **New**.

Once you've set up these configurations, every project will require you to create a new debug config but will auto-fill the settings we want.
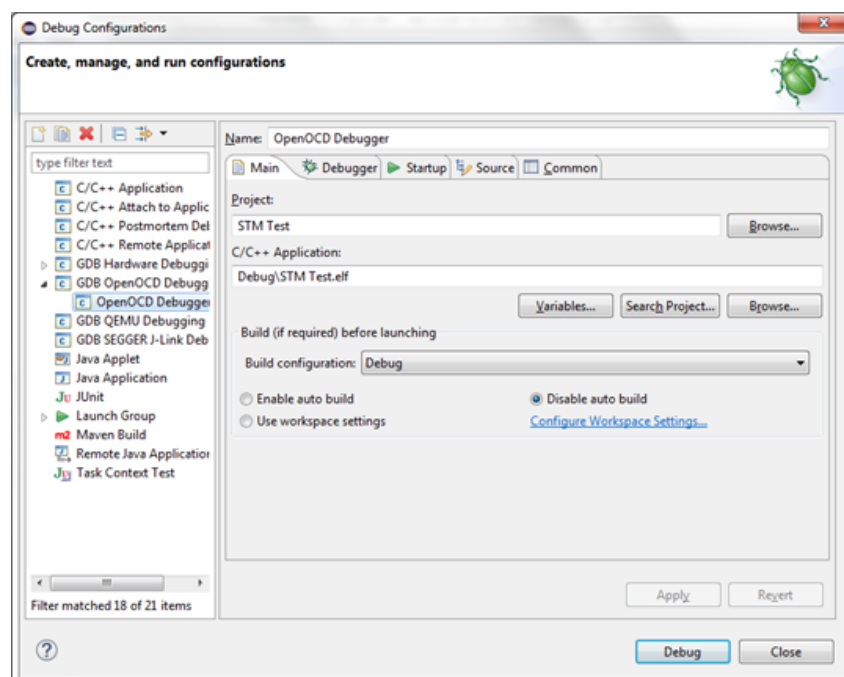
Call this new configuration **"OpenOCD Debugger"** and in the **Main** tab you should see some parameters. By default, in the <u>Project</u> field, your project name should be filled in. *If you haven't yet built your project then Eclipse won't fill in anything here. You obviously want to direct the debugger to work with the* **Project** *you are currently working on. You can select* **Browse** *to view the projects you currently have open.*
For my case, it's STM Test. Similarly in the <u>C/C++ Applications</u> field it should say:
"Debug\(Project_ Name.elf) or something along those lines.
Similarly, if you haven't built your project the program will be looking for a .elf file that doesn't exist, you can select **Search Project . . .** *to select the .elf files of the built projects you've got open.*
For each project you'll need to ensure you are going to be debugging the correct project, this is doubly important if you are working on multiple projects at one time. We can use the **System Variables** within Eclipse to automatically fill in the details we need, in the <u>Project field</u>, fill in:
$project_ name
This will point to which ever project you've currently selected to debug. And finally in the <u>C/C++ Application</u> field:
$project_ loc/Debug/$project_ name.elf
Which obviously looks in the projects directory, in the Debug folder, for an .elf binary file with the same name as the project we are working on. Next, in the **Debugger** tab, make sure that
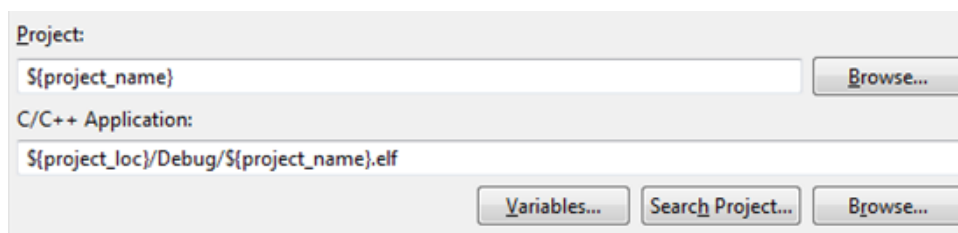
OpenOCD is selected to start locally and under the <u>Executable</u> field, click the **"Browse** button and locate where you have installed the **OpenOCD.exe** executable.
*OpenOCD was installed in Section: [**?**]*
Make sure the <u>GDB port</u> is port **3333** and that the <u>telnet port</u> is **4444**. These are the communication ports on which the ST-Link listens for communications from a host device. Next, and most importantly, in the <u>Config Options</u> field, paste the following:
<div align="center">

-f interface/stlink-v2.cfg âĂŞf target/stm32f0x_ stlink.cfg
</div>

This tells OpenOCD that we are interfacing with the St-Link, version 2, and it should use the associated configurations and secondly that the target processor is from the STM32F0x family. You can read up about why we do this on the <u>OpenOCD website</u>[9].
In the <u>GDB Client Setup</u>, make sure the Executable is:
arm-none-eabi-gdb
By default it could be set to just gdb, which we don't want. The Commands field should be filled out automatically but if it isn't fill it in as shown above. Then click **Apply**, there are no more settings in the Debugger tab we need to change.

---

[9]http://openocd.org/
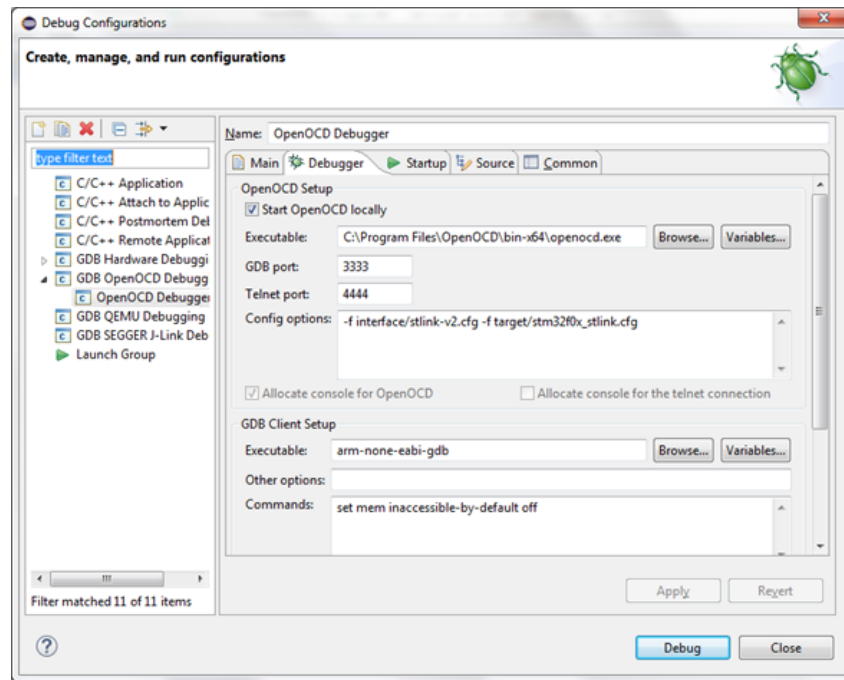
In the Startup tab, scroll all the way down to the bottom and make sure a **breakpoint** is set at main. I'll explain how to use/set up breakpoints later but they just tell the debugger to pause the processor at a particular point so we can see what memory values are set, we can't configure the processor to step through only 1 line of code at a time so we insert breakpoints. Finally, in the Common tab, select this **Debug Configuration** to Display in favorites menu and finally click **Apply**.
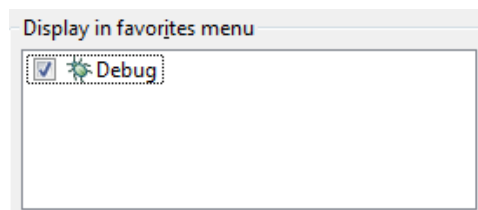
You can now press the **Debug** button; you'll be asked if you want to save some selected resources. Tick to save **"main.c"** and then select **Always save resources before launching**.
The first time you debug you'll be presented with a window asking to switch a **debug perspective**. Select yes and tick for the compiler to always remember your decision. The Debug Perspective is a different window specifically for the debugging process with a different layout.
Your first run through will be paused (remember we put a **Breakpoint** at main, pressing the **resume button** which will begin execution of the code you've flashed onto the micro. We will discuss everything that can be done from this perspective later in [**?**].

You can switch between perspectives (i.e back to the C/C++ Perspective using the **Perspectives Manager** in the top right hand corner.
We're now going to discus what **perspectives** are in Eclipse and how to use them.

1.5 Perspectives in Eclipse

Eclipse is a multi-window IDE, meaning we can view different windows for different situations, each view is called a **Perspective**. By default the **C/C++ Perspective** is always open.
You can customize a perspective to look like or do what ever you want, that is why Eclipse is such a powerful and useful IDE. You can open different tabs and add them to your perspective through:
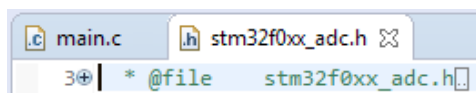**Window → Show View → ...**
The most useful part about the C/C++ Perspective is the **Project Explorer**, this is the section of the window in the far left hand side of your screen. It displays the projects you have in your working directory. You can expand the open projects and view what files are included.
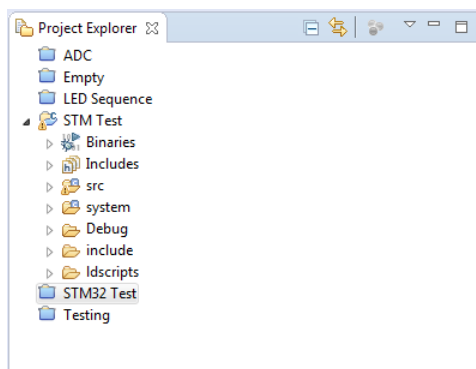
> If you can't see the **Project Explorer** then display it in the perspective by selecting:
> **Window → Show View → Project Explorer**

It isn't really wise to have multiple projects open at any one time as this may often confuse you so be sure to close a project if you're busy working on another.
In the center of your window you'll see your **Text Editor** which will most probably have the **main.c** source file open. If you want to look at any of your header files, you can open them up and switch between them by selecting the different tabs in the top part of this window.


At the bottom of your screen, the **Console Window** is open. This window shows any returns when commands are executed, there are consoles associated with the GDB and OpenOCD programs which are displayed here. When you build your project or debug it you'll see different bits of information being displayed in the console or any errors which show up with your code at the compiling stage.


All the information displayed in the console is usually to help you figure out why you're getting an error or what isn't working. It's difficult to understand initially but you'll get the hang of it.

To change between the different consoles you have at your disposal, use the **"Display Selected Console"** drop down menu.
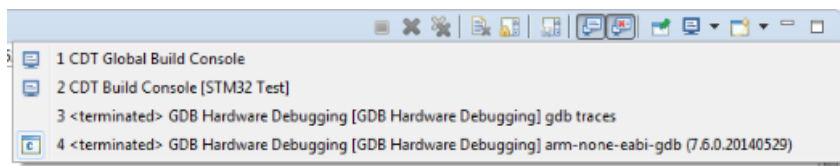

If you make any changes to your perspective view, you can right click on the particular perspective in the perspective tab and save the changes you made. There are lots of different ways you can set up your perspective to fit your needs so try experimenting with some. The windows are pretty self-explanatory so you can decide what you'll need or won't need and don't worry, you can always Reset a perspective back to its default if you don't like the changes you've made. In the perspectives tab you can open a new perspective to view the debugger perspective or perhaps another one.

For our purposes here, you'll be using the **C/C++ Perspective** and the **Debugger Perspective**.
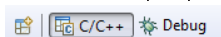
You won't be able to see the icon for the Debug Perspective until you've run a debug process.

The Debugger perspective is probably the most important one as this is what we use to execute the code on our STM32. It is a very powerful tool which we will now discuss. To launch the debugger from the C/C++ perspective select the debugger drop-down menu and choose which debug configuration you'll be using.
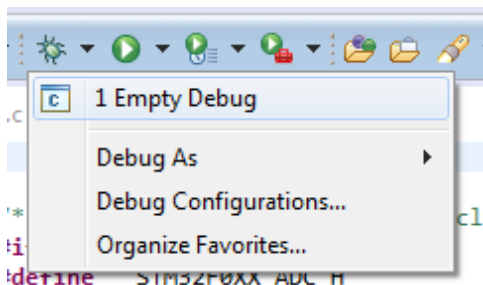
Eclipse will automatically switch to the **Debugger perspective** once **OpenOCD** has made a connection and the Debugger has access to the ST-Link. Once in the **Debug Perspective**, you are presented with a lot of windows/information. Some of it is the same as in the **C/C++ Perspective** (but they now have different functions). Starting from the top, in the **Debug Tab** you are shown a cascaded program tree of what is currently being executed by the program.

So the tree obviously shows that you are working with the OpenOCD Debugger, youâĂŹve executed the debug process with a particular binary (**Project_ Name.elf**) and that we are running OpenOCD.exe and arm-none-eabi-gdb. Under Thread # 1 is a list of "Breakpoints", breakpoints are stages of the program which, once executed, halt operation of the processor so you can inspect registry elements. By default there is only one break point (at main, seem familiar?) but we will add more later. You can set the debugger to **skip all breakpoints** ,but that isn't all that useful.

The first thing to do is **Resume** execution of your program and the processor will step through all the instructions you've programmed until it reaches another breakpoint when it will halt again. Once running, you can **Pause** your program or completely **Terminate** its execution.