



UNIVERSITY
of York

SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

Systems Programming for ARM Laboratory Scripts

Dr. Andy Pomfret

Laboratory 1

Introduction to the μ Vision IDE

1.1	Aims	1
1.2	The ARM Board	2
1.3	Introducing μ Vision	3
1.3.1	Project Files	3
1.3.2	Using the IDE	3
1.3.3	Testing the IDE and Board	4
1.4	The μ Vision Debugger	6
1.4.1	Introduction	6
1.4.2	Breakpoints, Watchpoints and Stepping	6
1.4.3	Using the μ Vision Debugger	7
1.5	Optimisation	10
1.6	Summary and Conclusions	11

1.1 Aims

In this laboratory session, you will:

- Learn about the Cortex-M4 Processor Board and the μ Vision Integrated Development Environment;
- Write some simple C code to run on the board;
- Experiment with the integrated debugger.



Before you start this lab session, extract the provided .zip file to somewhere convenient on your computer. All file-related instructions will apply relative to whatever location you've chosen.



A bug in the IDE means that the debugger will behave erratically if you attempt to work from a network drive. Ensure that wherever you unzip the folder, it's on a drive that's physically located in the machine you're using.

1.2 The ARM Board

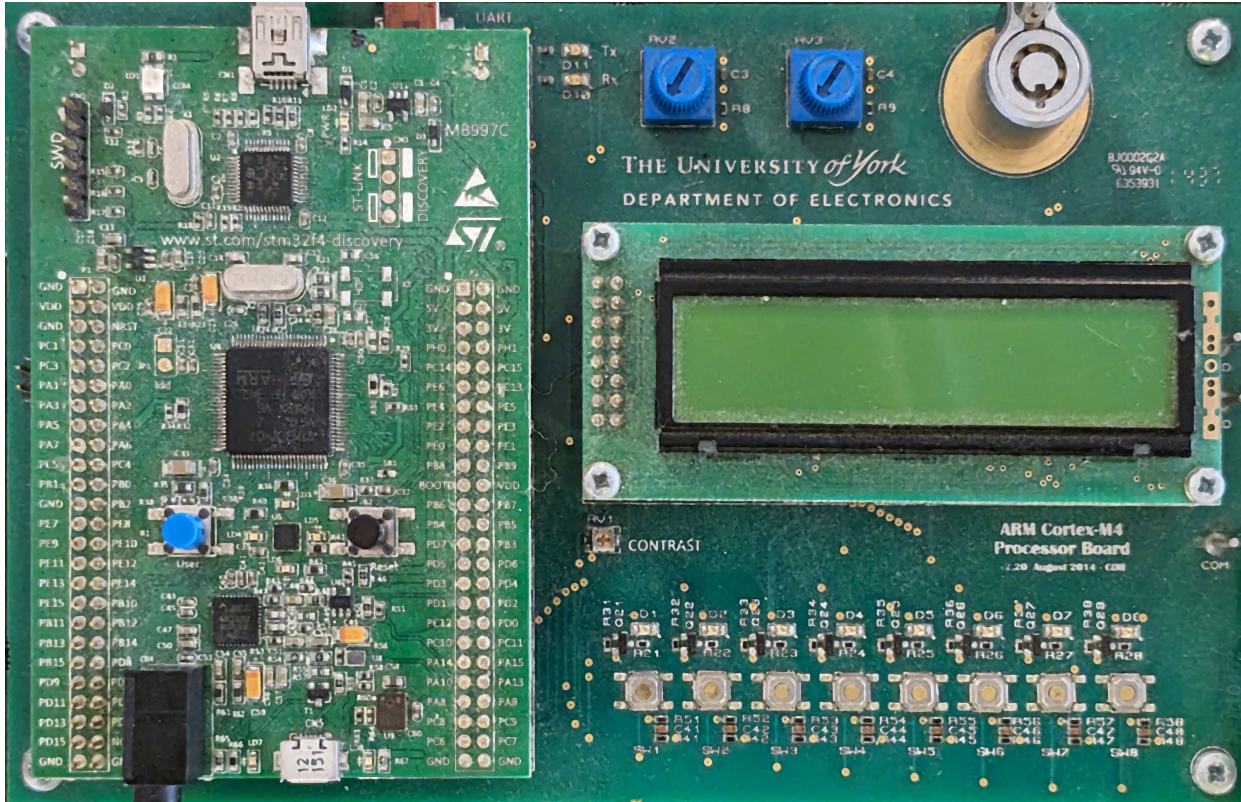


Figure 1.1: The Cortex-M4 Processor Board

Figure 1.1 shows the Cortex-M4 Processor Board. The most important features for now are:

- The STM32F4DISCOVERY daughterboard. This carries:
 - The microcontroller, an ST STM32F407VG. This has an ARM Cortex M4F processor core that can run up to 168MHz, 192KB RAM, 1MB Flash ROM, and a lot of peripherals.
 - The primary USB connection. This is used to load compiled code onto the microcontroller, and to interact with the debugger. You'll learn more about the debugger in Section 1.4.
- The USB serial connection. This is provided by a standard USB RS232 serial interface device, located underneath the STM32F4DISCOVERY. It allows for simple communication between the PC and the evaluation board, which can be useful for debugging.
- The pushbuttons and LEDs, which are connected to general-purpose input and output (GPIO) ports on the microcontroller.



Connect the board to your PC using the USB cable supplied.

1.3 Introducing μ Vision

The μ Vision Integrated Development Environment (IDE) provides you with a C compiler that can generate machine code for a variety of ARM-based target devices. In the projects you'll be using, it will be configured to target the STM32F407VG. It also contains a simulator, which can be used to run and test code without having a physical target present. And it comes with a debugger, which can be used to investigate the operation of the program as it runs – either in the simulator, or on the target device itself.

1.3.1 Project Files

For all of the laboratory sessions, project files have been provided. These can be loaded into μ Vision and contain all the settings and C source files required to start each section of the laboratory. You will be told when to load each new project file.



If you opt to create a new project from scratch, rather than loading an existing project, several of the project options will be set incorrectly. To avoid problems, always modify an existing project rather than creating your own.

1.3.2 Using the IDE



Start the IDE by opening “Keil μ Vision5”. When it loads select **Project**→**Open Project...** from the menus, and load the first project from `Introduction\intro.uvprojx`.

The IDE should now look something like the screenshot in Figure 1.2.

The most important features of the IDE are highlighted on the figure:

- The **project window** shows you a tree of all the source files that form part of the current project.
- Double-clicking any source file in the project window will load it into the **main window**. Tabs at the top of the main window will allow you to switch between open source files.
- The **rebuild** button will attempt to build all of the source files in the project.
- Any warnings or errors that are generated during the build process will be reported in the **build output window**.
- Once your code is built (compiled and linked), clicking the **download** button will send it to your target, providing the board is properly connected.
- The **target settings** button allows various aspects of the compilation, linking and download processes to be configured. We'll look more at this another time.

- Clicking the **start/stop debugger** button starts the debugger, so that the operation of compiled code can be checked a line at a time. This changes the layout of the IDE windows quite drastically. More details of the debugger and its operation will follow in Section 1.4.

1.3.3 Testing the IDE and Board

The example project you have loaded contains code that should work straight away. Before building and downloading the project, let's have a look at what the code does.

```
configClock();
configUSART2(38400);
```

These two lines set up the serial port and the LCD screen respectively. The actual code to do this is in other source files in the project, there's nothing hidden away – but you don't have to know how it works, for now.

```
while(1) {
```

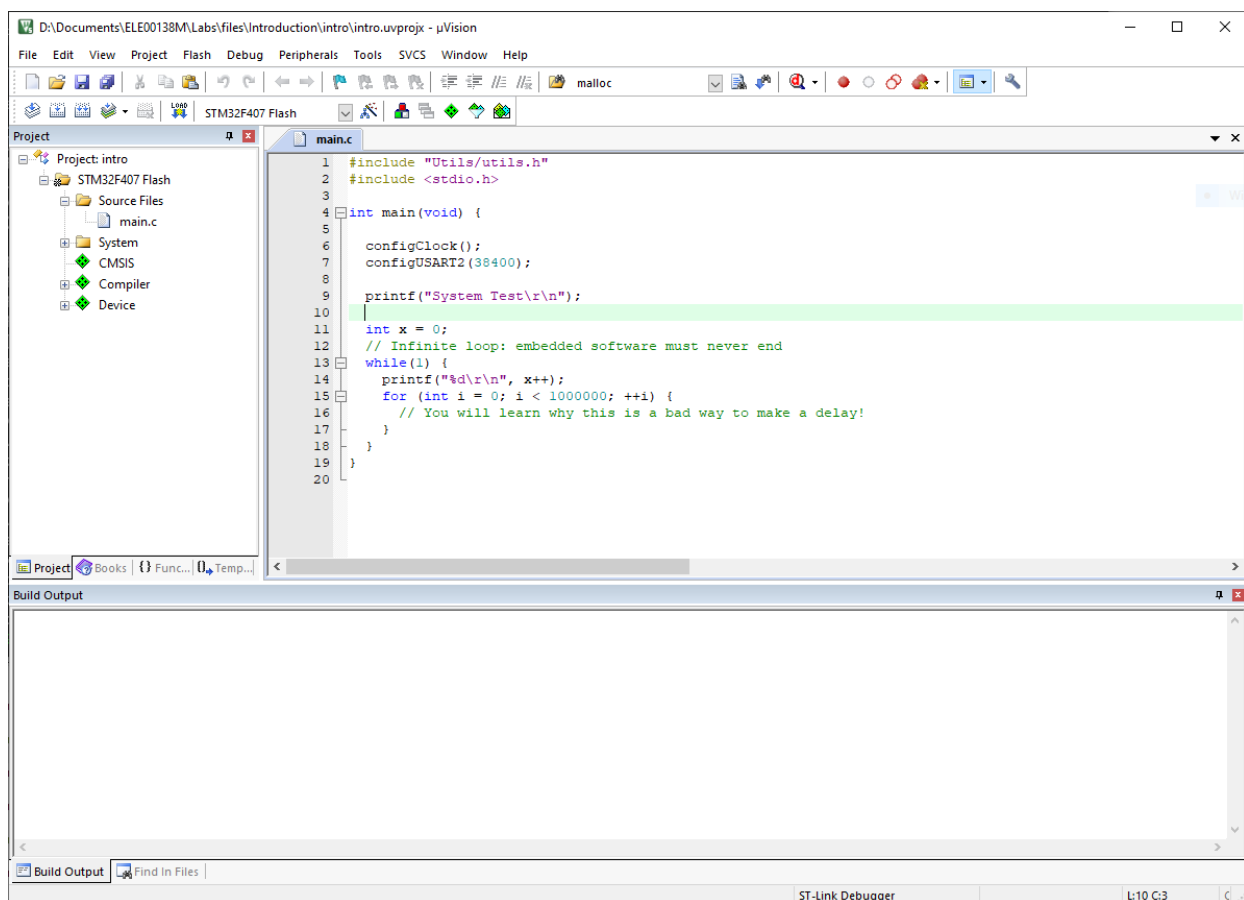


Figure 1.2: The Keil μVision Integrated Development Environment

```
printf("%d\r\n", x++);
for (int i = 0; i < 1000000; ++i) {
}
}
```

These lines will print an incrementing counter value. But there's no monitor connected to the evaluation board, so where does the text go? Well, some code in another project file 'retargets' the output of `printf()`, and sends it to a serial port (USART2). This port is connected to the USB serial adapter, which sends the data to the PC. So to see the strings being printed, we'll need to launch a program on the PC that prints out everything that's received on the serial port. More on that soon.

- ❗ Note the use of `\r\n` to end each line. These are the ASCII codes for 'carriage return' and 'new line' respectively. The ASCII standard originally required both of these characters to be present to represent the end of a line of text. For various reasons this has not always been respected across different platforms; Windows, for example, uses only `\n` in text files. Some serial terminal programs require both characters, and others don't, but you should include both as a matter of habit.

1.3.3.1 CoolTerm

To view characters that are sent to the serial port of the PC, you'll need to run a "serial console" program. We recommend CoolTerm ([direct download link](#)) because it's simple and it works, but if you have a different serial console that you like using, that's no problem.

- ⚙ To install CoolTerm, simply extract the zip file to somewhere convenient, and double-click the CoolTerm executable. No actual installation is required. Once you've run it once, you'll be able to find it by typing its name in the Windows Start menu if you wish, or right-click it and pin it to the taskbar.

To make life easier on yourself, connect the ARM board to the PC *before* you start CoolTerm. This will allow CoolTerm to automatically detect the serial port. If you forget, you can re-scan for serial ports by pressing the 'Options' button.

There are likely to be multiple serial ports on the machine you're using. There's no magic way to work out which is connected to the ARM board, unfortunately. The Device Manager in Windows is perhaps the easiest way – the device is usually called "USB Serial Port" and its port number will show up alongside it.

- ⚙ The settings for the serial port are: 38400 baud, 8 data bits, no parity, 1 stop bit. Most of these match the default settings in CoolTerm, but **not the baud rate**, so press 'Options' and select 38400 baud from the drop-down list, as well as selecting the correct port.

Now press the 'Connect' button.

1.3.3.2 Running the Test Project



You're now ready to build and run the test project.

- Build the test project by clicking the **build** button or pressing F7.
- Assuming it builds without errors (which it should!), download it to the board by clicking the **download** button or pressing F8.
- Once the download is complete, press the black Reset button on the STM32F4DISCOVERY daughter board to start the code running.

You should see that the CoolTerm window displays the text sent down the serial connection from the `printf()` lines.

If anything doesn't work as expected, or you have any questions, ask any of the demonstrators to help.

1.4 The μ Vision Debugger

As you progress through these laboratory sessions, you may find it helpful to be able to use the debugger. In this section, you will learn about its basic capabilities.

1.4.1 Introduction

The debugger in μ Vision does not work alone. The ARM Cortex-M4 core contains a large amount of dedicated debug hardware, and ST have added some of their own support logic to this for the STM32F4xx processors. (This is described in detail in Chapter 33 of the STM32F4xx reference manual, starting from page 1375, in case you're interested.) This allows the debugger to interact directly with the core and the rest of the device in several ways. The most important for now are:

- Setting *breakpoints* and *watchpoints*;
- Inspecting registers and memory contents.

The debugger also provides a *disassembly* function, allowing the actual instructions that the processor is executing to be seen alongside the source code that generated them.

1.4.2 Breakpoints, Watchpoints and Stepping

The ARM Cortex-M4 has a selection of *watchpoint registers*. These will automatically halt the processor when the pattern of bits loaded into a watchpoint register matches the pattern of bits on the address bus, the data bus, and/or a selection of control lines. These watchpoint registers can be used in three ways:

- To set *breakpoints*. These halt the processor when the processor fetches a specified instruction, meaning you can force the processor to halt at a specified point in the code.
- To set *watchpoints*. These halt the processor when a particular memory location is modified, or when a certain value is written to it. This allows you to halt execution on, for example, a particular iteration of a `for` loop by adding a watchpoint on the loop counter.
- To implement *stepping*, allowing you to step *over* any line of code, or *into* or *out of* any function.

You don't have to interact with the watchpoint registers directly, they're used by the debugger to perform all of the tasks listed above.

1.4.3 Using the μ Vision Debugger

Click the **start/stop debugger** button (see Figure 1.2) to start the debugger. You will see the window layout change dramatically.

1.4.3.1 Debugger Layout

The IDE should now look like the screenshot in Figure 1.3.

The **source window** shows your source code, just as before. The **disassembly window** shows the machine code, and associated assembly language instructions, that the CPU is actually executing. When the CPU is halted, highlighted lines in these two windows show the machine code instruction, and the corresponding line of C code, that it will execute next.

When the debugger starts, the code starts to run, but the CPU is automatically halted on the first line of `main()`, as shown by the highlighted lines in both the source window and the disassembly window.⁰

The **run** and **stop** buttons do exactly as you would expect: if the code is running, the **stop** button will cause the CPU to halt immediately, and the **run** button will cause it to resume.

The **reset** button causes a soft CPU reset, so execution is restarted from the beginning. This is subtly different to resetting the CPU by using the button on the board, which causes a hard CPU reset and may cause the debugger to lose communication with the board.

⁰Your `main()` function is not actually the first thing that the STM32F407VG runs when it is reset or powered on. There's some assembler code in a file called `startup_stm32f4xx.s`, which you can look at if you want; it sets up the oscillator, and performs various other housekeeping tasks, some of which are written in C and contained in a file called `startup_stm32f4xx.c`, before calling your `main()` function. But the debugger is set up to halt the CPU as soon as `main()` is entered.

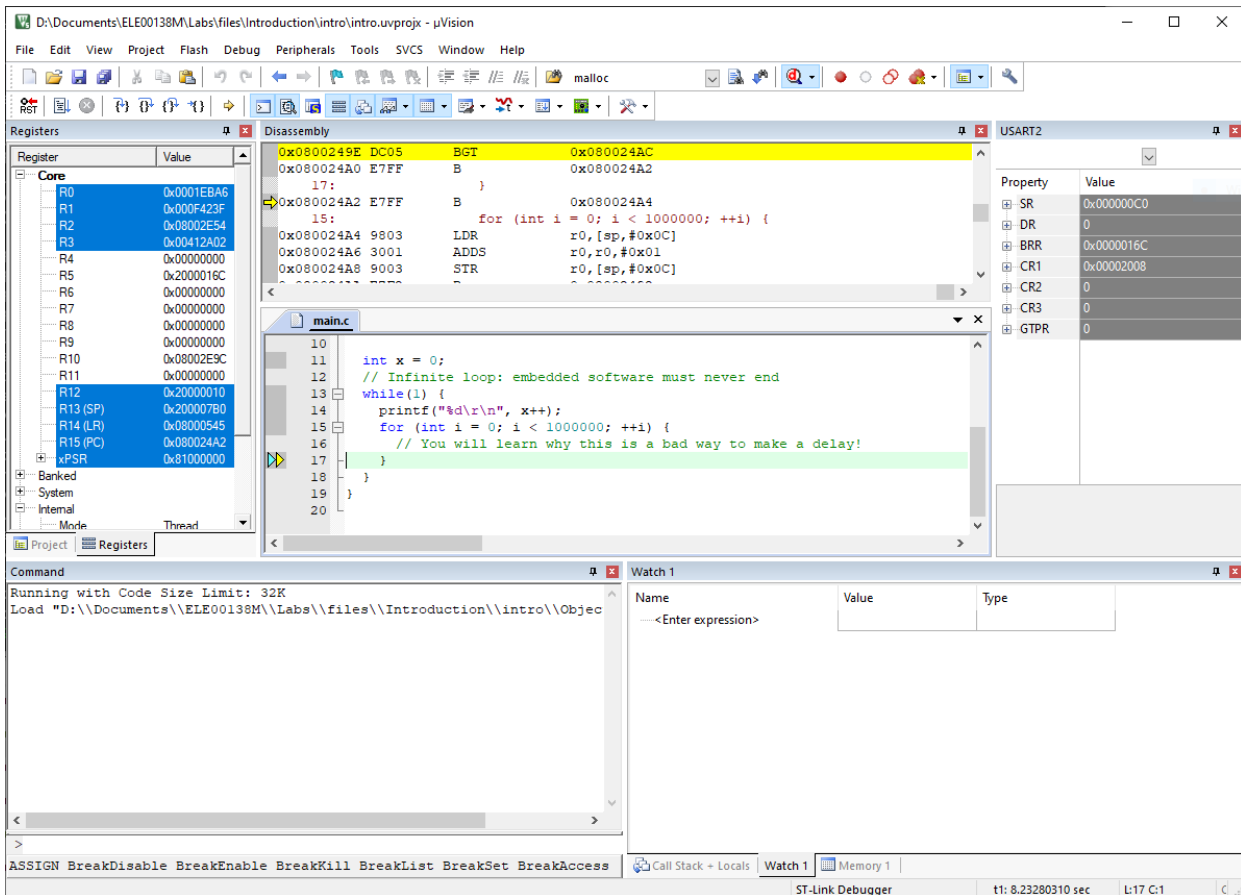


Figure 1.3: The Keil μVision Debugger

1.4.3.2 Stepping

Stepping is conducted by clicking the **step**, **step over** and **step out** buttons – or equivalently by using their keyboard shortcuts, which are F11, F10 and Ctrl+F11 respectively.

Step, often known as *step into*, executes the next line of code and then stops again immediately. If the line to be executed is a function call, the CPU will enter the function and then halt. **Step over** is similar, but if the line to be executed is a function call, the function is executed in its entirety, and the CPU halts again as soon as the function returns. **Step out** allows the CPU to run until it leaves the function it's currently executing, at which point it will halt.

At this point it's also worth mentioning the **run to cursor** button, which can be used to step over several lines of code at once: click on the line you wish to run to, click on **run to cursor**, and the CPU will resume running until the selected line is reached.

- ① Note that the debugger has two stepping modes. *Source stepping* is when each click on a stepping button advances the CPU by one line of the C source code. *Machine code*

stepping is when each click advances by one line of the compiled machine code, which is usually less than one line of the source code. If the source window has the focus, then source stepping will result; if the disassembly window has the focus, then machine code stepping will result instead.

⚠ Source stepping is really an illusion! The debugger divides the machine code into small chunks representing the individual lines of source code, so when you step over one line of source code, that's translated into a step over a chunk of machine code instructions automatically. This process has some limitations, as we'll see later on. Also note that if you step into a library function for which the source code is not available – `printf()`, for example – then source stepping will not be available. If you accidentally step into a function like this, the best option is usually to use **step out** to come straight back out again!

⚙ Try stepping through the example project. Use **step over** to begin with, making sure that the source window has the focus first, and watch CoolTerm to see how each line behaves. Once you're comfortable with that, experiment with the behaviour of the **step** and **step out** buttons. Note that the loop inside `main()` will never end, and so attempting to step out of `main()` will just cause the code to run forever. You can force it to halt again by clicking the **stop** button.

1.4.3.3 Breakpoints

Inserting a breakpoint in the code is simple. Click the grey margin by the line on which you want to place the breakpoint, and choose **Debug**→**Insert/Remove Breakpoint**, or press **F9**. You will see a red marker appear next to the line, indicating that the breakpoint has been set. You can also double-click the marker, or the equivalent place on other lines, to clear or set breakpoints.

You can also disable breakpoints rather than removing them, if you want to switch off a breakpoint without losing all trace of where it was. Do this by selecting **Debug**→**Enable/Disable Breakpoint**, or pressing **Ctrl+F9**.

⚠ Breakpoints have limitations. Firstly, you can't place a breakpoint on a source line that does not explicitly generate a machine code instruction. An example would be the line `int i;` – this declares a variable, but doesn't actually perform any action, so it generates no machine code and cannot be used as a breakpoint location. Secondly, the STM32F407VG has only a limited number of breakpoint and watchpoint registers. One of these will be used by the debugger to perform stepping. If you have too many active breakpoints, the debugger will warn you.

⚙ Set a breakpoint on one of the lines in the loop, perhaps on the call to `printf()`. Now click **run** and verify that the CPU halts as required. Click **run** a few more times, and observe the output in the CoolTerm window.

1.4.3.4 Inspecting Variables

The **register window** shows the values in all of the CPU's registers whenever it is halted. This is extremely useful for debugging assembly language code, but when you're debugging C and you don't have explicit control over the usage of the registers, it's of limited use until you reach the point of knowing what the compiler is likely to be doing with each register.

The **watch window** contains a variety of tools for inspecting the values of C variables while the CPU is halted. By clicking the **call stack** tab, and expanding the tree, the values of most local variables and function parameters can be seen. The **locals** tab shows the values of all of the local variables in the function that's being executed at the time.

By default, the displayed values are in hexadecimal. Right-click on an entry in the watch window if you want to change its display to decimal.

The watch window has other features, too. You probably won't need them, but you can find out about them using the [Keil \$\mu\$ Vision documentation](#).

1.4.3.5 Watchpoints

Breakpoints work by asking the CPU to halt when a particular line of code is executed. Watchpoints work by asking the CPU to halt when a particular memory location is written or read.

Watchpoints can be extremely useful for certain types of debugging, and particularly for identifying the source of apparent corruption of values stored in memory. Again, though, they have their caveats. Many *variables* are not mapped directly to memory locations; they are sometimes stored in registers, and can also move around in memory. Adding a watchpoint that is sensitive to a C variable requires a *conditional breakpoint*, and these are not supported on the STM32F407VG. You can add a watchpoint to a memory location though, and that includes memory-mapped peripheral control registers.


As you might have guessed, the mechanism for setting watchpoints is quite complex and versatile, and is described in some detail in the Keil μ Vision documentation.

1.5 Optimisation


We'll see in a bit more detail soon what the actually happens when you press the "build" button in the IDE. But one of the things that happens is that the compiler performs *optimisation*: it tries to make the compiled code as small as possible or as fast as possible (these two aims often conflict).


By default, optimisation is disabled for the projects in this module. That's because *optimised code is much harder to debug*. The compiler may do any number of things to your

code to make it faster or smaller: it may remove variables, “unroll” loops, remove functions, change the order of operations, and many other things. As you’ll see, sometimes this needs some thought when programming to avoid unintended consequences when optimisation is applied: basically you will sometimes need to tell the compiler, in a variety of ways, that something is important and should not be touched during optimisation. But it also makes debugging much harder because there is no longer guaranteed to be a clear correlation between your C code and the generated machine code. Source stepping often fails because it only works reliably when there is a simple, consistently-ordered relationship between the source lines and the machine code. Watching variable values may be difficult or impossible as well.

 Close the debugger. Click the *options for target* icon on the taskbar or choose Project→Options for Target ‘STM32F407 Flash’.... Find and click the “C/C++ (AC6)” tab. In the dropdown box next to “Optimisation” it should currently say -O0, which means “no optimisation”. Choose instead -O3, click “OK”, and rebuild the project. Restart the debugger.

Try some of the same things you were doing earlier with the debugger. Can you set a breakpoint in the `for` loop? Can you inspect the value of the variable `i`?

 The compiler has completely eliminated the `for` loop, because it has no effect. It has also eliminated the variable `i` because its value is never used. This is one reason why an empty `for` loop is a poor way to generate a delay!


 Close the debugger and allow the code to run. What do you notice?

1.6 Summary and Conclusions

By now, you should:

- Have a basic understanding of the operation of the µVision Integrated Development Environment and the Debugger;
- See how care needs to be taken with even basic code to prevent unintended interactions with the compiler.

If you have any questions about what you have learned, please ask a demonstrator.

 Remember to copy your project files to a network drive so you can keep them for reference and revision purposes.