Microprocessor Systems Lab 4

Checkoff and Grade Sheet

Partner 1 Name: _____

Partner 2 Name: _____

Grade Component	Max.	Points Awarded Partner 1	Points Awarded Partner 2	TA Init.s	Date
Performance Verification: Task 1	10 %				
Task 2	10 %				
Task 3	10 %				
Task 4	10 %				
Task 5 [Depth]	10 %				

→ Laboratory Goals

By completing this laboratory assignment, you will learn to use:

1. the Analog-to-Digital Converter (ADC),
2. the Digital-to-Analog Converter (DAC),
3. the Digital Signal Processor (DSP) and Floating Point Unit (FPU),
4. ARM Assembly instructions to increase efficiency.

→ Reading and References

R1. Mastering STM32: Chapters 12 (ADC), 13 (DAC)

R2. UM1905-stm32f7_HAL_and_LL_Drivers.pdf: Chapters 6 (ADC), 15 (DAC)

R3. RM0410-stm32f76xxx_Reference_Manual.pdf: Chapters 15 (ADC), 16 (DAC)

R4. PM0253-stm32f7_Programming_Manual.pdf: Instruction sets: Sections 3.1 (summary), 3.5 (general processing), 3.6 (multipy and divide), 3.11 (floating-point)

R5. stm32f769xx_Datasheet.pdf

R6. AN4841-stm32_DSP_using_CMSIS.pdf: Overview of DSP functionality provided through assembly or CMSIS commands. Skim all.

R7. AN3116-stm32_ADC_modes.pdf: Overview of ADC - Section 1 Only.

R8. OperandsAndConstraints.pdf::A list of inline assembly operand and constraint modifiers.

R9. Lab04_Template.zip: Project Template for Lab 4

Analog To Digital Converter (ADC)

→ ADCs on the STM32

The STM32F769NI has 3 built-in 12-bit successive approximation ADCs with up to 19 channels (16 external: ADC_IN0-ADC_IN15, 2 internal, and 1 on V_{BAT}). These channels are shared between the three ADCs as described in Figures 71-73 of R3. Resolution is configurable with 6, 8, 10, and 12-bit modes, and data conversion can be done via single, continuous, scan (samples a pre-selected group of channels), or discontinuous sampling modes.

Sampling of multiple channels in succession is done through channel grouping. There are two kinds of channels groups: regular and injected. A regular group is a set of channels that are set and used whenever a program calls for a conversion, while an injected group is similar to an interrupt: an external trigger causes the injected group to interrupt the conversion of a regular group. If an injected group is triggered during the conversion of a regular group, the current conversion is canceled and resumed after the conversion of the injected group is completed, as shown in Figure 1.

A regular group can be composed of up to 16 conversions, with the conversion order of the selected conversions fully configurable, including having channels be repeated in the sequence. For example, it is possible to configure a regular group to convert in the following sequence: ADC_IN3, ADC_IN15, ADC_IN3, ADC_IN10, ADC_IN3. Additionally, each of these channels may have varying sampling times configured

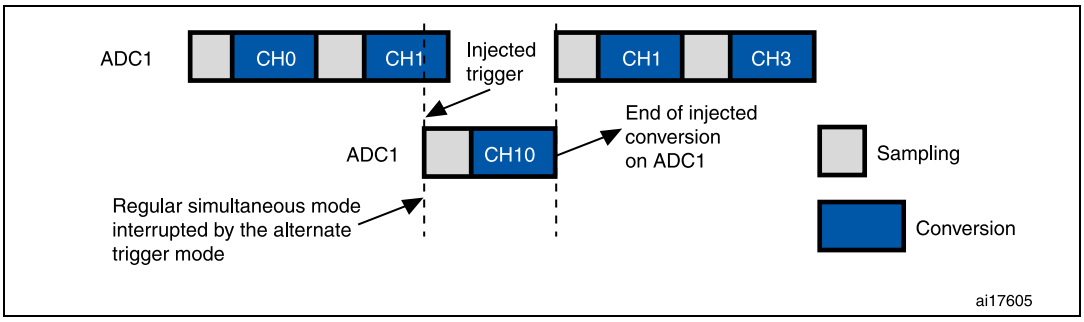

Figure 1: Injected ADC conversions interrupting regular group. Reproduced from Figure 6 of R7.

as well, though a single channel cannot have multiple different sampling times even if it appears multiple times in a sequence. In multichannel (scan), single conversion mode, the entirety of the regular group would be sampled in one continuous sequence, with the ADC waiting for instruction after the conversions are completed. In continuous mode, the regular group would continuously be sampled without pause.

An injected group can be composed of up to 4 conversions, again, with the sequence and timing of each conversion being fully configurable.

The output generated from a conversion of 1 channel is given by the Equation 1:

$$\text{ConversionResult} = \begin{cases} 0 & V_{in} < 0, \\ \left\lfloor (2^{n_{bits}} - 1)\, \frac{V_{in}}{V_{ref}} \right\rfloor & 0 \leq V_{in} \leq V_{ref}, \\ 2^{n_{bits}} - 1 & V_{in} > V_{ref}, \end{cases} \tag{1}$$

where n_{bits} is the resolution of the ADC (6, 8, 10, or 12 bits), V_{in} is the voltage input to the ADC and V_{ref} is the ADC reference voltage, which for this labs purposes will be 3.3 V. In other applications, V_{ref} may be specified through a connection to the V_{REF+} pin.[1]

The simplest mode to operate the ADC in is polling mode in single channel, single conversion mode, where the program requests one conversion at a time and returns that result. When using multichannel and/or continuous modes, care must be taken to ensure that the previously converted value is received from the ADC (from register `ADCx_DR`) before the next conversion completes, else the data will be overwritten and lost. For these modes, it may be more convenient to operate in a interrupt or DMA mode. For both modes, the samples may be recorded after each conversion without continuously polling, significantly freeing up processing time. Note that injected conversions are stored individually into the `ADCx_JDRy` registers and therefore only need to be recorded after the complete injected group has completed instead of independently for each conversion.

[1] To use an external V_{ref}, resistor R110 on the DISCO board would need to be removed. By default, this resistor connects the analog power supply VDDA (3.3 V) to V_{REF+}.

→ Configuring and Using the ADCs

The following steps are an example sequence of instructions to enable and use the ADCs in polling mode through the use of the HAL drivers:

1. Turn on the ADC peripheral clock from APB2,
2. Configure the GPIO pins for use as analog inputs within the `HAL_ADC_MspInit())` callback function,
3. Configure the ADC module through the `ADC_HandleTypeDef` struct and `HAL_ADC_Init()` function,
4. Add channels to the regular group using `ADC_ChannelConfTypeDef` and `HAL_ADC_ConfigChannel()`[2]
5. Start a conversion by calling `HAL_ADC_Start()`
6. Wait for conversion to complete, continuously checking either `HAL_ADC_PollForConversion()` or the macro `__HAL_ADC_GET_FLAG` and the flag `ADC_SR_EOC`[3].
7. Retrieve the converted value through `HAL_ADC_GetValue()`.

[2]Injected channels are not supported by the HAL ADC Generic Driver, the ADC Extension Driver would be needed.

[3]If using the macro to wait for conversion to complete, the flag does not need to be manually cleared through `__HAL_ADC_CLEAR_FLAG`, as it is automatically cleared when the converted value is read.

◇ Task 1: Simple Voltmeter

Write a program to sample a voltage on any of the analog pins available from the DISCO board *except* PA4 (Arduino pin A1) every 100 ms. The program should use a single channel/single conversion sampling mode using the full 12-bit ADC resolution.

The program display should have the following information: (i) the current hexadecimal ADC value (ii) the calculated decimal voltage, printed with a resolution of 1 mV, (iii) a running average of the decimal voltage over the last 10 s, and (iv) the minimum and maximum voltages measured over the last 10 s.

Check the output results with a voltmeter and hand calculation to verify proper operation of the program through several cycles. The program must be tested with a DC voltage and a varying voltage (e.g., a slow sine wave).

Warning: **Arduino pin A1 is not 5 V tolerant. Applying a voltage outside GND-3.3 V may damage the pin.** Arduino pins A0, A2-A5 are 5 V tolerant; however, any voltage outside of the GND-5 V range may damage the pin and possibly the whole microcontroller. It is highly recommended that before you connect any signal generator to the DISCO board you measure its amplitude and ensure it is within GND-5 V. Additionally, feeding an analog voltage to the pin with a large series resistor (e.g., 10 kΩ) will also help in preventing damage, though is not necessary. Finally, a potentiometer may be used instead in order to produce a safe and controllable, though inaccurate, analog source.

NOTES:

1. It is suggested that students consider using a circular-, or ring-, buffer to keep track of the 10 s span.
2. Many of the items in the struct `ADC_HandleTypeDef` may be ignored in the single channel, single sample mode.
3. Be aware the timing restrictions and behaviors of the ADC. Specifically, the ADC clock frequency, sampling time, sampling rates, total conversion and startup times. See Table 71 of R5. Note that the APB2 CLK is 108 MHz, as configured in `init.c`
4. There is no distinction between GPIO analog inputs and outputs in the HAL. Simply setting the mode to `GPIO_MODE_ANALOG` will suffice.

Digital To Analog Converter (DAC)

→ DACs on the STM32

The STM32F769NI has 2 built-in 12-bit DACs with 1 output channel each. Resolution is configurable with 8 and 12-bit modes, and each DAC channel has its own converter, so simultaneous conversions are possible via dual mode (e.g., use both DACs at once). Unfortunately, one of the DACs (DAC_OUT2) is on a pin shared with the USB 2.0 On-the-Go host controller and is connected via PCB trace to pin 1 of the USB3320C-EZK chip. This pin is vital to achieve USB 2.0 speeds, so we can only use DAC_OUT1 connected to PA4 on Arduino A1.

Operation and configuration of the DACs is much less flexible than the ADCs. The resolution of the DAC may be configured to be either 8- or 12-bits, which is useful in matching the format of an ADC channel, though 6- and 10-bit resolution is not available.

Triggering of the DAC may be done by simply writing a value to the DAC when using trigger type `DAC_TRIGGER_NONE`. Additionally, timers, external lines, and software triggering may be used as well. The voltage produced by the DAC is given by Equation 2.

$$V_{\text{DAC}} = V_{ref} \frac{N_{DAC}}{2^{n_{bits}}} \tag{2}$$

V_{ref} is the reference voltage, again, for this lab's purposes is 3.3 V, N_{DAC} is the input value given to the DAC, or the digital value to be converted to analog, and n_{bits} in this case is the resolution of the DAC. Note that the denominator in Equation 2 is $2^{n_{bits}}$ and not $2^{n_{bits}} - 1$, as for the ADC (Equation 1), therefore, the DACs' voltage resolution is *slightly* smaller than that of the ADCs', assuming that n_{bits} is the same for each.

→ Configuring and Using the DACs

The following steps are an example sequence of instructions to enable and use the DACs write-triggering mode through the use of the HAL drivers:

1. Turn on the DAC peripheral clock from APB1,
2. Configure the GPIO pins for use as analog outputs within the `HAL_DAC_MspInit())` callback function,
3. Configure the DAC module through the `DAC_HandleTypeDef` struct and `HAL_ADC_Init()` function. Note that very little is needed to be done for this step.
4. Configure the triggering (`DAC_TRIGGER_NONE`) and buffering[4] of the DAC_OUTn signal using the struct `DAC_ChannelConfTypeDef` and `HAL_DAC_ConfigChannel()` function.
5. Start the DAC via `HAL_DAC_Start()`.
6. Set the output value and associated resolution with `HAL_DAC_SetValue()`.

[4]Each DAC channel has an optional output buffer to allow driving of external loads. It will not be required here.

◇ Task 2: DAC Output

Write a program to convert digital values to analog signals and output them to the DAC_OUT1 pin.

The program should be developed in two steps:

First, write a short routine that continuously increments an unsigned integer starting from 0. Once the integer reaches the max allowed DAC output value, it should start back at 0 again. This will generate a 3.3 V sawtooth wave on the DAC output. Confirm this with an oscilloscope. Notice the output frequency might not be as high as you expected (if you expected MHz). *Why*?

Next, enable ADC1 and connect it to a signal generator. Set up the generator to send a sine wave. Take the conversions generated by ADC1, and output them through DAC1. The signal you see through the oscilloscope should be very close to the signal that is being sent from the signal generator *at lower frequencies*. As the frequency increases, the quantization effects from the ADC and DAC should become evident as the output begins to deviate from the input.

NOTES:

1. Similar timing characteristics as considered for the ADC should be considered for the DAC. See Table 83 of R5.
2. Verification for this task should include a case where the output does not match the input due to quantization effects.

Digital Signal Processing

Devices conforming to the ARM CMSIS specification include dedicated hardware for mathematical operations. This hardware collectively forms the DSP subsystem of the Cortex ARM core. A short overview of the functionality and use cases is given in R6. There are many ways to access the Cortex-M7 DSP functionality on the DISCO board, though primarily this would be done through:

1. Through automatic compiler optimizations,
2. Through CMSIS functions provided in the `arm_math.h` library,
3. Directly with ARM assembly instructions.

Each of these methods are useful in different situations. The first method, using automatic compiler optimizations, is by default enabled through the project templates provided in this course. Essentially, if an equation is provided in `C` to the compiler, the compiler will attempted to make this equation as efficient as possible, within reason. This primarily includes using specialized assembly functions without programmer intervention, to perform the evaluations. A prime example of this is the evaluation of floating point numbers. Normally, floating point arithmetic requires many system clock cycles per single operation to complete; however, if the microcontroller has a *Floating-Point Unit* (FPU), then the compiler will instead employ the FPU to perform the operation in significantly less time. Again, this is done without the programmer necessarily needing to specify the use of the FPU for the evaluation as long as the compiler knows that the FPU is present[5].

Using the prebuilt functions provided through `arm_math.h`[6] is a great way to implement very complex mathematics without having to write from scratch and be assured that the function used is efficient. For example, the function `arm_bilinear_interp_f32()` is provided which performs a bilinear floating point interpolation.

Through assembly commands, a finer control of evaluations may be achieved, where each instruction required to run is specified explicitly. This is useful for making extremely efficient code as the compiler will not be capable or aware of all possible optimizations[7]. There are three main subsets of assembly DSP instructions: Saturation, Multiply and Accumulate (MAC), and Single Instruction Multiple Data (SIMD). The saturation instructions are alternatives to the normal addition and subtraction functions, where instead of overflowing the output of an operation, these instructions will "saturate" to either that maximum or minimum value. The SIMD commands allow multiple operations to be done over a single instruction, such as adding two sets of numbers at once. The MAC commands are useful when both a multiplication and addition are required. The MAC commands are extremely useful in filtering applications as successive samples of incoming data are multiplied by weights and then added to a running total:

$$y[k] = \sum_{n=0}^{2} w_n x[k-n] \tag{3}$$

Consider the rudimentary example equation given in Equation 3. Using normal addition and multiplication assembly instructions, a simplified set of operations is given below (left)[8], where each operation

[5]The presence of the FPU is indicated to the compiler through the compiler directives `-mfloat-abi=hard -mfpu=fpv5-d16`, which may be found under *Project Properties* → *C/C++ Build* → *Tool Settings* → *MCU GCC Compiler* → *All options:*

[6]See arm-software.github.io/CMSIS_5/DSP/html/index.html for documentation on the functions available

[7]In many cases the compiler will produce more efficient assembly code than a programmer.

[8]primarily, data movement from memory to registers is omitted in these examples

requires one instruction cycle. Similarly a simplified set of operations for when employing MAC commands is also given below (right). For each set of operations, the corresponding assembly instruction name is also given. From this example, it could be expected that using the MAC commands over independent addition and multiplication could result in a 50 % reduction in evaluation time for the filter. This is not necessarily true for all cases as the supporting commands used to prepare for both methods (data movement, looping, etc.) will have associated overhead and therefore the actual performance gain would be less.

1. $n = 0$: `MUL` $tmp = w_0 x[k-0]$
2. $n = 0$: `ADD` $y[k] = y[k] + tmp$
3. $n = 1$: `MUL` $tmp = w_1 x[k-1]$
4. $n = 1$: `ADD` $y[k] = y[k] + tmp$
5. $n = 2$: `MUL` $tmp = w_2 x[k-2]$
6. $n = 2$: `ADD` $y[k] = y[k] + tmp$

1. $n = 0$: `MLA` $y[k] = y[k] + w_0 x[k-0]$
2. $n = 1$: `MLA` $y[k] = y[k] + w_1 x[k-1]$
3. $n = 2$: `MLA` $y[k] = y[k] + w_2 x[k-2]$

The above example commands only considering integer math; `MUL`, `ADD`, and `MLA` are all integer instructions. Floating point math is possible using the FPU. The FPU uses similar instructions, where FPU commands are commonly just the integer instruction names with a "V" prepended; e.g., `VMUL`, `VADD`, `VMLA`, etc. For a comprehensive list of both types of assembly instructions, among many others, see Chapter 3 of R4. Within the documentation for the FPU assembly commands, it is shown that the FPU does not use the core registers `r0-r12`, but instead has its own "floating point" registers: `s0-s31`, which are 32-bit single precision registers (`float` type), and `d0-d15`, which are 64-bit double precision registers (`double` type).

→ Assembly Instructions

It is necessary to understand the register layout of the microcontroller in order to use the assembly instructions properly. The STM32F769NI has 13 32-bit "general purpose" processor core registers, labeled `r0-r12`. The data to be operated on generally needs to reside in these registers and not in memory (RAM). This implies that to be able to add two variables together, for example, that the contents of those variables first needs to be moved from memory to any of the `r0-r12` registers. Once moved, the ADD command would be issued, with the result being stored into another register, which then must be moved from that register to the output variable's memory location. Considering this, it could take four or more assembly commands to simply add two numbers! Fortunately, for some commands there is flexibility in where the data may reside, be it `r0-r12`, other core registers (e.g., `SP`), or in memory locations. The format for each command is also available in R4, where the command structure is explicitly specified for each variation of operand types.

Continuing with the above example, consider the following command in `C`:

```
var3 = var1 + var2;
```

When this command is compiled *without optimizations*, the assembly code produced may look something like the following[9]:

```
ldr  r3, .L4+4    ; load the memory address of var1 to r3
```

[9] In assembly code, a semicolon denotes a comment, among other characters as well.

```
ldr  r2, [r3]     ; load value of var1 into r2
ldr  r3, .L4+8    ; load the memory address of var2 to r3
ldr  r3, [r3]     ; load value of var2 into r3
add  r3, r3, r2   ; add r2 and r3 together, save in r3
ldr  r2, .L4+12   ; load the memory address of var3 to r2
str  r3, [r2]     ; save the value of r3 in the memory location in r2
```

Of course, if the programmer was writing this in plain assembly, it would be necessary to track the memory addresses of the variables used and move them in and out of memory often, which would become cumbersome. Alternatively, one can use *inline assembly* commands in `C` to implement specific commands, with the supporting commands provided by the compiler. The above equation may be implemented in inline assembly through the command[10][11]:

```
asm("ADD %[out],%[in1],%[in2]"
       :[out] "=r" (var3)
       :[in1] "r" (var1),[in2] "r" (var2));
```

This command will produce code functionally equivalent[12] to the assembly code presented previously without having to write each command individually. Obviously, for this particular application above, using inline assembly is completely useless as it does not provide any advantages over the instructions that the `C` command produces.

The example command above is considered an "extended" inline assembly command, as opposed to "basic" inline assembly. Each type is discussed in the following two sections.

→ Basic Inline Assembly[13]

Basic inline assembly commands are essentially pure assembly code written in `C` . The format for a basic inline assembly command is simply:

`asm ( `*AssemblerInstruction*` )`

where *AssemblerInstruction* is a string explicitly stating the desired code. For this case, there is no interaction between `C` variables and the assembly instruction. For example, consider the following commands:

```
asm("LDR r2, r4"); // Copy value in r4 to r2
asm("LDR r3, =0x000102FA"); // Load the value 0x000102FA into r3
asm("ADD r4, r2 , r3"); // Add r2 and r3, store into r4
```

It is clear what the first and third command are accomplishing, but the format of the second is different. In this case, a constant value is explicitly defined to be loaded into `r3`. This is a generalization of the command syntax:

```
LDR rt, [rn, #offset]
```

[10] This command structure is discussed in the Extended Inline Assembly subsection.

[11] This command may be placed all one line. It was broken into three lines for document clarity.

[12] In fact, when testing this code the compiler produced the exact same set of assembly commands, registers, and memory addresses.

[13] This and the following section were originally adapted from https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/Using-Assembly-Language-with-C.html#Using-Assembly-Language-with-C, which has subsequently been substantially edited.

where the second term in this command is specifying a location in memory to load a value from. Instead, the inline assembly provides the value at an unknown memory location and allows the compiler to select the source memory location (memory location stored in `rn` plus some `offset` value, in bytes) to store that value. If desired, multiple assembly commands may be written with one `asm` statement using `\r\n`, as such:

```
asm("LDR r2, r4 \r\n LDR r3, =0x000102FA \r\n ADD r4, r2 , r3");
```

→ Extended Inline Assembly

Extended inline assembly greatly enhances the flexibility of `C` style inline assembly code. Using extended assembly, it is possible to directly assign `C` variables as the inputs and outputs of assembly instructions. The general format for an extended assembly command is as follows:

`asm` [`volatile`] (*AssemblerTemplate* : *OutputOperands* [: *InputOperands* [: *Clobbers*]])

In this command, *AssemblerTemplate* is an assembly command but with the input and output registers or addresses replaced with placeholders. This is similar to how `printf()` uses the codes `%d`, `%c` or `%x`, for example, as markers for replacement of variable values. In this case, the `%` character is also used in order to indicate a placeholder, with the type specifiers, or "format strings," following. These will be discussed more in depth later in this section. A summary of the available format strings may be found in R8.

The command portion *OutputOperands* provides the `asm` command with the `C` variables to use as the outputs of the assembly command, with *InputOperands* providing the input `C` variables. Finally, the optional *Clobbers* section specifies the registers that are directly modified in the assembly command; necessary in order for the compiler to keep track of which register is in use. Note that both the volatile keyword and the *InputOperands* and *Clobbers* fields are optional, as denoted by the surrounding brackets.

A simple implementation of a this is command to store a register to a variable:

```
uint32_t var;
asm("STR r2,%0" : "=m" (var));
```

In this example, the only format string present in the *AssemblerTemplate* is `%0`. This specifies that the first operand listed is to be used in it's place. Similarly, if `%1` and `%2` were present in the *AssemblerTemplate*, the second and third operand would be matched to those. In this case, the first operand listed is the output operand `"=m" (var)`. The `"=m"` specifies that our output variable is stored in a valid memory address (as opposed to, say, another register) via the letter `m`, and the `=` sign denotes that this variable is to be written to (but not read from). The `C` variable used to store the output is listed within the parenthesis: `(var)`. The format string may have also alternatively taken the form of `%[tag]`, where the string contained within the brackets, in this case `tag`, is an arbitrary name shared between the format string and the desired insertion variable and has not meaning outside of the specific `asm` it is used within. For example:

```
asm("STR r2,%[mps]" : [mps] "=m" (var));
```

The `tag` in this case is `mps`. The tag `mps` also appears within the operand specification, explicitly specifying the relationship of the named operand within the assembler template. The label `mps` does not need to be specified anywhere external to this command as it is only a temporary label and not, for example, a variable.

In the above example, since the assembly commands second operand is a location in memory, it is specified through the "m" *constraint* that the variable should be treated as a location in memory. For other commands, such as ADD, the variable may also be treated as a register via the constraint "r":

```
asm("ADD r0,%[tmp],r1" : : [tmp] "r" (var));
```

Here, var is being used as an input operand: note the two colons and the blank between them, specifying an empty *OutputOperands* field. Since var is being read from and not being written to, the = sign present in the previous command's constraint is omitted. Likewise, if a command requires that a variable be read from and written to in the same operation, then the modifier + would be used; e.g., "+r". The AND instruction requires that each of the input/outputs used be registers, requiring the usage of "r". If instead it was attempted to treat the input variable as a memory location, incorrectly, by using "m", the compiler would throw an error.

For evaluations using the FPU and its associated registers, the letter r would be replaced with either t for the s# registers or w for the d# registers[14].

This type of operation as shown above, when compiled, will not generate one assembly command but instead will generate multiple, as the assembly commands required to move the var variable's value into one of the core register's with be implemented automatically.

Below is an example of a more complex instruction.

```
double c,a,b;
// ...
asm("VMLS.F64 %P[dest],%P[fac1],%P[fac2]"
                      : [dest] "+w" (c)
                      : [fac1] "w" (a) , [fac2] "w" (b));
```

This asm command issues a floating point multiply and subtract instruction (similar to multiply and accumulate) , implementing the C style equation c-=a*b. For this instruction, the "output" variable c is a double type, w, and must be read from and written to, hence the "+w" constraint. The input variables a and b are also of double type but are only read from, therefore the "w" constraint. Note that w is used here in all three cases as the VMLS.F64 instruction requires all values to exist in the d# registers and not in a memory location. Also note the suffix to the VMLS command: .F64, which denotes that the operation is a 64-bit operation. Alternatively, the suffix .F32 may be used to specify single precision operations, of course, implying the use of the s# registers as well. Finally, the format modifier P inside the format strings %P[dest], etc., explicitly specifies that the variable and register used here is double precision. Addition of this or similar modifiers for most cases is not necessary, although it is needed in this case as the compiler will throw a "*selected FPU does not support instruction...*" error without it.

Finally, since the compiler is generally not aware of which registers are being used and reused in certain contexts, it is sometimes necessary to ensure that a specific register allocated to a variable is left alone and not cleared/overwritten within the instruction set. In order to do this, adding the volatile keyword after the asm as well as possibly within the definition of the used variables. This essentially tells the compiler to be more careful with the optimization of the command. Additionally, adding a & to the variable's constraint may help as well:

```
volatile double c;
double a,b;
```

[14] See R8 for a complete list of the operand constraints

```
// ...
asm volatile ("VMLS.F64 %P[dest],%P[fac1],%P[fac2]"
       : [dest] "+&w" (c)
       : [fac1] "w" (a) , [fac2] "w" (b));
```

In a similar vein, a command that mixes use of inputs/outputs and explicit access of registers may incorrectly reuse an explicity allocated register to load a variable, such as the example `ADD` command already used (repeated below).

```
asm("ADD r0,%[tmp],r1" : : [tmp] "r" (var));
```

In this case, the compiler will not be aware of the explicit use of `r0` and `r1` and may use either of these to load in the value of `var`. To avoid this, the compiler must be told to avoid these registers. This is done using the *Clobbers* field:

```
asm("ADD r0,%[tmp],r1" : : [tmp] "r" (var) : "r0" , "r1");
```

Here, `"r0"` and `"r1"` are added as clobbers.

In debugging these commands, it will be very useful to review the generated assembly file produced by the compiler.

◇ Task 3: Simple Assembly Math and Logic

Write a short program that does four separate things and printing the result of each (via `printf()`). Note that there are two sets of tasks, one for each group member.

Task Set 1:

1. In basic inline assembly only: load (into registers) and add two integer numbers (hard-coded). Use extended assembly to pass the resultant value to a `C` variable and print it.
2. In extended assembly: multiply 2 `int32_t` variables (signed long).
3. In extended assembly: evaluate the equation $\frac{2x}{3} + 5$ through 32-bit integer math: addition (`ADD`), multiplication (`MUL`), and division (`DIV`). See Note #1.
4. In extended assembly: evaluate the previous equation using integer MAC commands where applicable.

Task Set 2:

1. In basic inline assembly only: load (into registers) and add two integer numbers (hard-coded). Use extended assembly to pass the resultant value to a `C` variable and print it.
2. In extended assembly: multiply 2 single precision floats.
3. In extended assembly: evaluate the equation $\frac{2x}{3} + 5$ through floating point addition (`VADD`), multiplication (`VMUL`), and division (`VDIV`).
4. In extended assembly: evaluate the previous equation using floating point MAC commands where applicable.

NOTES:

1. Without special care, the integer implemented function will generally be inaccurate for small values. Why? This does not need to be fixed for this task.
2. The project template has been changed from previous templates to allow for the saving of the generated assembly code and program listings. You can find the generated assembly for a source file under *Debug/src/*[filename]*.asm*. The complete program listing can be found under *Debug/*[projectname]*.lst*.
3. If it is unclear how to perform an operation in assembly, it may be useful to write the command in `C` and check the generated assembly file.
4. Make sure to use the correct constraints for each operation, e.g.: `"r"` read from `C` variable to `r#` register, `"=r"` write to `C` variable from register, or `"+r"` read and write `C` variable to and write register.
5. If the written assembly is correct, take a look at the generated assembly or disassembly (within Debug: Window→Show View→Disassembly) . The compiler might be changing it if it's not specified as `volatile`, and possibly even if it is!

◇ Task 4: IIR Filter Implementation

Combine codes from Task 2 and Task 3 to produce a IIR (infinite impulse filter), acting upon a received signal from the ADC and outputting the filtered signal using the DAC. The filter to be implemented will have a response based on pole-zero diagram in Figure 2 and the transfer function:

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\frac{10}{32}\left(z + \frac{5}{13} + j\frac{12}{13}\right)\left(z + \frac{5}{13} - j\frac{12}{13}\right)}{z\left(z - \frac{19}{20}\cdot\frac{10}{32}\right)} \tag{4}$$

which leads to the difference equation:

$$y(k) = \frac{10}{32}\left(x(k) + \frac{10}{13}x(k-1) + x(k-2) + \frac{19}{20}y(k-1)\right) \tag{5}$$

where $x(k)$ is the input from the ADC at sample time k, $x(k-1)$ is the previous sample, etc. Similarly, $y(k)$ is the output to the DAC. In order to implement this filter, you will need to store the current and previous two ADC readings as well as the previous output. Note that the fractions in the equation above were left fractions in order to ease the implementation of the filter using integer math. These may be simplified into decimals, resulting in the equation:

$$y(k) = 0.312500x(k) + 0.240385x(k-1) + 0.312500x(k-2) + 0.296875y(k-1) \tag{6}$$

Implement this equation using in two ways: using a `C` style floating point equation and also using either floating point or integer MAC assembly instructions.

ADC conversion should be set up to put the results in the low 12-bits of the output register. The DAC values should be similarly formatted, with the DAC output buffer enabled. Additionally, the ADC should be set to run continuously such that the maximum speed of the filter may be achieved.

Each of the implemented filters should be characterized through the use of a signal generator connected to the ADC, generating a sine wave, and an oscilloscope monitoring both the input and output signals.

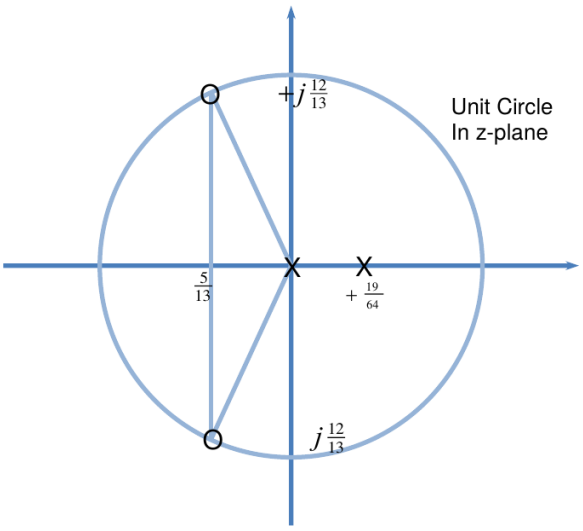

Figure 2: Pole-Zero diagram for desired IIR filter.

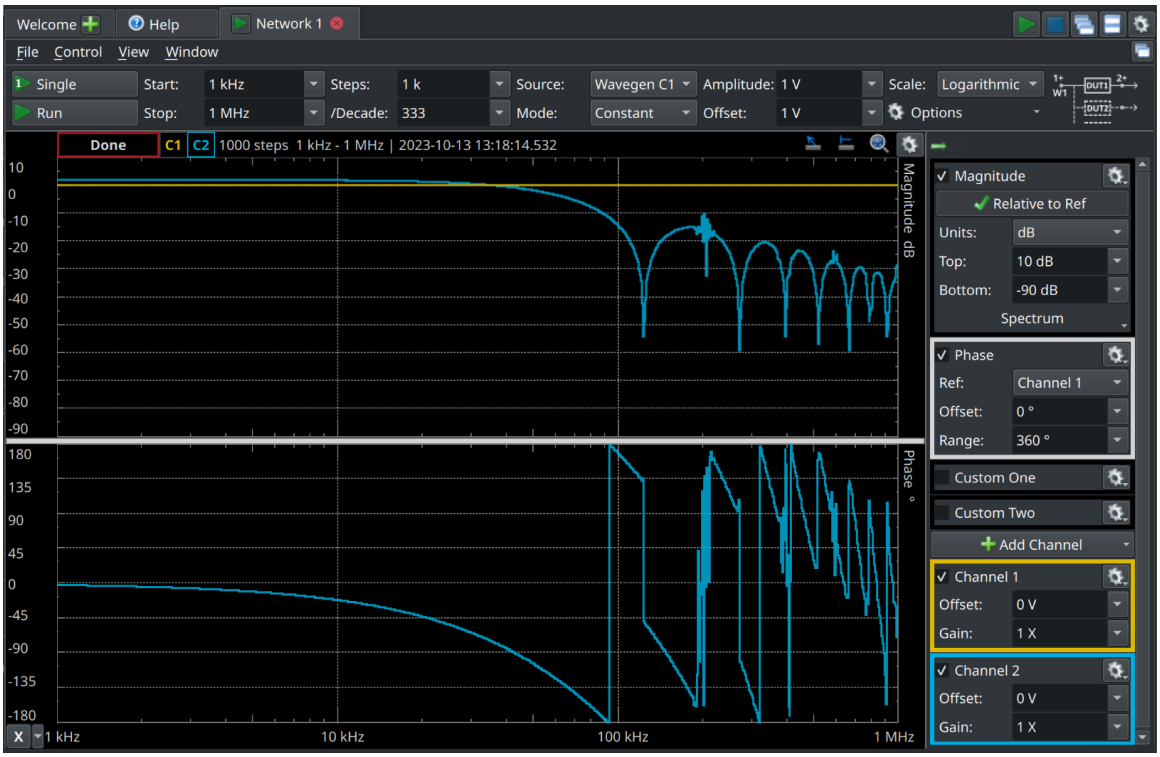

Figure 3: Measured response of implemented IIR filter. Frequency range is dependent on program speed.

Further, the sampling rate for each filter should be calculated and/or measured. Explain your results. . Ultimately, the filter response should look something like in Figure 3; however, the filter notch will likely materialize at a different frequency (it is shown at approximately 125 kHz in this example). The first zero in the filter's response is located at 112.61986° on the unit circle, corresponding to the notch in the filter at $\frac{112.61986°}{180°}\frac{f_s}{2}$, where f_s is the ADC and DAC sampling frequency. This implies that for Figure 3 f_s is approximately 400 kHz for the specific ADC configuration. It is not expected that groups reproduce the same sampling frequency and associated notch frequency as shown in Figure 3: a group's result should still have the same general shape but frequency shifted. The resulting sample rate and notch frequencies should be measured, verified, and reported, along with the methods to do so.

NOTES:

1. The floating point equation implementation is simple: simply evaluate the equation as-is.
2. If using integer MAC, be careful of both overflow and truncation of the evaluations. Scaling the constants and inputs may be required such that larger numbers are used to represent smaller numbers to prevent significant truncation. For example, add a scaling factor to the inputs and outputs $x(k) \rightarrow x(k) * 2^n$. Bit shifting in the assembly equations may be useful here (`LSL`, `LSR`).
3. As long as there is no floating point math being done in `C` (which there shouldn't be), it can be assumed that the contents of `s#` or `d#` will not change unless specifically commanded through the assembly commands. This may be used to decrease the filters instructional overhead.
4. The sampling frequency may be determined by toggling a GPIO pin each filter iteration. The GPIO should then have frequency half that of the filter sampling frequency; half because the output will

be low for one cycle and high for one cycle.

5. Figure 3 was produced with an Analog Discovery in *Network Analyzer* mode. The Analog ADALM2000 has similar functionality, though the high frequency measurement range will be much more noisy. If using a signal generator and oscilloscope, the notch frequency must be found by manually changing the signal generator frequency to locate the point at which the filter output is very small/zero.
6. The DAC output buffer should be enabled for this task. While the filter will work without the buffer enabled, the noise floor of the filter will be much higher without it enabled causing the sharp downward peaks in Figure 3 to be less pronounced among other effects.
7. The output of the filter near DC will have a >1 gain, which may lead to data overflows. Do not worry about fixing this other than by reducing the input amplitude; although explanation of the result is expected.
8. DO NOT use oscilloscope probes on anything other than oscilloscopes. They will not work as the connection between the signal generator and the ADC.
9. If the implemented filter is not working as expected, several intermediate steps may be taken:
   (a) Set all gains in the equation to 0 except the coefficient of $x(k)$, which should be set to 1. This should produce unity gain through your filter. If the output does not match the input, then something is possibly wrong with the gains associated with the inputs and outputs from/to the ADC/DAC.
   (b) Additionally set the gain of $x(k-1)$ to 1. This should result in a 1^{st} order, low pass filter, with a gain of 2 and an output of 2 when the input frequency is half the sampling frequency. The gains may need to be set to 0.5 to avoid low-frequency signal overflow.
   (c) The DC gain of the filter (with all coefficients) should be slightly higher than 1.
10. A completely successful implementation of the assembly filter *is not* required for checkoff, though a non-working one will result in point deduction. This will depend on how well the filter was implemented and what is missing.
11. Take extreme care when doing inline assembly within a function. Function arguments are passed using the core registers, usually starting at `r4`. If your assembly instructions use one of the argument registers, they will be overwritten.
12. If the written assembly is correct, take a look at the generated assembly or disassembly (within Debug: Window→Show View→Disassembly) . The compiler might be changing it if it's not specified as `volatile`, and possibly even if it is!

◇ Task 5: [Depth] Frequency Mixer

Implement the frequency mixer DSP block diagram as shown in Figure 4. Both inputs, sine wave signals with f_1 and f_2, must be measured using two ADC peripherals being sampled simultaneously (e.g., ADC1 and ADC2). The triggering for the ADCs **must be directly provided by a timer** and not through code. The rate should be relatively fast (e.g., >100 kHz). Once the ADC samples are acquired, they should be "Mixed," which is just multiplying them together. Filter the result using the band-pass IIR filter given in equations 7 and 8. This mixing and filtering should take place within an interrupt callback triggered by the completion of the ADC conversion. Finally, output the result of the filter through the DAC as in the previous task.

$$H(z) = \frac{Y(z)}{X(z)} = \frac{0.001 - 0.002z^{-2} + .001z^{-4}}{1 - 3.166z^{-1} + 4.418z^{-2} - 3.028z^{-3} + 0.915z^{-4}} \tag{7}$$

$$\begin{aligned} y(k) = {} & 0.001x(k) - 0.002x(k-2) + .001x(k-4) \\ & + 3.166y(k-1) - 4.418y(k-2) + 3.028y(k-3) - 0.915y(k-4) \end{aligned} \tag{8}$$

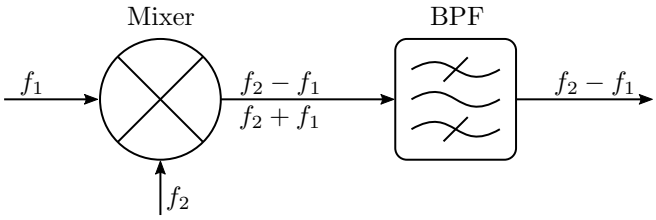

Figure 4: Mixer block diagram

NOTES:

1. The filter is a 4^{th} order Butterworth whose center is $f_s/10$ with a bandwidth of $f_s/100$.
2. Since the filter is done in an interrupt, polling of *both* the ADCs is not acceptable.
3. Multi-ADC mode is not required; though it would ensure that both ADCs are done converting per each interrupt. Alternatively, flags may be set/cleared in the ADC interrupts to track ADC completion. For example, if ADC2 completes first, a flag is set; then, ADC1's interrupt is triggered, this flag is read and the filter is executed.
4. Testing of this mixer will require two separate frequency generators. The target is to produce two signals that multiply such that $f_2 - f_1$ or $f_2 + f_1$ lies within the filter's pass frequency. Given that the acquisition rate and filter are known, this value should be straight forward to determine.
5. Initial testing of the mixing may be done by omitting the filter.
6. *It is not possible* to use a network analyzer (e.g., Figure 3) to verify the mixer operation as it is a non-linear operation (frequency in $\neq$ frequency out); however, a spectrum analyzer may be used.
7. You *can* test the filter using the network analyzer, however. To do so: remove the mixer prior to the filter and pass only one of the ADC inputs into the filter.

8. The DC biases of the input signals should be removed prior to multiplication in order to prevent additional frequency components. Add the DC bias back in for DAC transmit. The DC Biases may be assumed to be constant and hardcoded into the program. This may require that the DC biases be measured each setup to get an accurate value, however.
9. Similarly, care must be taken considering amplitudes of the output. As the mixer multiplies the input signal, very large values are expected out of the mixer if the math is done in ADC units; the employed data-types should account for this. It is suggested to correct this scaling after filtering is completed.
10. When testing and demonstrating the filter, avoid aliasing due to the sample rate.
11. There is no requirement for assembly to be used in this task.
12. When "mixing" two sin waves with frequencies f_1 and f_2, the result will be the superposition of two sin waves with frequencies $f_2 - f_1$ and $f_2 + f_1$. An example of this is given in Figure 5 where signal 1, $f_1 = 200$ kHz, is mixed with signal 2, $f_2 = 250$ kHz, to produce a third signal with frequencies $f_{3,a} = 50$ kHz and $f_{3,b} = 450$ kHz. The purpose of the the bandpass filter after the mixer would be to remove $f_{3,b}$, leaving just an output wave with frequency f_3, a.

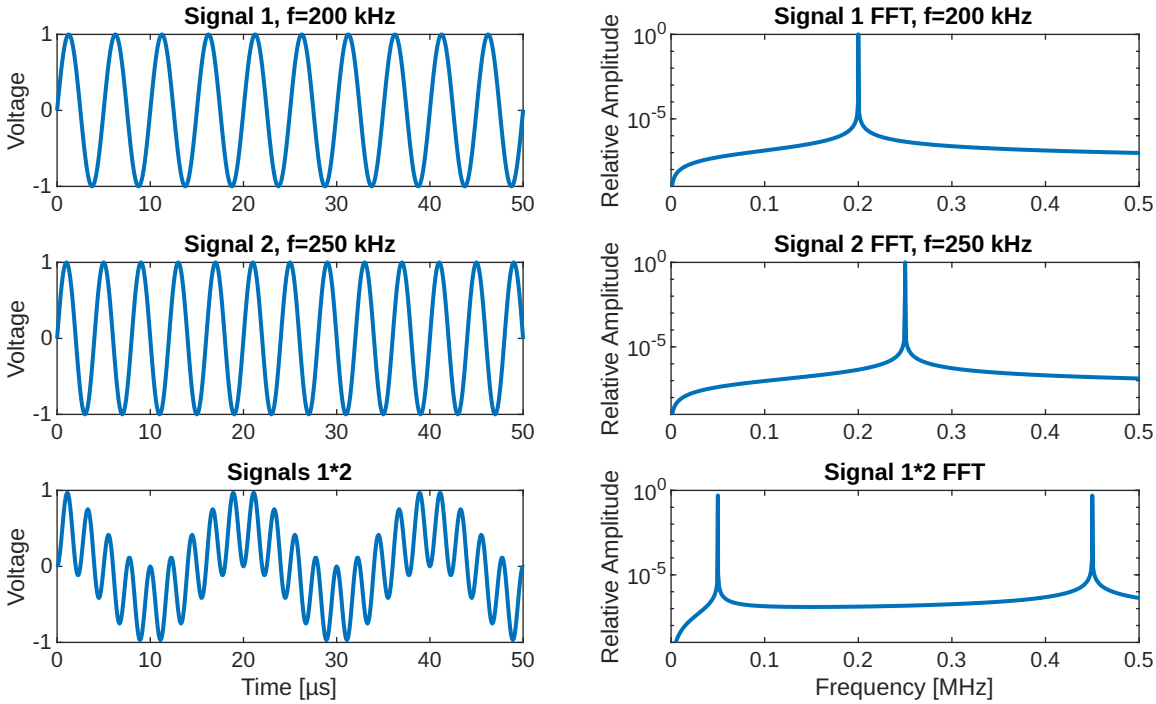

Figure 5: Frequency Mixing Example.