***Embedded System Design Course***

**LAB 1: Programming a processor with Arm Assembly Language and C**

Contents

[1 Introduction 1](#_Toc86679933)

[1.1 Learning Outcomes 1](#_Toc86679934)

[2 Lab Exercise 1](#_Toc86679935)

[2.1 Mixing Assembly Language and C Code 1](#_Toc86679936)

[2.2 Main 1](#_Toc86679937)

[2.3 Register Conventions 2](#_Toc86679938)

[2.3.1 Calling functions and Passing Arguments 2](#_Toc86679939)

[2.3.2 Temporary storage 2](#_Toc86679940)

[2.3.3 Preserved Registers 2](#_Toc86679941)

[2.3.4 Returning from Functions 2](#_Toc86679942)

[2.4 String Copy 2](#_Toc86679943)

[2.5 String Capitalization 3](#_Toc86679944)

[3 Lab Procedure 5](#_Toc86679945)

[4 Please answer the following questions based on your program debugging 7](#_Toc86679945)

[5 Troubleshooting](#_Toc86679946) 9

# Introduction

## Learning Outcomes

In this module, we will program an Arm processor on a Fixed Virtual Platform using a mixture of Assembly language and C code.

At the end of this lab, you will be able to:

* Modify a C program with assembly subroutines to perform string copy and capitalization operations.
* Compile a C program and execute the program on a Fixed Virtual Platform in the Vitis tool.
* Demonstrate how to step through the code and examine register and variable values in debug mode.
* Identify known issues related to connection issues in the debug configuration and apply troubleshooting solutions.

# Lab Exercise

## Mixing Assembly Language and C Code

We will use Arm Development Studio with a C program, but add assembly subroutines to perform string copy and capitalization operations.

Note that some embedded systems are coded purely in assembly language, but most are coded in C with assembly language used only for time-critical processing, if at all. This is because the code development process is much faster (and hence less expensive) when writing in C when compared to assembly language.

Writing assembly code as functions that can be called from C code as C functions result in modular programs, which gives us the best of both worlds: the fast, modular development of C and the high performance of assembly code. It is also possible to add inline assembly code to C code, but this requires much greater knowledge of how compilers generate code.

## Main

First, we will create the main C function. This function contains two variables (a and b) with character arrays.

int main(void)

{

const char a[] = "Hello world!";

char b[20];

my\_strcpy(a, b);

my\_capitalize(b);

while (1)

;

}

## Register Conventions

There are certain register use conventions that we need to follow if we would like our assembly code to coexist with C code.

### Calling functions and Passing Arguments

When a function calls a subroutine, it places the return address in the link register (LR). The arguments (if any) are passed in registers R0 through R3, starting with R0. If there are more than four arguments, or if they are too large to fit in 32-bit registers, they are passed on the stack.

### Temporary storage

Registers R0 through R3 can be used for temporary storage if they were not used for arguments, or if the argument value is no longer needed.

### Preserved Registers

Registers R4 through R11 must be preserved by a subroutine. If any must be used, they must be saved first and restored before returning. This is typically done by pushing them to and popping them from the stack.

### Returning from Functions

Because the return address has been stored in the LR, the BX lr instruction will reload the Program Counter (PC) with the return address value from the LR. If the function returns a value, it will be passed through register R0.

## String Copy

The function my\_strcpy has two arguments (src, dst). Each is a 32-bit long pointer to a character. In this case, a pointer fits into a register, so argument src is passed through register R0 and dst is passed through R1.

Our function will load a character from memory.

\_\_asm void my\_strcpy(const char \*src, char \*dst)

{

loop

LDRB r2, [r0] ; Load byte into r2 from memory pointed to by r0 (src pointer)

ADDS r0, #1 ; Increment src pointer

STRB r2, [r1] ; Store byte in r2 into memory pointed to by r1 (dst pointer)

ADDS r1, #1 ; Increment dst pointer

CMP r2, #0 ; Was the byte 0?

BNE loop ; If not, repeat the loop

BX lr ; Else return from subroutine

}

## String Capitalization

Let’s look at a subroutine to capitalize all the lowercase letters in the string. We need to load each character, check to see if it is a letter, and if so, capitalize it.

Each character in the string is represented with its ASCII code. For example, “A” is represented with a 65 (0x41), “B” with 66 (0x42), and so on up to “Z,” which uses 90 (0x5a). The lowercase letters start at “a” (97, or 0x61) and end with “z” (122, or 0x7a). We can convert a lowercase letter to an uppercase letter by subtracting 32.

\_\_asm void my\_capitalize(char \*str)

{

cap\_loop

LDRB r1, [r0] ; Load byte into r1 from memory pointed to by r0 (str pointer)

CMP r1, #'a'-1 ; compare it with the character before 'a'

BLS cap\_skip ; If byte is lower or same, then skip this byte

CMP r1, #'z' ; Compare it with the 'z' character

BHI cap\_skip ; If it is higher, then skip this byte

SUBS r1,#32 ; Else subtract out difference to capitalize it

STRB r1, [r0] ; Store the capitalized byte back in memory

cap\_skip

ADDS r0, r0, #1 ; Increment str pointer

CMP r1, #0 ; Was the byte 0?

BNE cap\_loop ; If not, repeat the loop

BX lr ; Else return from subroutine

}

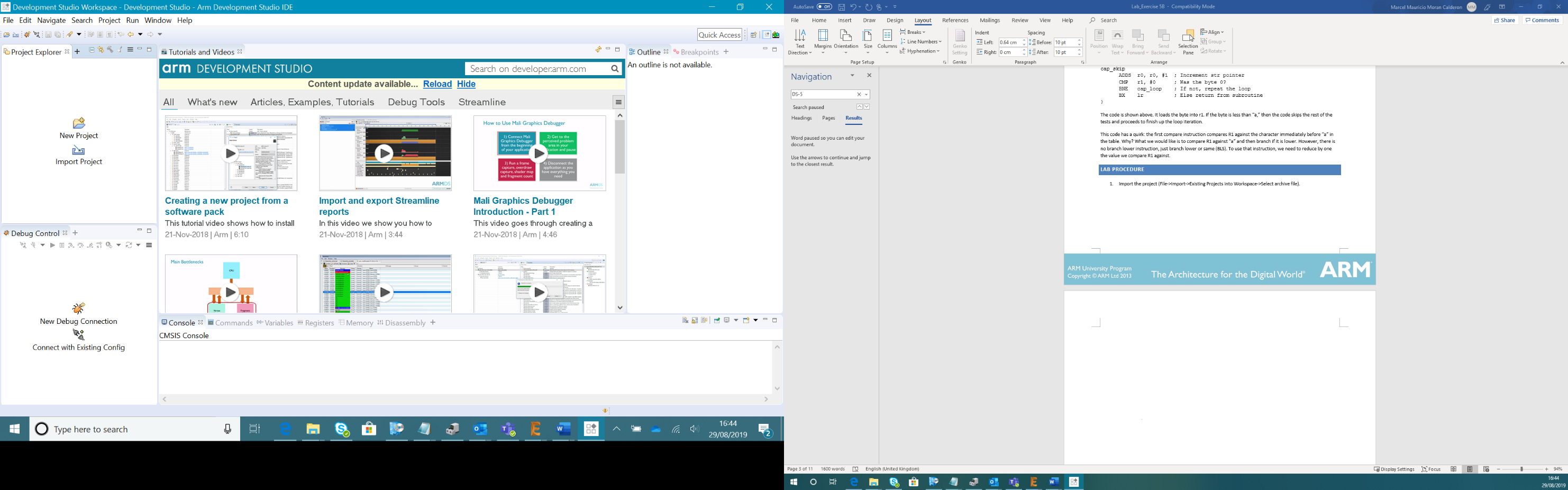
The code is shown above. It loads the byte into r1. If the byte is less than “a,” then the code skips the rest of the tests and proceeds to finish up the loop iteration.

This code has a quirk: the first compare instruction compares R1 against the character immediately before “a” in the table. Why? What we would like is to compare R1 against “a” and then branch if it is lower. However, there is no branch lower instruction, just branch lower or same (BLS). To use that instruction, we need to reduce by one the value we compare R1 against.

# Lab Procedure

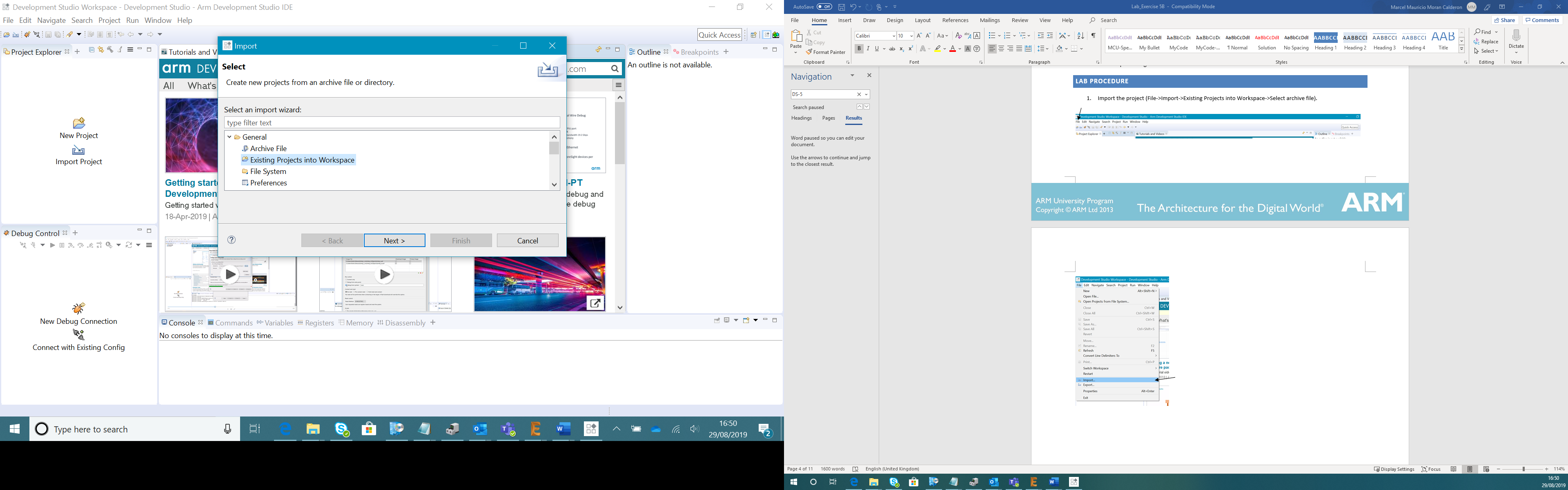
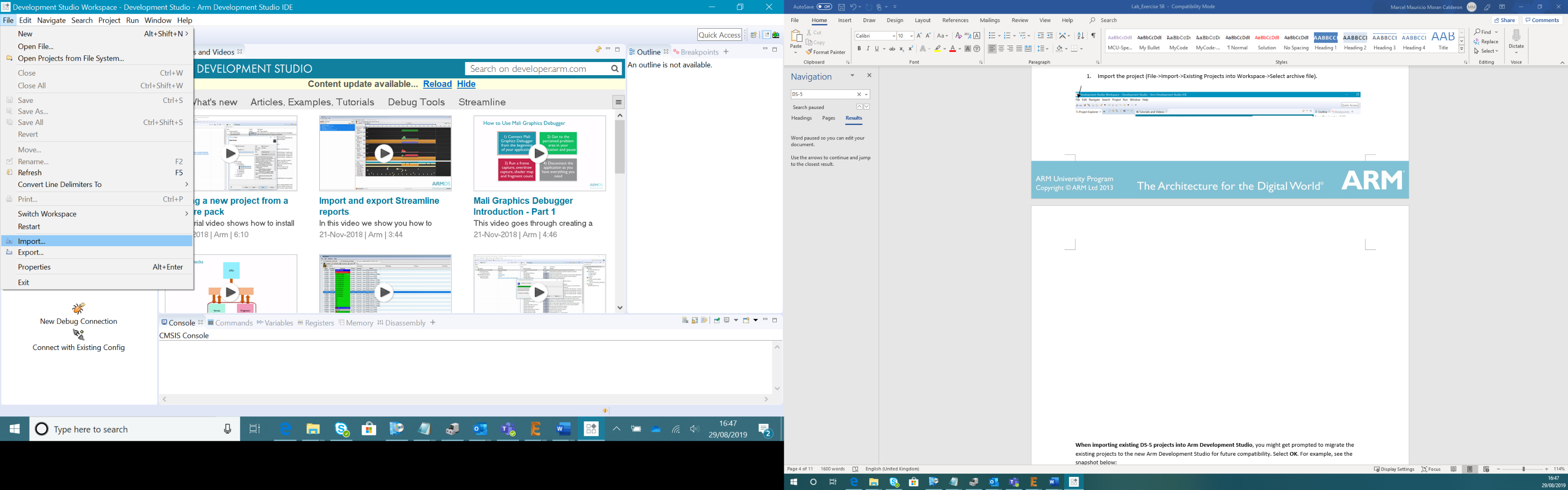
1. Import the project (File->Import->Existing Projects into Workspace->Select archive file).

a



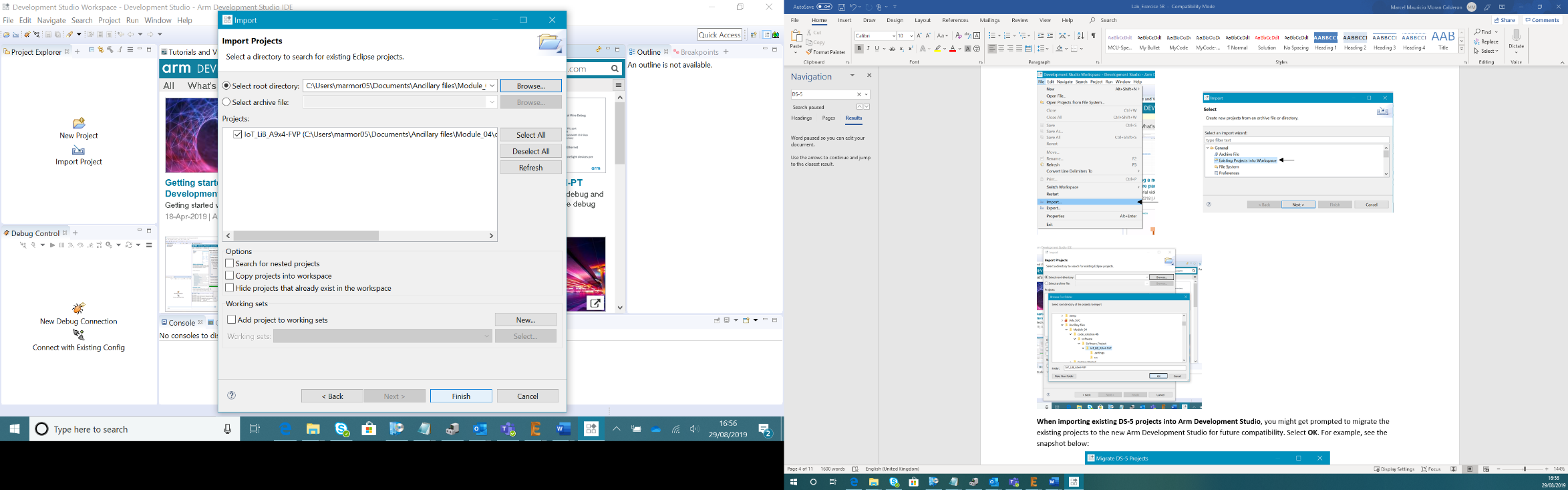
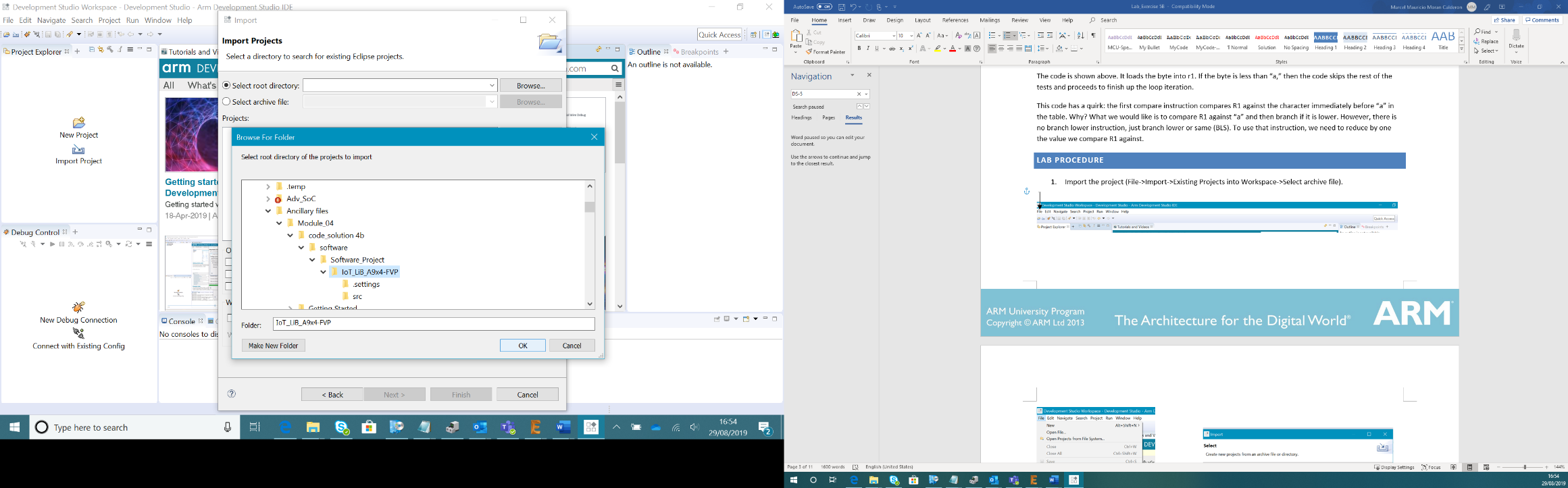
b

c



e

d



**Note: The project file supplied was done in Arm DS-5. You will need to import the existing DS-5 project files into Arm Development Studio.**

**When importing existing DS-5 projects into Arm Development Studio**, you might get prompted to migrate the existing projects to the new Arm Development Studio for future compatibility. Select **OK**. For example, see the snapshot below:

A picture containing text, screenshot, computer

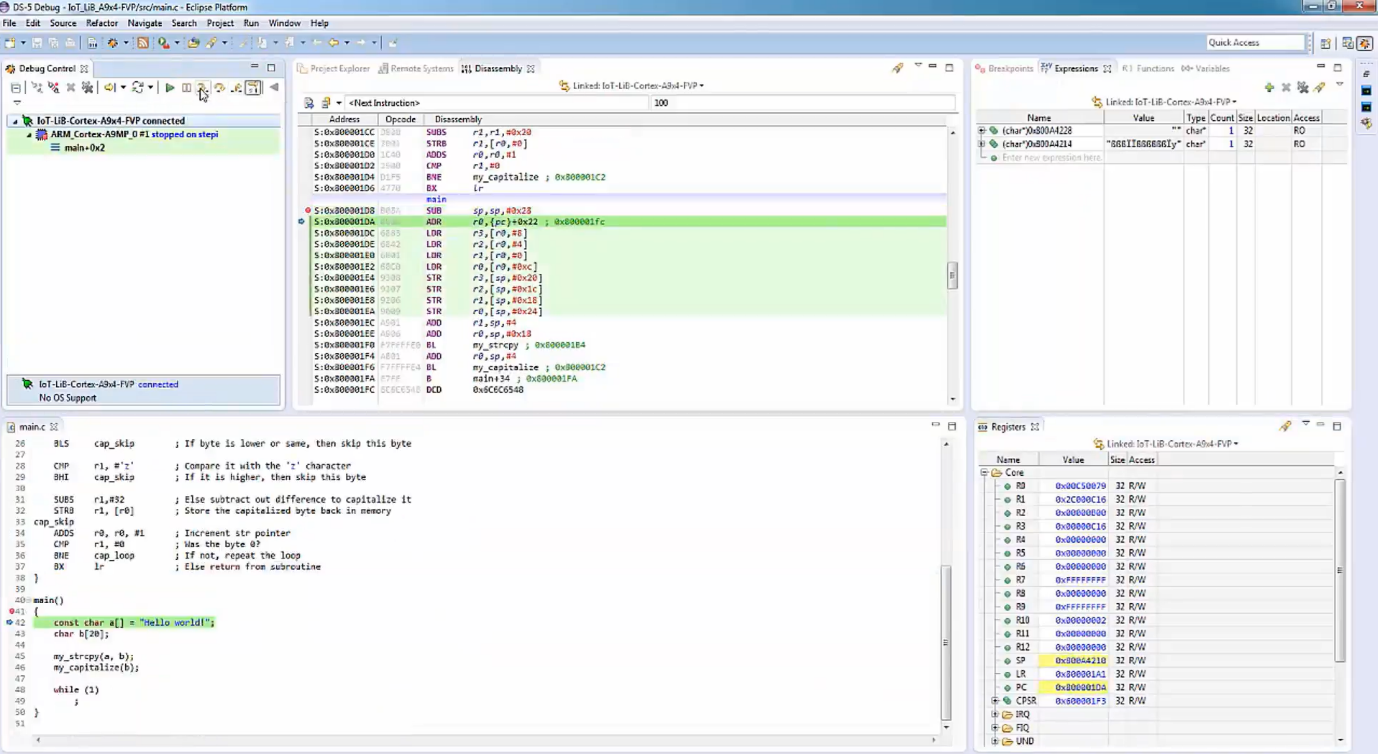
Description automatically generated

1. Run debugging and connect to Cortex-A9x4 FVP (Run->Debug Configurations, double click on IoT-Lib-Cortex-A9x4-FVP).

**Note**: If you experience an error at this stage, see the **Troubleshooting** section at the end of this document.

Graphical user interface, text, application, email

Description automatically generated



# Please answer the following questions based on your program debugging

1. Run the program until the opening brace in the main function is highlighted. Open the Registers window (Window->Show View->Registers). What are the values of the stack pointer (SP), LR, and the PC? (Insert screenshot of your editor.) **(5 points)**

Write answer in different text color

1. Open the Disassembly window (Window->Show View->Disassembly). Which instruction is highlighted, and what is its address? How does this address relate to the value of PC? (Insert screenshot of your disassembly window.) **(5 points)**

Write answer in different text color

1. Switch to “Step by Instruction” mode by clicking  in the Debug Control window. Step one machine instruction using the F5 key while the Disassembly window is selected. Which two registers have changed (they should be highlighted in the Registers window), and how do they relate to the instruction just executed? (Insert the screenshot.) **(5 points)**

Write answer in different text color

1. Look at the instructions in the Disassembly window. Do you see any instructions that are four bytes long? If so, what are the instructions? (Insert screenshot.) **(5 points)**

Write answer in different text color

1. Continue execution (using F5) until reaching the BL my\_strcpy instruction. What are the values of the SP, PC, and LR? (Insert screenshot.) **(5 points)**

Write answer in different text color

1. Watch the Variables window (Window->Show View-> Variables) to analyze the variables “a” and “b” (Insert screenshot). What is the value of “a”? What is the value of “b”? **(5 points)**

Write answer in different text color

1. Which registers hold the arguments to my\_strcpy, and what are their contents? (Insert screenshot.) **(5 points)**

Write answer in different text color

1. Use the Expressions window to watch the values in the address held in R0 and R1. Do the values match variables “a” and “b”? (Insert screenshot.) **(5 points)**

Write answer in different text color

1. Execute the BL instruction. What are the values of the SP, PC, and LR? What has changed and why? Does the PC value agree with what is shown in the Disassembly window? (Insert screenshot.) **(5 points)**

Write answer in different text color

1. Single step through the assembly code watching the “Expressions” window to see the string being copied character by character from a to b. Which register holds the character? **(5 points)**

Write answer in different text color

1. What are the values of the character, the src pointer, the dst pointer, the LR, and the PC when the code reaches the last instruction in the subroutine (BX lr)? (Insert screenshot.) **(5 points)**

Write answer in different text color

1. Execute the BX lr instruction. Now what is the value of the PC? **(5 points)**

Write answer in different text color

1. What is the relationship between the PC value and the previous LR value? Explain. **(5 points)**

Write answer in different text color

1. Now step through the my\_capitalize subroutine and verify it works correctly, converting from “Hello world!” to “HELLO WORLD!”. ((Insert final screenshot.) **(5 points)**

Write answer in different text color

1. Please explain your debugging experience in your own words **(10 points)**

Write answer in different text color

1. Video Demonstration **(20 points)**

# Troubleshooting

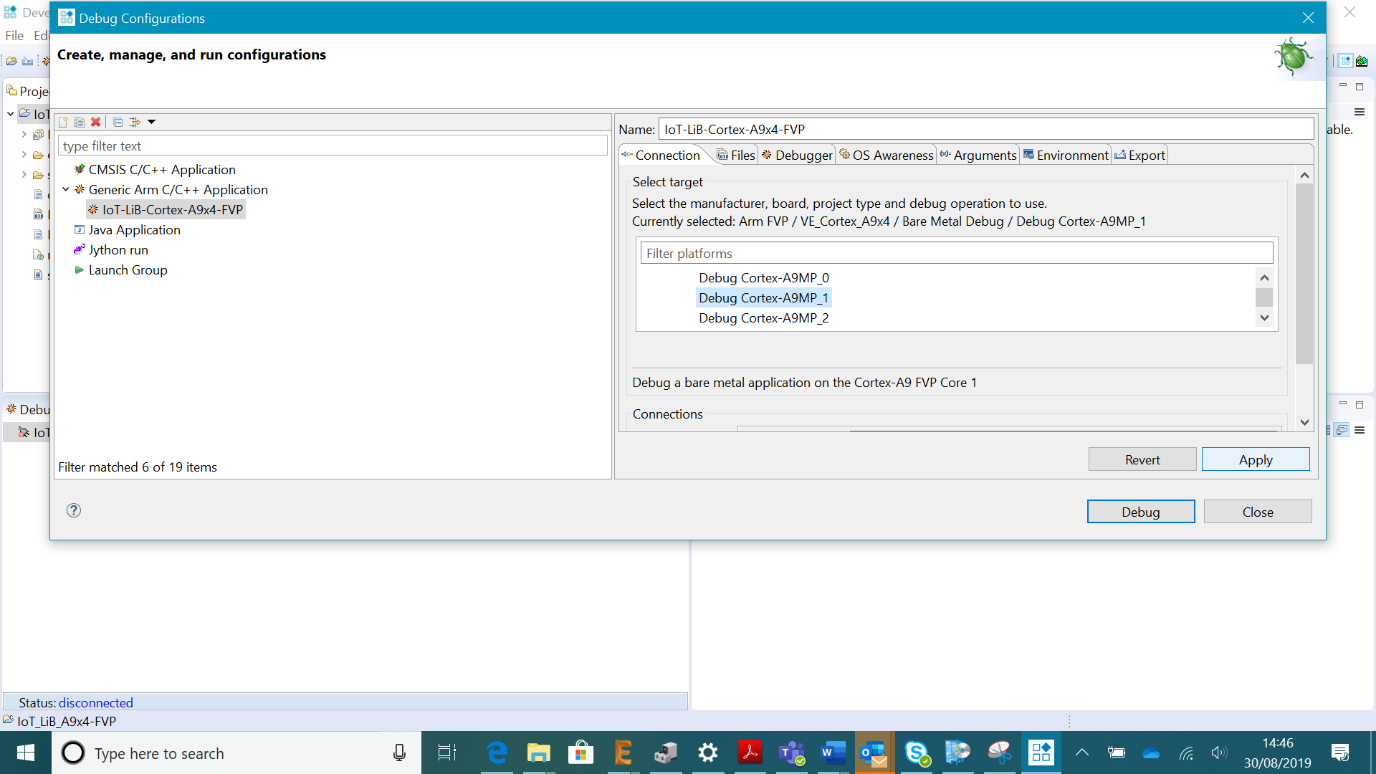
When debugging the program, you may experience a connection error as shown below:

Graphical user interface, application, Word

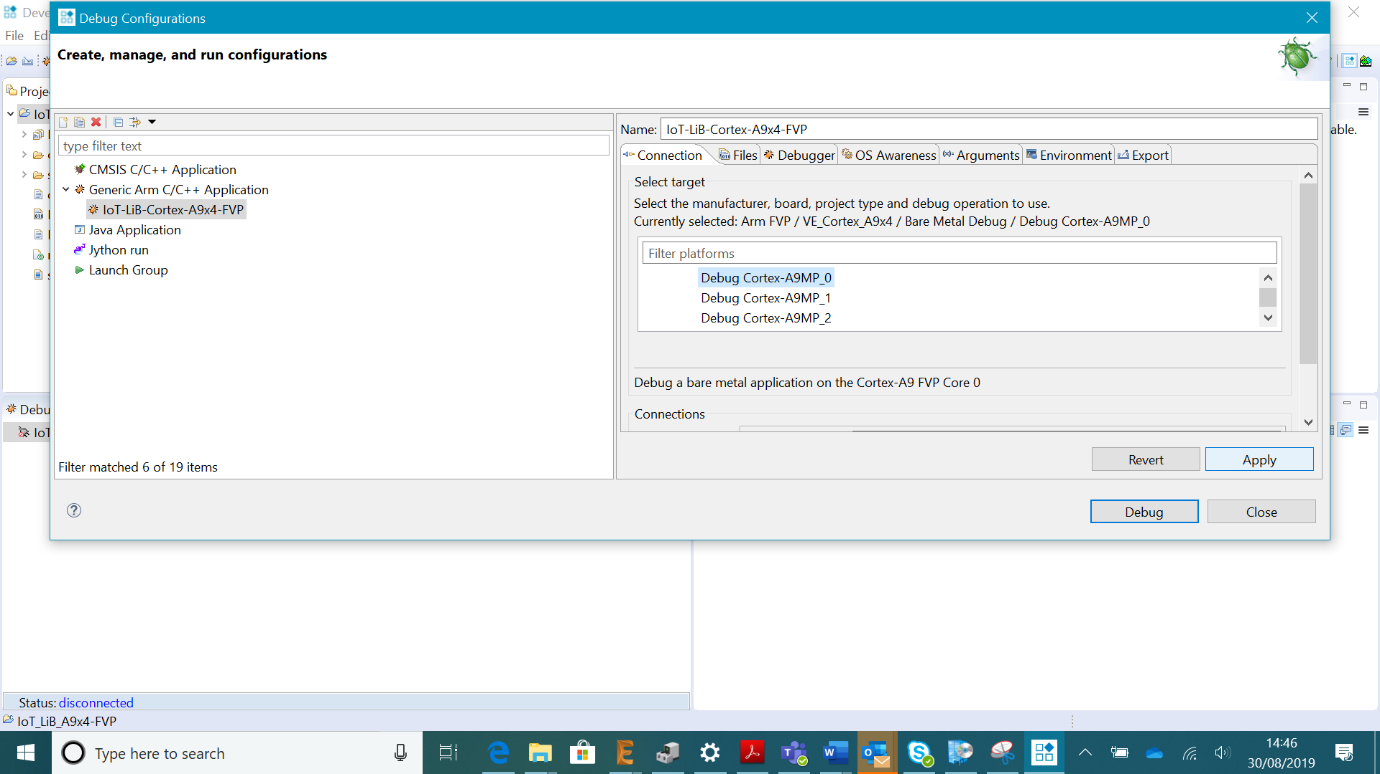
Description automatically generated

As a workaround, you can try the following steps:

1. Change Debug Cortex-A9MP\_0 to Debug Cortex-A9MP\_1 in Debug Configurations.



1. Go back to the old configurations and select Debug again.



1. The program should now start.

A screenshot of a computer

Description automatically generated