ECE 362 Lab Experiment 6: Analog Input/Output

Home About References Lecture Homework Labs

Introduction

Interfacing a microcontroller with real-world devices often means working with analog voltages that are not constrained by digital logic's high/low dichotomy. In this lab, you will gain experience using the digital-to-analog converter and the analog-to-digital converter built in to your microcontroller. You will also practice writing code to manipulate the converted values as well as display them on an output device.

Direct Memory Access (DMA) allows you to automatically transfer a region of memory to a peripheral, a different location in memory, or any memory mapped region of the device. In this lab, you will also gain experience using DMA.

Instructional Objectives

- To understand the concept of analog-to-digital conversion
- To understand the concept of digital-to-analog conversion

- To learn how to use STM32F0 peripherals to measure and process analog information
- to use DMA to automatically transfer data to and from peripherals

Table of Contents

2.5.2	set_freq()	5
2.5.3	setup_dac()	10
2.5.4	init_tim7()	5
2.5.5	Timer 7 ISR	10
3	Submit your postlab results	*
	Total:	100

* All the points for this lab depend on proper completion of and submission of your post-lab results.

When you are ready for your lab evaluation, review this checklist.

Step 0: Prelab Exercises:

- Be familiar with lectures up to and including (15) Analog-to-Digital Conversion.
- Scan Chapters 13 and 14 of the STM32F0 Family Reference Manual to become familiar with the DAC and ADC subsystems.
- Read Chapter 11 of the Family Reference Manual to understand how to configure the proper DMA channel.
- Read this entire lab document.
- Leave the devices and wiring you did for Lab 5 in place and add the things described in section 1.7.
- After doing the previous steps, including reading the entire lab document, then do the prelab exercises and submit them **before** attempting the lab experiment.

Step 1: Background

1.1 Analog-to-Digital Conversion

Audio signals, analog sensor input waveforms, or input waveforms from certain types of input devices (sliders, rotary encoders, etc.) are all examples of analog signals which a microcontroller may need to process. In order to operate on these signals, a mechanism is needed to convert these analog signals into the digital domain; this process is known as analog-to-digital conversion.

In analog-to-digital conversion, an analog signal is read by a circuit known as an analog-to-digital converter, or ADC. The ADC takes an analog signal as input and outputs a quantized digital signal which is directly proportional to the analog input. In an n-bit analog-to-digital converter, the voltage domain of the digital logic family is divided into 2^n equal levels. A simple 3-bit (8-level) analog-to-digital quantization scheme is shown in figure 1, below:

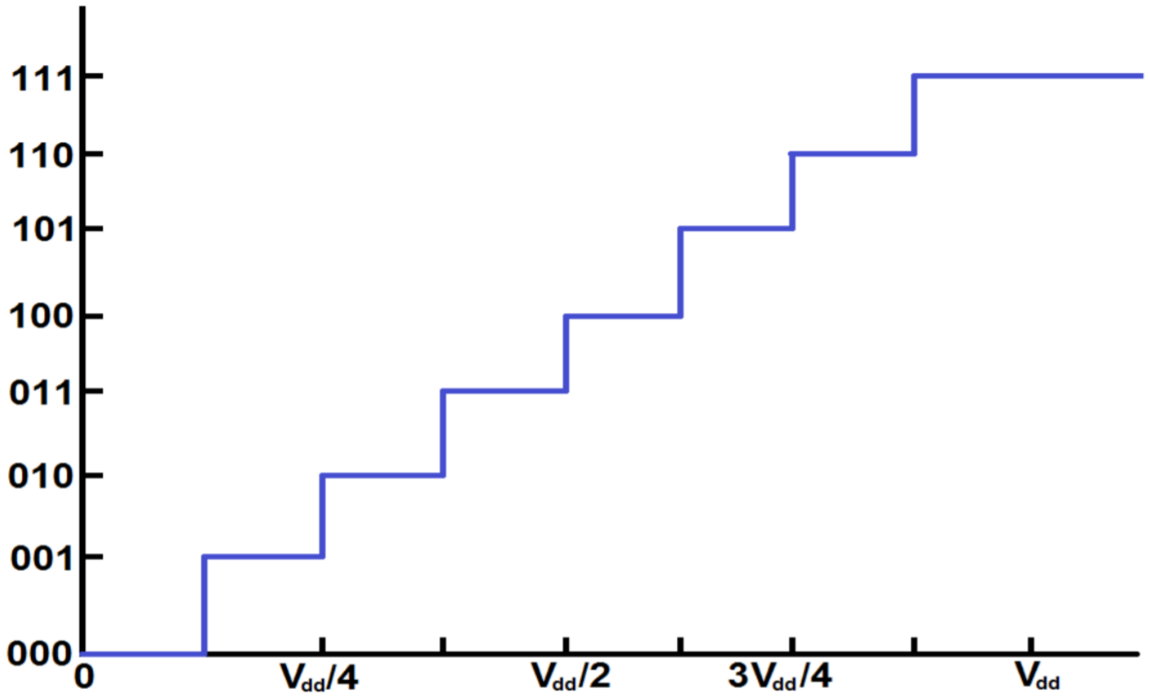

Figure 1: 3-bit analog-to-digital quantization scheme

1.2 Configuring the STM32F0 ADC

The STM32F0 microcontroller has an integrated 12-bit analog-to-digital converter. The converter has 19 channels, allowing for readings from up to 16 independent external pins and three different internal sources. The ADC is capable of converting signals in the 0-3.6V range, although it is limited by the reference voltage on the development board (approximately 2.95 V). It is able to capture and convert a new sample every 1μs. Many other ADC features are described in greater detail in the microcontroller documentation.

Before any ADC configuration can be done, a few initializations must be done. First, the ADC clock must be enabled using reset clock control (RCC), as was done with other peripherals in previous experiments. Which bit in which register? It is up to you to consult the documentation for the RCC registers to determine the bit to set.

Next, the inputs which are to be configured as analog I/O must be configured for analog mode in the corresponding GPIOx_MODER register. This is described in further detail in section 9.3.2, "I/O pin alternate function multiplexer and mapping", of the STM32F0 family reference manual.

Prior to performing any configuration of the ADC, the peripheral must first be enabled. ADC activation must take place before any other ADC configurations are performed; failure to follow this order may result in the ADC entering an unknown state, at which point the ADC must be disabled and restarted. Enabling the ADC requires writing '1' to the ADC enable bit, ADEN, in the ADC control register, ADC_CR. Once this is done, software must wait until the ADC is in a ready state, indicated by the bit ADRDY (located in the ADC status register, ADC_ISR) being read as '1' to software.

For the purposes of this experiment, we are interested in configuring the microcontroller ADC for single conversion, software-driven mode. This means that the ADC will perform a single conversion, initiated in software. The ADC will then await the next software trigger before another conversion is initiated. This is the default operating mode of the ADC, and no special configurations will be needed within the ADC peripheral. In the event that special operating conditions (DMA, continuous conversion mode, etc.) need to be performed by the ADC, operating modes can be specified via the ADC configuration registers: ADC_CFGR1 and ADC_CFGR2.

When performing a conversion, the ADC will read and convert values on any active channels. By default, all ADC channels are disabled. Prior to use, it is important to enable those channels on which conversions will be performed. This is done through the ADC channel selection register, ADC_CHSELR.

1.3 Using the STM32F0 ADC

Once the ADC has been properly enabled and configured, initiating an analog-to-digital conversion is as simple as writing a '1' to the ADSTART bit in the ADC_CR register. Writing a '1' to this bit instructs the converter to automatically begin conversion of all ADC channels. This bit will automatically be cleared by hardware once the conversion has completed. Converted ADC data is ready to be processed by the core once the end of conversion bit, EOC, is activated in the ADC_ISR register.

In some situations, particularly when disabling the ADC, it may be desirable to stop a conversion which is already in process. This can be done by writing a '1' to bit ADSTP of the ADC_CR register. Doing so will stop the ongoing conversion process, discarding any data, and will automatically clear the ADSTART bit.

Once an analog-to-digital conversion has been completed, the converted data can be read from the ADC data register, ADC_DR. Data will be right-aligned or left-aligned, depending on configuration settings (right-aligned by default).

It is worth reiterating that, when an external pin is configured for analog operation, it puts delicate internal circuitry at risk. If you expose an external pin to greater than 4.0 V even for a fraction of a second, it will permanently, irreparably damage either the pin or the entire microcontroller. Disconnect your microcontroller from power when you are wiring circuits. **Don't connect anything to the 5 V power pin of the microcontroller.** Check your circuitry before applying power to your development board.

In this lab, you will be using pins on Port A for analog operations. This means you will modify the GPIOA_MODER configuration. Remember that, if you modify the configuration for pins PA13 or PA14, you will lose the ability to debug or even re-program the microcontroller. Double-check your MODER updates to make sure they will not change pins 13 or 14.

When you misconfigure GPIO Port A, remember that you can restore the ability to use the debug/programming interface by:

1. repairing your program,
2. pressing and holding the reset (SW1) button,
3. pressing "Run" on SystemWorkbench to reprogram the microcontroller

Depending on what kind of mistake you made, you might have to hold down the reset button for only one second after pressing "Run". You may also need to reprogram the microcontroller twice before it works again. Be patient. Do not give up. Ask a TA for help with this process.

1.4 Digital-to-Analog Conversion

A digital-to-analog converter (DAC) is similar to analog-to-digital conversion in reverse. An n-bit digital quantity is written to a data register, the conversion is *triggered*, and the *quantized* analog value appears on an output pin. One significant difference is that, since the conversion mechanism uses only a resistor network, the conversion is nearly instant. There is no need to continually check a DAC to find out if the conversion is complete.

Like the ADC input, the DAC output cannot represent all values. For an n-bit DAC, there are 2^n distinct values. A digital value of n zeros usually represents an analog conversion of 0 V. A digital value of all ones usually represents an analog conversion of the reference voltage (V_{REF}). If treated as an n-bit integer, each increment results in an approximately $V_{REF} / 2^n$ V increase in output voltage.

1.5 Configuring the STM32F0 DAC

The STM32F091 microcontroller has an integrated 12-bit DAC with two output channels. Samples can be written to the DAC, converted, and alternately written to either channel to produce stereo audio. For this lab experiment, however, we will use only the first channel. (Read section 14.4 of the Family Reference Manual if you want to learn more about "Dual-Mode" output control.)

To configure and use the DAC, the RCC clock must first be enabled. The pin to use for analog output must be configured for analog mode by setting the appropriate value in its GPIOx_MODER register field. Next, DAC channel 1 must be enabled by setting the DAC_CR_EN1 bit in the DAC_CR register. Finally, the type of trigger to use to initiate a conversion must be selected. For this lab experiment, we'll use a software trigger.

1.6 Using the STM32F0 DAC

Because it does not require polling to determine the completion of a conversion, use of the DAC is less complicated than the ADC. All that is necessary to produce an analog value on an DAC output when configured for software trigger mode, is to write a value into the appropriate Data Holding Register (DHR) and initiate the conversion by writing a '1' to the trigger bit of the DAC_SWTRIGR register. The DAC is ready to receive another sample in the next APB clock cycle.

Like the ADC, the DAC has variable bit sizes as well as left-aligned and right-aligned data handling capabilities, but they do not need to be selected in advance. If you wish to output an 12-bit left-aligned data value, you write it to the DAC_DHR12L1 register before triggering the conversion. If it is a 12-bit *right-aligned* value write it to the DAC_DHR12R1 register instead. The only other bit size available is an 8-bit right-aligned register, DAC_DHR8R1. There is no left-aligned 8-bit data holding register, nor are there any other sizes.

It is important to understand that writing to any of the channel 1 holding registers (DAC_DHR12L1, DAC_DHR8R1, DAC_DHR12R1) effectively creates a holding register value equivalently. For example, doing any of the following three writes

- Writing `0x0000`**`d700`** to DAC_DHR12L1 or...
- Writing `0x0000`**`0d70`** to DAC_DHR12R1 or...
- Writing `0x000000`**`d7`** to DAC_DHR8R1

will deposit the 12-bit value 0xd70 into the 12-bit hardware register waiting to be converted by the DAC.

1.7 Wiring for this lab

For this lab, you will use a USB oscilloscope (e.g., the Digilent Analog Discovery 2) to observe the output of the DAC.

You will use potentiometers to connected between 3V and Gnd with the center tap acting as a voltage divider. The center tap of the potentiometer will be connected to analog inputs. As long as you are certain to connect the potentiometers only between **3V and Gnd**, no damage to the analog inputs is possible.

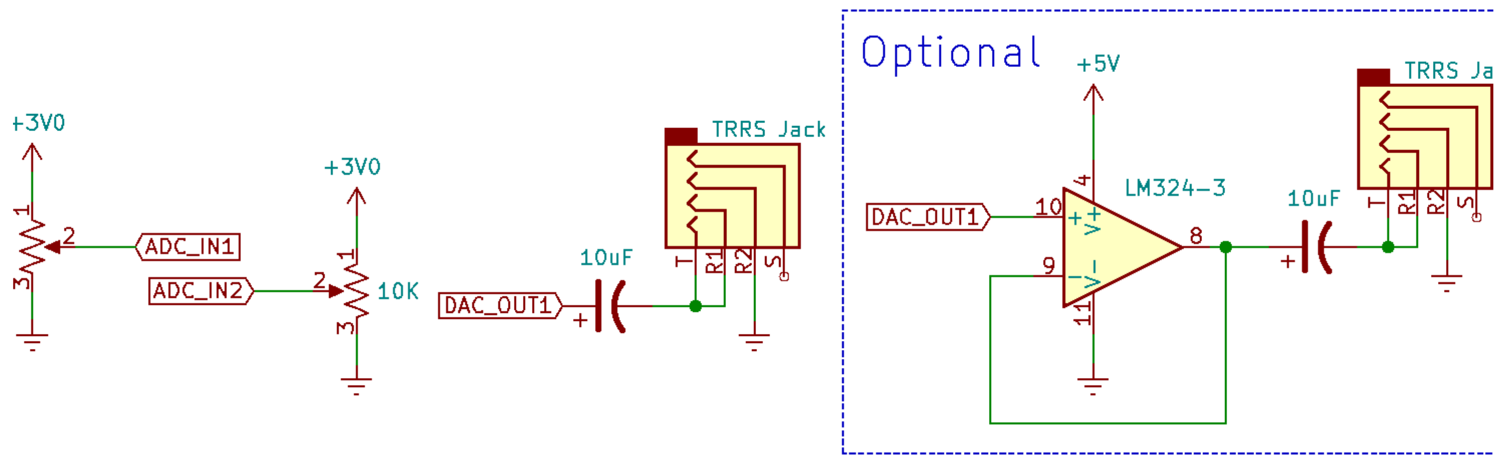

Figure 2: Potentiometer and headphone jack

1.8 Direct Memory Access

The STM32F09x family of microcontrollers have 2 Direct Memeory Access (DMA) controllers, each with 7 and 5 channels respectively that can autonomously move one or more words of various sizes between memory and peripherals. Once values are written to a DMA channel’s configuration registers, the CPU can be used for other purposes while the transfers operations continue. Each incremental DMA operation can be triggered by a peripheral’s readiness to be written to or read from. Some microcontrollers allow DMA channels to be arbitrarily associated with peripheral devices, but the STM32F0 requires that each peripheral be used with only the specific DMA channel that it is wired to. Table 32 and 33 of the Family Reference Manual shows the peripheral-to-DMA channel mapping.

Under normal circumstances, when moving a number of words of memory (indicated by the CNDTR register) to or from a single peripheral address (in the CPAR register), the register that points to the memory address (CMAR) will be incremented by the size of the word. This increment is optional and must be set with the MINC bit of the channel’s CCR register. A DMA request is activated by setting the EN bit of the channel’s CCR register.

Once the requested number of words has been moved, the EN bit for that channel is cleared. Each DMA channel can be configured for circular operation by setting the CIRC bit in the CCR register. In circular mode, the DMA channel moves the specified number of words (CNDTR) between the addresses in CMAR and CPAR, incrementing these registers as requested, but when finished, instead of disabling the EN bit, the CMAR, CPAR, and CNDTR registers are reset to their original values, and the channel request is restarted. A circular request is one that continually moves words between a region of memory and a peripheral register.

Each DMA channel can give updates on its progress by invoking interrupts. Three interrupts are possible. First, the Transfer Complete interrupt indicates that all words of the DMA request have been moved. Second, the Half Transfer interrupt indicates that half of the words of the DMA request have been moved. (This is useful when using a circular buffer to indicate that the first half of the buffer is ready.) Finally, a Transfer Error interrupt indicates that an access was attempted to a memory location that was not permitted for the type of operation requested.

1.9 Debouncing a Keypad

In lab experiment 5, you learned how to read and debounce a keypad matrix using a history mechanism. In this lab experiment, you will do that again (in C), and fix the problems with multiple interrupt sources, as well as do translation of keys.

Briefly, the history method keeps track of the last eight readings from each button on the keypad in a byte of memory. When the byte is `00000000`, it has been idle for a long time. When the byte is `00000001`, it has been read as pressed. When the byte is something like `10101011`, it was pressed some time ago, and it is still bouncing. When the byte is `11111111`, it has been pressed for a long time and is no longer bouncing. Similar categorizations can be made for buttons when they are released. In this lab, we will care only about the instant a button is first pressed.

In lab experiment 5, there was discussion about how the history values for the keypad were sampled after certain interrupts, and extra interrupts would corrupt the readings. To overcome that, the subroutine (the scanning ISR) that reads the columns from the keypad, and updates the history values should also recognize the `00000001` (`0x01`) in the history byte and should make a note in a separate data structure to indicate that a key was pressed. To build this data structure, you will use a small *circular queue* to indicate the most recently pressed buttons. For the purposes of this lab experiment, a two-element queue will be enough. Since the ISR is always shifting new values into the history bytes, it will never have successive 0x01 values in a history byte. Therefore, it will not insert the button event twice into the queue.

The subroutine to wait on keys and return their value will sleep waiting on interrupts and check the queue every time it wakes up. If there is something in the queue to indicate a button press, it will note what it is, translate it into the ASCII character representation, clear the queue entry, and advance to watching the next entry in the queue. By clearing each queue item after it is read, subsequent reads of the queue can recognize the presence of a new button press by looking for a *non-empty* queue entry.

In this lab, you will be using pins on Port C for the keypad. This means you will modify the GPIOC_MODER and GPIOC_AFR configuration. Remember that, if you modify the configuration for pin PC12, you will lose the ability to debug or even re-program the microcontroller. Double-check your MODER updates to make sure they will not change pin 12.

When you misconfigure GPIO Port C, remember that you can restore the ability to use the debug/programming interface by:

1. repairing your program,
2. pressing and holding the reset (SW1) button,
3. pressing "Run" on SystemWorkbench to reprogram the microcontroller

Depending on what kind of mistake you made, you might have to hold down the reset button for only one second after pressing "Run". You may also need to reprogram the microcontroller twice before it works again. Be patient. Do not give up. Ask a TA for help with this process.

2.0 Experiment

Create a project in SystemWorkbench called "lab6", and replace the automatically-generated main.c file with the main.c skeleton file provided for you.

At some point, a pre-compiled object file will be prepared that contains test functions to automatically test the subroutines you write for this lab. It is called autotest.o. To incorporate it into your System Workbench project, follow the instructions in the ECE 362 References page for Adding a pre-compiled module to a project in System Workbench. Errors will be reported on the serial port display in exactly the same manner that you have seen in previous lab experiments.

2.1 Configure GPIO

This is similar to the GPIO configuration you've done in previous labs, but now we will do it using the CMSIS constants in C. As you will see, this is much more readable.

Fill out the subroutine **`enable_ports`** with the following configurations:

- Enables the RCC clock to GPIOB and GPIOC without affecting any other RCC clock settings for other peripherals
- Configures pins PB0 – PB10 to be outputs
- Configures pins PC4 – PC7 to be outputs
- Configures pins PC0 – PC3 to be inputs
- Configures pins PC0 – PC3 to be internally *pulled low*

As usual, you should not alter the configuration for any other pins. For instance, if you were to improperly alter the MODER setting for PC12, the serial port would stop working.

2.2 Configure DMA transfers

Here, you will set up a circular DMA transfer to the GPIOB peripheral's ODR register. The trigger will be triggered whenever TIM2 has an update event. As soon as the timer reaches the ARR value, the DMA channel will be triggered to write a new value into the data register.

Remember that TIM2 works with one particular channel of the DMA1 controller. You will use that default channel.

2.2.1 setup_dma()

Write a C subroutine named **`setup_dma()`** that does the following things:

- Enables the RCC clock to the DMA controller and configures the following channel parameters:
  - **Turn off the enable bit for the channel.**
  - Set CPAR to the address of the GPIOB_ODR register.
  - Set CMAR to the msg array base address
  - Set CNDTR to 8
  - Set the DIRection for copying from-memory-to-peripheral.
  - Set the MINC to increment the CMAR for every transfer.
  - Set the memory datum size to 16-bit.
  - Set the peripheral datum size to 16-bit.
  - Set the channel for CIRCular operation.

Write one more C subroutine named **`enable_dma()`** that does only the following operation:

- Enable the channel.

There is a lot of variation in students' work, and it's difficult for autotest to check all possible cases once a DMA channel is enabled and running. That's why we keep it separate.

2.2.2 init_tim2()

Fill out the subroutine **`init_tim2()`** to enable TIM2's clock in RCC and trigger a DMA request at a rate of 1 kHz.

Once you implement these subroutines and uncomment the calls for them as well as the other SPI calls in **`main()`**, you should see ECE 362 in the 7-seg LED array. The interesting thing to notice here is that the circular DMA operation continually and automatically copies the buffer array **`msg`** to the display. This creates the illusion that the memory is *mapped* into the display.

2.2.3 append_display(char val)

Fill out the subroutine **`append_display()`**, which updates `msg` to display a given font character on the rightmost 7-segment display. Existing values are shifted to the left.

Remember that each entry in `msg` contains all the bits that need to be written to `GPIOB_ODR` every millisecond.

2.3 Debouncing a Keypad

Similar to the previous lab, we will set up a software-debounced keypad. This time we will employ a queue of length 2 to parse key presses. And of course, this time it is in C.

2.3.1 drive_column()

Write a C subroutine named **`drive_column()`** that does exactly what the similarly-named subroutine did in lab 5. Namely,

- Use the column number (indexed from 0) in the first arguent and use it to turn on one of PC7, PC6, PC5, or PC4. E.g., if the argument is 2, then PC6 should be set high, and the other three outputs should be set low.

2.3.2 read_rows()

Write a C subroutine named **`read_rows()`** that does exactly what the similarly-named subroutine did in lab 5. Namely,

- Read the 4-bit value from PC[3:0] and return it.

2.3.3 push_queue()

Write a C subroutine named **`push_queue()`** that accepts a single argument that represents the entry in the history array for a button that was just pressed and puts into the **`queue`** array at the position **`qin`**. For instance, the 'D' button is element number 0 in the history array, the 'C' button is element number 1, and so on (check the keymap array for the exact mapping). Since the entry to push may be zero, and since you must be able to distinguish a valid entry from an empty (zero) entry, set the most significant bit in the queue entry. Also the parameter is an int and **`queue`** stores 8 bits (and the 8th has special meaning), so only capture the lowest 7 (or 8) bits. Finally, update the **`qin`** variable to point to the other queue element. Basically:

- AND the value 0x7f with the parameter
- OR the value 0x80 with the parameter.
- Store the value into **`queue[qin]`**
- Toggle the value of **`qin`** to refer to the other element of the queue.

2.3.4 pop_queue()

Write a C subroutine named **`pop_queue()`** that returns the value in the **`queue`** at position **`qout`**. You must remove the MSB we set to differentiate it from zeros. You must also toggle the **`qout`** value to point to the other queue element. Basically:

- Store **`queue[qout]`** in a temporary variable.
- AND the value 0x7f with the temporary variable.
- Set **`queue[qout]`** to zero.
- Toggle the value of **`qout`** to refer to the other element of the queue.
- Return the value of the temporary variable

2.3.5 update_history(int cols, int rows)

Write a C subroutine named **`update_history()`** that does exactly what the similarly-named subroutine did in lab 5. Namely, update the four **`history`** elements for the column specified by the first parameter by shifting them one bit left and ORing in the values from the row passed in the second parameter.

Add the following functionality: If any of the four history values is updated to a 0x1, call **`push_queue()`** with the element's index in the **`history`** array.

2.3.6 Timer 6 ISR

Create a C subroutine to act as the ISR for Timer 6. It should do what the the Timer 7 handler did in lab 5.

2.3.7 init_tim6()

Write a C subroutine named **`init_tim6()`** that does exactly what `init_tim7()` did in lab 5.

2.3.8 get_keypress()

Write a C subroutine named **`get_keypress()`** that waits for an interrupt (any interrupt) by using an inline assembly statement,

```
            asm volatile("wfi");
```

Each time, upon waking after an interrupt, the subroutine should check the **`queue[qout]`** element. If it is zero, it should continue waiting.

If the queue element is non-zero, call **`pop_queue()`** and use the keymap array to translate the returned value. Return the translated value.

2.4 Reading an Analog Voltage

2.4.1 setup_adc_dma()

Write a C subroutine named **`setup_adc_dma()`** that does the following things:

- Enables the RCC clock to the DMA controller and configures the following channel parameters:
  - **Turn off the enable bit for the channel**
  - Set CPAR to the address of the ADC1->DR register.
  - Set CMAR to the volume valriable's address
  - Set CNDTR to 1
  - Set the DIRection for copying from-peripheral-to-memory.
  - Set the MINC and PINC to not increment.
  - Set the memory datum size to 16-bit.
  - Set the peripheral datum size to 16-bit.
  - Set the channel for CIRCular operation.

Like before, write one more C subroutine named **`enable_adc_dma()`** that does only the following operation:

- Enable the channel.

2.4.2 setup_adc()

Write a C subroutine named **`setup_adc()`** that initializes the ADC peripheral and configures the pin associated with ADC_IN1 to be analog mode. To write this subroutine, you should carefully examine the example on page 15 of lecture 15 on Analog-to-Digital Conversion. You should also consider examples A.7.2 and A.7.4 of the Family Reference Manual appendix. The subroutine should do the following:

- Enable the clock to GPIO Port A
- Set the configuration for analog operation only for the appropriate pins
- Enable the clock to the ADC peripheral
- Turn on the "high-speed internal" 14 MHz clock (HSI14)
- Wait for the 14 MHz clock to be ready
- Enable the ADC by setting the ADEN bit in the CR register
- Wait for the ADC to be ready
- Set the ADC to continous mode
- Set the ADC to trigger a DMA request (remember to set the DMACFG bit as well for continuous mode DMA).
- Select the corresponding channel for ADC_IN1
- Wait for the ADC to be ready
- Start the ADC

Your subroutine can use the default configuration of right-aligned 12-bit conversion. Really, not much more need be done than is shown in the example on page 15 of the lecture. We just want to do the minimum needed to make the ADC work.

2.5 DAC Initialization

2.5.1 Initialize a Wavetable

Write a subroutine named **`init_wavetable()`** that computes an N-entry sine wave and writes it into the array named **`wavetable[]`**. Each entry of the array is a signed 16-bit integer which can represent values from -32768 to 32767. For reasons of symmetry, your sine wave values should range from -32767 to 32767. To compute this, use the following code:

```
for(int i=0; i < N; i++)
    wavetable[i] = 32767 * sin(2 * M_PI * i / N);
```

The STM32F091 has no floating-point hardware, so this will be computed with slow, emulated mechanisms. It is still fast enough that it can be done at program startup without being noticed. You must, however, avoid floating-point calculations in time-critical situations such as interrupt handlers. That is the entire idea behind fixed-point calculation described in Lecture 13 on Polyphonic Sound Generation.

2.5.2 set_freq()

Write the subrouting **`set_freq()`**. It should set the global variable **`step`** to an appropriate value given the passed frequency. (See the notes on sound generation.)

2.5.3 setup_dac()

Write a C subroutine named **`setup_dac()`** that initializes the DAC peripheral and configures the output pin for DAC_OUT1 for analog operation. Consider the example on page 22 of lecture 12 on Digital-to-Analog Conversion as well as the examples in Appendix A.8. Your subroutine should do the following:

- Enable the clock to GPIO Port A

- Change the configuration only for DAC_OUT1 for analog operation
- Enable the RCC clock for the DAC
- Select a software trigger for the DAC
- Enable the trigger for the DAC
- Enable the DAC

You don't need to disable the buffer-off (BOFF) bit or first disable the DAC enable before you set values. Just do the minimum necessary steps to enable the DAC.

2.5.4 init_tim7()

A symbolic constant, **`RATE`** is #defined in main.c. Write a subroutine **`init_tim7()`** that initializes timer 7 so that it's interrupt is called at RATE times per second (20000 for this exercise). It would be best if the code used RATE to calculate the values to use to set up the timer so that you can change RATE later.

2.5.5 TIM7 ISR

Write the ISR for TIM7. **It should trigger the DAC immediately upon entering.** It should then do the following to setup the next value to stage the next value for the DAC:

- Remember: Trigger the DAC first.
- Increment **`offset`** by the **`step`** amount
- If the integer portion of **`offset`** is greater than N, find the fixed-point representation of N and subtract it from the offset. (See the notes on sound generation.)
- Get the sample from the **`wavetable`** at index of the integer portion of offset.
- Multiply the sample by **`volume`**.

- Shift the sample to the right by 16.
- Add 2048 to the sample.
- Write the sample to the 12-bit right-aligned data holding register of the DAC.

The **`volume`** variable is a 12-bit number, so it has a maximum value of 4095. If this is multiplied by a sample the product will range between +/-134180865 (which is +/- 2^{27}). When this value is shifted right by 16, it will result in a signed 12-bit value. Adding 2047 to it will put it in the range 0 - 4095, which is appropriate for DAC output.

3 Submit your postlab results

Once your lab experiment works to your satisfaction, and autotest tells you how many parts work correctly on the serial port, it will create a completion code similar to that of some previous labs. This code is a cryptographic confirmation of your completion of the lab experiment as well as your identity and your score. (Remember to change the "login" variable from "xyz" to your login.) It is normal for the code to change every time you run it. Choose one of the codes and carefully enter all 32 characters into the post-lab submission for this lab experiment.

For this lab, and most to follow, you must also submit the program that you wrote so that it can be checked by the course staff. Either upload the file or copy it from SystemWorkbench and paste it into the text box. Make sure that your entire program is shown there.

Lab Evaluation Checklist

Normally, we'll have a checklist of things that you should review before going into your evaluation. There will be no evaluation for this lab experiment.

- [ ] You did not share your microcontroller with anyone else, right? We will know if you did.