University of Victoria
Electrical and Computer Engineering
ECE 355: Microprocessor-Based Systems
Laboratory Manual

By

Ali Jooya, Kevin Jones, Daler Rakhmatov,

and Brent Sirna

INTRODUCTORY LAB

Part 1: Embedded Software Development with ECLIPSE

Objective

This part of the introductory lab will help you get started with the ECLIPSE software development kit (SDK). You will learn how to use ECLIPSE to develop embedded software for 32-bit ARM® Cortex™-M0-based microcontroller platforms – specifically, the **STM32F0 Discovery** board by STMicroelectronics, featuring the STM32F051R8T6 MCU. Upon completion of Part 1 of the introductory lab, you will know how to:

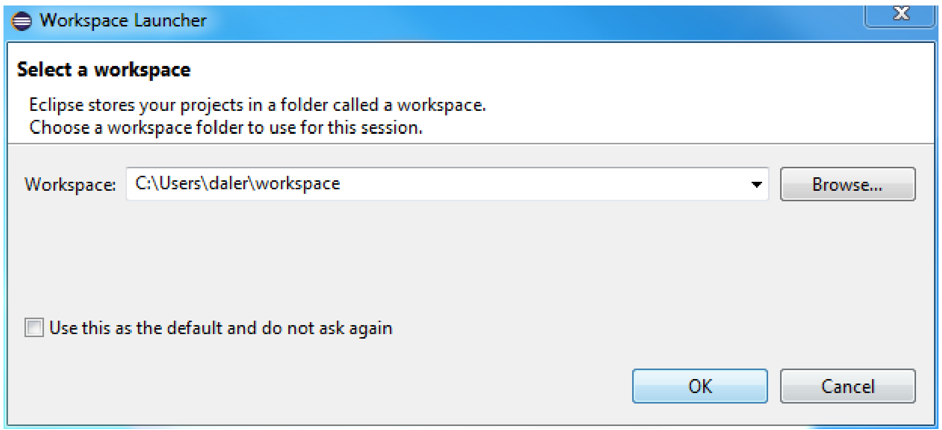
a) Create C-based projects using the ECLIPSE SDK;
b) Use a cross-compiler for building binary executables;
c) Use a debugger for loading and troubleshooting executables;
d) Use a console to observe the output of your embedded application.

General Guidelines

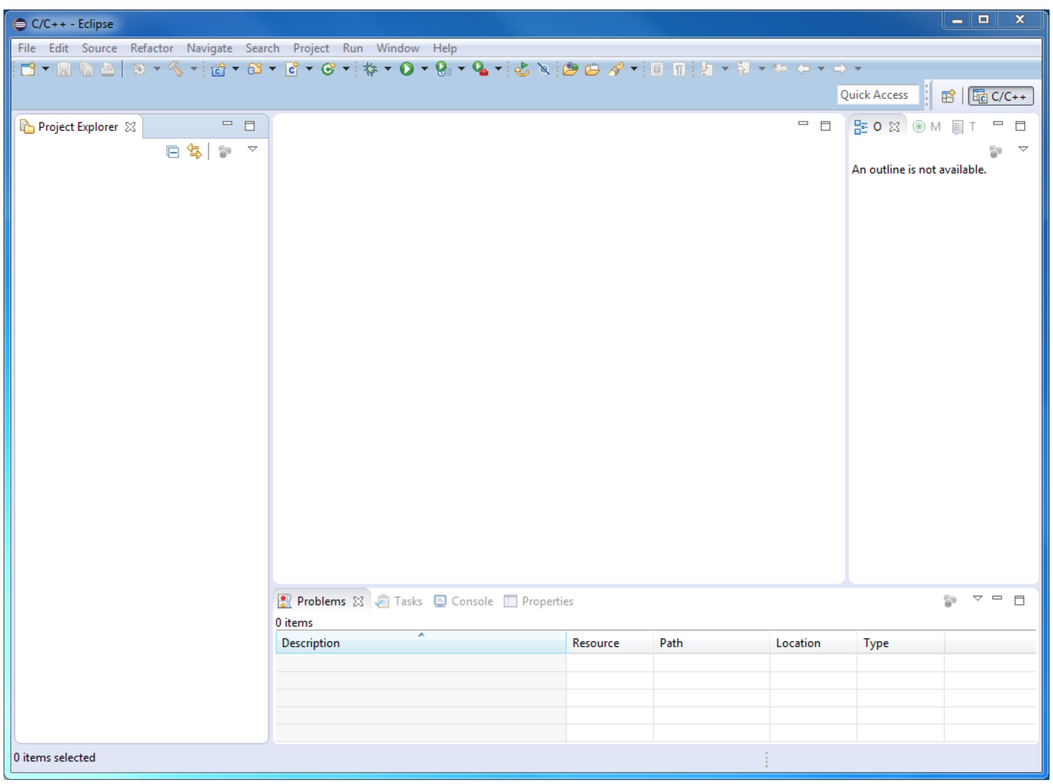
- **IMPORTANT:** *Save your work on the **M:** drive, which is your password-protected **ENGR** home directory, which is backed up regularly.*
- Please handle the hardware components with due care. To minimize potential damage due to electrostatic discharge (ESD), make contact with a grounded surface before touching the board or anything else connected to it.
- The lab equipment may be damaged if you attempt to power up an electrically faulty circuit. If in doubt, please ask your lab TA to check your circuit before powering it up.
- When working with a Function Generator, please use the SYNC output, whose voltage is limited to the digital signal range (from 0 to +3.3 V).
- Please keep your backpacks, food/drinks, and other personal items away from the lab equipment (instruments, boards, wires, etc.), thus helping prevent accidental damage and hardware malfunctioning. If you encounter a hardware-related problem, please notify your lab TA and the technical support staff person (see the lab's white board).

Setup Procedure

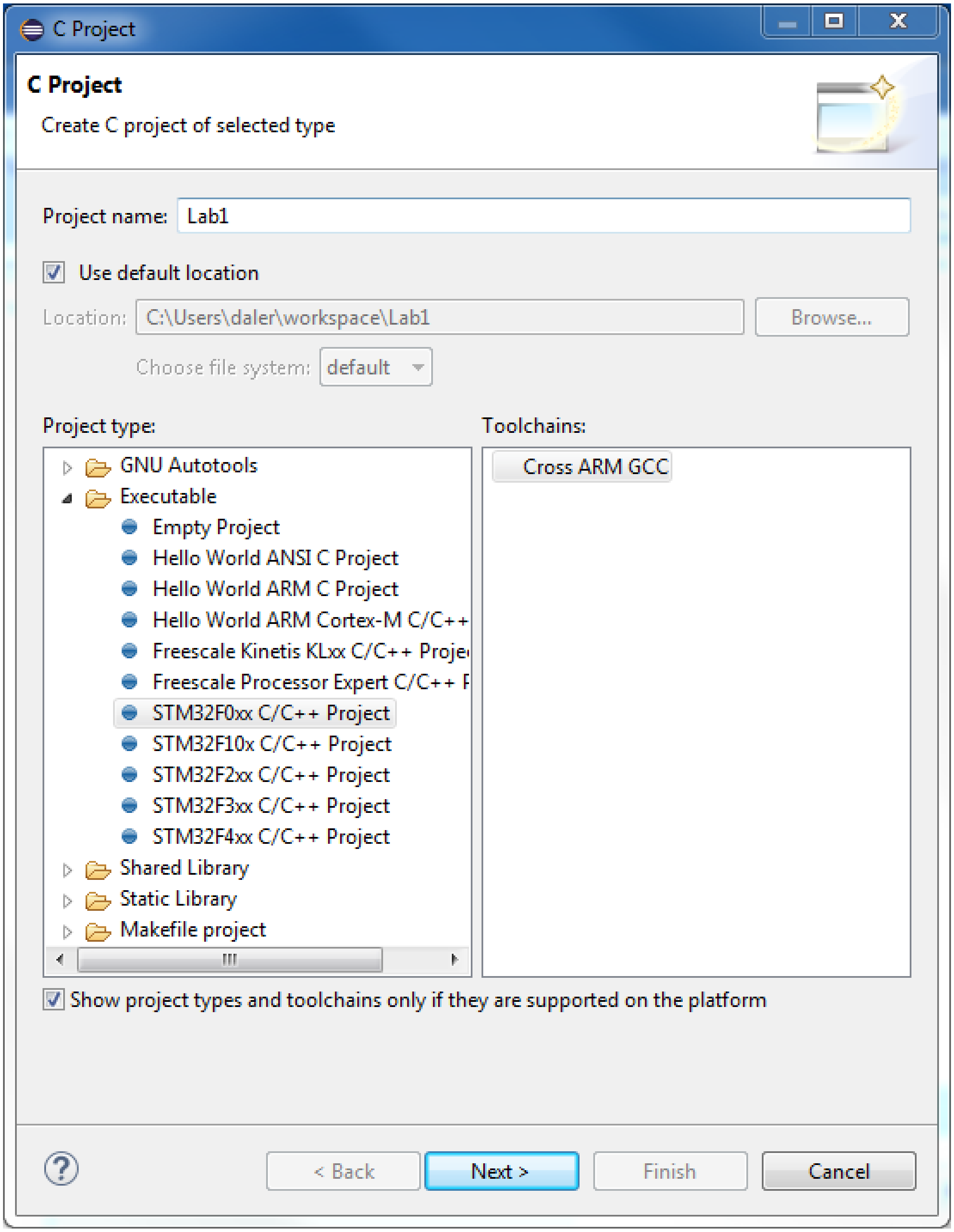
1. Login to one of the lab computers, open the **ECE355** folder on the **Desktop**, and double-click on **eclipse**, which brings up the Workspace Launcher window. Type ***C:\Users\\<u>YourUserName</u>\workspace*** in the "Workspace" box (e.g., see below).



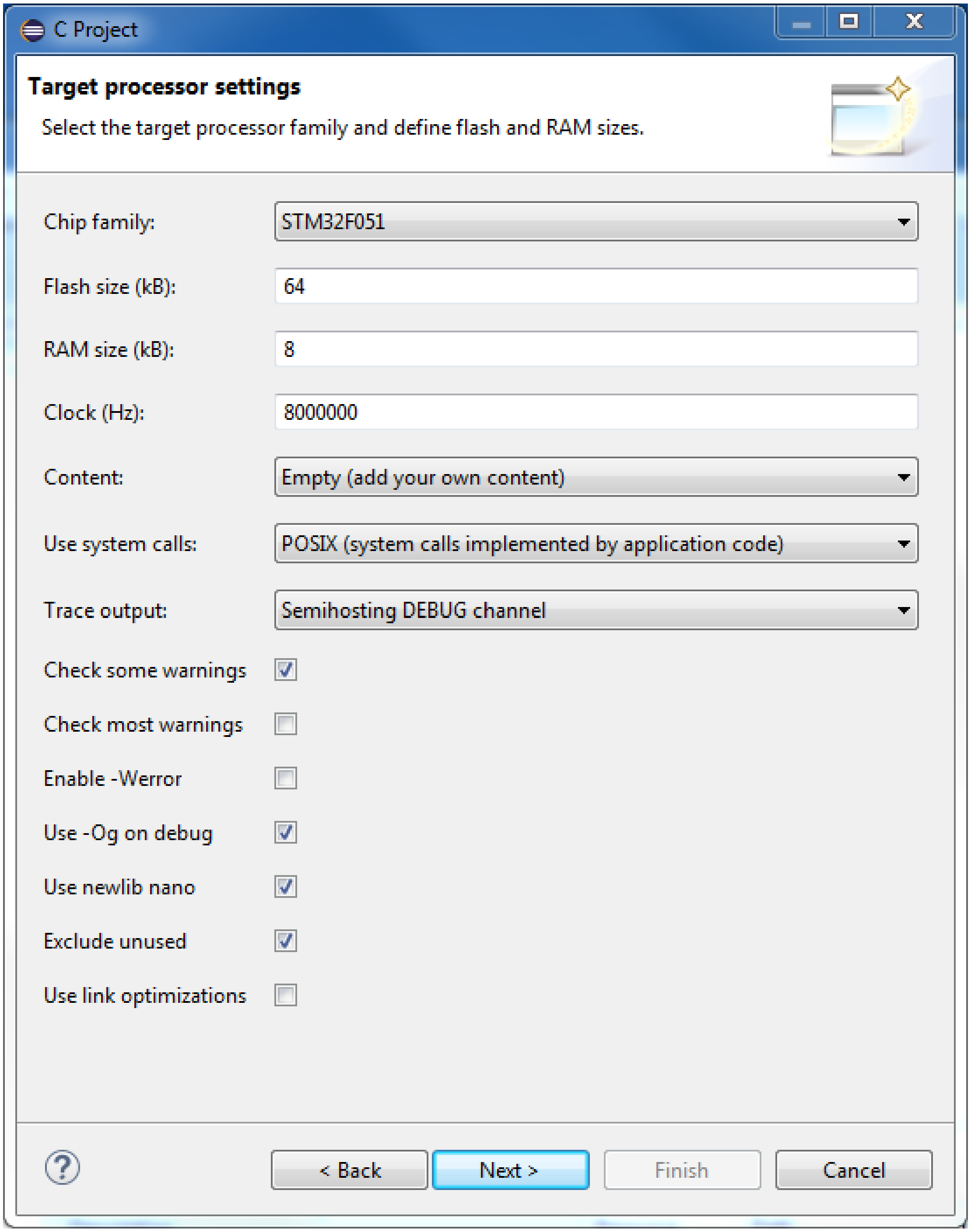
2. The Eclipse window will appear on the screen, as shown below (close the "Welcome" tab, if needed):



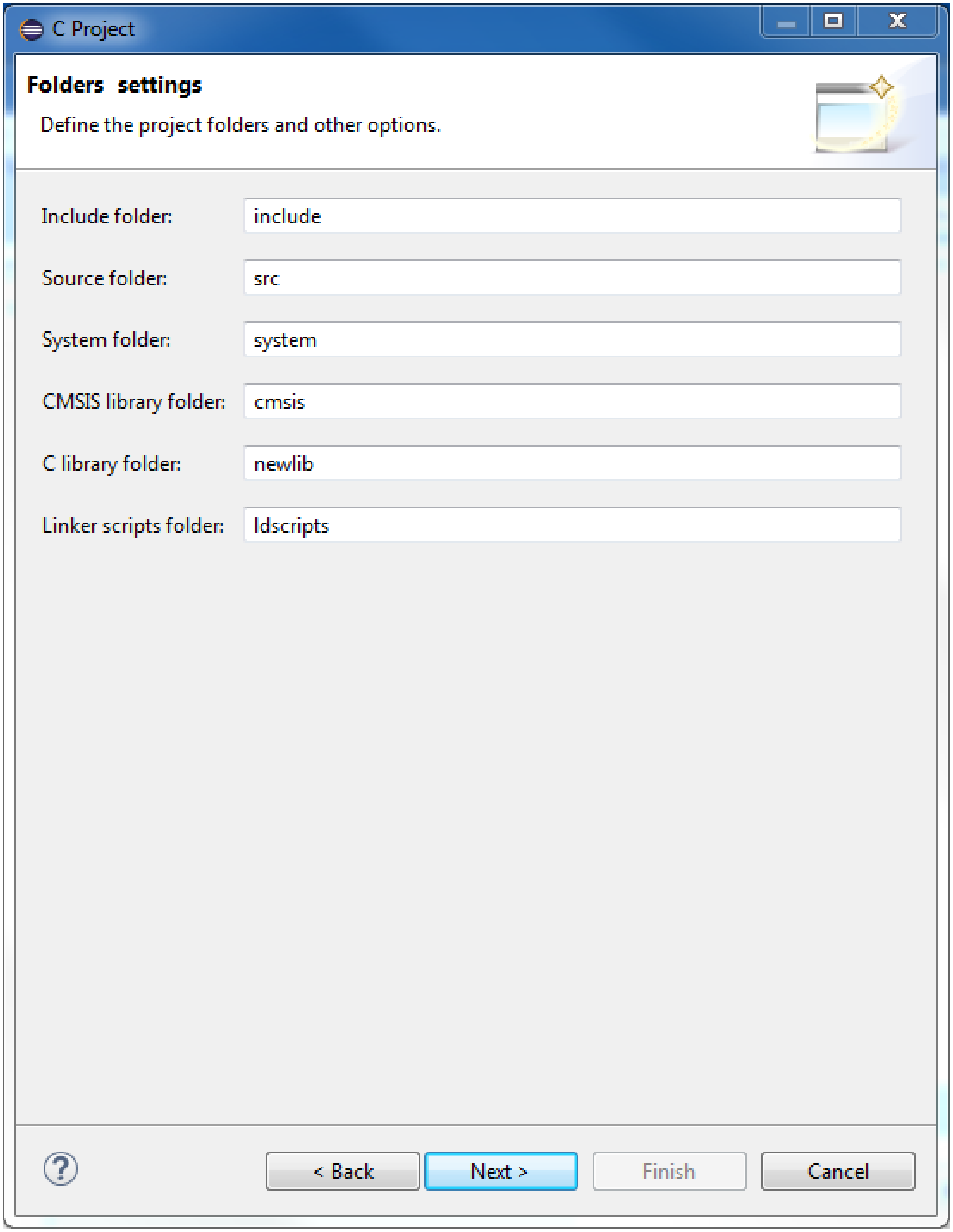
3. In the ECLIPSE window, select **File > New > C Project**. In the C PROJECT window, type ***Lab1*** in the "Project name" box and select ***STM32F0xx C/C++ Project*** in the "Project type" panel. Click **Next.**



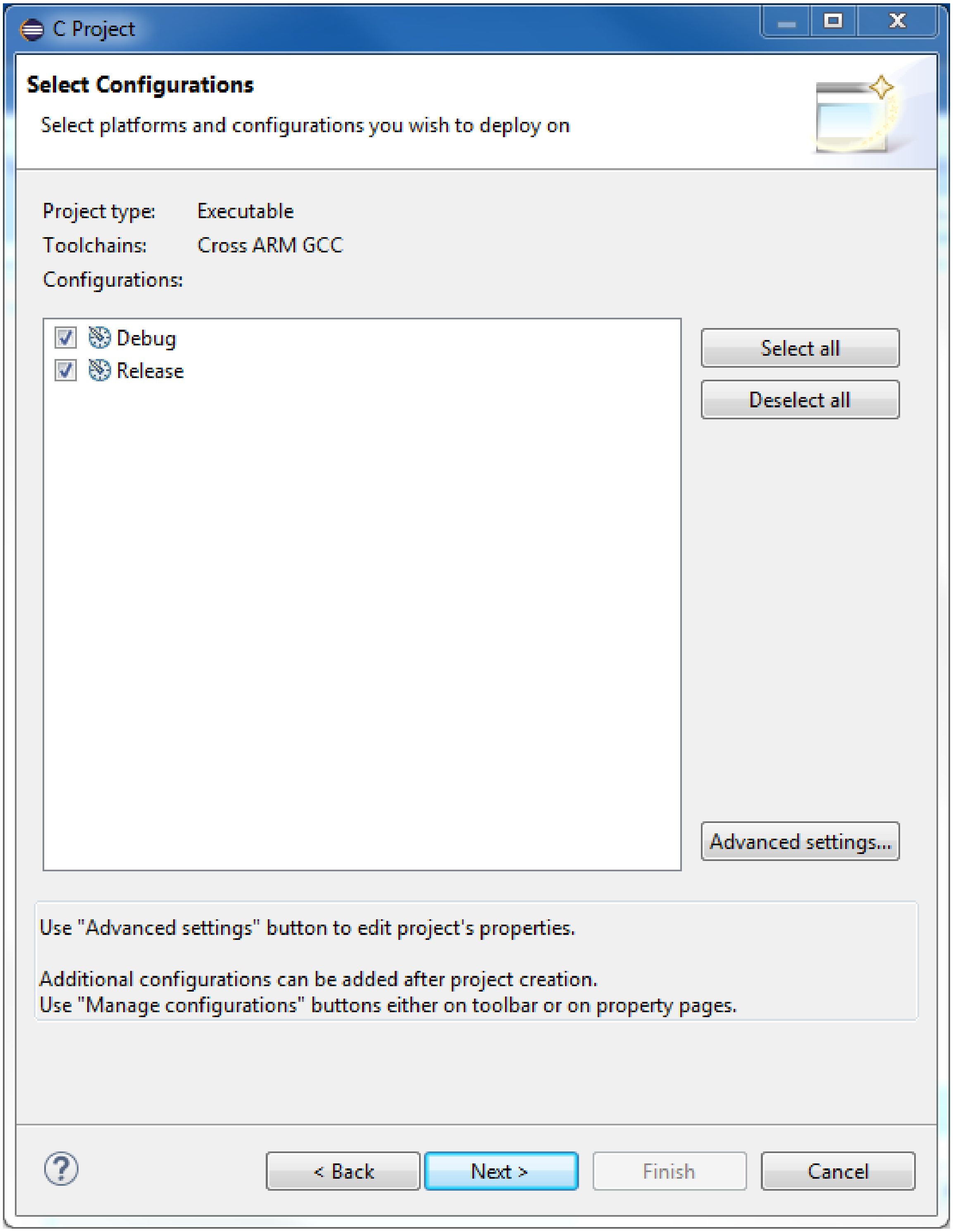
4. In the C PROJECT window, ensure that your **Target processor settings** selections are as shown below. Notice that the "Content" box reads: ***Empty (add your own content)***. Click **Next.**



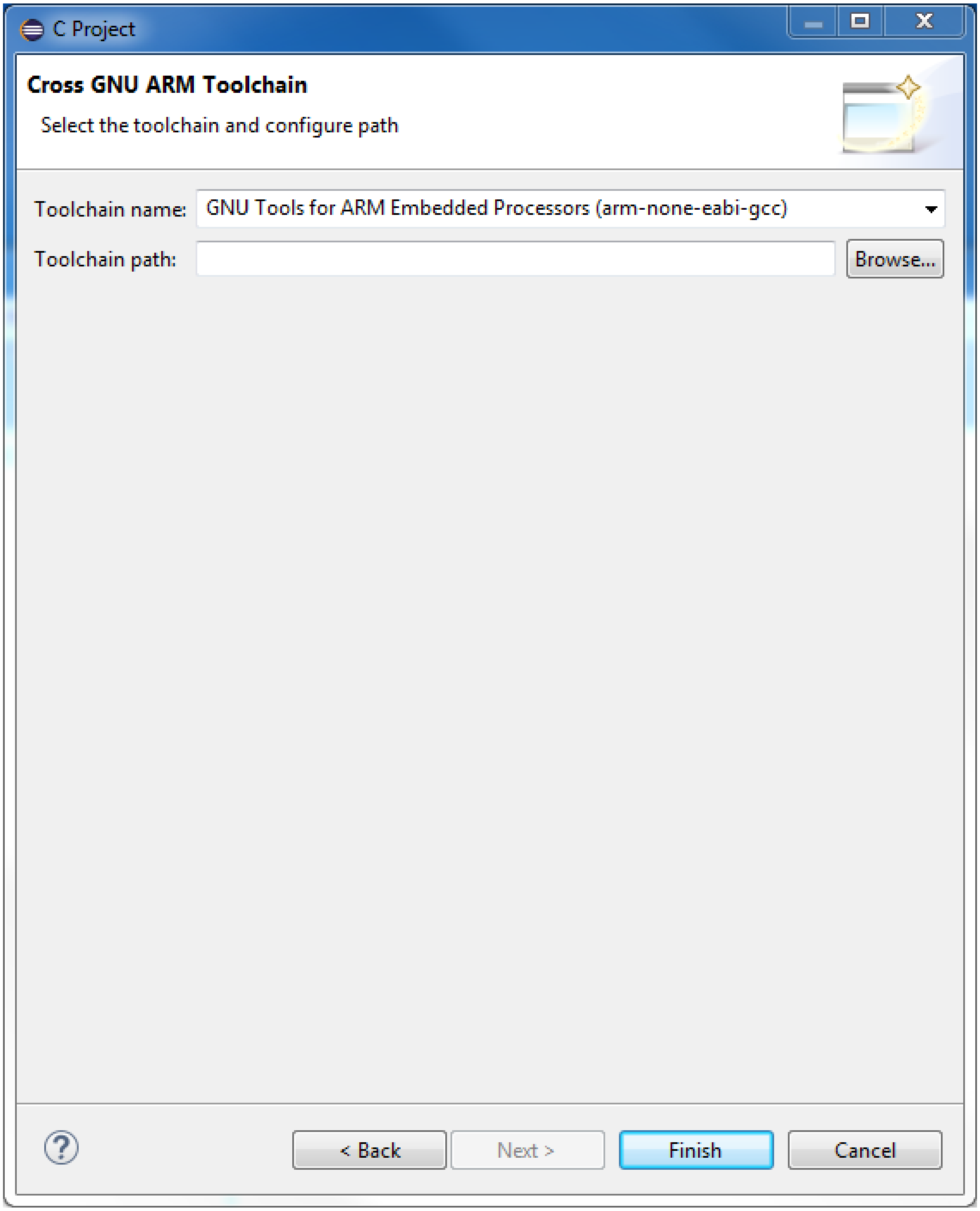
5. In the C PROJECT window, ensure that your **Folder settings** selections are as shown below. Click **Next.**



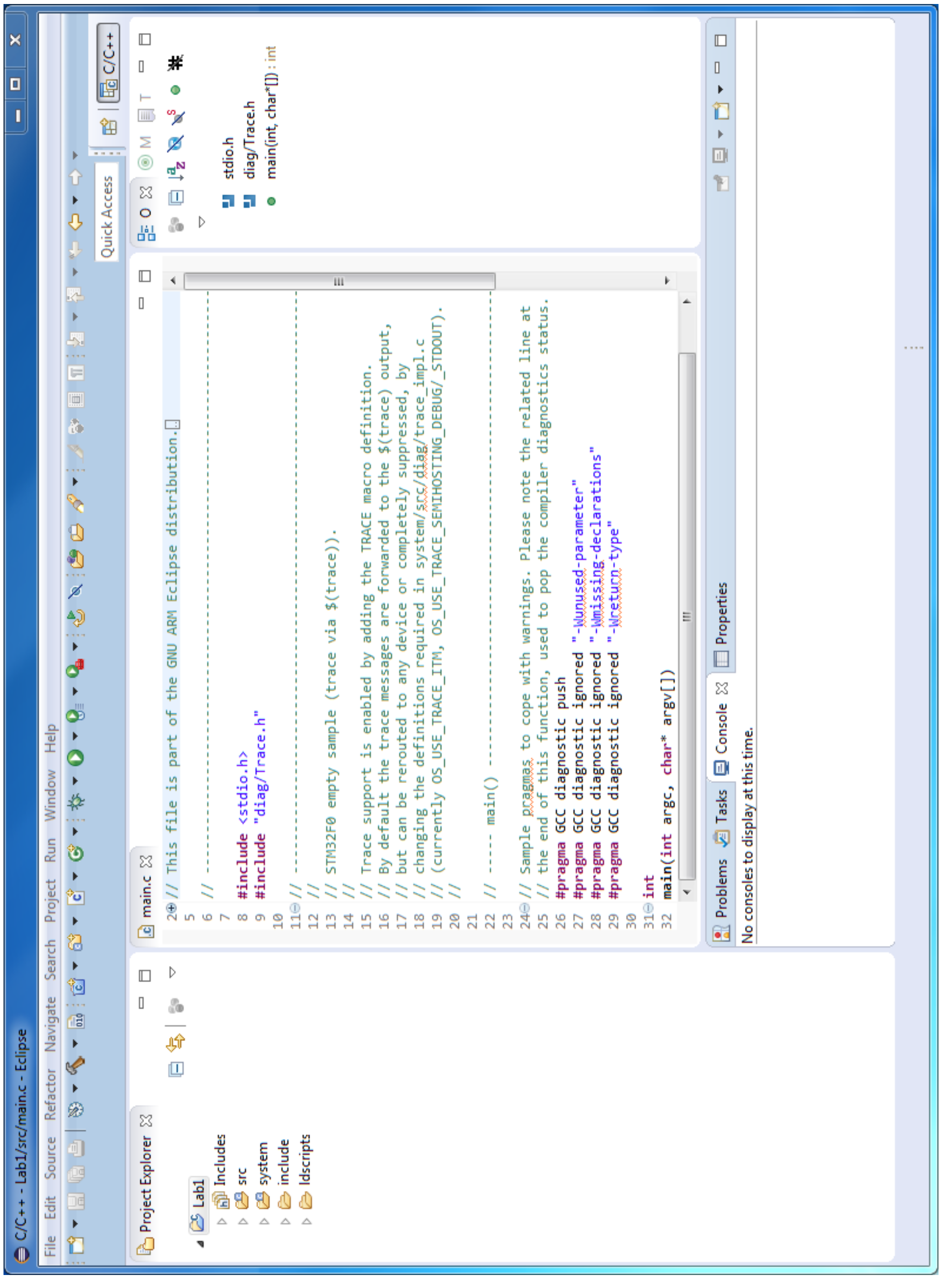
6. In the C PROJECT window, ensure that your **Select Configurations** selections are as shown below. Click **Next.**



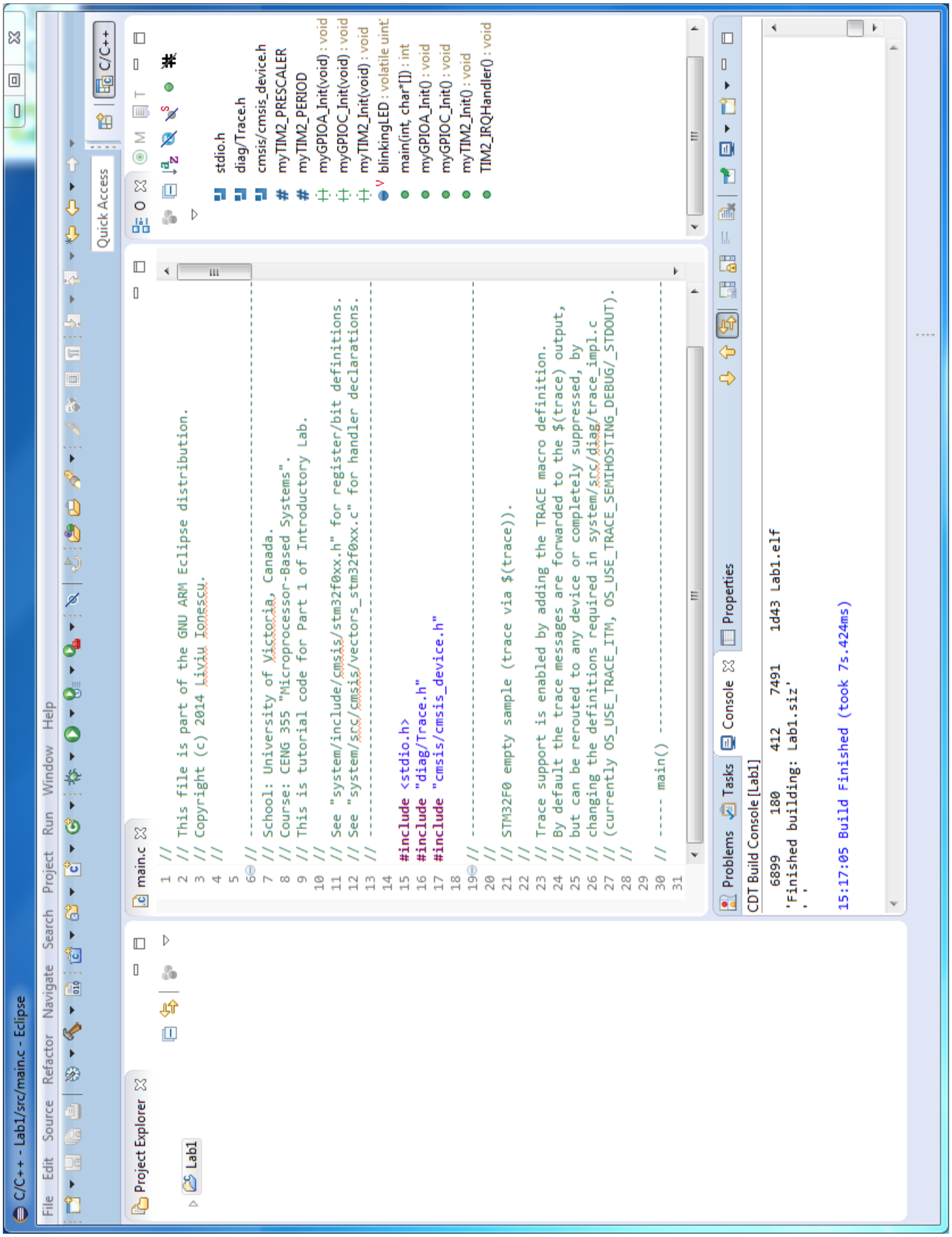
7. In the C PROJECT window, ensure that your **Cross GNU ARM Toolchain** selections are as shown below. Click **Finish.**



8. In the ECLIPSE window, expand the **Lab1** folder in the "Project Explorer" tab in the left panel and switch to the "Console" tab in the bottom panel (see below).

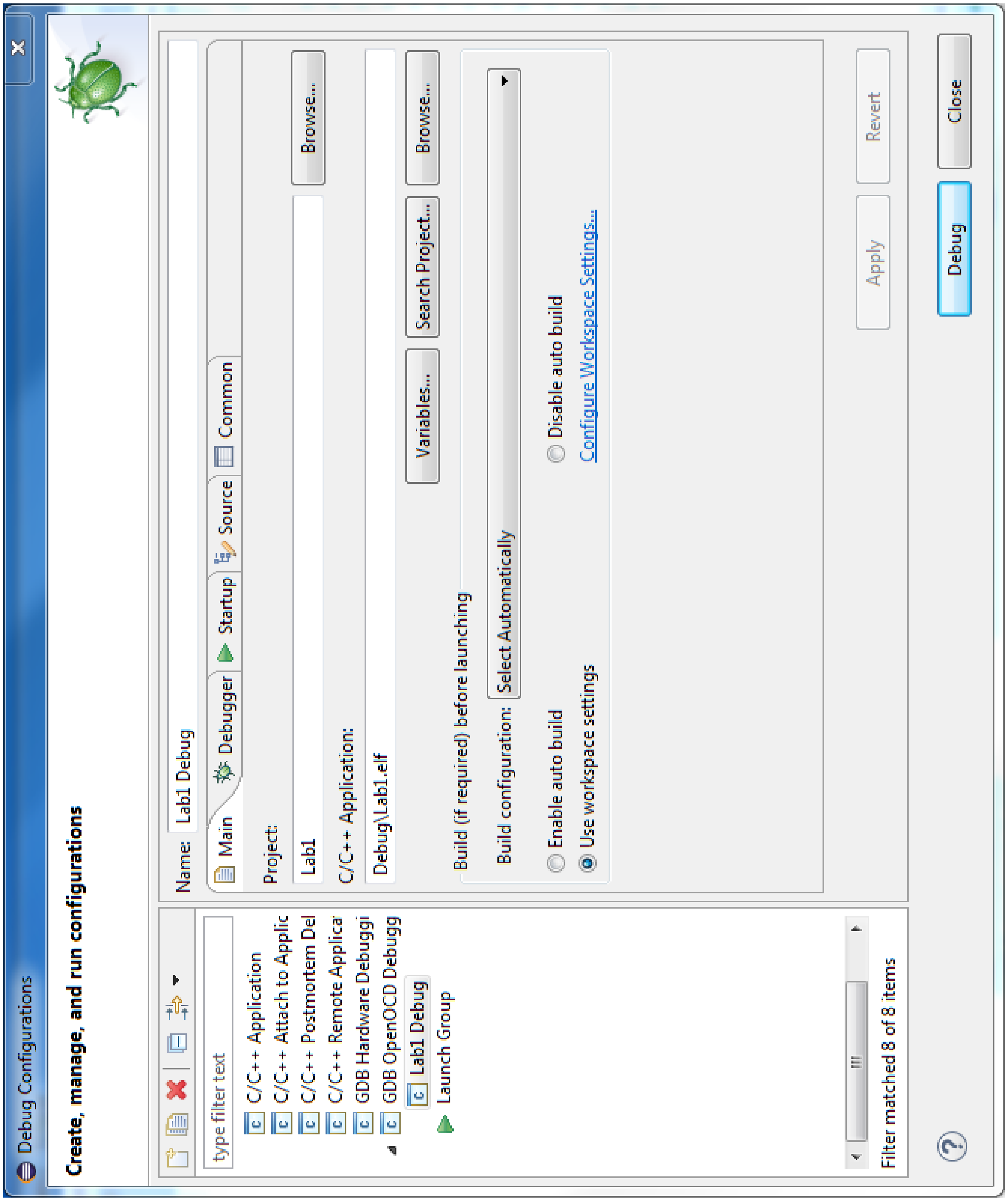


9. In the ECLIPSE window, select the “main.c” tab in the center panel and replace the existing C code with that provided in APPENDIX A (also available on the lab website). Select **File > Save** and then select **Project > Build Project**, thus compiling and linking your ***main.c*** code. The status of the project building process appears in the “Console” tab in the bottom panel (see below). If any errors are found, you can switch to the “Problems” tab to view them. There should be no errors reported.

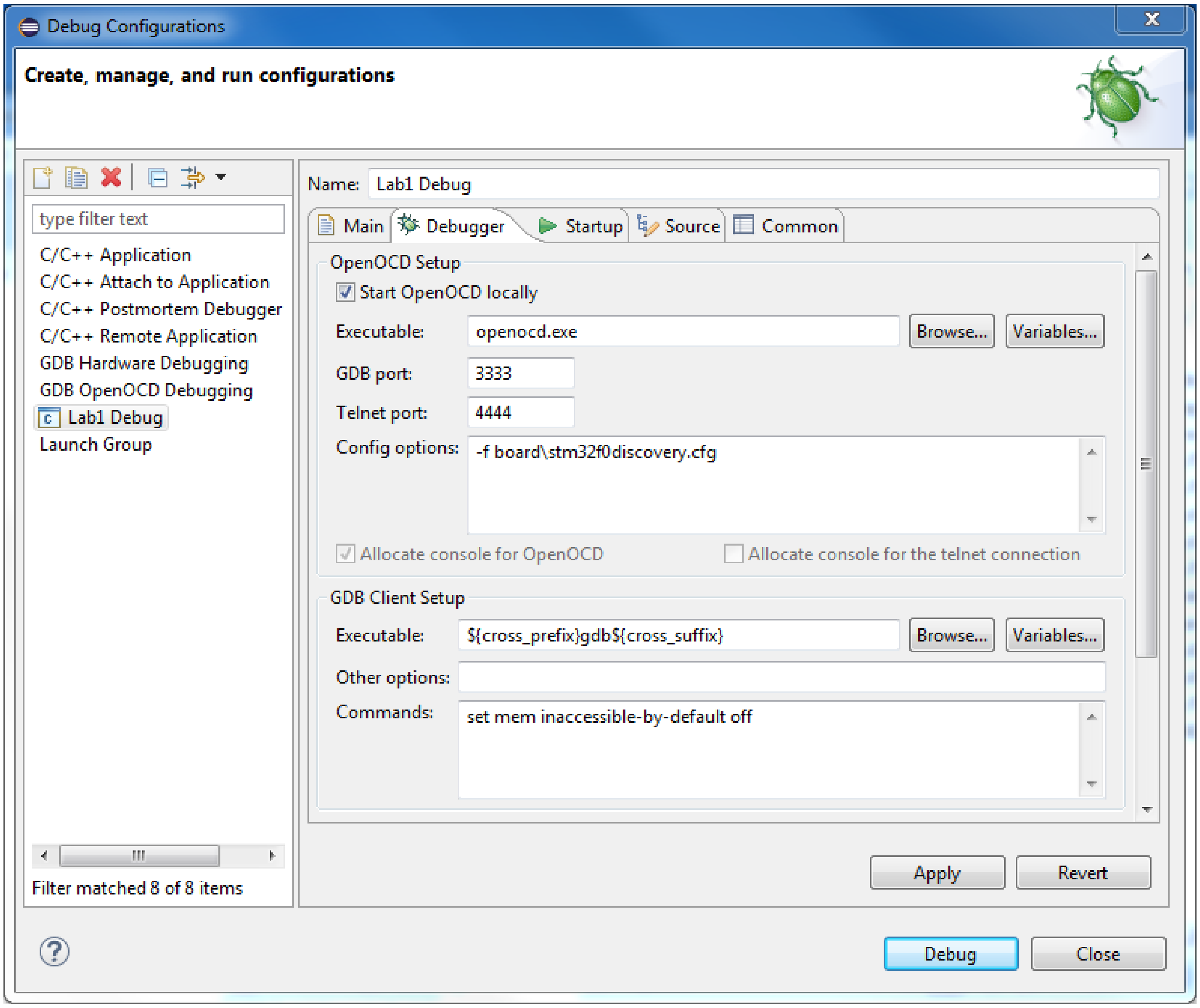


10. Connect the USB cable to the **STM32F0 Discovery** board.

11. In the ECLIPSE window, select **Run > Debug Configurations**, which brings up the DEBUG CONFIGURATIONS window. Double-click on **GDB Open OCD Debugging** in the left panel. The DEBUG CONFIGURATIONS window should appear as shown below:



12. In the DEBUG CONFIGURATIONS window, select the "Debugger" tab and ensure that your selections are as shown below. Click **Apply**.



OpenOCD Setup – "Executable" Box:

openocd.exe

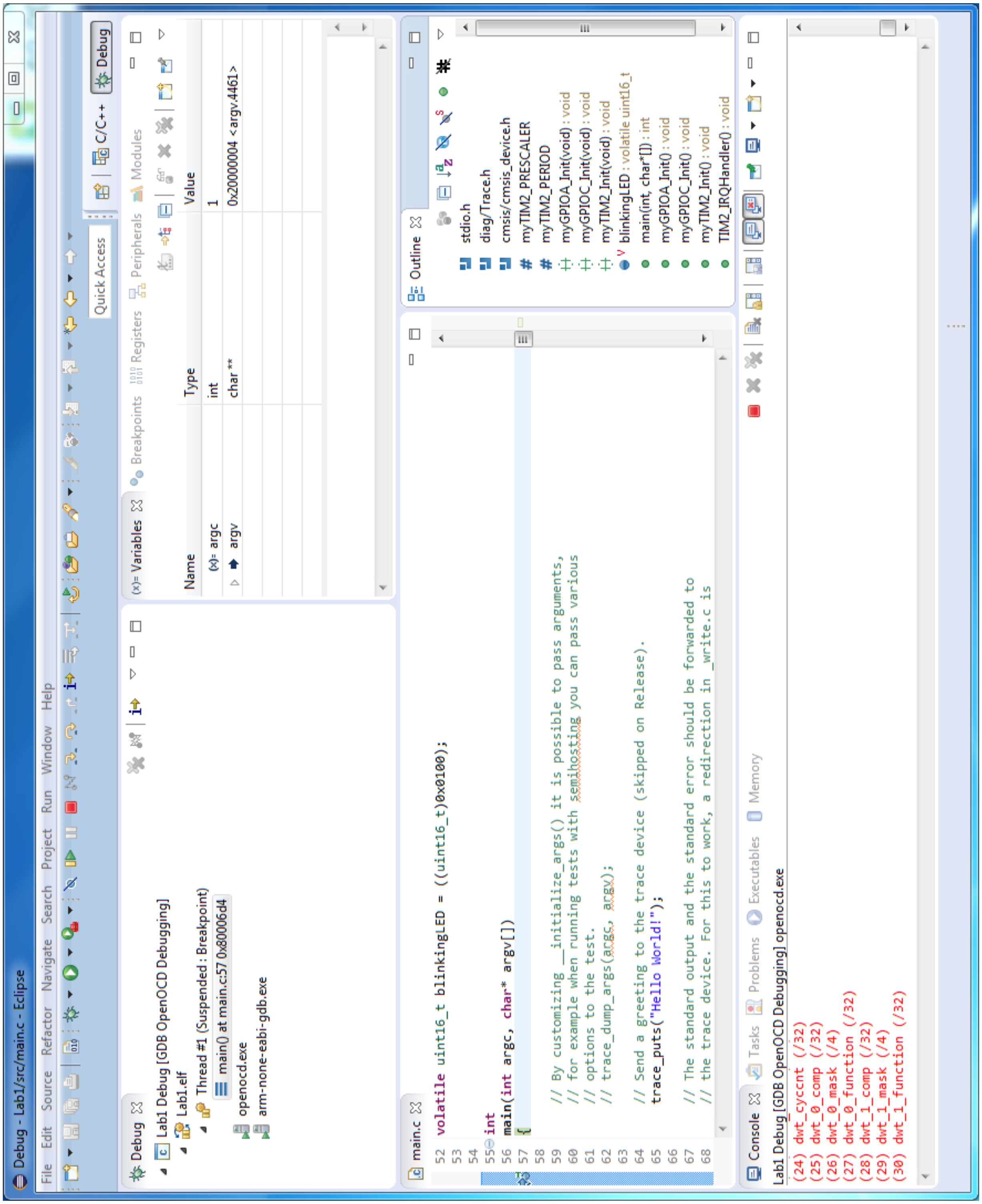
OpenOCD Setup – "Config options" Box:

-f board\stm32f0discovery.cfg

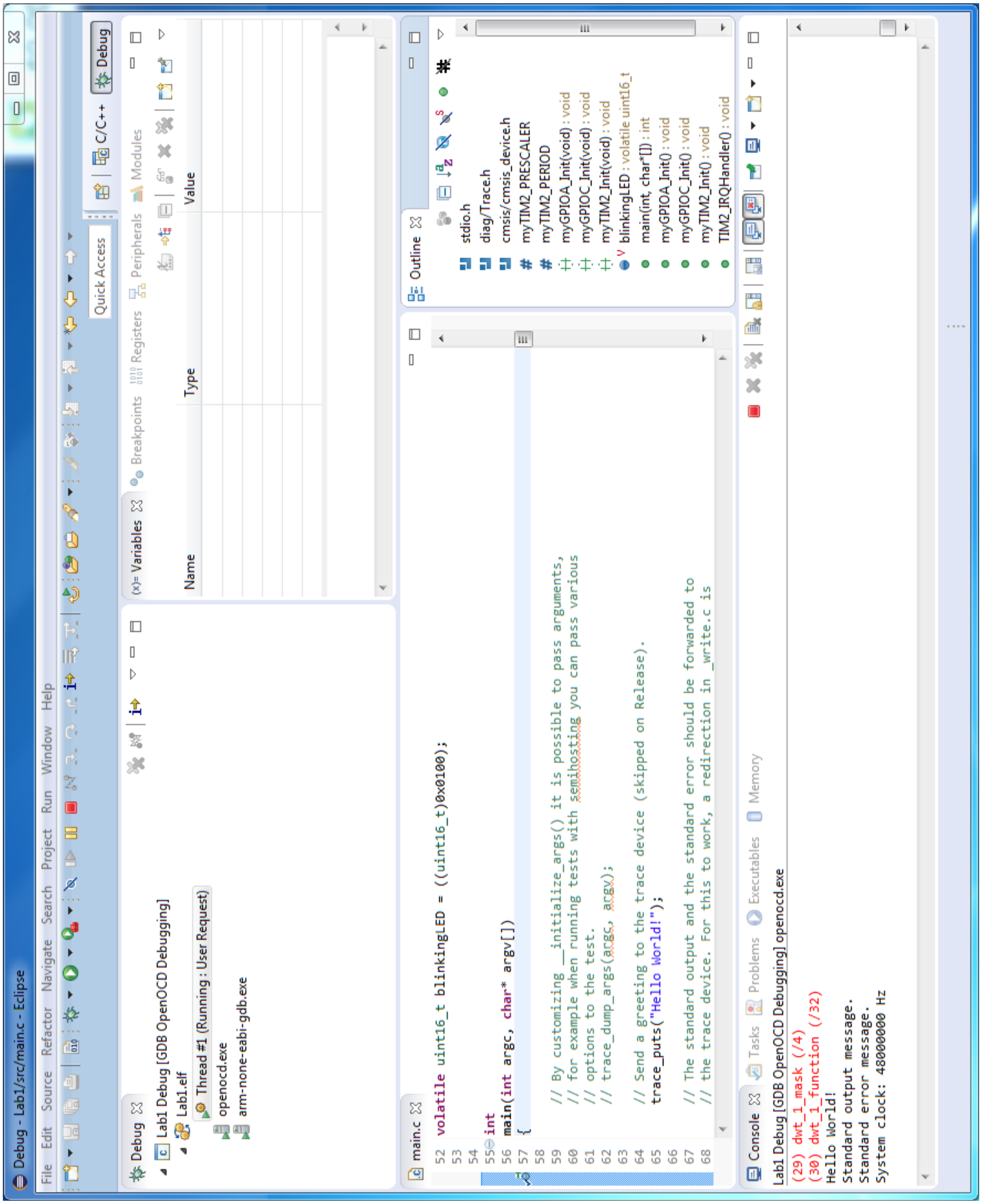
GDB Client Setup – "Executable" Box:

${cross_prefix}gdb${cross_suffix}

13. In the DEBUG CONFIGURATIONS window, click **Debug**. In the CONFIRM PERSPECTIVE SWITCH window, click **Yes**. The ECLIPSE window should appear as shown below:



14. In the ECLIPSE window, select **Run > Resume** (or press F8 key). Four messages are printed in the "Console" tab in the bottom panel (see below), and the blue LED on the **STM32F0 Discovery** board starts blinking.



15. On the **STM32F0 Discovery** board, press the blue button (USER). The blue LED is turned off, and the green LED starts blinking. Pressing the USER button switches the blinking LED and prints ***"Switching the blinking LED…"*** in the "Console" tab in the bottom panel of the ECLIPSE window. This is the intended functionality of the ***Lab1*** executable code.

Note: Before proceeding to the next step, make sure that the blue LED is blinking.

16. In the ECLIPSE window, select **Run > Suspend**. In the "main.c" tab in the center panel, set breakpoints at lines 179 and 184 by double-clicking on them (see below).

main.c

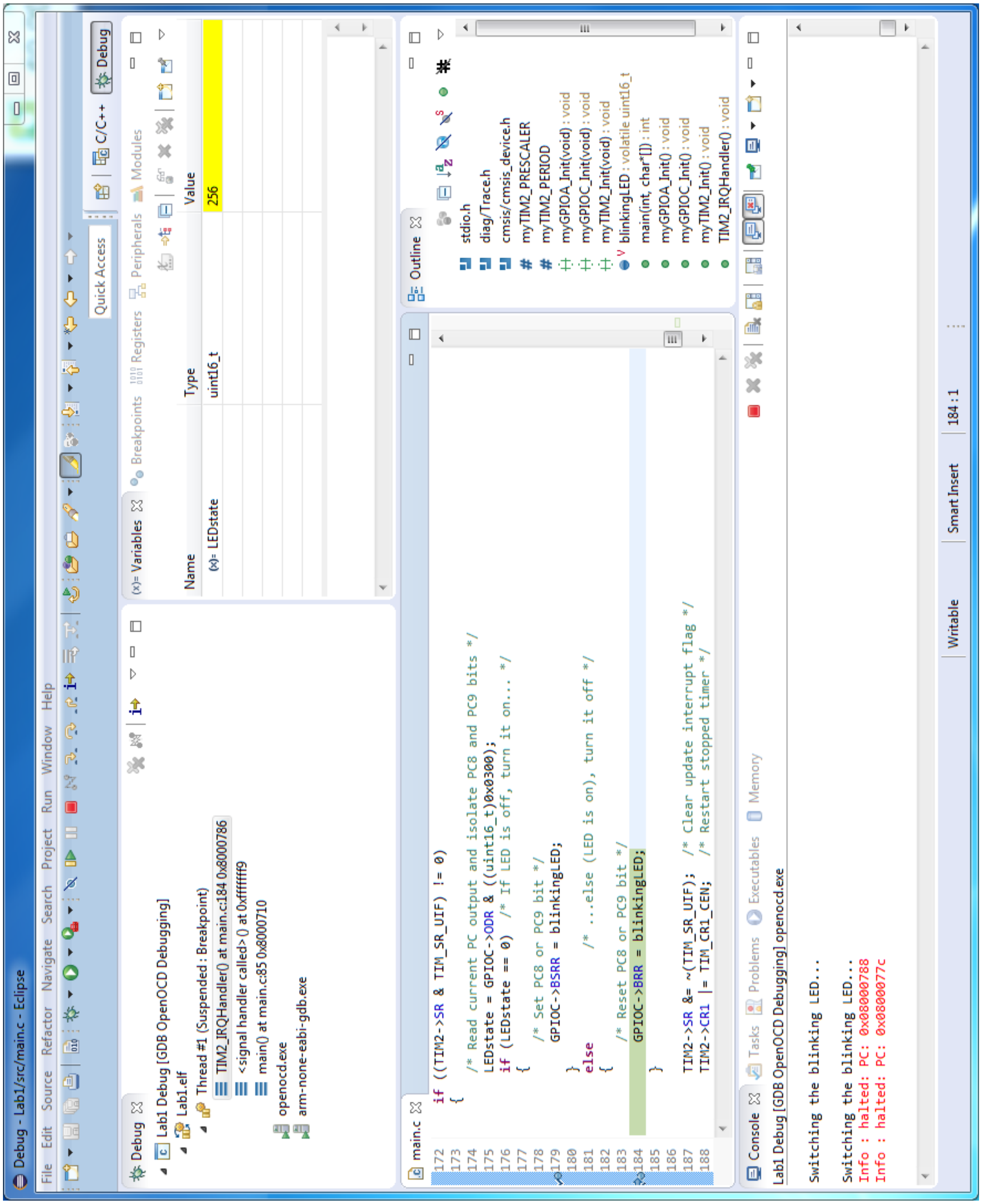
```
169     uint16_t LEDstate;
170
171     /* Check if update interrupt flag is indeed set */
172     if ((TIM2->SR & TIM_SR_UIF) != 0)
173     {
174         /* Read current PC output and isolate PC8 and PC9 bits */
175         LEDstate = GPIOC->ODR & ((uint16_t)0x0300);
176         if (LEDstate == 0)  /* If LED is off, turn it on... */
177         {
178             /* Set PC8 or PC9 bit */
179             GPIOC->BSRR = blinkingLED;
180         }
181         else                /* ...else (LED is on), turn it off */
182         {
183             /* Reset PC8 or PC9 bit */
184             GPIOC->BRR = blinkingLED;
185         }
186
```

17. In the ECLIPSE window, select **Run > Resume**. The program execution should stop either at line 179 (the blue LED is off) or at line 184 (the blue LED is on).

If the program execution stops at line 179, the ***LEDstate*** variable should be equal to ***0***.

If the program execution stops at line 184, the ***LEDstate*** variable should be equal to ***256*** (***0x0000100*** in hex) – see below.

Selecting **Run > Resume** switches between the two breakpoints, thus switching the blue LED state between off and on.



18. In the "main.c" tab in the center panel of the ECLIPSE window, remove breakpoints at lines 179 and 184 by double-clicking on them. Select **Run > Resume** (the blue LED starts blinking again).

19. Press the USER button, so that the green LED is blinking. Repeat steps **16** and **17**. Whenever the green LED is on (i.e., whenever the program execution stops at line 184), the ***LEDstate*** variable should be equal to ***512*** (***0x0000200*** in hex). Repeat step **18** (the green LED starts blinking again).

20. To make any changes to your ***Lab1*** code, follow the sequence of steps listed below:
    – Suspend your program execution (if running): **Run > Suspend**.
    – Terminate your debugging session (if active): **Run > Terminate**.
    – Switch to the **C/C++** perspective (see the top right corner of the ECLIPSE window).
    – Make any desired changes to your code in the center panel of the ECLIPSE window.
    – Save your changes: **File > Save**.
    – Rebuild your project: **Project > Build Project** (there should be no errors reported).
    – Restart debugging: **Run > Debug History > Lab1 Debug**.

21. When you are done using the **STM32F0 Discovery** board, make sure to suspend your program execution and terminate your debugging session. Then select **File > Exit**.

22. **IMPORTANT:** Copy your ***workspace*** folder from the **C:** drive into the **M:** drive for safekeeping.

This is the end of Part 1 of the introductory lab, where you have learned how to use the ECLIPSE SDK to build, run, and debug your embedded software code targeting the STM32F0 Discovery board.

Additional up-to-date information can be found on the ECE 355 lab website: http://www.ece.uvic.ca/~ece355/lab.

Part 2: Signal Frequency Measurement

Objective

Using the ECLIPSE SDK and the **STM32F0 Discovery** board, you are to develop a system that measures the frequency of a square-wave signal generated by a Function Generator. The signal period and frequency are to be displayed on the console. The minimum and the maximum detectable frequencies must also be determined.

Specifications

- The input signal will be a square wave with the amplitude ranging from 0 to +3.3 V, generated by a Function Generator – use the SYNC output!
- Your will use the **TIM2** general purpose timer to measure the frequency of the input signal. Your goal is to determine the number of timer pulses (clock cycles) elapsed between two consecutive rising (or falling) edges of the input signal. A current count value of the timer pulses is recorded in the **TIM2_CNT** counter register of **TIM2**: it must be configured to increment every cycle of the **TIM2**'s clock, whenever **TIM2** is enabled to count.
- On the **STM32F0 Discovery** board, you will use the microcontroller's **PA1** I/O pin, serving as the **EXTI1** external interrupt line, to generate interrupt requests when a rising (or falling) edge of the input signal is detected. Your **EXTI1** interrupt handler will need to access **TIM2** (e.g., start/stop the counting process, read **TIM2_CNT**, etc).
- Your C code must calculate the period and the frequency of the input signal, and then display those values on the console.
- You will need to determine the range of detectable frequencies of the input signal.
- You can find a basic code template (to get you started) on the lab website.

Hints and Advice

- You will need to study Chapters 7, 9, 12, 17 of the **STM32F0xx Reference Manual** providing necessary technical details on RCC (reset and clock control), GPIO (general purpose I/Os), interrupt controllers (NVIC and EXTI), and general purpose timers (TIM2 and TIM3).

- You will need to study the **PBMCUSLK User Guide** providing necessary technical details on the **PBMCUSLK** board connections.
- Your goal is to determine the number of clock cycles between two consecutive rising (or falling) edges of the input signal. On the first edge, enable the timer to start counting. On the second edge, stop the counting process, calculate the signal period and frequency, and then display those values on the console.
- Additional hints and advice are available

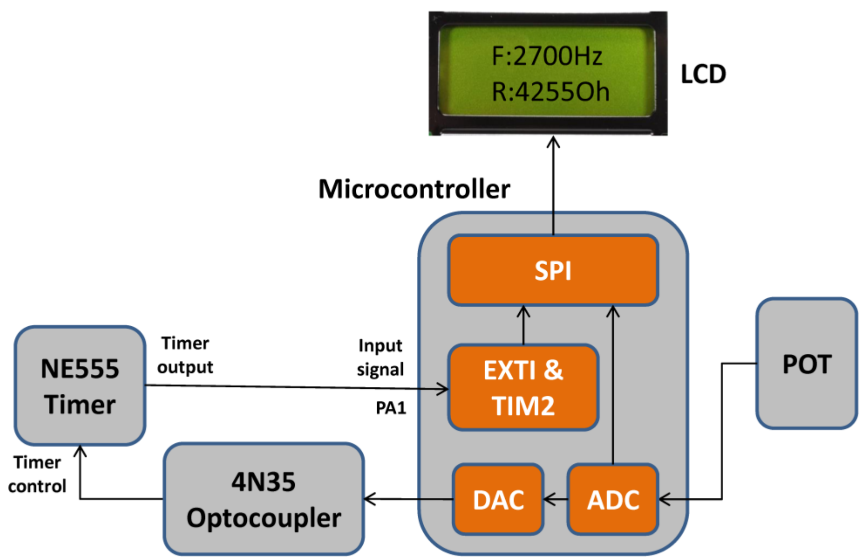
Deliverables

- **Demonstration.** You must demonstrate a working system at during your second lab session. You must also determine the limitations of your system (the maximum and the minimum detectable frequencies) and be able to explain why your system has such limitations. This deliverable is worth **10%** of your final lab grade.

PROJECT: PWM Signal Generation and Monitoring System

Objective

You are to develop an embedded system for monitoring and controlling a pulse-width-modulated (PWM) signal that will be generated by an external timer (**NE555** IC). An external optocoupler (**4N35** IC), driven by the microcontroller on the **STMF0 Discovery** board, will be used to control the frequency of the PWM signal. The microcontroller will be used to measure the voltage across a potentiometer (POT) on the **PBMCUSLK** board and relay it to the external optocoupler for controlling the PWM signal frequency. The microcontroller will also be used to measure the frequency of the generated PWM signal. The measured frequency and the corresponding POT resistance are to be displayed on the LCD on the **PBMCUSLK** board. The overall system diagram is shown below.



Specifications

- In Part 2 of the introductory lab, you have used a Function Generator to generate a square-wave signal and to display its period and frequency on the console. For this project, you will use the **NE555** timer (instead of a Function Generator) to generate the square-wave signal, and you will use the LCD (instead of the console) to display the signal frequency and the POT resistance. You should be able to measure the signal frequency reusing most of the code you have developed in Part 2.

- The STM32F051R8T6 MCU mounted on the **STM32F0 Discovery** board features built-in analog-to-digital converter (ADC), digital-to-analog converter (DAC), and serial peripheral interface (SPI). The DAC will be used to drive the **4N35** optocoupler to adjust the signal frequency of the **NE555** timer, based on the potentiometer voltage read by the ADC, and the SPI will be used to communicate with the LCD.
- The analog voltage signal coming from the potentiometer on the **PBMCUSLK** board will be measured continuously by the ADC – this task is to be accomplished using a polling approach. Using those voltage measurements, you will need to calculate the corresponding potentiometer resistance value. You must also determine the lower and the upper limits of the measurable voltage.
- You will use the digital value obtained from the ADC to adjust the frequency of the PWM signal generated by the **NE555** timer. For that purpose, you will use the DAC to convert that digital value to an analog voltage signal driving the **4N35** optocoupler.
- To display the signal frequency and the potentiometer resistance, you will use the SPI (to be appropriately configured) to drive the LCD on the **PBMCUSLK** board.
- The LCD is a 4-bit, 2-by-8 character display, and there is no direct write access the LCD pins. The only way to control the LCD pins is through the 8-bit **74HC595** shift register on the **PBMCUSLK** board. The **74HC595** shift register will need to receive 8-bit words from the SPI via the serial **MOSI** port, appropriately timed using the latch clock **LCK** and the serial shift register clock **SCK**. Each 8-bit word sent by the SPI will need to include LCD data bits **D3-D0**, register-select bit **RS**, and enable bit **EN**.
- As an option (during your software development), you may use the CMSIS-defined software library functions for SPI access. Please see the lab website for "Project Tips" showing how to include SPI library for use with your code. For any other peripherals involved in this project, your code must access relevant I/O registers explicitly (for educational purposes).

Hints and Advice

- Successful completion of the project requires a thorough knowledge of the available technical documentation. You will need to study Chapters 7, 9, 12, 13, 14, 17, 27 of the **STM32F0xx Reference Manual**, as well as the **PBMCUSLK User Guide**, the **NE555** timer data sheet, and the **4N35** optocoupler data sheet.

- Use an Oscilloscope to confirm the period of the square-wave signal being measured, the analog voltage produced by the DAC, or any other signal parameters of interest.
- The supply voltage of the **NE555** timer is 5 V. Do <u>NOT</u> supply 5 V to anything else.
- To identify and resolve software bugs related to the SPI-to-LCD interfacing task, you can wire the LCD signals (accessible via connector **J9**) to the LED1-LED8 (accessible via connector **J10**) on the **PBMCUSLK** board. These LEDs will indicate the values of individual bits of every 8-bit data word sent by the SPI, helping you make sure that the LCD is receiving the right data. The **J9** connector pinouts are shown below.

GND	**1**
5V	**2**
CONTRAST	**3**
RS	**4**
R/W*	**5**
EN	**6**
DB0	**7**
DB1	**8**
DB2	**9**
DB3	**10**
LCD_D4	**11**
LCD_D5	**12**
LCD_D6	**13**
LCD_D7	**14**

SPI data bit definitions to LCD Port:
LCD_D[7..4] – LCD data bits D[3..0]
DB[3..0] – Unused, 10K ohm pull-downs installed
R/W – Read/Write pin, set to 0 volts, Read only
EN – LCD enable input, 1 = LCD enable
CONTRAST – LCD contrast input
RS – Register Select, 0 = LCD Command, 1 = LCD Data

Deliverables

- **Demonstration.** You must demonstrate a working project during your last lab session. You must also determine the limitations of your system and be able to explain why your system has such limitations. Your lab TA will inspect your code and ask each group member (individually) a series of technical questions. Your (individual) mark for the project demonstration will be based on your ability to answer your lab TA's questions. This deliverable is worth **30%** of your final lab grade.
- **Report.** A substantial project report is required from each student group – it should be structured as a standard engineering technical report. The report must describe system

specifications and outline your design approach (including circuit diagrams and source code), as well as present and analyze experimental results, including a discussion on your system limitations. The report should contain enough information so that another engineer could easily reproduce your system. Each group member will earn the same mark for that group's report. This deliverable is worth **60%** of your final lab grade.

Additional Materials

Additional up-to-date information, including "Project Tips", can be found on the ECE 355 lab website:

http://www.ece.uvic.ca/~ece355/lab.

APPENDIX A: File ***main.c*** for Part 1 of the Introductory Lab

```
//
// This file is part of the GNU ARM Eclipse distribution.
// Copyright (c) 2014 Liviu Ionescu.
//

// ----------------------------------------------------------------------------
// School: University of Victoria, Canada.
// Course: ECE 355 "Microprocessor-Based Systems".
// This is tutorial code for Part 1 of Introductory Lab.
//
// See "system/include/cmsis/stm32f0xx.h" for register/bit definitions.
// See "system/src/cmsis/vectors_stm32f0xx.c" for handler declarations.
// ----------------------------------------------------------------------------

#include <stdio.h>
#include "diag/Trace.h"
#include "cmsis/cmsis_device.h"

// ----------------------------------------------------------------------------
//
// STM32F0 empty sample (trace via $(trace)).
//
// Trace support is enabled by adding the TRACE macro definition.
// By default the trace messages are forwarded to the $(trace) output,
// but can be rerouted to any device or completely suppressed, by
// changing the definitions required in system/src/diag/trace_impl.c
// (currently OS_USE_TRACE_ITM, OS_USE_TRACE_SEMIHOSTING_DEBUG/_STDOUT).
//

// ----- main() ---------------------------------------------------------------

// Sample pragmas to cope with warnings. Please note the related line at
// the end of this function, used to pop the compiler diagnostics status.
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wunused-parameter"
#pragma GCC diagnostic ignored "-Wmissing-declarations"
#pragma GCC diagnostic ignored "-Wreturn-type"


/* Clock prescaler for TIM2 timer: no prescaling */
#define myTIM2_PRESCALER ((uint16_t)0x0000)
/* Delay count for TIM2 timer: 1/4 sec at 48 MHz */
#define myTIM2_PERIOD ((uint32_t)12000000)


void myGPIOA_Init(void);
void myGPIOC_Init(void);
void myTIM2_Init(void);


/* Global variable indicating which LED is blinking */
volatile uint16_t blinkingLED = ((uint16_t)0x0100);
```

```
int
main(int argc, char* argv[])
{

	// By customizing __initialize_args() it is possible to pass arguments,
	// for example when running tests with semihosting you can pass various
	// options to the test.
	// trace_dump_args(argc, argv);

	// Send a greeting to the trace device (skipped on Release).
	trace_puts("Hello World!");

	// The standard output and the standard error should be forwarded to
	// the trace device. For this to work, a redirection in _write.c is
	// required.
	puts("Standard output message.");
	fprintf(stderr, "Standard error message.\n");

	// At this stage the system clock should have already been configured
	// at high speed.
	trace_printf("System clock: %u Hz\n", SystemCoreClock);


	myGPIOA_Init();		/* Initialize I/O port PA */
	myGPIOC_Init();		/* Initialize I/O port PC */
	myTIM2_Init();		/* Initialize timer TIM2 */

	while (1)
	{
		/* If button is pressed, switch between blue and green LEDs */
		if((GPIOA->IDR & GPIO_IDR_0) != 0)
		{
			/* Wait for button to be released (PA0 = 0) */
			while((GPIOA->IDR & GPIO_IDR_0) != 0){}

			/* Turn off currently blinking LED */
			GPIOC->BRR = blinkingLED;

			/* Switch blinking LED */
			blinkingLED ^= ((uint16_t)0x0300);

			/* Turn on switched LED */
			GPIOC->BSRR = blinkingLED;

			trace_printf("\nSwitching the blinking LED...\n");
		}
	}

	return 0;

}
```

```
void myGPIOA_Init()
{
	/* Enable clock for GPIOA peripheral */
	RCC->AHBENR |= RCC_AHBENR_GPIOAEN;

	/* Configure PA0 as input */
	GPIOA->MODER &= ~(GPIO_MODER_MODER0);
	/* Ensure no pull-up/pull-down for PA0 */
	GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPDR0);
}


void myGPIOC_Init()
{
	/* Enable clock for GPIOC peripheral */
	RCC->AHBENR |= RCC_AHBENR_GPIOCEN;

	/* Configure PC8 and PC9 as outputs */
	GPIOC->MODER |= (GPIO_MODER_MODER8_0 | GPIO_MODER_MODER9_0);
	/* Ensure no pull-up/pull-down for PC8 and PC9 */
	GPIOC->PUPDR &= ~(GPIO_PUPDR_PUPDR8 | GPIO_PUPDR_PUPDR9);
	/* Ensure push-pull mode selected for PC8 and PC9 */
	GPIOC->OTYPER &= ~(GPIO_OTYPER_OT_8 | GPIO_OTYPER_OT_9);
	/* Ensure high-speed mode for PC8 and PC9 */
	GPIOC->OSPEEDR |= (GPIO_OSPEEDER_OSPEEDR8 | GPIO_OSPEEDER_OSPEEDR9);
}


void myTIM2_Init()
{
	/* Enable clock for TIM2 peripheral */
	RCC->APB1ENR |= RCC_APB1ENR_TIM2EN;

	/* Configure TIM2: buffer auto-reload, count up, stop on overflow,
	 * enable update events, interrupt on overflow only */
	TIM2->CR1 = ((uint16_t)0x008C);

	/* Set clock prescaler value */
	TIM2->PSC = myTIM2_PRESCALER;
	/* Set auto-reloaded delay */
	TIM2->ARR = myTIM2_PERIOD;

	/* Update timer registers */
	TIM2->EGR = ((uint16_t)0x0001);

	/* Assign TIM2 interrupt priority = 0 in NVIC */
	NVIC_SetPriority(TIM2_IRQn, 0);
	// Same as: NVIC->IP[3] = ((uint32_t)0x00FFFFFF);

	/* Enable TIM2 interrupts in NVIC */
	NVIC_EnableIRQ(TIM2_IRQn);
	// Same as: NVIC->ISER[0] = ((uint32_t)0x00008000);

	/* Enable update interrupt generation */
	TIM2->DIER |= TIM_DIER_UIE;
	/* Start counting timer pulses */
	TIM2->CR1 |= TIM_CR1_CEN;
}
```

```
/* This handler is declared in system/src/cmsis/vectors_stm32f0xx.c */
void TIM2_IRQHandler()
{
	uint16_t LEDstate;

	/* Check if update interrupt flag is indeed set */
	if ((TIM2->SR & TIM_SR_UIF) != 0)
	{
		/* Read current PC output and isolate PC8 and PC9 bits */
		LEDstate = GPIOC->ODR & ((uint16_t)0x0300);
		if (LEDstate == 0)  /* If LED is off, turn it on... */
		{
			/* Set PC8 or PC9 bit */
			GPIOC->BSRR = blinkingLED;
		}
		else                /* ...else (LED is on), turn it off */
		{
			/* Reset PC8 or PC9 bit */
			GPIOC->BRR = blinkingLED;
		}

		TIM2->SR &= ~(TIM_SR_UIF);        /* Clear update interrupt flag */
		TIM2->CR1 |= TIM_CR1_CEN;         /* Restart stopped timer */
	}
}


#pragma GCC diagnostic pop

// ----------------------------------------------------------------------------
```