**MPS**

**Analog Conversion & Signal**

**Processing Lab Exercise**

**Analog Conversion and Low-Level DSP**

Student's name & ID (1): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Partner's name & ID (2): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Your Section number & TA's name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Notes:**

You must work on this assignment with your partner. Hand in a printed copy of your software listings for the team. Hand in a neat copy of your circuit schematics for the team.

These will be returned to you so that they may be used for reference.

------------------------------- do not write below this line -----------------------------

|  |  |  |  |
| --- | --- | --- | --- |
|  | POINTS (1) (2) | | TA init. |
| Grade for performance verification (50% max.) |  | |  |
| Part 1 (8% max.) |  | |  |
| Part 3 (7% max.) |  | |  |
| Part 4 (5% max.) |  | |  |
| Part 5 (30% max., 15% max. for non-MAC solution) |  | |  |
| Grade for answers to TA's questions (20% max.) |  |  |  |
| Enhancement (5% max.) |  | |  |
| Grade for documentation and appearance (25% max.) |  | |  |
|  |  |  | TOTAL |

Grader's signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Analog Conversion and Low-Level DSP**

**GOAL**

By doing this lab assignment, you will learn to use:

1. The Analog-to-Digital converter

2. The Digital-to-Analog converter

3. The Digital Signal Processor and Floating Point Unit

4. ARM Assembly instructions in C code

**PREPARATION**

• References: *769 Reference Manual (Register Map).pdf*

Ch. 15 (ADC), 16 (DAC; skip 16.4)

*STM32 ADC Modes.pdf*

All (ADC; it’s very short.)

*Mastering STM32*

Ch. 12 (ADC; skip 12.3), 13 (DAC)

*STM32 DSP Overview.pdf*

(Stop at Ch. 4)

*Cortex-M7 Programming Manual (CMSIS Peripheral Registers and CPU Model).pdf*

Ch. 3.6 (Reference for MAC instructions)

*769 Description of HAL Drivers.pdf*

Ch. 6 (ADC), 15 (DAC)

(Reference only; alternatively, “stm32f7xx\_hal\_adc.h/.c” and

“stm32f7xx\_hal\_dac.h/.c” contain more or less the same info in their

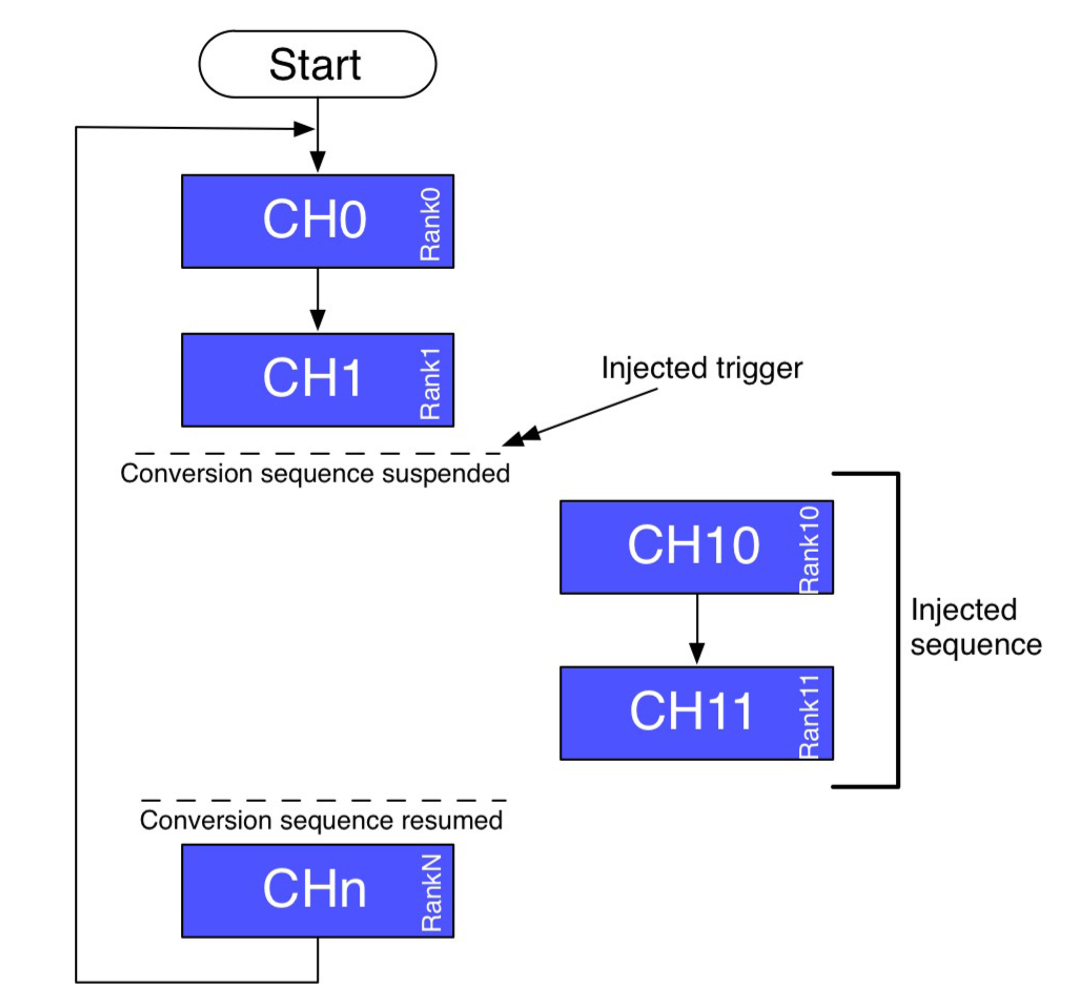
comments)

**ANALOG TO DIGITAL CONVERTER (ADC)**

**1. Introduction to the ADC**

The STM32F769NI has 3 built-in 12-bit successive approximation ADCs with up to 19 channels (16 external, 2 internal, and 1 on VBAT).[[1]](#footnote-1) 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[[2]](#footnote-2) sampling modes.

There are two kinds of groups of channels: regular groups and injected groups. A regular group is a set of channels that are set and used whenever a program calls them, while an injected group is like an interrupt: an external trigger causes the injected group to interrupt the conversion of a regular group, perform conversions on the injected group, and then resume the regular group when finished. Figure 1, taken from *Mastering STM32* Ch. 12.2.1.5, illustrates this process nicely:



**Figure 1: From Mastering STM32, Ch. 12.2.1.5 Injected Conversion Mode**

From Ch. 15.3.4 in *769 Reference Manual (Register Map).pdf*:

“There are 16 multiplexed channels. It is possible to organize the conversions in two groups: regular and injected. A group consists of a sequence of conversions that can be done on any channel and in any order. For instance, it is possible to implement the conversion sequence in the following order: ADC\_IN3, ADC\_IN8, ADC\_IN2, ADC\_IN2, ADC\_IN0, ADC\_IN2, ADC\_IN2, ADC\_IN15.

* A regular group is composed of up to 16 conversions. The regular channels and their order in the conversion sequence must be selected in the ADC\_SQRx registers. The total number of conversions in the regular group must be written in the L[3:0] bits in the ADC\_SQR1 register.
* An injected group is composed of up to 4 conversions. The injected channels and their order in the conversion sequence must be selected in the ADC\_JSQR register. The total number of conversions in the injected group must be written in the L[1:0] bits in the ADC\_JSQR register.

If the ADC\_SQRx or ADC\_JSQR registers are modified during a conversion, the current conversion is reset and a new start pulse is sent to the ADC to convert the newly chosen group.”

The actual conversion process occurs according to the following formula:

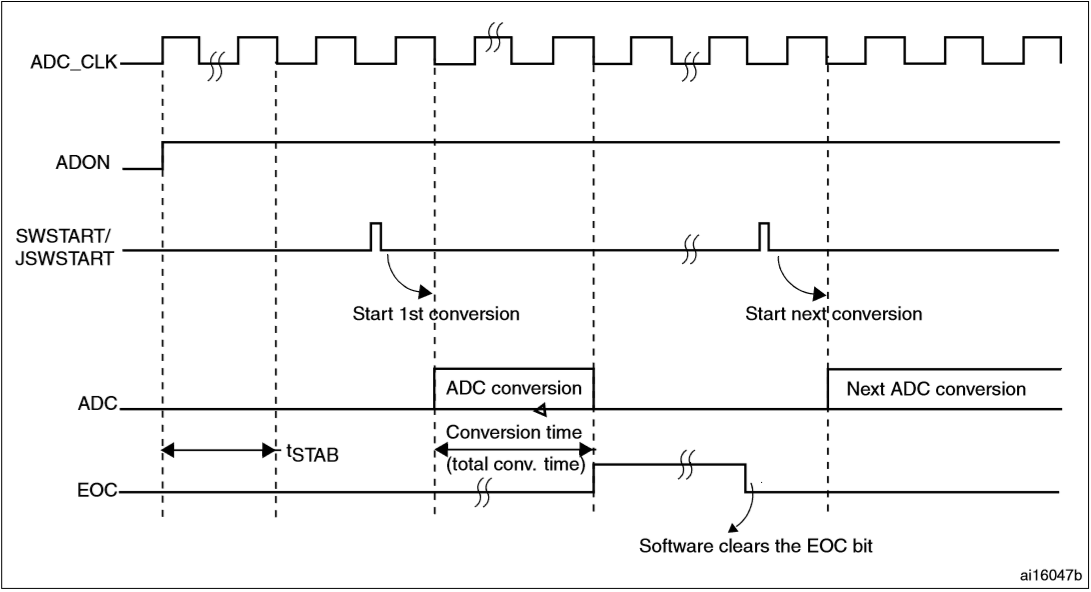
The *output value* is an unsigned 16-bit integer in the ADC\_DR register, *VREF* is internally connected to 3.3V, and *MaxNumber* is the maximum value allowed by the ADC bit setting, 2Resolution – 1.

A neat feature of the 769 chip is that it can have an external *VREF* applied, but the DISCO board was not designed to allow this. Thus, we are stuck with *VREF* at 3.3V, so interfacing with lower-voltage sensors would require taking this into account.

**2. Configuring the ADC**

1. Turn on the ADC by setting ADON in the ADC\_CR2 register (clear the bit to turn it off)
2. Configure the ADC channels as desired (resolution, sampling time, etc.)
   1. Ch. 4 in *769 Description of HAL Drivers.pdf*
      1. Ch. 4.1.4 for the ADC\_HandleTypeDef structure
      2. Ch. 4.1.1 for the ADC\_InitTypeDef structure
      3. Ch. 4.1.2 for the ADC\_ChannelConfTypeDef structure (for individual channels)
      4. Ch. 4.2 for configuration tips
3. Enable the ADC peripheral clock (APB2)
4. Enable the ADC clock, and wait for the stabilization time of the ADC before the first conversion
   1. The stabilization time, tSTAB, and similar parameters are provided in Ch. 5.3.4 of *769 Datasheet (Alternate Functions etc).pdf*.
5. Start conversion by setting either SWSTART (regular) or JSWSTART (injected)
6. End of conversion is signaled by an interrupt (or monitor EOC/JEOC in ADC\_SR)
7. Converted data can then be read from ADC\_DR

Here is a timing diagram of the general process in *769 Reference Manual (Register Map).pdf*, Ch. 15.3.7:



**Figure 2: ADC timing diagram from 769 Reference Manual (Register Map).pdf, Ch. 15.3.7**

Some points to consider:

* The ADC has multiple clock schemes. The clock for the digital interface, which is used for register read/write, is derived from the APB2 system clock (which has been set to run at 108MHz in “init.c”). The clock for the analog circuitry is configurable and can be APB2 or a slower derivative such as APB2/2, /4, /6 or /8. Note the max ADC clock is 36MHz, so plan accordingly.
* External ADC connections (A0-A5 on the Arduino connector) are ADC1\_IN4, ADC1\_IN6, ADC1\_IN12, ADC3\_IN6, ADC3\_IN7, and ADC3\_IN8. These inputs are all 3.3V max. See the Arduino connector table at the end of this document for port pinout and connector assignment.
* Internal ADC connections on the STM32:
  + The internal temperature sensor & VBAT are both on ADC1\_IN18, and only one can be converted at a time. If both are set, VBAT has priority.
  + Ch. 15.10 in *769 Reference Manual (Register Map).pdf* has an excellent run-down of how to use the temperature sensor, as well as how to obtain Celsius temperature from it
    - Avg\_Slope and V25 are in *769 Datasheet (Alternate Functions etc).pdf*, Ch. 5.3.25
  + VREFINT is on ADC1\_IN17, and is defined in Ch. 5.3.27 of the datasheet
* Ch. 5.3.24 of *769 Reference Manual (Register Map).pdf* contains the electrical specifications of the ADC, as well as accuracy values (measurement error), frequency information, and other very useful information.
* Interrupt timers can also be configured to trigger at both the start and end of conversions.
* Watchdog timers can be set to send interrupts if analog values are above or below defined thresholds.

**3. Polling**

The simplest modes the 769 supports are single conversion mode and continuous polling mode. Single conversion mode does one conversion and then stops, while continuous mode keeps running conversions over and over again (it starts the next conversion right after it finishes the previous one, and only applies to regular groups). Scan, discontinuous, and the very nifty dual/triple ADC modes will not be covered here, but you may use them as enhancements if you’d like.

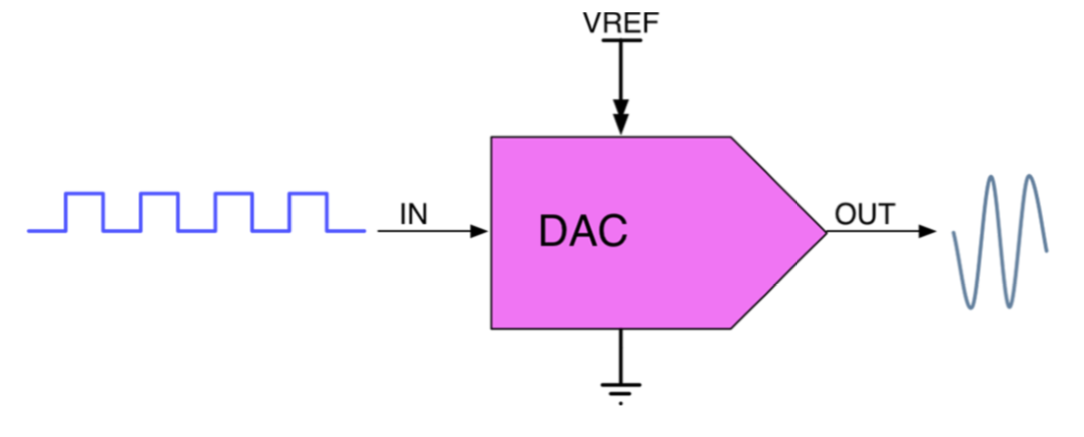
Reading from the ADC data registers differs slightly between regular and injected groups, where injected groups are allowed to be negative (signed) based on a subtraction offset set in ADC\_JOFRx (note: x corresponds to one of the 4 possible injected data channels), while regular group data are unsigned. Regular group conversion data are stored in ADC\_DR, while injected group data are stored in ADC\_JDRx (same x as before).

ADC\_DR will only contain the last converted regular group channel data, and gets overwritten every regular group conversion. The EOC flag is set in ADC\_SR upon every completion, and is cleared by reading the ADC\_DR register (or by user software, if no read was performed). If enabled (EOCIE), an interrupt is generated upon completion, as well.

Note: In *769 Reference Manual (Register Map).pdf* Ch. 15.13.1, in the bits for ADC\_SR, the description of the EOC bit has an issue: “(EOCS=0)” should be “(EOCS=1)” and vice versa.

**DIGITAL TO ANALOG CONVERTER (DAC)**

**1. Introduction to the DACs**



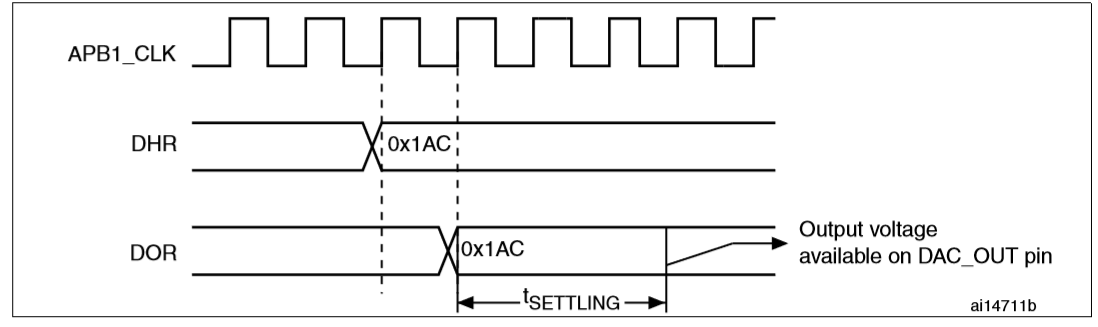
**Figure 3: The general structure of a DAC, from Mastering STM32 Ch. 13.1**

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’s channels have their own converter, so simultaneous conversions are possible via dual mode. 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.[[3]](#footnote-3),[[4]](#footnote-4)

**2. Configuring the DAC**

Similar to the ADC, the DAC must be enabled and configured properly before use:

1. Turn on the DAC’s only usable output channel by setting EN1 on DAC\_CR
2. Configure the DAC
   1. Enable the output buffer (BOFF1 in DAC\_CR) to negate the need for an external op-amp
   2. See Ch. 13 in *769 Description of HAL Drivers.pdf* for the usual HAL handler and init structure information
3. Enable the DAC peripheral clock (APB1) and wait for tWAKEUP to elapse before use
4. Write data to be output to DAC\_DHRyyy1 (DAC\_DHR12R1, DAC\_DHR12R2, or DAC\_DHR8R1, depending on how you want to format the data)
   1. Refer to Ch. 16.3.3, *769 Reference Manual (Register Map).pdf*
5. Data written to one of the DHRs (data holding registers) will be output on DAC\_DOR1 on the next APB1 cycle, after a time tSETTLING (see figure below)
   1. We have APB1 running at 54MHz



**Figure 4: Timing diagram for DAC DHR to DOR to output, from 769 Reference Manual (Register Map).pdf, Ch. 16.3.4**

**3. Performing a Conversion**

Assuming all has been configured properly, simply writing to the data holding register will change the output on the next APB1 clock cycle. Like the ADC, DAC tWAKEUP and tSETTLING can be found in Ch. 5.3.28 of *769 Datasheet (Alternate Functions etc).pdf*.

The DAC conversion is *almost* precisely the opposite of the ADC’s in 12-bit mode:

*Output value* is an unsigned 16-bit integer in the DAC’s Data Output Register (DAC\_DOR), *VREF* is internally connected to 3.3V, and *MaxNumber* is the maximum value allowed by the bit setting, i.e. 2Resolution (this differs by 1 from the ADC). Note that the maximum output value that can be used is actually 2Resolution – 1 (see Ch 13.1 in *Mastering STM32* for an explanation).

**DIGITAL SIGNAL PROCESSOR (DSP) *{Do programming tasks 1-3 first!}***

**1. Introduction to the DSP**

Devices conforming to the ARM CMSIS specification include dedicated hardware for mathematical operations, and this hardware collectively forms the DSP subsystem of the Cortex ARM core. There are 2 ways to access the Cortex-M7 DSP functionality on the DISCO board: one is via CMSIS functions provided by the “arm\_math.h” library, which provides a wide variety of math functions, and the other is via ARM assembly instructions.

The latter is the only way to run single instructions on input data with guaranteed complete control over what happens and where it goes, and it’s not particularly hard to implement with GCC, so when implementing specific hardware math functions later in this lab exercise we will be making use of these assembly instructions. One can think of the DSP library as a collection of prebuilt use cases for the ARM instructions, so going the assembly route is like getting a “behind the scenes” look at how digital signal processors are implemented. Assembly is also faster; simple instructions need only *one* clock cycle each.

The DSP subsystem can also use the Cortex-M7 floating point unit, which you may optionally read about in *STM32 Using the Floating Point Unit (FPU).pdf* if you wish [*Recommended*]. The primary takeaway is that FPU usage is automatic based on compiler directives, specifically “-mfloat-abi=hard -mfpu=fpv5-d16,” which we do have enabled. These directives inform the compiler that there is a hardware FPU compatible with the FPv5-D16 standard[[5]](#footnote-5) and that it ought to use FPU instructions where possible.

The M7’s FPU can handle single (32-bit), double (64-bit), and half (16-bit) precision float formats.

**2. DSP Features**

The DSPs in the wider Cortex-M family utilize 3 kinds of instructions: Saturation, MAC (Multiply ACcumulate), and SIMD (Single Input Multiple Data). You can read about these in *STM32 DSP Overview.pdf* Ch. 2. At the very least, you will be required to use MAC instructions in programming task part 4, but you may also use SIMD and saturation instructions as enhancements.

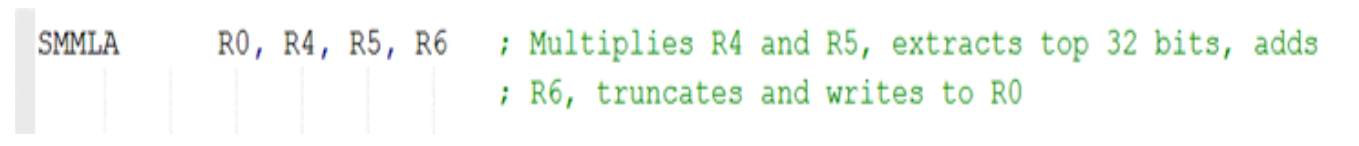
To that effect, the overview of MAC instructions from the DSP Overview document is included below:

“**2.2 MAC instructions**

Multiply ACcumulate (MAC) instructions are widely used in DSP algorithms, as in the case of the Finite Impulse Response (FIR) and Infinite Impulse Response (IIR).

Executing multiplication and accumulation in single cycle instruction is a key requirement for achieving high performance.

The following example explains how the SMMLA (Signed Most significant word MuLtiply Accumulate) instruction works.



[Note: R0, R4, etc. are 32-bit general purpose processor core registers. Registers R0-R12 are free to use. See Ch 2.1.3 in *Cortex-M7 Programming Manual (CMSIS Peripheral Registers and CPU Model).pdf.*]”

Ch. 3.6 in *Cortex-M7 Programming Manual (CMSIS Peripheral Registers and CPU Model).pdf* contains a very useful reference for the several other multiply/divide/multiply + accumulate instructions. The most basic MAC instruction is MLA (Multiply with Accumulate, 32-bit result), which behaves as so:

MLA Rd, Rn, Rm, Ra ; Multiply with accumulate

“The MLA instruction multiplies the values from Rn and Rm, adds the value from Ra, and places the least significant 32 bits of the result in Rd.” (Ch. 3.6.1, CMSIS PDF)

Example invocation:

MLA R10, R2, R1, R5; multiply with accumulate, R10 = (R2 x R1) + R5

On the Cortex-M7, the registers can contain floats or integers for these operations, and running the operation only takes one cycle (so you can read the result on the next cycle).

Below is an example of how to use ARM MAC assembly instructions in C code:

asm("MLA %[out], %[x], %[y], %[z]" : [out] "=r" (t) : [x] "r" (q), [y] "r" (r), [z] "r" (s));

While this may look daunting, not to worry, it’s actually pretty simple—but it’s very, very powerful.

**3a. Inline Basic Assembly in C with GCC 5.4.x**

*(The following information has been adapted from here:* [https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/Using-Assembly-Language-with-C.html](https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/Using-Assembly-Language-with-C.html#Using-Assembly-Language-with-C) and *Cortex-M7 Programming Manual (CMSIS Peripheral Registers and CPU Model).pdf*)

asm [ volatile ] ( *AssemblerInstructions* )

The C function “asm()” informs the compiler that the contents within () is assembly for the target platform, where *AssemblerInstructions* is a string with the actual assembly code. Let’s use something simpler as an example:

asm("LDR r3, =0xABCDEFGH\n\t"

"LDR r2, =0x12345678\n\t"

"LDR r4, =0xFFFF0000"

);

This is 3 separate assembly statements (\n\t, or newline + tab, means “end of this statement”), and in each statement the instruction LDR (load register) loads a general purpose register, r3, r2, or r4, with a hex value. The syntax “=0xAAAAAAAA” is a generalization of [rX, #y], where [rX, #y] means read the value found at the address of register rX offset #y (offset in number of bytes). In other words:

asm("LDR r3, =0xABCDEFGH") == asm("LDR r3, [register whose base address we need, #offset, in bytes, relative to base register address where the data, 0xABCDEFGH, is -- default #0]");

The brackets mean “contents at memory address specified by register base + offset.” Recall the registers being referred to are ARM core registers like r0, r1, and r12, not peripheral registers like ADC\_SR. One of the components of the compiling process is to intelligently shuffle data throughout core registers, so in reality all program code will be converted into assembly form. This means that manually inserting inline assembly forces the compiler to work around the user-input code block, since it also needs to use those registers when the user does not need to explicitly set them. Rule of thumb for inline assembly is *do you what you need to do, then get out of the compiler’s way*.

At the same time, the reason we use the “pseudo-instruction” syntax asm("LDR r3, =0xABCDEFGH") is because we don’t expressly know where the register base and offset are—but the compiler and assembler do when they turn that code into well-defined register addresses. Using the pseudo-instruction informs GCC that, wherever it decides to put 0xABCDEFGH, grab it from that place and load it into r3.

That’s called “Basic Assembly,” and it is the simplest usage of inline assembly. You can read more about it here (it’s a very short read): <https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/Basic-Asm.html>, though note that the “volatile” qualifier doesn’t actually do anything because, in basic inline assembly, asm() is always volatile.

**3b. Extended Assembly**

Basic Assembly is great for single-cycle data reads/writes, as well as for executing special instructions like MAC instructions, but only when working entirely within the confines of pure assembly. In order for us to integrate the ARM instructions with our C code, which would allow us to call the low-level instructions we want specifically, we need to use “Extended Assembly.” The syntax is built similarly to Basic Assembly, but with extra arguments in the asm() function:

asm [volatile] ( *AssemblerTemplate*

: *OutputOperands*

[ : *InputOperands*

[ : *Clobbers* ] ])

Here, *AssemblerTemplate* is the assembly instruction like before, but this time it accepts “operands.” Operands are just like printf() escape characters %d, %s, %c, etc., except they are stricter with allowable values. Output operands are like variables that are for writing TO, and input operands are like variables that are for reading FROM. “Clobbers” is a list of registers that are modified by the desired instruction, not including the output operands (they obviously change). Also, in the above template, brackets mean optional parameters.

A simple Extended Asm case looks like this:

uint32\_t d;

asm("STR r9, %[some2]"

: [some2] "=m" (d)

);

This just means “store the contents of r9 into the memory address denoted by ‘some2.’” The % has the same meaning as the printf() %, i.e. the parameter is defined outside of the string, and in this case the output operand is whatever is at the [some2] memory location. In the above example, the C unsigned 32-bit integer “*d*” is what the [some2] location refers to. The “=m” means that *d* is at an arbitrary valid memory address (the *m* part) that can be written to (the = part). Syntax requires the C variable must always be in parenthesis. Ultimately, this is how we write (C variable) *d* = (assembly register) r9.

As a quick aside, we could use MOV instead of STR, though STR is meant to handle 32-bit data (the core registers are all 32-bit). MOV is limited to 8/16-bit and requires a second instruction, MOVT, to copy the upper half-word. MOV also has the source and destination flipped compared to STR. However, if certain optimization levels are enabled (e.g. via the GCC compiler -O flag) and the compiler finds that an STR or LDR instruction could done faster with a MOV and/or MOVT, it will make an appropriate substitution. Not only that, but also, depending on the optimization level, compilers like GCC will only make the substitution if they are 100% certain that a MOV instruction will always work at that point in the program. For example, if a program is trying to load a constant 12-bit value into a core register using LDR, the compiler will just turn it into a MOV and save a cycle or two. Pretty cool! (For the curious, we use -Og, which means “optimize for debugging.”)

Now we can finally look at the more complex example of an Extended Asm statement from 3a:

asm("MLA %[out], %[x], %[y], %[z]" : [out] "=r" (t) : [x] "r" (q), [y] "r" (r), [z] "r" (s));

Rewriting it vertically makes it much easier to see the different parts:

uint32\_t q, r, s, t; // q, r, and s do need to be given values

asm("MLA %[out], %[xerneas], %[yveltal], %[zygarde]"

: [out] "=r" (t)

: [xerneas] "r" (q), [yveltal] "r" (r), [zygarde] "r" (s)

);

We have the MAC instruction, MLA, taking in the C variables *q*, *r*, and *s* and outputting to the C variable *t*, so we get *t* = *q \* r + s* computed in hardware exactly the way we want it. We did not need to specify any clobbers, so that parameter has been omitted.

The last aspect to go over is the “constraint.” That’s the “=m” from earlier or the “r” and “=r” from above. These specify what kind of data type the passed C expressions need to be treated as such that they match what the assembly instruction expects. For instance, MLA works on registers, so we need to tell the compiler to treat the C expressions as assembly registers when used in this assembly statement. LDR and STR can read from and write to memory regions, so we use “m.” All output operands need “=” in front of constraints by convention, as it means “writable.”

Constraints are a really important concept to grasp when writing inline assembly. Without them, your assembly code won’t understand that there is a difference between things like general registers, single-precision floating-point registers, and double-precision floating-point registers! (For more details on the floating-point registers, see the first page of Ch. 3 in *STM32 Using the Floating Point Unit (FPU).pdf*.)

A useful reference with a condensed set of constraints is here:

<http://www.ethernut.de/en/documents/arm-inline-asm.html>

(You might consider saving this page as a PDF via “Print to PDF.” It’s really good!)

A bigger ARM constraints list, as well as useful operand modifiers:

<http://hardwarebug.org/2010/07/06/arm-inline-asm-secrets/>

Here are a few more examples of syntax:

uint32\_t e;

asm("mov %[some], $4" // $ denotes a constant

: [some] "=r" (e) // MOV is actually a copy, not a move

);

uint32\_t q, r; // q and r do need to be given values

asm("LDR r5, %[some1]\n\t"

"LDR r6, %[some2]"

: // No outputs

: [some1] "m" (q), [some2] "m" (r) // Input variables for multiple statements

:); // No clobbers

A complete list of constraints and modifiers is included at the end of this document. These have proven invaluable for resolving weird inline ASM errors. For example, you need to explicitly specify doubles with operand modifier “P” when using them in operands, i.e. by using %P[mem] instead of %[mem]. Otherwise, GCC inline assembly will try to use single precision registers with a double precision instruction like VMUL.F64 – even if correctly specifying the “w” constraint. Not using the “P” modifier in this situation will also cause the assembler to throw an error for what might otherwise look like valid code – specifically it will report “Error: selected FPU does not support instruction…”

For further reading on Extended Assembly, see the official GCC documentation on the subject:

<https://gcc.gnu.org/onlinedocs/gcc-5.4.0/gcc/Extended-Asm.html>

*My ASM code is right, but my program isn’t doing anything (or is doing something weird)! What gives??*

Sometimes GCC will decide that an assembly register you absolutely need to remain constant is not important and will overwrite it later – or even straight up delete it from the compiled code altogether – if it isn't specified as an explicitly constant register value. For example, normally you would do something like this to do a single-cycle floating-point add and output it to the variable *dacfloat* for use in C code:

**float** dacfloat;

**float** floatone = 12.3;

**float** floattwo = 2.5;

**asm** **volatile** ("VADD.F32 %[dac], %[a], %[b]"

: [dac] "=t" (dacfloat)

: [a] "t" (floatone), [b] "t" (floattwo)

);

But sometimes that doesn’t work, indicating a more explicit approach is needed. Explicitly defining a register as immutable to GCC’s optimization algorithms looks like this:

// Define the memory address of single-precision float register s15 as a bona fide register.

// This basically tells GCC that, when we use the variable “dacfloat,” we mean register s15.

register float \*dacfloat asm("s15");

float floatone = 12.3;

float floattwo = 2.5;

// We need to explicitly specify “volatile” to tell GCC to leave this asm statement alone

// and not delete it behind our backs. The “volatile” term is only necessary if there are

// output operands present, otherwise it is implied by ‘asm.’

asm volatile ("VADD.F32 %[dac], %[a], %[b]"

: [dac] "=&t" (dacfloat)

: [a] "t" (floatone), [b] "t" (floattwo)

);

// Lastly, the “&” in “=&t” tells GCC not to reuse or modify (in an undesired way) the output

// register in question during this instruction. Sometimes GCC will overwrite the output

// register when it loads/moves one of the inputs from memory into a usable register, and

// you’ll be left wondering why your code is outputting 0 (or weird values) when it looks

// and compiles fine. The “&” prevents this behavior, so the above floating-point add should

// output 14.8 into s15.

It’s possible that GCC will ignore the “register” declaration and stick a different register in the output of VADD (despite dacfloat still referring to s15). When that happens, we need to declare “dacfloat” as “volatile,” which means “DON’T SCREW WITH THIS, GCC! I MEAN IT!” (if you’re angry) or “this variable might change for reasons unrelated to nearby C code, so optimizing it risks causing unexpected behavior” (if you’re not).

volatile float dacfloat;

float floatone = 12.3;

float floattwo = 2.5;

asm volatile ("VADD.F32 %[dac], %[a], %[b]"

: [dac] "=&t" (dacfloat)

: [a] "t" (floatone), [b] "t" (floattwo)

);

Another solution might be to condense multiple ASM statements into one, like in this earlier example:

uint32\_t q, r; // q and r do need to be given values

asm("LDR r5, %[some1]\n\t"

"LDR r6, %[some2]"

: // No outputs

: [some1] "m" (q), [some2] "m" (r) // Input variables for multiple statements

:); // No clobbers

**PROGRAMMING TASKS**

**PART I – Simple Voltmeter**

For this lab you will write your program for use with a circuit on a prototyping board. This builds on what you have learned so far.

This lab uses a protoboard. You will be using the ADC in 12-bit mode, and you will need configure the blue push button to pull its pin (PA0) high when pushed.[[6]](#footnote-6) The TAs will caution you about the power supply and its connections. NEVER WORK ON CIRCUIT WIRING WITH THE POWER ON!

**THE PROTOBOARD:**

To obtain an analog voltage to be used as the input to converter, use a potentiometer (10kΩ). Connect the potentiometer between +3.3 V and GND (these can be found on the DISCO board itself). Connect the wiper to Arduino pin A0. Only when you are sure you have no shorts and that you did NOT accidentally plug into the 5V connector should you turn on the power supply.

If +3.3V is unavailable on the board for any reason, there are DC power supplies at most of the lab stations that will output an appropriate voltage. On those supplies, dialing in 3.00 V will provide a safe voltage level that won’t damage the converter, but you must make absolutely certain that the 10 - 20V button to the left of the dial and display is NOT pressed in.

The 10kΩ potentiometer is one of 4 in the parts kit, mounted on a circuit board. Refer to the end of this document for the circuit schematic and pinouts.



Write an analog-to-digital conversion program to operate the EVB as a digital voltmeter. Display the hexadecimal raw result and the corresponding decimal voltage value on the terminal display.

Some of what you will need to do:

* Write a main program that calls the necessary routines to implement your program design.
* Write a routine to configure the ADC to convert the single analog voltage on ARD\_A0 using single conversion mode. Conversion should start when PA0 is pulled low. The program should poll this pin waiting for the signal.
* Write a routine to poll ADC1 until a conversion is complete.
  + There is more than one way to do this.
* Write a routine to display the low, high, and average result in hexadecimal, and display the corresponding voltages in decimal. Display the value of the decimal voltage to 6 decimal places and update the display whenever PA0 is pulled high. In doing this, you will need to keep track of the low and high values and maintain a running average (e.g. the average of last 16 or so samples).
* Check your results with a voltmeter and hand calculate a few voltage readings to verify that your program is working correctly. Think about the maximum voltage this setup can convert correctly relative to the 3.3V or 3.0V potentiometer supply.

# PART II – Advanced Voltmeter and Frequency Counter {This is OPTIONAL}

**THE PROTOBOARD:**

To obtain an analog voltage to be used as the input to converter, use a potentiometer (10kΩ) as in Part I. Again, connect the potentiometer between +3.3V and GND. Connect the wiper to ARD\_A0, and, as before, only when you are sure you have no shorts should you consider turning on the power supply.

This program will use some of the alternative features of the A/D converter as well as processor arithmetic operations to calculate some values for an input waveform. Unfortunately, the STM32F7 line does not feature differential inputs for its chips’ ADCs, so we are limited to waveforms with positive voltage values. That means that we cannot use something like a true differential floating voltage signal, which would look similar to the figures below with external batteries for power:



Note: the 2 connection points for these sources are true floating differential signals and when connected to differential analog inputs would normally allow conversion over the full range -3.0V to +3.0V.

However, it is possible to use injected groups to produce negative values based on offsets, so the STM32F769I-Discovery board can be configured to use the range 0V to 3.3V, *effectively* -1.65V to 1.65V, or even some combination of either two settings. You could even use both settings on the same channel, although in that situation -1.65V as measured by an injected channel would equal 0V as measured by a regular channel (no matter what we are constrained to the same 3.3V total voltage range).

The part 2 program should continuously sample a waveform input on ARD\_A0 and display the minimum, maximum, and a running weighted average based on (or weighting the current input with and the previous average with ). Make sure you do these calculations using at least 16-bit numbers. The display should be updated approximately every second. A separate reset pushbutton should reset the stored minimum and maximum readings.



Optional Feature:

Assuming the maximum frequency of the input waveforms are much below half the sampling frequency of the processor system, calculate the frequency of the waveform by measuring the time between repeat patterns on the signal (i.e. peak to peak) for sine and triangle waves. This can be done by counting the number of samples in one period and multiplying the number by the sample period or using timers. Test your program on frequencies in the range of ~0.3 Hz to 30 kHz and document its useable range. Better accuracy on lower frequency signals may be achieved by adjusting the timer and A/D converter clock frequencies and redoing the measurements. Display the frequency in Hz on the console, but do not update it more frequently than about once a second. Use an oscilloscope to verify measurements of each parameter. Have the program calculate the A.C. RMS value of the waveform. This is where is the running weighted average. The mean should be calculated using some form of integration over one period.



# PART III – DAC Output

Write a program to convert digital values to analog signals and output them to the DAC\_OUT1 pin. 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 3.3V sawtooth wave on the DAC output. Confirm this with an oscilloscope. Notice the output frequency is 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.

If you use the input from a program written for part 2, and in that program opted for using the injected channel range -1.65V to 1.65V, you will need to figure out how to deal with a bipolar input passed to a unipolar output. Probably the easiest way is to scale the signal so that the peak-to-peak voltage will fit in the 0 – 3.3V range of the output DAC, which is for this chip would involve simply adding an offset equivalent to half the range (1.65V). Now the output will have the same shape as the input.

*Consider this:* Given the current STM32 setup (12-bit ADC/DAC, 27MHz ADC clock), what’s the theoretical max frequency of waveform that our system can pass through at full resolution? *Hint:* Ch. 5.3.4 of *769 Datasheet (Alternate Functions etc).pdf*. How might you be able to measure this experimentally to verify your number?

***Once finished, go back to page 8 and read the DSP section before continuing to Part 4.***

**PART IV – Assembly**

Before moving on to Part V, write a short program that does three separate, simple things:

1. In basic inline assembly only**,** loads and adds two numbers
   1. You’ll need to use extended assembly to get the result to print it, though
2. In extended assembly, multiplies 2 floats or doubles and prints them
3. In extended assembly, evaluates via float instructions and via MAC instructions
   1. For the MAC instruction equation, you can use a shift to divide

In order to enable printing of floats, you’ll need this line at the top of your .c file under your #includes:

asm (".global \_printf\_float");

This informs the compiler to enable printf() to use floats, which is disabled by default because we need to compile with the “Newlib-Nano” library for space savings. Printf float support adds several KB to programs, which is a lot for our uses. Alternatively, adding “-u \_printf\_float” to the linker’s command line does the same thing, but using the assembly code allows us to enable the functionality only when we need it, saving space. If you need scanf() for floats, use the same statement and replace “\_printf\_float” with “\_scanf\_float” (and you can use both the printf and scanf asm() statements in the same program).

Remember, *Cortex-M7 Programming Manual (CMSIS Peripheral Registers and CPU Model).pdf* is a reference for all the available Cortex-M7 instructions (MAC instructions are just in Ch. 3.6).

# PART V – FIR Filter

This program will build on the previous parts, and filter the data coming in before outputting it through the DAC. Create a simple IIR (infinite impulse response) filter using MAC instructions. The filter to be implemented will have a response based on the transfer function (and pole-zero diagram) below:



Here, *x*(*k*)is the input from ADC1 at sample time *k*, *x*(*k-1*) is the value of the previous sample etc., and *y*(*k*) is the value being sent to the DAC. To implement this filter, you will need to store the current and previous two ADC readings as well as the previous output, and use MAC instructions to multiply *x*(*k-1*) by and *y*(*k-1*) by as well as the overall scaling of , necessary to limit the D.C. gain to a reasonable value. Note that in the pole-zero plot.









O

O

X

X

Unit Circle

In z-plane



There are 3 ways of solving the difference equation: With floats, large integers, or binary fractions. You will be required to implement the float method + your choice of one of the other two.

ADC conversion should be set up to put the results in the low 12 bits of the output register, and DAC output should be formatted as right-aligned, as well.

The sequence followed to evaluate the difference equation with raw instructions is:

1. Start a new conversion
2. Update previous output & previous 2 inputs for the cycle
3. Get the new sample
4. Add *x*(*k*) and *x*(*k -* 2) in assembly and store in *output* *variable*
5. Multiply-accumulate *output variable* + 10/13 \* *x*(*k -* 1)
6. Multiply-accumulate *output variable* + 19/20 \* *y*(*k -* 1)
7. Multiply *output variable* by 10/32
8. Convert *output variable* into unsigned 32-bit integer for the DAC
9. Write the *output variable* to the DAC

If you configured the DAC and ADC to use different resolutions, you might need to do some scaling. **It is also worth noting that printf() might have trouble printing floats that have just been operated on in assembly, so using an integer to get the integer part of the float for printing may be necessary**.

Some quick notes about each implementation method:

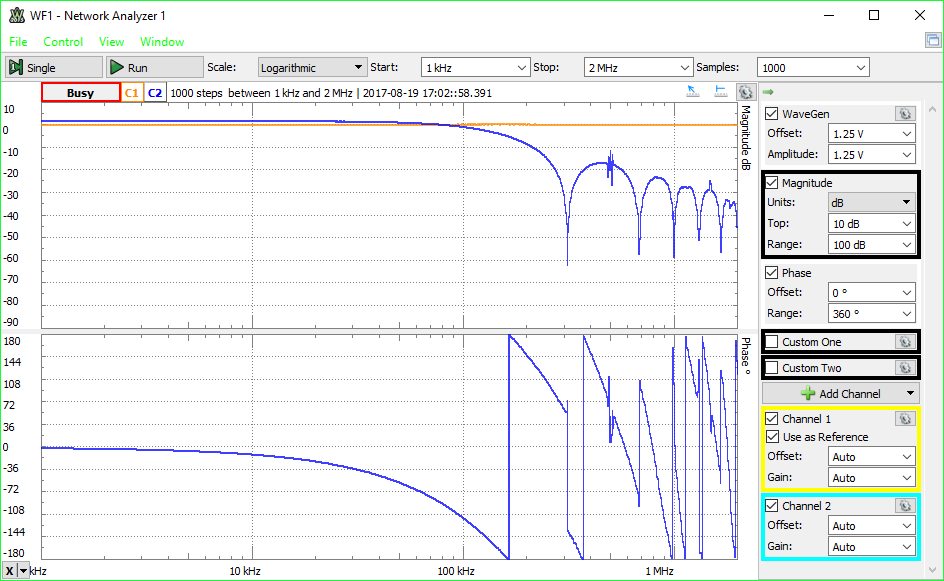
* Implementing the difference equation with floats is simple; you can implement the equation as-is.
* If using large integers, make sure you don’t miss any orders of magnitude in your calculations.
* If you use binary fractions, the difference equation would be written as:



* + The largest 32-bit positive value less than 1 is 1 - 2-31 ≈ 0.9999999995343, which in hexadecimal is 0x0.FFFFFFFE (or 0x7FFFFFFF bit-reversed). The largest usable, non-float value less than 1 on the 769 is 1 - 2-63 via 64-bit integers.

After implementing the filter, connect the output to an oscilloscope and input different frequencies from the signal generator. Explain your results. You should be able to show in the z-plane that the discrete time system has a zero on the unit circle where the output is a minimum and a pole near the unit circle where the output is a maximum. The output for waveforms near DC will have high gains and may cause data overflows. Don’t worry about fixing this other than by reducing the input amplitude, but be ready to explain what is observed.

Ultimately, the filter should look something like this:



In digital filters, frequency increases counterclockwise along the unit circle with 0 Hz at z = 1 and fs/2 (half the sample frequency) at z = -1. The sample frequency is determined by 1/T where T is the time for which an output voltage is held. The angle to the zero on the unit circle is 112.61986° so that frequency corresponds proportionally to . Verify that the notch appears at the correct frequency: This is the frequency of an input sine wave that gives an output amplitude of zero. H(1) is the D.C. gain of a filter. Calculate this value and verify experimentally that it is correct with the rounded coefficients used in the implementation compared to the ideal coefficients.



It is easy to implement the filter with a standard C expression calculating *y*(*k*) instead of using the MAC. This filter will behave the same, except potentially at a much slower sampling rate. This is useful for characterizing and debugging to verify correct operation. Since most are not familiar with how digital filters work, it is now mandatory that the software implementation be done first to aid in the difficult task of troubleshooting the MAC version. This can be observed and analyzed to determine how the filter should behave and used for comparison while fixing the MAC implementation. Also, it will be a safety net since you will receive partial credit for demonstrating a working software implementation if you are unable to get the MAC working.

An optimized C HAL implementation can get results very close to raw instructions (*Why?*), but using assembly instructions correctly will result in more consistent manipulations of higher frequency waves (i.e. > 300 kHz) in more stringent conditions than we have set up on the DISCO board for this lab. You can actually see this for yourself if you use ADC\_SAMPLETIME\_3CYCLES with a 500 kHz to 1 MHz input wave and compare minimal HAL vs. optimized assembly implementations.

*Good programmer's tip:* Design the program top-down. Then write the routines bottom-up. Write them one at a time and thoroughly test each one before integrating them. This way you will have isolated any errors to the routine that you are currently writing. Good programmers follow this method.

***Important note regarding oscilloscopes:***

One of the apparent serious deficiencies of oscilloscope use is the lack of understanding of how scope probes should be used. The 10x scope probes that are in the utility bags attached to the oscilloscopes are precision balanced circuits designed to minimize the loading on the circuits to which they are attached. The equivalent impedance is 1 MΩ or higher. As a result, the probes are 10x attenuating probes, which means the voltage seen at the scope is 1/10 that of the actual voltage at the probe end. Most modern scopes will automatically adjust the voltage gain on y-axis scale to accommodate for the attenuation. Occasionally the user must verify this, or be off by a factor of 10 in all voltage measurements.

The bottom line is that scope probes must only be used as input lines to oscilloscopes. They must **never** be used as simple alligator to BNC cables on the output of a function generator—or really any other connection beyond what they are intended for.

*Tip for combining two 32-bit values into a 64-bit value:*

Don’t use a pointer to combine two adjacent 32-bit values in an array into one 64-bit value. These methods are much faster (shifts are extremely fast – a single clock cycle in the best circumstances).

To concatenate small values into a larger one:

uint64\_t largevalue = (uint64\_t) highword << 32 | lowword;

To split a large value into smaller ones:

uint64\_t largevalue;

uint32\_t highword = largevalue >> 32;

uint32\_t lowword = largevalue;

An int is 16-bits (half word, also uint16\_t), a uint32\_t is a long int (word, 32-bits), and uint64\_t is a long long int (double word, 64-bits).

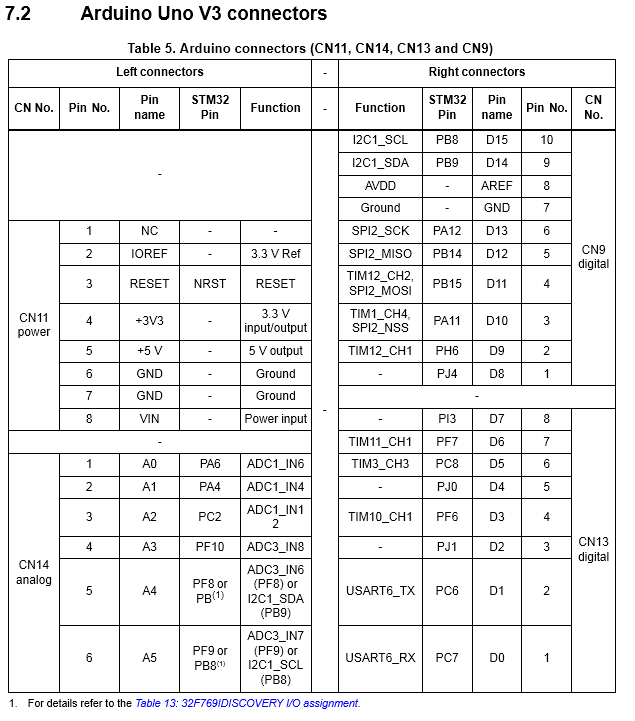
*More hints to help with troubleshooting:*

If you follow **all** the above steps correctly and can verify all the intermediate results, but get to the final full 2nd order filter implementation and it does not behave correctly, there are several things that can be done quickly to determine where things are going wrong. First, set all gains to zero except the coefficient in the *x*(*k*) term – this should be set to 1. Now this 0th order filter should set the output to be the same as the input. If this is not observed there is probably an offset problem or a shift problem in the output. It is possible the result is using a value that is shifted to the right or left of what it should be.

After verifying results with the first coefficient set to unity, set the first two coefficients to unity (gain on both the *x*(*k*) and *x*(*k* - 1) terms) while leaving the rest at zero. This is a 1st order low-pass filter with a D.C. gain of 2 and an output of zero when the input frequency is half the sampling frequency. The peak-peak value of the input signal may need to be reduced at low frequencies to prevent overflowing output.

Another simple check is the D.C. gain of the final filter. With Laplace transforms, the D.C. gain is the value of the transfer function at s = 0, or H(0). In z-transforms, the equivalent value is for z = 1, or evaluating the transfer function H(z) when z is set to 1. Putting in a constant voltage should output a constant voltage that is higher by the D.C. gain of the filter. Anything else means there is a problem with the final shifting in the calculations to normalize values.

Troubleshooting skills are valuable and this lab in particular is one where being able to figure out solutions on your own without using up scarce TA time will help your performance grade for the course. Creating known dummy input values for the filter against which you can check calculated values from the program to hand-calculated results should be useful for revealing problems and indicating solutions.



**Figure 5: Page 25, 769 Discovery Kit User’s Manual.pdf**

**A/D Converter Input Voltage Board**

Note: Replace all instances of +5V with +3.3V.

AD-Conv_pots

**OperandsAndConstraints.txt**

A Complete List of GCC ARM Assembly Operand Modifiers and Constraints

Valid as of Saturday, July 22, 2017.

#######################################################################

######################### ASM Operand modifiers #########################

#######################################################################

Valid operand modifiers for use in asm statements.

A : A memory operand for a VLD1/VST1 instruction.

a : Print as a memory address.

c : Don't print "#" before an immediate operand.

P : Print a VFP double precision register.

q : Print a NEON quad precision register.

y : Print a VFP single precision register as indexed double.

B : Bitwise inverse of integer or symbol without a preceding #.

L : The low 16 bits of an immediate constant.

M : A register range suitable for LDM/STM.

m : The base register of a memory operand.

R : The most significant register of a pair.

Q : The least significant register of a pair.

e : The low doubleword register of a NEON quad register.

f : The high doubleword register of a NEON quad register.

H : The highest-numbered register of a pair.

h : A range of VFP/NEON registers suitable for VLD1/VST1.

0-9 : %0 means "first operand, regardless of whether it's input or output," %1 means "the next operand listed after the first one," etc.

Example: Using %P[thing] in an asm instruction explicitly defines that a double is located at the memory address represented by variable "thing"

Above operands were found in LLVM's ARMAsmPrinter.cpp.

#######################################################################

################## Input/Output Operand Constraint Modifiers #################

#######################################################################

These are placed in front of constraints in operands if needed (like "=r" for a write-only output general register).

"=" : Can write to the operand only.

"+" : Can read from and write to the operand.

"&" : Output operand is stored before the need for the input operands (and their index registers) is over.

"%" : This operand and the next operand are commutative (order interchangeable).

#######################################################################

######################### GCC Common Constraints #######################

#######################################################################

Register constraints

"r" : Matches any general register.

"g" : Any register, memory or immediate integer operand is allowed, except for registers that are not general registers.

Memory constraints

"m" : Matches any valid memory.

"o" : Matches an offsettable memory reference.

Normal constraints

"0-9" : Use this operand for the #th operand (for example: asm("mov %0, r1" : "=r" (value) : "0" (value)); means use the same (value) register that is used for input for output)

(#) Similar to +

"V" : Matches a non-offsettable memory reference.

(#) "V" matches TARGET\_MEM\_CONSTRAINTs that are rejected by "o".

(#) This means that it is not a memory constraint in the usual sense, since reloading the address into a base register would make the address offsettable.

(#) "o" and "V" are like components of "m" in a way.

"<" : Matches a pre-dec or post-dec operand.

">" : Matches a pre-inc or post-inc operand.

(#) Like "V", "<" and ">" are not memory constraints, since reloading the address into a base register would cause it not to match.

"i" : Matches a general integer constant.

"s" : Matches a symbolic integer constant.

"n" : Matches a non-symbolic integer constant.

"E" : Matches a floating-point constant.

"F" : Matches a floating-point constant.

(#) There is no longer a distinction between "E" and "F".

"X" : Matches anything.

Address constraints

"p" : Matches a general address.

########################################################################

####################### ARM-SPECIFIC CONSTRAINTS ####################

########################################################################

============================================================================

**NOTE: Anything marked @internal should NOT BE USED in inline asm statements.**

These are included purely for edification and completeness.

============================================================================

The following register constraints have been used:

- in ARM/Thumb-2 state: t, w, x, y, z

- in Thumb state: h, b

- in both states: l, c, k, q, Cs, Ts, US

In ARM state, 'l' is an alias for 'r'

'f' and 'v' were previously used for FPA and MAVERICK registers.

The following normal constraints have been used:

- in ARM/Thumb-2 state: G, I, j, J, K, L, M

- in Thumb-1 state: I, J, K, L, M, N, O

'H' was previously used for FPA.

The following multi-letter normal constraints have been used:

- in ARM/Thumb-2 state: Da, Db, Dc, Dd, Dn, Dl, DL, Do, Dv, Dy, Di, Dt, Dp, Dz

- in Thumb-1 state: Pa, Pb, Pc, Pd, Pe

- in Thumb-2 state: Pj, PJ, Ps, Pt, Pu, Pv, Pw, Px, Py

- in all states: Pf

The following memory constraints have been used:

- in ARM/Thumb-2 state: Uh, Ut, Uv, Uy, Un, Um, Us

- in ARM state: Uq

- in Thumb state: Uu, Uw

- in all states: Q

Register constraints

"t" : The VFP registers s0-s31.

"w" : The VFP registers d0-d15, or d0-d31 for VFPv3.

"x" : The VFP registers d0-d7.

"y" : The Intel iWMMX co-processor registers.

"z" : The Intel iWMMX GR registers.

"l" : In Thumb state the core registers r0-r7.

"h" : In Thumb state the core registers r8-r15.

"Ts" : For arm\_restrict\_it the core registers r0-r7. GENERAL\_REGS otherwise.

@internal "k" : The stack register.

@internal "q" : In ARM state with LDRD support, core registers, otherwise general registers.

@internal "b" : Thumb only. The union of the low registers and the stack register.

@internal "c" : The condition code register.

@internal "Cs" : The caller save registers. Useful for sibcalls.

Normal constraints

"j" : A constant suitable for a MOVW instruction. (ARM/Thumb-2)

"I" : In ARM/Thumb-2 state a constant that can be used as an immediate value in a Data Processing instruction. In Thumb-1 state a constant in the range 0-255.

"J" : In ARM/Thumb-2 state a constant in the range -4095 to 4095. In Thumb-1 state a constant in the range -255 to -1.

"K" : In ARM/Thumb-2 state a constant that satisfies the "I" constraint if inverted. In Thumb-1 state a constant that satisfies the "I" constraint multiplied by any power of 2.

"L" : In ARM/Thumb-2 state a constant that satisfies the "I" constraint if negated. In Thumb-1 state a constant in the range -7 to 7.

"M" : In Thumb-1 state a constant that is a multiple of 4 in the range 0-1020.

"N" : Thumb-1 state a constant in the range 0-31.

"O" : In Thumb-1 state a constant that is a multiple of 4 in the range -508 to 508.

"G" : In ARM/Thumb-2 state the floating-point constant 0.

"Pf" : Memory models except relaxed, consume or release ones.

@internal The ARM state version is internal...

@internal In ARM/Thumb-2 state a constant in the range 0-32 or any power of 2.

(#) The above two lines are not typos.

@internal "Pj" : A 12-bit constant suitable for an ADDW or SUBW instruction. (Thumb-2)

@internal "PJ" : A constant that satisfies the Pj constrant if negated.

@internal "Pa" : In Thumb-1 state a constant in the range -510 to 510.

@internal "Pb" : In Thumb-1 state a constant in the range -262 to 262.

@internal "Pc" : In Thumb-1 state a constant that is in the range 1021 to 1275.

@internal "Pd" : In Thumb state a constant in the range 0 to 7.

@internal "Pe" : In Thumb-1 state a constant in the range 256 to 510.

@internal "Ps" : In Thumb-2 state a constant in the range -255 to 255.

@internal "Pt" : In Thumb-2 state a constant in the range -7 to 7.

@internal "Pu" : In Thumb-2 state a constant in the range +1 to 8.

@internal "Pv" : In Thumb-2 state a constant in the range -255 to 0.

@internal "Pw" : In Thumb-2 state a constant in the range -255 to -1.

@internal "Px" : In Thumb-2 state a constant in the range -7 to -1.

@internal "Py" : In Thumb-2 state a constant in the range 0 to 255.

@internal "Pz" : In Thumb-2 state the constant 0.

@internal "Dz" : In ARM/Thumb-2 state a vector of constant zeros.

@internal "Da" : In ARM/Thumb-2 state a const\_int, const\_double or const\_vector that can be generated with two Data Processing insns.

@internal "Db" : In ARM/Thumb-2 state a const\_int, const\_double or const\_vector that can be generated with three Data Processing insns.

@internal "Dc" : In ARM/Thumb-2 state a const\_int, const\_double or const\_vector that can be generated with four Data Processing insns. This pattern is disabled if optimizing for space or when we have load-delay slots to fill.

@internal "Dd" : In ARM/Thumb-2 state a const\_int that can be used by insn adddi.

@internal "De" : In ARM/Thumb-2 state a const\_int that can be used by insn anddi.

@internal "Df" : In ARM/Thumb-2 state a const\_int that can be used by insn iordi.

@internal "Dg" : In ARM/Thumb-2 state a const\_int that can be used by insn xordi.

@internal "Di" : In ARM/Thumb-2 state a const\_int or const\_double where both the high and low SImode words can be generated as immediates in 32-bit instructions.

@internal "Dn" : In ARM/Thumb-2 state a const\_vector or const\_int which can be loaded with a Neon vmov immediate instruction.

@internal "Dl" : In ARM/Thumb-2 state a const\_vector which can be used with a Neon vorr or vbic instruction.

@internal "DL" : In ARM/Thumb-2 state a const\_vector which can be used with a Neon vorn or vand instruction.

@internal "Do" : In ARM/Thumb2 state valid offset for an ldrd/strd instruction.

@internal "Dv" : In ARM/Thumb-2 state a const\_double which can be used with a VFP fconsts instruction.

@internal "Dy" : In ARM/Thumb-2 state a const\_double which can be used with a VFP fconstd instruction.

@internal "Dt" : In ARM/ Thumb2 a const\_double which can be used with a vcvt.f32.s32 with fract bits operation.

@internal "Dp" : In ARM/ Thumb2 a const\_double which can be used with a vcvt.s32.f32 with bits operation.

Memory constraints

@internal "Ua" : An address valid for loading/storing register exclusive.

@internal "Uh" : An address suitable for byte and half-word loads which does not point inside a constant pool.

@internal "Ut" : In ARM/Thumb-2 state an address valid for loading/storing opaque structure types wider than TImode.

@internal "Uv" : In ARM/Thumb-2 state a valid VFP load/store address.

@internal "Uy" : In ARM/Thumb-2 state a valid iWMMX load/store address.

@internal "Un" : In ARM/Thumb-2 state a valid address for Neon doubleword vector load/store instructions.

@internal "Um" : In ARM/Thumb-2 state a valid address for Neon element and structure load/store instructions.

@internal "Us" : In ARM/Thumb-2 state a valid address for non-offset loads/stores of quad-word values in four ARM registers.

@internal "Uq" : In ARM state an address valid in ldrsb instructions.

@internal "Q" : An address that is a single base register.

@internal "Uu" : In Thumb state an address that is valid in 16bit encoding.

@internal "Uw" : In Thumb state an address that is valid in 16bit encoding, and that can be used for unaligned accesses.

(#) The 16-bit post-increment LDR/STR accepted by thumb1\_legitimate\_address\_p are actually LDM/STM instructions, so cannot be used to access unaligned data.

@internal "US" : US is a symbol reference.

@internal "Uz" : A memory access that is accessible as an LDC/STC operand.

We used to have constraint letters for S and R in ARM state, but all uses of these now appear to have been removed.

Additionally, we used to have a Q constraint in Thumb state, but this wasn't really a valid memory constraint. Again, all uses of this now seem to have been removed.

All GCC constraints were found in /gcc/config/arm/contraints.md and /gcc/common.md.

1. Channels are shared as per the figures in Ch. 15.3.2 in *769 Reference Manual (Register Map).pdf*. [↑](#footnote-ref-1)
2. Discontinuous mode: given a group of pre-selected input channels (like for scan mode), the ADC samples a subset of pre-selected size on each trigger until all have been sampled. E.g., for a group of 8 channels, it can sample the first 3 on trigger #1, the next 3 on trigger 2, etc. It rolls over only after converting the whole group. The max subset size is 8 channels per trigger. [↑](#footnote-ref-2)
3. But don’t worry, we will be using the USB 2.0 OTG port (CN15) in a later lab. Yes, you are going to learn USB. :) [↑](#footnote-ref-3)
4. However, if you’re feeling really creative, the audio codec on the Serial Audio Interface does contain 4 DAC channels and 2 ADC channels… <http://origin.cirrus.com/cn/pubs/proDatasheet/WM8994_v4.5.pdf> [↑](#footnote-ref-4)
5. See Ch. 3 in *STM32 Using the Floating Point Unit (FPU).pdf* for a description of what this means. [↑](#footnote-ref-5)
6. It turns out that PA0 is on a permanent pull-down, as you can see on page 49 of the Discovery kit schematic. [↑](#footnote-ref-6)