GEORGIA INSTITUTE OF TECHNOLOGY
SCHOOL of ELECTRICAL and COMPUTER ENGINEERING

ECE 4550 — Control System Design — Summer 2020

Lab #4: D-to-A and A-to-D Conversion

Contents

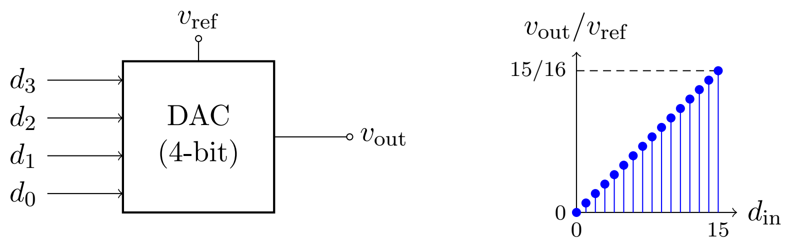
1 Background Material

1.1 Introductory Comments

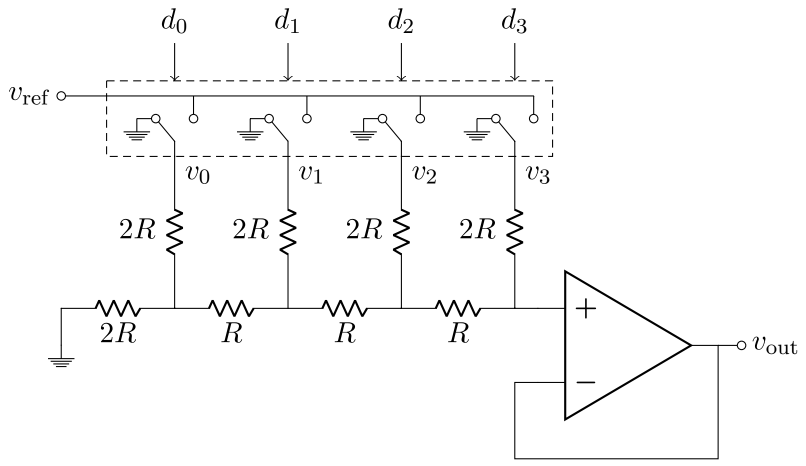
The objective of this lab is to perform digital-to-analog conversion and analog-to-digital conversion periodically in time using a programmable conversion frequency. To achieve this objective we must understand how to program the timer subsystem, the DAC subsystem and the ADC subsystem, and how to use the interrupt subsystem with the ADC subsystem as an interrupt source. Tasks 1 and 2 of this lab build on Lab 3 in the sense that periodic operation of GPIO pins will now be augmented with periodic reconstruction of continuously-valued analog voltage signals at DAC pins and periodic sampling of continuously-valued analog voltage signals at ADC pins; these extensions are essential features of digital signal processing systems, as well as digital control systems that utilize actuators with analog inputs and sensors with analog outputs.[1]

1.2 Fundamentals of D-to-A Converter Circuits

Figure 1 provides two representations of a D-to-A converter, the first emphasizing the high-level function of the component and the second revealing the internal circuitry of the component. To keep the diagrams simple, a 4-bit converter is used as an example, whereas we will be working with 12-bit converters on the F28379D device.



(a) Systems perspective.



(b) Circuits perspective.

Figure 1: The high-level function and internal structure of a 4-bit D-to-A converter.

From the systems perspective, we may think of a D-to-A converter as being a component characterized by the block diagram and input-output relation of Figure 1a. This component has a digital input (bits d_3, d_2, d_1, d_0) and an analog output (voltage $0 \leq v_{\text{out}} < v_{\text{ref}}$).

From the circuits perspective, the D-to-A converter would be implemented as shown in the schematic diagram of Figure 1b (half digital, half analog). The digital input determines switch states and ultimately node voltages according to

$$v_i = \begin{cases} v_{\text{ref}} & , \ d_i = 1 \\ 0 & , \ d_i = 0 \end{cases}$$

[1]Some control systems are implemented using actuators and sensors that are designed to permit communication with the microcontroller in a fully digital way, using pins multiplexed to special-purpose microcontroller modules; some of these modules will be explored in future labs this semester, especially PWM and QEP.

and these node voltages feed a resistor ladder and buffer amplifier to yield analog output voltage

$$v_{\text{out}} = \left(\frac{d_3}{2^1} + \frac{d_2}{2^2} + \frac{d_1}{2^3} + \frac{d_0}{2^4} \right) v_{\text{ref}}.$$

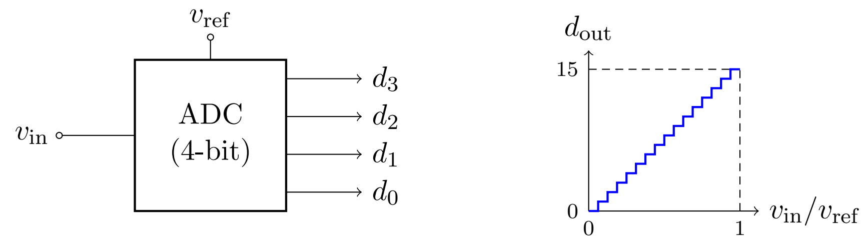
Generalizing this circuit to N bits would thus provide an output voltage

$$v_{\text{out}} = \left(\frac{N\text{-bit unsigned integer}}{2^N} \right) v_{\text{ref}}$$

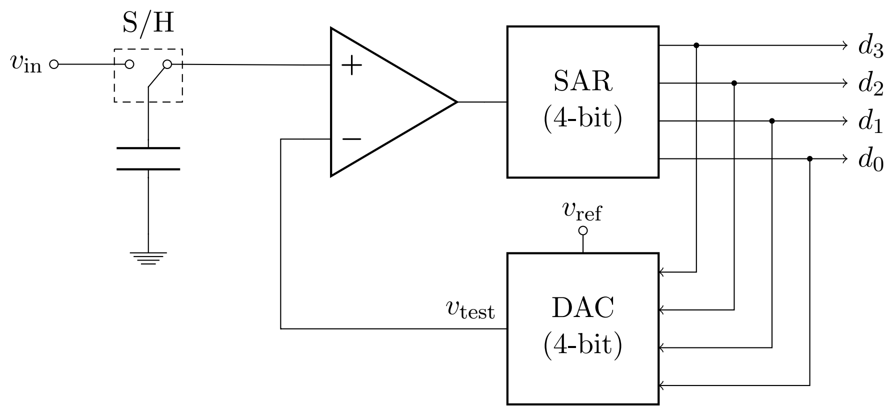
in response to a given N-bit unsigned integer.

1.3 Fundamentals of A-to-D Converter Circuits

Figure 2 provides two representations of an A-to-D converter, the first emphasizing the high-level function of the component and the second revealing the internal circuitry of the component. To keep the diagrams simple, a 4-bit converter is used as an example, whereas we will be working with 12-bit converters on the F28379D device.



(a) Systems perspective.



(b) Circuits perspective.

Figure 2: The high-level function and internal structure of a 4-bit A-to-D converter.

From the systems perspective, we may think of an A-to-D converter as being a component characterized by the block diagram and input-output relation of Figure 2a. This component has an analog input (voltage $0 \leq v_{\text{in}} < v_{\text{ref}}$) and a digital output (bits d_3, d_2, d_1, d_0).

From the circuits perspective, the A-to-D converter would be implemented as shown in the schematic diagram of Figure 2b (half analog, half digital). The first stage of the conversion process is the *sample-and-hold stage*, during which the switch connects the pin at voltage v_{in} to the sample-and-hold capacitor. Once the capacitor has charged, the switch then connects the capacitor to the comparator to begin the second stage of the conversion process, the *successive approximation stage*. During this stage, the iteration shown in Figure 3 is used to determine the digital output.

```
set $d_3 = 0$, $d_2 = 0$, $d_1 = 0$, $d_0 = 0$
for $i = 3, 2, 1, 0$
    set $d_i = 1$
    set $v_{\text{test}} = \left(\frac{d_3}{2^1} + \frac{d_2}{2^2} + \frac{d_1}{2^3} + \frac{d_0}{2^4}\right) v_{\text{ref}}$
    if $v_{\text{in}} > v_{\text{test}}$
        set $d_i = 1$
    else
        set $d_i = 0$
    end
end
```

Figure 3: Pseudo-code of successive approximation method used by A-to-D converters.

According to Figure 3, in bit-by-bit fashion individual bits are tentatively set to 1 and then ultimately fixed at 1 or 0 according to the comparator output. By comparing the value of v_{in} to the value of v_{test}—the trial voltage produced by an internal DAC fed by a trial combination of bits—the successive approximation method is guaranteed to converge to the correct digital output. As an example, suppose that $v_{\text{in}} = 0.6v_{\text{ref}}$. For step 1, $d_3 = 1$ yields $v_{\text{test}} = 0.5v_{\text{ref}}$, resulting in a final choice of $d_3 = 1$. For step 2, $d_2 = 1$ yields $v_{\text{test}} = 0.75v_{\text{ref}}$, resulting in a final choice of $d_2 = 0$. For step 3, $d_1 = 1$ yields $v_{\text{test}} = 0.625v_{\text{ref}}$, resulting in a final choice of $d_1 = 0$. For step 4, $d_0 = 1$ yields $v_{\text{test}} = 0.5625v_{\text{ref}}$, resulting in a final choice of $d_0 = 1$. Therefore, the digital output ultimately settles on `0b1001`, which corresponds to the unsigned integer $9 \in \{0, 1, \ldots, 15\}$.

Generalizing this circuit to N bits, the analog input voltage v_{in} would produce the digital output

$$N\text{-bit unsigned integer} = \text{floor}\left(2^N \frac{v_{\text{in}}}{v_{\text{ref}}}\right), \quad 0 \leq v_{\text{in}} < v_{\text{ref}}$$

after N successive approximation steps. Clearly, A-to-D conversion is more complicated and more time consuming than D-to-A conversion; in fact, the former actually incorporates the latter.

1.4 Variations on Interrupt-Based Code Flow

As you contemplate your microcontroller code development, keep in mind that the actuation system and the sensing system are related by reciprocal structures, as exhibited in Figure 4. The actuation system begins with a command value, whereas the sensing system ends with a measurement value, and both of these numbers would preferably be expressed using floating-point data types to explicitly describe the physical signals of actuation and sensing in SI units; since the DAC and ADC modules intrinsically operate with unsigned integer data types, software mappings (scale factors) must be included in the application code. Both systems require circuitry at the microcontroller pins to interface the internal microcontroller modules to the necessary external components.

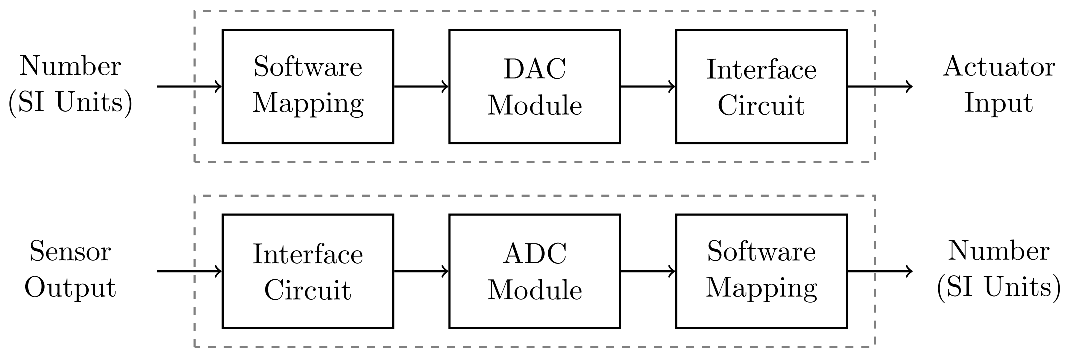


Figure 4: The reciprocal structures of DAC actuation and ADC sensing.

Interrupt-based code flow will be used to implement the DAC actuation and ADC sensing systems just described, as shown in Figure 5. Timer hardware is used to trigger time-periodic ADC start of conversion (SOC) events, resulting in corresponding time-periodic ADC end of conversion (EOC) events; the EOC events are offset from the SOC events by the length of the ADC conversion process, which consists of a sample-and-hold stage and a successive approximation stage. The EOC events will trigger ADC interrupts, and the associated ADC interrupt service routine provides a means for reading from ADC sensors and writing to DAC actuators in time-periodic fashion.

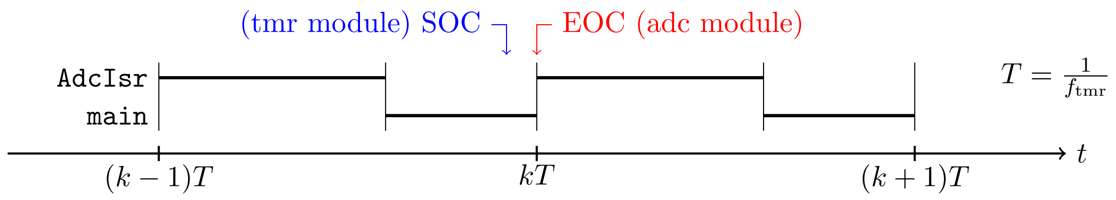


Figure 5: Interrupt-based code flow with DAC actuation and ADC sensing.

To achieve tight time synchronization, tasks performed in the ISR could be as follows.

```
interrupt void AdcIsr (void)
{
    // 1. (Fast) Write an output voltage value to the DAC module.
    // 2. (Fast) Read an input voltage value from the ADC module.
    // 3. (Slow) Compute and store the next output voltage value.
    // 4. (Fast) Acknowledge the interrupt.
}
```

1.5 Relevant Microcontroller Documentation

The overall objective of these lab projects is to teach you how to do embedded design with microcontrollers in a general sense, not just how to approach one specific application using one specific microcontroller. Therefore, the guidance provided herein focuses more on general thought processes and programming recommendations; step-by-step instructions of an extremely specific nature have been intentionally omitted. Use fundamental documentation as your primary source of information as you work through details of implementation. Being able to read and understand such documentation is an important skill to develop, as similar documentation would need to be consulted in order to use other microcontrollers or other application hardware. By making the effort to extract required details from fundamental documentation yourself, you will have developed transferable skills that will serve you well in your engineering career. For this lab, review:

- F28379D Launch Pad Schematic
- F28379D Datasheet
  - Figures 4-1 through 4-4 and Table 4-1, for pin descriptions.
- F28379D Technical Reference Manual
  - §11.1, for DAC module description.
  - §11.4, for DAC module registers.
  - §10.1, for ADC module description.
  - §10.4, for ADC module registers.

1.6 Target Hardware Schematic Diagrams and Data Sheets

First consult F28379D Datasheet to determine which F28379D pins are associated with the DAC module and the ADC module. Then consult F28379D Launch Pad Schematic to determine which DAC pins and which ADC pins are hardwired to the header pins referred to in the task specifications.

2 DAC Module: Step-by-Step Guidelines

The F28379D device has three DAC modules (A, B and C), each having one buffered analog output pin; since these pins are shared, an application will be able to use them as analog output pins or analog input pins (i.e. ADC pins), but not both. The F28379D Launch Pad headers provide access to two of these analog output pins. The range of the analog output pins is assigned by externally supplied reference voltages. On the F28379D Launch Pad board, a common reference voltage equal to 3 V is supplied to all three DAC modules. The analog output pins provide 12-bit resolution. Figure 5-27 of Datasheet and Figure 11-1 of Technical Reference Manual provide block diagrams of the DAC modules.

2.1 Initialize the DAC Registers

The first action that must be taken is to turn on the clock signal that feeds the DAC module; until this action is taken, writes to the DAC module registers will have no effect. The HAL variables used to perform this step appear in the code snippet shown below; the short delay introduced by the two “no operation” assembly language instructions is critical to guarantee that subsequent writes to DAC module registers will be recognized.

```
CpuSysRegs.PCLKCR16.bit.DAC_? = ?;
asm(" NOP"); asm(" NOP");
```

The next configuration step is to select the DAC module reference voltage. We will use VREFHI, which is 3 V on the Launch Pad. The HAL variables used to perform this step are

```
Dac?Regs.DACCTL.bit.DACREFSEL
```

The next configuration step is to enable DAC module output; this step is required because DAC modules and ADC modules share pins. The HAL variables used to perform this step are

```
Dac?Regs.DACOUTEN.bit.DACOUTEN
```

The final configuration step is to assign the desired initial value for DAC output voltage. Details of this step appear in the following section. The HAL variables used to perform this step are

```
Dac?Regs.DACVALS.all
```

2.2 Utilize the DAC Pins

Figure 6 visualizes the signal flow associated with each DAC module. On the hardware side, the DAC output pin connects the module to a driven load circuit. On the software side, the desired output voltage is requested as a floating-point value in SI units. Registers represent the interface between the hardware and software sides; the DACVALS registers (one per module) are the means by which the desired output voltage is requested.

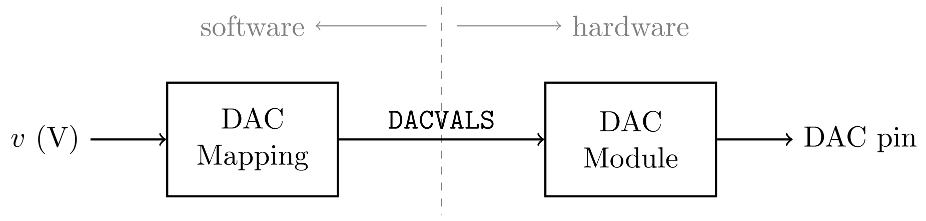


Figure 6: DAC module input-output signal flow.

Since the DAC modules have 12-bit resolution, the scale factor will be $2^{12} = 4096$. The unsigned integer value written to the DACVALS register is computed from the commanded output voltage Vcmd, and the reference voltage Vref, according to

```
Dac?Regs.DACVALS.all = 4096*Vcmd/Vref;
```

where Vcmd and Vref are float32 values and the casting is implicit. The smallest output voltage equal to $(0)\times$Vref is commanded by writing DACVALS = 0x000. The largest output voltage equal to $(4095/4096)\times$Vref is commanded by writing DACVALS = 0xFFF.

3 ADC Module: Step-by-Step Guidelines

The F28379D device 337-ball variant has four ADC modules (A, B, C and D). In combination, these ADC modules provide 24 analog input pins; since 3 of these pins are shared, an application will be able to use them as analog input pins or analog output pins (i.e. DAC pins), but not both. The F28379D Launch Pad headers provide access to 16 of these analog input pins, and 2 of these will function as analog input pins or analog output pins. The range of the analog input pins is assigned by externally supplied reference voltages, one per module. On the F28379D Launch Pad board, a common reference voltage equal to 3 V is supplied to all four ADC modules. The analog input pins can be configured for either 16-bit or 12-bit resolution, but we will be using the 12-bit mode. Figure 5-27 of Datasheet and Figure 10-1 of Technical Reference Manual provide block diagrams of the ADC modules.

Conversions are configured using three start-of-conversion (SOC) parameters: one parameter defines the trigger source used to initiate conversions; another parameter defines which pin voltage to convert; and another parameter defines the sample-and-hold duration. A typical way to achieve time-periodic conversion is to assign a CPU timer as the trigger source. The conversion process involves two phases, the sample-and-hold phase and the successive approximation phase, after which the value obtained from a conversion is stored in a conversion result register. An end-of-conversion (EOC) pulse is generated either at the end of the first or second phase of the conversion process, depending on a configuration setting. The EOC pulse may be used as an interrupt trigger, the goal being to automatically interrupt the main program whenever the conversion result register is ready to be read. See §10.1.4 and §10.1.8 of Technical Reference Manual for further details on the SOC, EOC and interrupt concepts.

3.1 Initialize the ADC Registers

3.1.1 Enable and Configure the Module Clock

The first action that must be taken is to turn on the clock signal that feeds the ADC module; until this action is taken, writes to the ADC module registers will have no effect. The HAL variables used to perform this step appear in the code snippet shown below; the short delay introduced by the two "no operation" assembly language instructions is critical to guarantee that subsequent writes to ADC module registers will be recognized.

```
CpuSysRegs.PCLKCR13.bit.ADC_? = ?;
asm(" NOP"); asm(" NOP");
```

The next configuration step is to set the internal clock frequency used within the ADC module; to operate within specification, 5 MHz $\leq f_{\text{clk,adc}} \leq$ 50 MHz according to Table 5-44 of Datasheet. The HAL variables used to perform this step are

```
Adc?Regs.ADCCTL2.bit.PRESCALE
```

Note that $f_{\text{clk,adc}}$ is merely an internal clock frequency of the ADC module, and therefore it does not determine the frequency of ADC conversion requests.

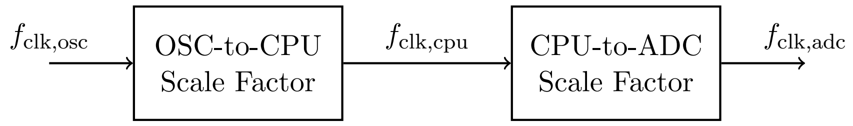


Figure 7: Configuring the ADC module clock frequency (not sampling frequency).

3.1.2 Enable the Module Power Supply

The analog supply within the ADC module is powered down by default, to improve the energy efficiency of applications that do not require this module. Before use, it is therefore necessary to power up the analog supply, and the HAL variables used to perform this step are

```
Adc?Regs.ADCCTL1.bit.ADCPWDNZ
```

In order to operate within specification, it is necessary to wait 500 μs after enabling the analog supply before performing any conversions, according to Table 5-45 of Datasheet. Any conversions requested during this startup interval will not yield accurate results. The preferred way to implement required wait intervals is to call a well-calibrated assembly language function; TI has supplied the function `DelayUs` for this purpose (it's on Canvas). Use `extern void DelayUs(Uint16);` to declare this function. One call to this function provides up to 65535 μs of delay.

3.1.3 Configure the Conversions

Each ADC module has 16 sets of SOC parameters (SOC0–SOC15) used to configure conversions. Each of these sets has three parameters: one parameter defines the trigger source that initiates the conversion; another parameter defines the input pin that will be converted; and another parameter defines the sample-and-hold duration to be used. If a trigger source attempts to simultaneously initiate the conversion of two or more input pins associated with a single module, then an SOC priority scheme is used to determine the conversion sequence; see §10.1.6 for details. The HAL variables used to perform this step are

```
Adc?Regs.ADCSOC?CTL.bit.TRIGSEL
Adc?Regs.ADCSOC?CTL.bit.CHSEL
Adc?Regs.ADCSOC?CTL.bit.ACQPS
```

The CPU timer hardware is typically used to provide a time-periodic SOC trigger source, in which case conversions will occur at the CPU timer frequency, say f_{tmr}.

3.1.4 Configure and Enable the Interrupts

Each ADC module has 4 interrupts (ADCINT1–ADCINT4) that may be used to systematize the reading of conversion result registers. Each SOC parameter set has a corresponding EOC pulse marking the completion of (i) the sample-and-hold phase or (ii) the successive approximation phase, depending on the assigned pulse position parameter; we prefer that the EOC pulse be issued at the true end of conversion (i.e. after completion of both previously mentioned phases), thus indicating the time instant at which a new conversion result is loaded into the conversion result register.

A visit to a particular ISR will be triggered by a particular EOC pulse, so proper configuration requires a closer look at this connection. Consider two typical situations:

1. Suppose it's desired to convert just one pin voltage in response to an SOC trigger (e.g. a timer event). Only one SOC parameter set is assigned, only one EOC pulse is available, and only one ISR is needed. In this case, there is a one-to-one correspondence between the single EOC pulse and the single ISR.

2. Suppose it's desired to convert more than one pin voltage in response to a single SOC trigger (e.g. a timer event). Multiple SOC parameter sets are assigned, multiple EOC pulses are available, multiple ISRs are possible but one ISR is typically sufficient. Assuming all conversions are performed by a single module, the SOC trigger initiates a sequence of conversions in order of priority. By choosing the lowest priority EOC pulse as the interrupt trigger for a single ISR, we can be assured that all conversions initiated by the most recent SOC trigger will be complete each time we visit the single ISR.

The motivation for using the ADC interrupt system is to avoid the need for continuous polling (i.e. wasting clock cycles) to determine if the conversion result registers have been updated with new valid data. Configuration of the ADC interrupt system requires assigning the pulse position parameter, associating an EOC pulse with an ADC interrupt, and enabling the ADC interrupt. The HAL variables used to perform these steps are

```
Adc?Regs.ADCCTL1.bit.INTPULSEPOS
Adc?Regs.ADCINTSEL?N?.bit.INT?SEL
Adc?Regs.ADCINTSEL?N?.bit.INT?E
```

Since the ADC interrupt is processed through the PIE system, it is necessary to register and enable it within the PIE system; follow the procedure already learned for the timer interrupt in the previous lab. The HAL variables used to perform these steps are

```
PieVectTable.ADC??_INT
PieCtrlRegs.PIEIER?.bit.INTx?
```

3.2 Utilize the ADC Pins

Figure 8 visualizes the signal flow associated with the ADC module. On the hardware side, the ADC input pins connect the module to driving source circuits. On the software side, floating-point representations of the sampled signals in SI units are obtained. Registers represent the interface between the hardware and software sides. Time-periodic conversions may be triggered directly by timer hardware; the `ADCRESULT?` registers contain the results of the conversions.

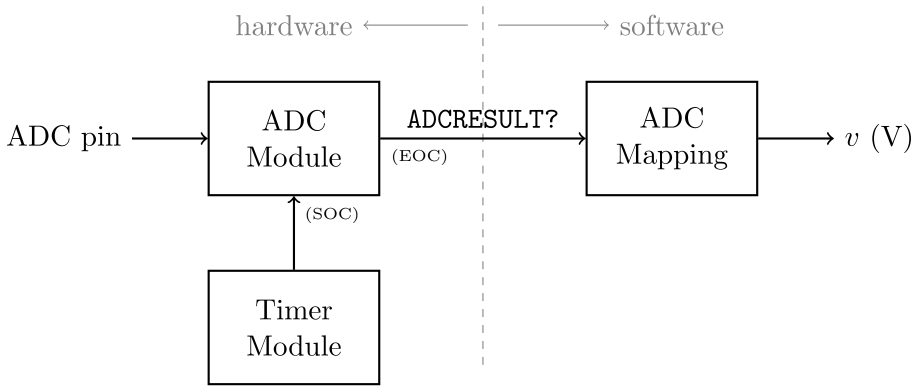


Figure 8: ADC module input-output signal flow.

3.2.1 Initiate a Conversion

By assigning timer hardware as the SOC trigger source (see §3.1.3), the timer hardware will automatically cause time-periodic initiation of ADC conversions in direct manner as shown in Figure 8; in this case, the ADC conversion frequency will be equal to the timer frequency.

3.2.2 Read and Map the Conversion Result

Whenever a conversion result is ready, an EOC pulse triggers an ADC interrupt which leads to a visit to the ADC interrupt service routine where the conversion result register may be read. Since we use the ADC modules in 12-bit mode, the scale factor will be $2^{12} = 4096$. The measured input voltage `Vmea` is computed from the unsigned integer value read from the `ADCRESULT?` register, and the reference voltage `Vref`, according to

```
Vmea = Adc?ResultRegs.ADCRESULT?*Vref/4096;
```

where `Vmea` and `Vref` are `float32` values and the casting is implicit. The smallest input voltage equal to $(0)\times$`Vref` is measured by reading `ADCRESULT? = 0x000`. The largest input voltage equal to $(4095/4096)\times$`Vref` is measured by reading `ADCRESULT? = 0xFFF`.

3.2.3 Clear the Interrupt Flag

Before leaving the ISR, it is necessary to clear the interrupt flag. The HAL variables used to perform this step are

```
Adc?Regs.ADCINTFLGCLR.bit.ADCINT?
```

3.2.4 Acknowledge the Interrupt

Before leaving the ISR, it is also necessary to acknowledge the interrupt within the PIE system. The HAL variable used to perform this step is

```
PieCtrlRegs.PIEACK.all
```

4 Lab Assignment

4.1 Preparation

Before attempting this lab project, do the following:

1. Read through this entire document.
2. Determine how the appropriate registers will be used to complete the tasks assigned in §4.2. Focus on those HAL variables identified in §2–3, and consult appropriate circuit diagrams.

4.2 Tasks

Table 1 specifies the clock frequencies, timer frequency and acquisition time that should be used to complete both of the assigned tasks. The ADC acquisition time of 0.2 μs has been chosen large enough to enable the sample-and-hold capacitor to fully charge, thereby guaranteeing accurate conversions. The ADC iteration time to complete a 12-bit conversion using the successive approximation method is equal to 12 ADC clock periods, which amounts to 0.24 μs when using a 50 MHz ADC clock. The sum of ADC acquisition time and ADC iteration time is the total ADC conversion time, which ends up being 0.44 μs. The chosen 100 kHz timer frequency corresponds to a 10 μs separation between SOC events. Hence, the specified parameters are feasible since 0.44 μs $\ll$ 10 μs allows sufficient time for the computations required to implement a digital filter.

Table 1: Specified Clock Frequencies, Timer Frequency and Acquisition Time

CPU clock frequency	$f_{\text{clk,cpu}}$	200 MHz	max value permitted by DATASHEET
ADC clock frequency	$f_{\text{clk,adc}}$	50 MHz	max value permitted by DATASHEET
CPU timer frequency	f_{tmr}	100 kHz	rate of conversion requests
ADC acquisition time	—	200 ns	for sample-and-hold stage
ADC iteration time	—	240 ns	for successive approximation stage

For both of the assigned tasks, you will need to work with a noise-contaminated 1 kHz sinusoidal voltage signal $u(t)$, sampled at 100 kHz, with values in the 0–3 V range. For this purpose, use the code provided below. The function call `rand_float(a)` returns a random floating-point value uniformly distributed between ±`a`. By pre-computing and storing values of $u(t)$ in array `U` prior to `while(1)`, your interrupt service routine will only be responsible for (i) writing to the DACs,

(ii) reading from the ADCs, (iii) updating the filter and (iv) possibly logging data. Since only 10 periods of the voltage signal will be pre-computed and stored, you will need to cycle through the signal values continuously (with index 999 followed by index 0).

- prior to `main`

```
#include "stdlib.h"

float32 rand_float (float32 a)
{
    return ((float32) rand()/RAND_MAX)*2*a-a;
}
```

- prior to `while(1)`

```
for (i=0; i<1000; i++)
{
    t = i*0.00001;
    U[i] = 1.5+sin(2*pi*1000*t)+rand_float(0.1);
}
```

For both of the assigned tasks, use just one ADC interrupt service routine and do not use any timer interrupt service routine. Use timer hardware as the ADC trigger source to achieve periodic conversion requests. Run your code using RAM debug sessions.

Microcontroller code that runs after entering the `while(1)` loop falls into two categories: (1) code that executes continuously and forever; and (2) code that executes only under special circumstances. You must write your code so as impose this two-category structure.

1. *Unconditional Execution*: Real-time tasks such as DAC actuation, ADC sensing and digital filtering should execute unconditionally.
2. *Conditional Execution*: Auxiliary tasks, such as logging data over a specified finite interval of time, should execute conditionally.

4.2.1 Digital-to-Analog and Analog-to-Digital Conversion

Develop code that uses one DAC to produce an analog voltage on header pin J3-10 and one ADC to measure an analog voltage on header pin J3-9. We connected these pins with wire, so you may use this setup to confirm proper DAC and ADC operation. In Figure 9, the blue portions are software and all else is hardware; note that DAC input signal `dacU` (in V) is continuously fed from the pre-computed 1000-element array `U` described previously, whereas the ADC output signal `adcU` (in V) feeds the initial 1000 measurements of $u(t)$ into array `Y`. Display `dacU` and `adcU` in the Expressions window with continuous refresh to verify that operation is sustained indefinitely. Use the graph tool to verify that the arrays `U` and `Y` are essentially identical.

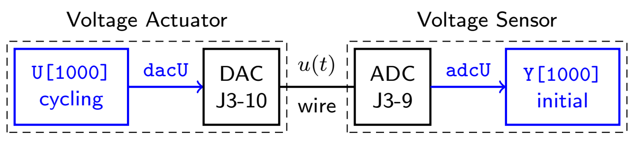


Figure 9: A test setup to demonstrate DAC and ADC operation.

4.2.2 Digital Filtering of Analog Signals to Remove Noise

Develop code that uses two DACs to produce analog voltages on header pins J3-10 and J7-10, and two ADCs to measure analog voltages on header pins J3-9 and J7-9, for the purpose of implementing a third-order Butterworth low-pass filter with cutoff frequency $\omega_c = 2\pi(2000)$ rad/s so that output $y(t)$ is a filtered version of input $u(t)$. Use the forward Euler method to transcribe the specified continuous-time filter into a discrete-time representation for real-time implementation on the microcontroller. In Figure 10, the blue portions are software and all else is hardware; we connected two sets of pins with wire as shown. Note that DAC input signal `dacU` (in V) is continuously fed from the pre-computed 1000-element array `U` described previously, whereas the ADC output signal `adcY` (in V) feeds the initial 1000 measurements of $y(t)$ into array `Y`. Display `dacU` and `adcY` in the Expressions window with continuous refresh to verify that operation is sustained indefinitely. Use the graph tool to compare the arrays `U` and `Y`.

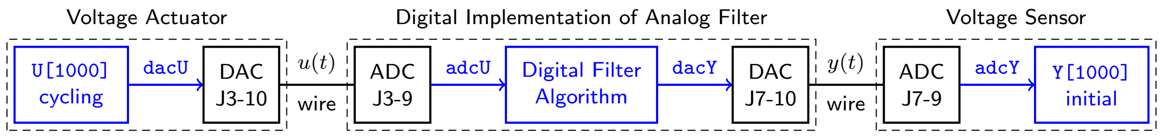


Figure 10: A digital implementation of an analog filter.

4.3 Deliverables

To be eligible for full credit, do the following:

1. Upload narrated videos showing completion of §4.2.1 and §4.2.2.
2. Upload your `main.c` file for §4.2.2.