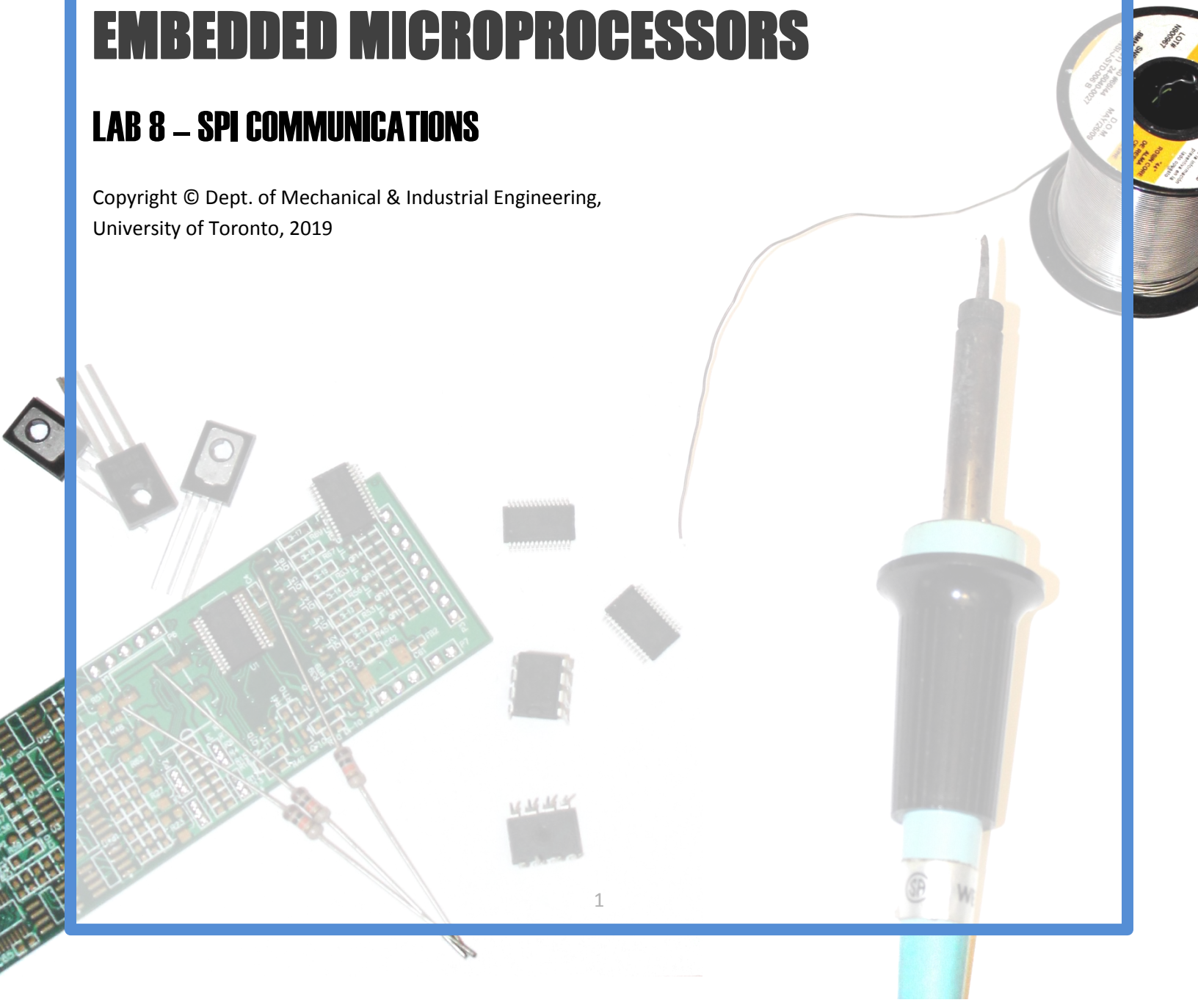




MIE438 – MICROCONTROLLERS & EMBEDDED MICROPROCESSORS

LAB 8 – SPI COMMUNICATIONS

Copyright © Dept. of Mechanical & Industrial Engineering,
University of Toronto, 2019



Pre-Reading: Please read this lab fully before attending!

Learning Objectives

You have now completed a basic communications protocols lab wherein you worked with an ad-hoc/software I2C variant. You have also seen and used many of the devices integrated onto the Dragon12-Plus2 EVB. Our next goal in the lab series is to broaden the scope of embedded-specific topics, design patterns, and peripherals that you have experience with. We will also examine complete embedded systems at the design level for conceptual patterns which you may reuse in your own designs. Lastly, we also want you to build experience with SPI as a second major protocol. In particular, we want you to see how one would use integrated hardware to implement SPI communication, as compared to the software-based protocol from the last lab. In this specific lab you will:

- Learn and implement the basic Serial-Peripheral Interface (SPI) protocol using integrated peripheral hardware, including setup and send/receive;
- Learn and implement an application layer for a specific external IC (DAC) over SPI;
- Observe actual SPI waveforms and practice digital debugging techniques using specialized hardware (logic analyzer / protocol analyzer);
- Implement one common design pattern for implementing software-driven high-speed SPI communication with high data throughput;
- Understand alternate design patterns using hardware-focused approaches (DMA, application-specific DDS);
- Learn about analog output and digital-to-analog conversion, and implement analog output functionality using a dedicated DAC chip.

Introduction

In this lab, we will make use of the EVB's dedicated digital-to-analog converter (DAC) chip to solve a meaningful problem. Our goal in this regard is to build a function generator which functions similarly to the desktop function generators available in the lab. It is a very useful piece of test equipment, and this application example will show that it is one which you could build and improve with very little hardware. Some key words describing our problem would include 'low-cost,' 'minimal size,' and 'minimal parts count.'

Engineering Specification

To make this a more formal example, we begin by crafting a specification for the problem at hand. In a client-based setting, this would of course be created in collaboration with the client, to address their specific needs. As we are the ‘client’ in this case, we can set our own target. Based on an understanding of the overall processing speed of the board available, and an assumed application in the audio band, the parameters below were selected. As a note, this is somewhat poor practice – forcing a design to fit existing embedded hardware is less preferred compared to iterating on the hardware or allowing the overall design to lead hardware selection. That being said, a possible specification is as follows:

$$\begin{aligned}V_{\text{OFFSET}} &= 2.5\text{ V (fixed)} \\V_{\text{AMPLITUDE}} &= 0\text{ V to } 5.0\text{ V (peak-to-peak, adjustable)} \\frequency &= 20\text{ Hz ... } 20\text{ KHz (adjustable)} \\Function &= \textit{Sine} \text{ (fixed)}\end{aligned}$$

To summarize, the fundamental goal of this mini-project is to use the available hardware (a given digital-to-analog converter, DAC, and a given microcontroller) to generate a sine wave with some adjustable parameters.

The offset is set to half of the microcontroller supply voltage ($5.0/2 = 2.5\text{V}$), which will give us the largest possible output amplitude range. In this manner, we can produce up to a 5.0V peak-to-peak sine wave by varying the DAC output from 0V to 5V . The DC offset could be easily removed by adding a capacitor to the output, if desired. Furthermore, the DAC (as we will see) has a second output channel. This could be used to add in an adjustable offset. However, this feature would require one op-amp (as a summing amplifier) and thus has been omitted for simplicity. If you plan to build this application example as a personal project, this should be one of your first modifications to the design.

Lastly, the frequency is selected based on the common audio range (20 Hz to 20 KHz is the range of human hearing, and a standard range for testing audio equipment). It is also convenient, in that the absolute maximum frequency the final design can produce is actually quite close to 20 KHz – the limiting factor is the microcontroller clock in this case.

The software design pattern we will use is borrowed from the concept of **direct digital synthesis**. This approach is used by the popular family of devices from Analog Devices (AD983X series and similar) to produce sine waves, albeit with improved overall capabilities. If you have a similar application but desire higher quality of output, consider a dedicated chip such as the above.

Design Pattern – Software/DAC-based Function Generator

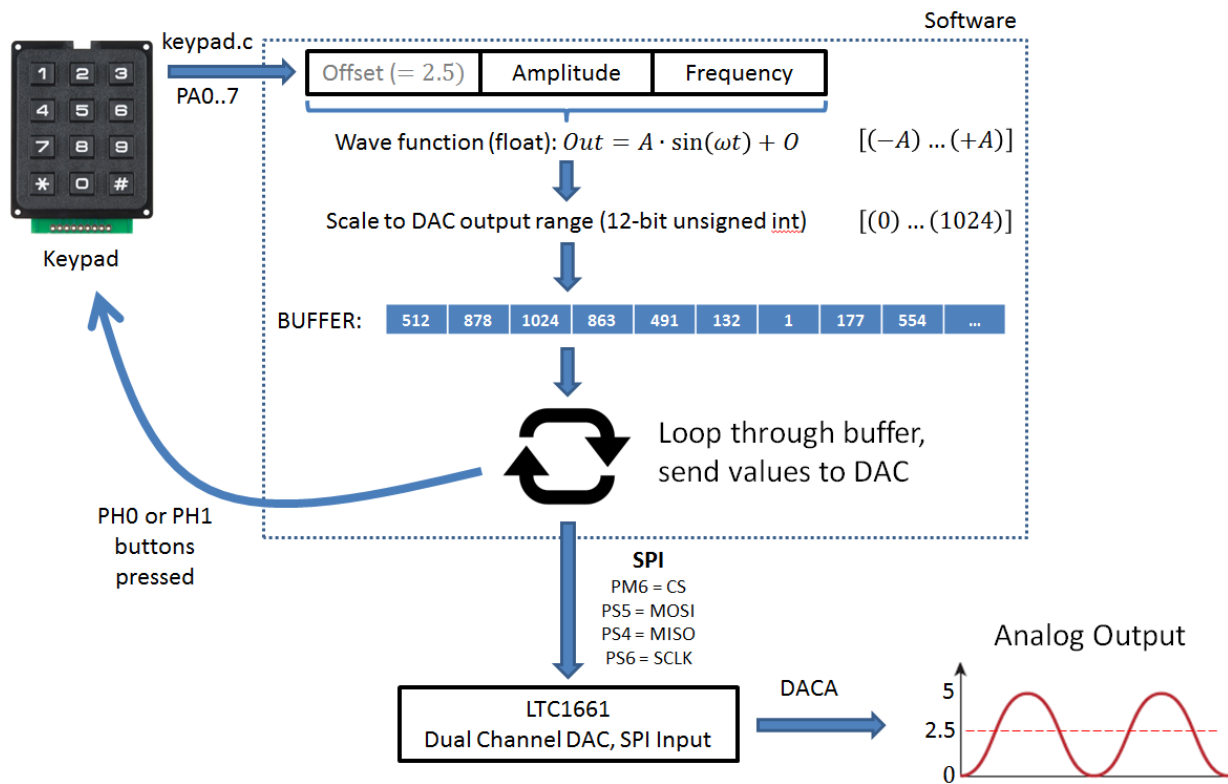


Figure 1 – Overview of software/DAC function generator for Lab 8

The ‘proposed solution’ to this problem is shown at a high level in Figure 1. It is useful to get into the habit of planning your system from the ‘top-down’ – start with a high-level diagram such as the above, showing major signals, user functions, sensors, communication paths and protocols, and other hardware. One can see that this system makes use of the Dragon12-Plus2’s existing hardware paths. As discussed, the second channel of the DAC (output DACB) could be utilized to add a controllable DC offset to the signal via a summing amplifier circuit.

To summarize hardware usage, one can see that the major output device in the system is the DAC (LT1661), which connects to the microcontroller via. SPI. A short loop supplies the DAC with a continuous stream of values read from a memory buffer. The major input devices are the numeric keypad and PORTH buttons, which use general-purpose I/O lines.

One can see that the design specifies a **low utilization** of available I/O pins (primarily due to the DAC’s SPI interface taking only 4 wires). However, it specifies a **very high utilization** of internal CPU resources, including cycles and internal memory. If this were to be a ready-to-manufacture product, the CPU selection would be a poor fit. A faster but ‘lighter’ chip (with fewer I/O but possibly more memory) would be a better choice.

The system can be further improved by selecting a microcontroller with hardware features suited to the task at hand. Figure 2 below shows a conceptual design for a more fully featured system with room for further functional expansion:

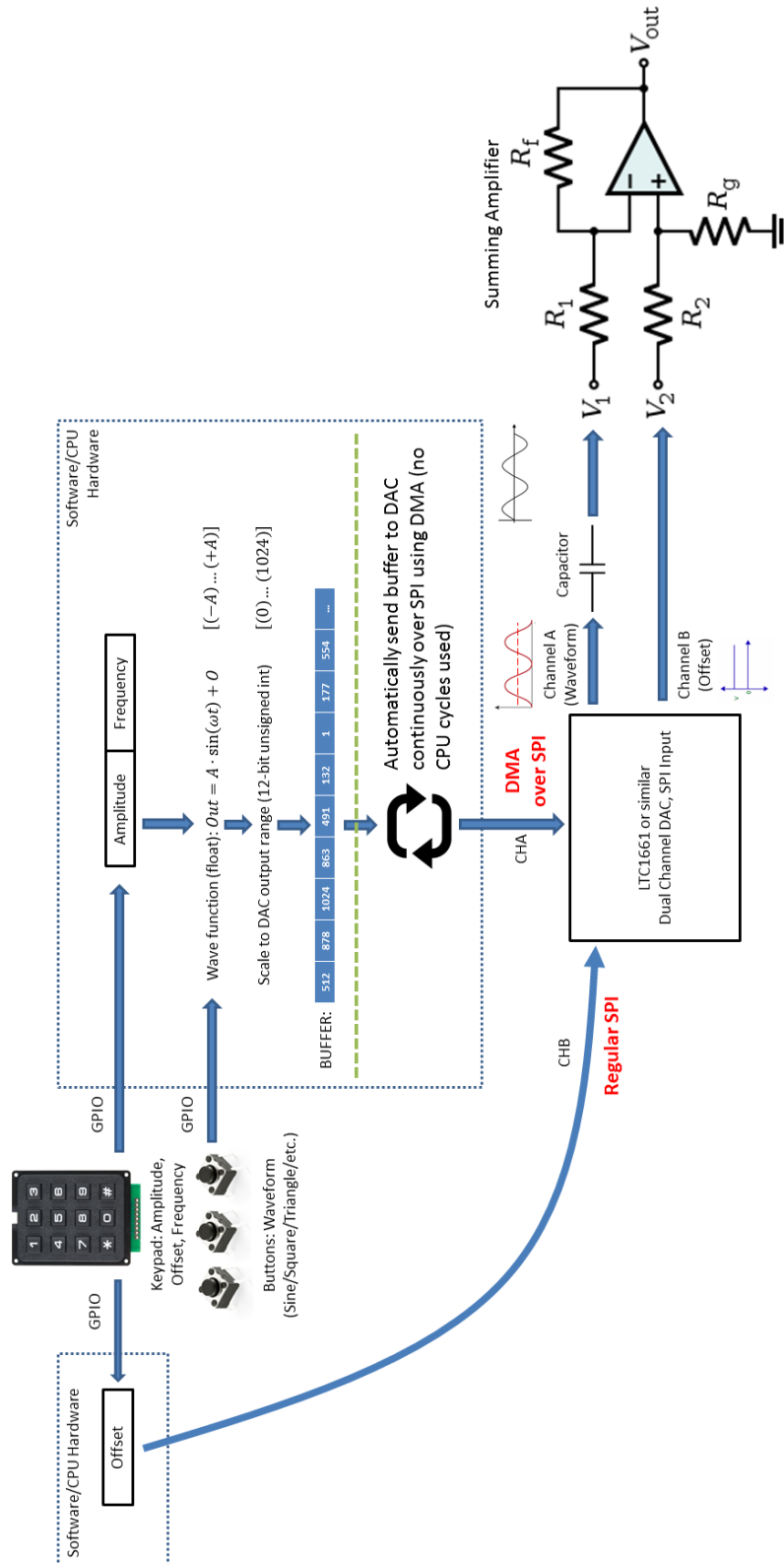


Figure 2 – Conceptual design for a fully featured function generator

The design shown in Figure 2 utilizes DMA (Direct Memory Access) to set up a continual transfer from the memory buffer (which stores the waveform) over SPI to the DAC. This transfer does not use CPU cycles beyond initial setup – this is the hardware-driven benefit of using DMA. The CPU is thus free to implement other functionality. Note that the 68HCS12 does not support DMA.

This design also shows how a controllable DC offset could be added to the signal. The second channel (B) of the DAC is used to create a controllable DC level. This level is fed to a summing amplifier, along with the waveform (with its own offset removed / set to zero). The summation of the two signals is a waveform with software-controllable amplitude, frequency, and DC offset.

The remaining bottleneck in the design is the inherent speed limit of the SPI protocol and the DAC itself and/or the DMA hardware on the CPU. If one wishes to push this design further (say into MHz-range frequencies while maintaining good fidelity of the output waveform), additional changes would be necessary. Options would include:

- Moving the DMA buffer out of main memory and onto an external memory chip which communicates only with the DAC. In this concept, the CPU would set up and mediate the DMA transfer, but would not directly move the data itself. New waveforms could still be loaded into memory chip by the CPU.
- Moving to an ASIC or ASSP (Application-Specific Integrated Circuit, Application-Specific Standard Product) designed for waveform generation. Sample options include the Analog Devices AD98XX DDS (Direct Digital Synthesis) series, which generate sinusoidal waveforms up to 100's of MHz at 10 – 14 bit resolution. These chips are controlled over SPI by the CPU (and may require an external clock signal), but otherwise handle all waveform calculations and DAC/generation internally. Some external circuitry (and possibly a separate DAC) may be needed to add control over amplitude and DC offset. This design concept can potentially achieve the maximum possible frequency, resolution, etc.


**ANALOG
DEVICES**

**400 MSPS, 14-Bit, 1.8 V CMOS,
Direct Digital Synthesizer**

Data Sheet
AD9954

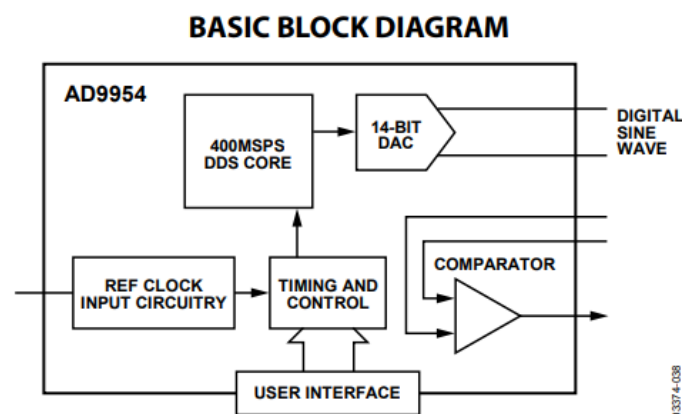


Figure 3 – Sample ASIC/ASSP for waveform generation: DDS chips (AD98XX)

Implementation – SPI Protocol

In this lab we will be using the **Serial Peripheral Interface** (SPI) to establish communication between the DAC and the CPU with high throughput. SPI is a synchronous serial communication interface specification. The concept was originally developed by Motorola, but eventually became a *de facto* standard implemented by multiple hardware vendors. Most modern microcontrollers and embedded microprocessors natively support SPI as an integrated peripheral. The standard is intended for **short-range** communication (no specific noise immunity or error detection/correction measures are included by default) with **high speed/throughput**.

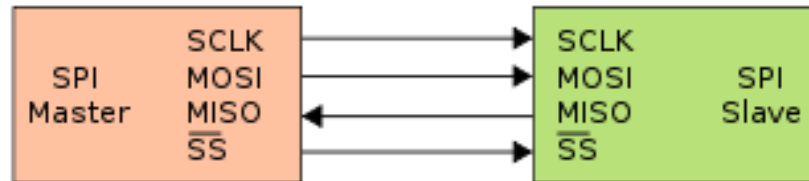


Figure 4 – Basic SPI connection between a master and slave device

At its core, the standard specifies **full duplex** communication, allowing for communication in both directions across the data bus (MOSI and MISO lines, Figure 4). Basic connectivity is as follows:

Pin on Device	Function
SCLK	Serial CLoCK; controls timing of each bit sent; always an output from the bus master
MOSI	Master Out, Slave In; always an output for the bus master and input for slave devices; data line
MISO	Master In, Slave Out; always an input for the bus master and output for slave devices; data line
SS	Slave Select; always an output from the bus master; used to notify a slave device of pending communication

Flow control is achieved through the **master-slave** architecture shown above wherein there is a single bus master which has full control over which device may communicate over the shared communication channel at a given time (via a dedicated slave-select line, SS). As a bus-based architecture, many slave devices can share a single data bus. However, expandability in this manner is limited by the need for individual slave select lines, one per slave device.

It is important to note that although the full standard specifies four wires, 3-wire variants exist. For example, if a slave only receives data but does not send data back to the master (common for many DACs, including the LTC1661 used on this EVB), then the MISO wire may be omitted. Similarly, MOSI may be omitted in applications where data is strictly received from the slave device.

Multiple devices may be connected in two primary configurations. In Figure 5, multiple devices share both MISO and MOSI data lines, allowing full-speed communication between the master and any slave device. In this variant, slave devices must implement some form of tri-state output

for their MISO lines – only one device should actively ‘drive’ the bus line at a given time, as determined by individual SS lines.

The downside of this approach is the larger number of interconnection wires needed, which may lead to noise/crosstalk/loading/layout issues when many devices share the bus. An alternate connection is shown in Figure 6, wherein the devices are daisy-chained (you may recognize the structure from Lab 7). This implementation uses fewer wires, but shares the same disadvantages as other ring topologies; data may need to flow through multiple devices before reaching its destination, causing delay.

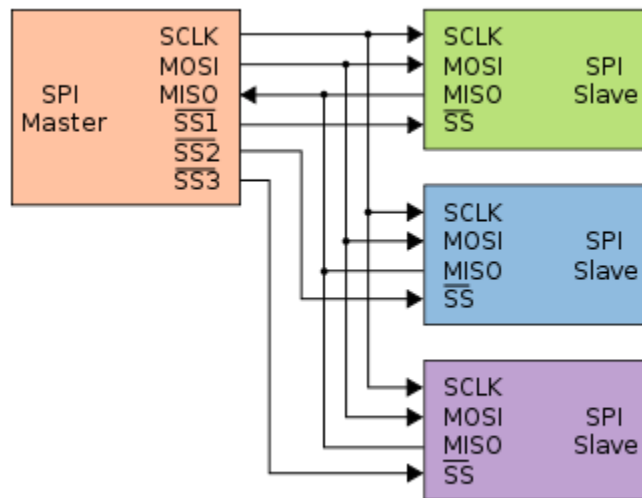


Figure 5 – Multiple slave devices on one bus, shared MISO/MOSI lines

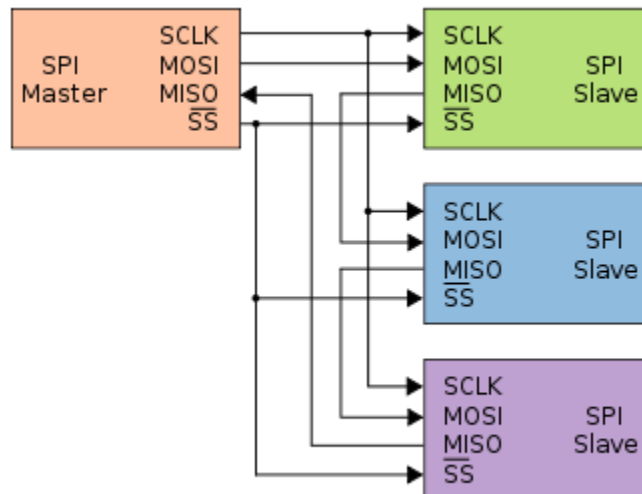


Figure 6 – Multiple slave devices, daisy-chain connection

Data input and output for each device is implemented through the use of **shift registers**, where data is ‘clocked’ into MOSI or MISO one bit at a time, as synchronized by the serial clock, SCLK. After n bits have been clocked in, a full word is available to be read out of the shift register:

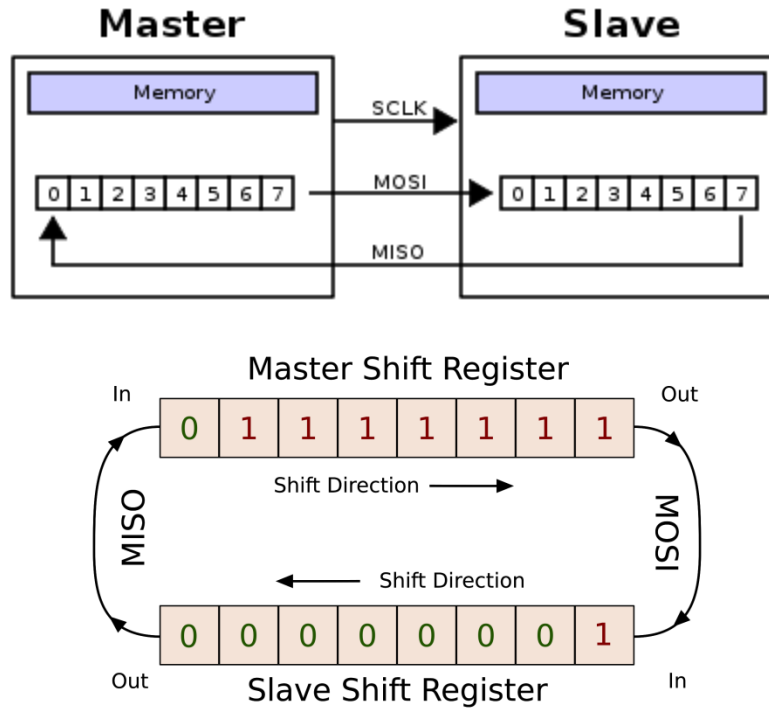


Figure 7 – Shift registers used for data transmission

Data is transmitted between master and slave devices in synchronous full duplex mode; that is, for every transmission of one word of data out from a device, a corresponding word is expected to be read in. As mentioned earlier, however, some devices use one of the two paths. Thus, to send one word of data (length n bits) the following occurs:

0. (*Pre-condition*) The master configures the data transmission parameters, including clock polarity and phase, as well as clock rate (frequency, up to 10+ MHz) based on the slave devices connected. If multiple slaves share the same bus, they must have compatible expectations for clock mode and comparable clock speed limits.
1. The master selects the slave device by asserting the SS line for said device. Some slave devices will expect a minimum wait period after this assertion before further action takes place.
2. One bit is read by the slave device and one bit by the master per clock cycle, as per full-duplex operations. Timing is enforced by the SCLK line – one edge of the clock signal triggers both devices to set their data lines to the bit being sent, and the subsequent edge triggers both devices to read what they see on their input line.
3. Step 2 is repeated n times until a full word is sent.
4. After one word is sent, the master de-asserts SS, returning the bus to idle. Note that some variants allow SS to remain asserted between words, but most require it to be de-asserted. Word size itself may vary between devices, with 8-bit, 10-bit, 12-bit, and 16-bit word sizes being common among modern devices.

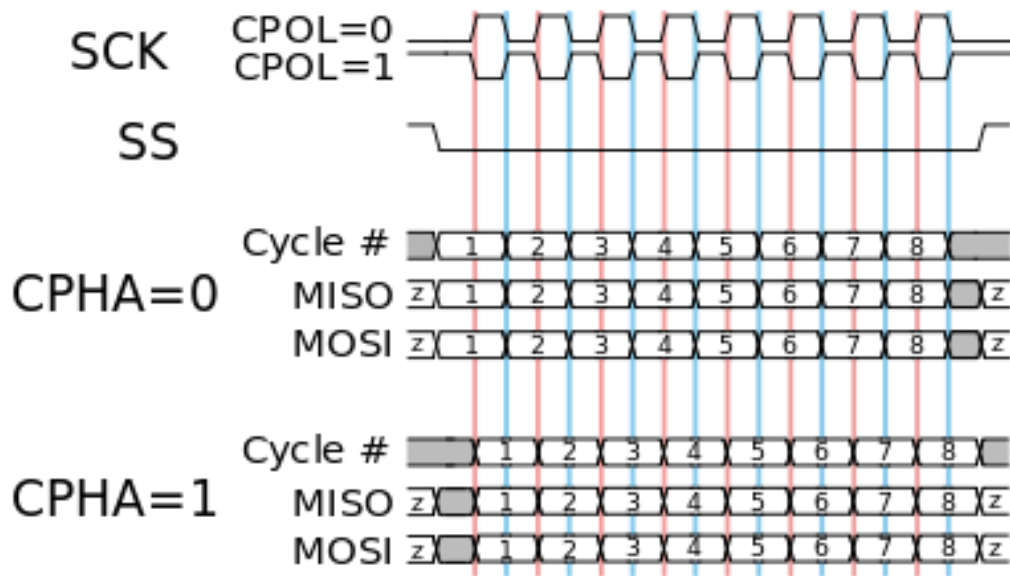


Figure 8 – Clock polarity and phase for SPI communication

Clock configuration allows for four combinations of polarity and phase. Clock polarity, CPOL, determines if the bus idle state is high (CPOL = 1) or low (CPOL = 0). Clock phase determines if the leading edge of a clock cycle tells devices to set up a new outgoing bit (CPHA = 1), or to read in a new bit (CPHA = 0). These settings must be the same for all slave devices sharing one bus. Many microcontrollers standardize the four combinations as follows:

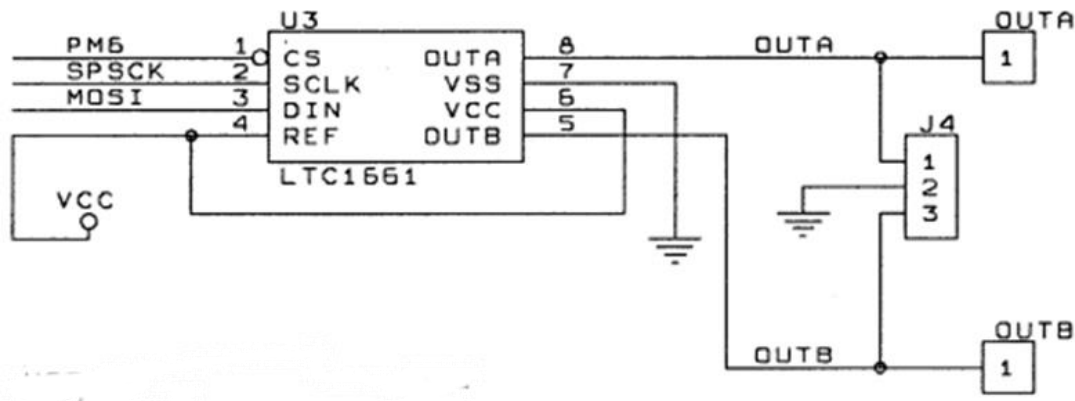
Mode	Clock Polarity CPOL	Clock Phase CPHA
0	0	1
1	0	0
2	1	1
3	1	0

Now that you have a basic understanding of the SPI standard, we can proceed to the specifics of the 68HCS12 setup and the connection of the DAC device.

Implementation – CPU to DAC SPI Connections

The DAC chip is connected to the SPI hardware on the 68HCS12 as follows for our EVB:

(From 'Lab Resource – Dragon12-Plus2 Schematic Page 3.pdf')



(From 'Lab Resource – Dragon12-Plus2 Schematic Page 1.pdf')

PS7/SS0	96	/SS
PS6/SCK0	95	SPSC
PS5/MOSI0	94	MOSI
PS4/MISO0	93	MISO
PS3/TXD1	92	TXD1
PS2/RXD1	91	RXD1
PS1/TXD0	90	TXD0
PS0/RXD0	89	RXD0

Figure 9 – Dragon12-Plus2 EVB connections for the DAC, LTC1661

We can see that the 68HCS12 variant used on the EVB has a built-in hardware SPI interface on PORTS. The DAC is configured as a slave device only, and does not send any data to the microcontroller. Thus, only the slave select, serial clock (SCLK/PS6 = SPSC), and master-out-slave-in (MOSI/PS5) lines are connected.

One thing to note is that although the 68HCS12 has a built-in slave select output (PS7) it is not connected to the DAC. The reason for this is a word-size incompatibility. The 68HCS12 only natively supports 8-bit words, and as such will return the SS output line to idle (high) between 8-bit words. The DAC chip, as we will see, has a 16-bit word size and expects SS to remain active (low) across the entire 16-bit transmission. To avoid this issue, we will manually set SS for the DAC using a general-purpose I/O line on Port M, PM6. This issue is common among microcontrollers developed in the transitional period from the early 2000s to 2010, where SPI was seeing increasing use as a *de facto* standard. Other comparable device families such as most PIC18 CPUs and early PIC24 CPUs share this issue.

Lastly, other important details can be seen. The reference voltage input of the LTC1661 is connected to V_{CC} , meaning that the output range of the digital-to-analog process will be from V_{SS} to V_{CC} , or 0 V to 5 V in this connection. We can also see the two outputs of the LTC1661 which are available. While this lab will only use OUTA, as discussed earlier the second output could be used to add controllable output offset.

Implementation – 68HCS12 SPI Configuration and Transmission

The SPI subsystem on the 68HCS12 is controlled through a series of SFRs, and is configurable to a wide variety of modes of operation. The CPU may be configured as a master or slave device, and to any of the clock modes discussed in the prior section. Its primary limitation is the inherent 8-bit word size. For more details on the following section, please refer to “MIE438 – Lab 8 – SPI Subsystem.pdf”.

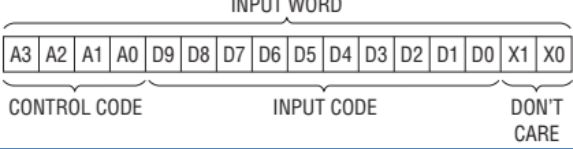
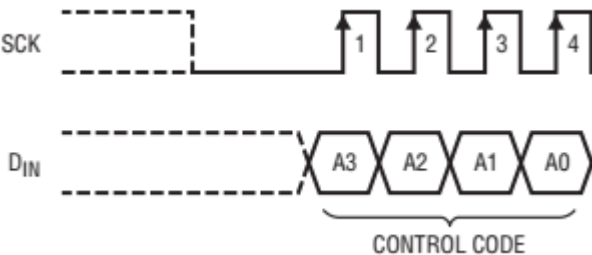
To interact with the SPI subsystem on the 68HCS12, we use the following SFRs:

1. **Data Direction Register M (DDRM)** – Since the DAC’s SS line is connected to PM6, we must set this as an output using DDRM.
2. **Data Direction Register S (DDRS)** – The SCK (PS6) and MOSI (PS5) lines need to be outputs for this application. If the slave device sent data back, MISO (PS4) would need to be an input. The built-in SS line (PS7) also needs to be an output to avoid issues with hardware fault detection, even though we are not using it.
3. **SPI Channel 0 Control Register 1 (SPI0CR1)** – This register determines basic functionality for the SPI system:

	Bit 7	6	5	4	3	2	1	Bit 0
R	SPIE	SPE	SPTIE	MSTR	CPOL	CPHA	SSOE	LSBFE
W								
Reset:	0	0	0	0	0	1	0	0

Bit	Function	Our Setting	Notes
SPIE	SPI Interrupt Enable	0	We are not using interrupts to send/recv. SPI data
SPE	SPI System Enable	1	We want to enable SPI
SPTIE	SPI Transmit Interrupt Enable	0	A second interrupt source related to transmissions over SPI; not needed here.
MSTR	SPI Master/Slave Mode Select	1	Selecting Master Mode for the 68HCS12
CPOL	SPI Clock Polarity	0	See below.
CPHA	SPI Clock Phase	0	See below.
SSOE	Slave Select Output Enable	0	We are not using the built-in SS output
LSBFE	SPI LSB-First Enable	0	See below.

To determine values for CPOL, CPHA, and LSBFE, we must look at the slave device we are connecting. From the datasheet (“Lab Resource – LTC1661.pdf”):

Datasheet	Page	Notes
	8	We can see that the word format is such that the MSB of the control code and input code (A3 and D7) are sent first, thus we want MSB-first (LSBFE = 0).
<p>D_{IN} (Pin 3): Serial Interface Data Input. Input word data on the D_{IN} pin is shifted into the 16-bit register on the rising edge of SCK. CMOS and TTL compatible.</p> 	6	Referring to Figure 8 from the prior section, reading data on the <i>rising</i> edge with a clock that idles low (as per diagram and description on the left) requires CPHA = 0 and CPOL = 0 (i.e., Mode 1 operation). We set:
	9	CPOL = 0 (SCK idles low) SPHA = 0 (Data loaded on rising edge)

4. **SPI Channel 0 Control Register 2 (SPI0CR2)** – Used to handle special functions related to SPI on the 68HCS12:

	Bit 7	6	5	4	3	2	1	Bit 0
R	0	0	0	MODFEN	BIDIROE	0	SPISWAI	SPC0
W								
Reset:	0	0	0	0	0	0	0	0

Bit	Function	Our Setting	Notes
MODFEN	Mode Fault Enable Bit	0	Partially determines SS pin function (checks for SS conflicts when enabled); not needed here (we are manually implementing SS).
BIDIROE	Output Enable in Bidirectional Mode	0	The CPU can be configured for bidirectional operation on MOSI (shared wire). We do not use this mode, so we disable this bit.
SPISWAI	SPI Stop in Wait Mode	0	Determines how SPI behaves when the CPU is in a low-power ‘wait’ mode; not needed here.
SPC0	Serial Pin Control 0	0	Used to set bidirectional mode in combination with MSTR from SPI0CR1; mode not used and thus disabled here.

5. **SPI Baud Rate Register (SPIBR)** – Recall that SPI devices may operate at a range of speeds. For the LTC1661 DAC:

(Datasheet, p. 3, **Maximum Ratings**)

SCK Frequency	Square Wave (Note 6)	●	16.7	MHz
---------------	----------------------	---	------	-----

Since our application requires us to send data to the DAC with high throughput, we want to maximize the baud rate used to prevent a bottleneck in sending said data. Baud rate is determined by SPIBR as follows:

R	0	SPPR2	SPPR1	SPPR0	0	SPR2	SPR1	SPR0
W								
Reset:	0	0	0	0	0	0	0	0

SPPR2–SPPR0 — SPI Baud Rate Preselection Bits

SPR2–SPR0 — SPI Baud Rate Selection Bits

$$BRD = (SPPR + 1) \times 2^{SPR+1}$$

$$Baud\ Rate = \frac{Bus\ Clock}{BRD}$$

We will set our bus clock in software to the maximum for this CPU, 24 MHz. We will also set $SPPR_{2..0} = [000]$ and $SPR_{2..0} = [000]$, yielding:

$$BRD = (0 + 1) \times 2^{0+1} = 2$$

$$Baud\ Rate = \frac{24000000}{2} = 12\ MHz$$

This is the fastest SPI clock we can achieve with this CPU. Notice that it is less than the LTC1661's maximum clock of 16.7 MHz by a significant margin (~40%), meaning we are not fully utilizing the DAC's potential. This clock speed will directly limit the frequency and quality of the waveform our DAC can produce – each new value for the DAC will require 16 SPI clock cycles, yielding an update rate of $\frac{12}{16} = 750\ KHz$ under perfect conditions. Based on sampling theory, our theoretical maximum waveform frequency without aliasing would be $\frac{750}{2} = 325\ KHz$. In practice, code overhead will significantly limit the above figure, however.

5. SPI Status Register (SPISR) – Normally used to implement SPI ISR sources:

	Bit 7	6	5	4	3	2	1	Bit 0
R	SPIF	0	SPTEF	MODF	0	0	0	0
W								
Reset:	0	0	1	0	0	0	0	0

In the above, SPIF and SPTEF are interrupt flags raised when either one word is sent (SPIF), or when a special transfer buffer is empty (SPTEF, not used in our application). Normally, these are used to trigger an ISR, but we will **poll them manually** to see when we can send new data to the DAC. The reason for polling is that we do not want the overhead of a jump to an ISR, and instead wish to send data to the DAC as fast as possible.

6. **SPI Data Register (SPIDR)** – This is a dual-purpose SFR which (i) we put data into to be sent to the selected slave device, and (ii) read data out of (coming from the slave device). Writing to SPIDR triggers a **one word send** automatically. However, the SPI standard is full duplex. We **must** also **read** a value from the data register for every value written. This must happen regardless of if the slave actually sends data (in this case, we know it cannot – the MISO is not even connected to the LTC1661). And, the read must happen **after** the SPI subsystem indicates a full word has been exchanged – this is indicated by SPIF becoming ‘1’ in SPISR. As mentioned prior, normally this would trigger an SPI interrupt, but we will check SPISR manually.

Given the above, to write one word (a byte) using SPI (forms the physical layer):

InitializeSPI() function

1. Set SPI0CR1 as per the above.
2. Set SPI0CR2 as per the above.
3. Set SPI0BR as per the above.
4. Set DDRS *such that SS/PS7 (unused), MOSI, and SCLK are outputs and MISO is an input.*
5. Set DDRM *such that PM6 (actual SS for LTC1661) is an output.*

SPI_Send(unsigned char data) function

1. Assert SS for the LTC1661 low.
2. Send an 8-bit word by setting SPI0DR equal to the byte we want to send (‘data’).
3. Wait for SPIF = 1.
4. ‘Read’ the incoming word from SPI0DR (it will be junk data, so we can discard it, but a read must actually happen).
5. De-assert SS back to idle (high).

We are now ready to implement basic SPI functionality and view actual SPI waveforms.

Lab Procedure

Part 1 – Basic SPI

1. Open 'Lab8_1.mcp' in CodeWarrior. This is a template for you to implement the basic SPI physical layer steps shown on the prior page.
2. Implement the 'InitializeSPI0' function as per the comments provided in the code template and the details on the prior page.
3. Implement the 'SPI_Send(data)' function in the same manner.
4. We will now use the oscilloscope to view the SPI waveform. You will need three scope probes and three jumper wires (to connect to the pin headers).
 - a. Connect one channel to SS (PM6), one channel to MOSI (PS5), and the last channel to SCLK (PS6). Disable the remaining channel (waveform off). Be sure to connect the ground clip from one probe to the ground loop on the board (near the power jack, labelled 'GND')
 - b. Set all channels to 5 V/div and the timebase to around 100 ns to start. Set the scope to trigger from the channel connected to SS, on a falling edge, with a level of around 2.5 V . Adjust the vertical position of each channel so that they do not overlap.
 - c. Run and debug your code until you see the expected SPI waveform; confirm the clock polarity and phase and check the data being sent on MOSI.
 - d. Use the 'measure' menu to check the frequency of the SCLK line (should be $\sim 12\text{ MHz}$).
 - e. When complete, copy the code for your two functions into **Q1** and **Q2** of the deliverable. Attach a capture of your final waveforms as **Q3** of the deliverable.

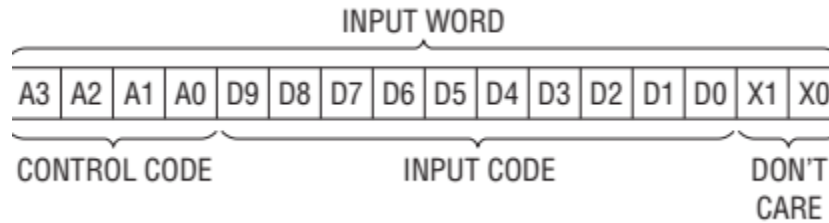
Complete?

☐☐☐☐☐☐☐☐

Implementation – LTC1661 Protocol and Temporary Application Layer

Now that you have completed Part 1, we are ready to implement the specific protocol that the LTC1661 expects. From its datasheet, the first thing one may notice is that the word size of the LTC1661 is 16 bits:

(Datasheet, p. 8)



This necessitates modifications to our physical layer code. Recall that SS cannot be de-asserted between the first and second halves of the word. This was our reason for avoiding the built-in SS output on the HCS12. Thus, we modify our SPI_Send(..) function as follows and move it to the protocol layer (as it is now specific to the LTC1661):

SPI_Send(unsigned char data1, unsigned char data2) function

1. Assert SS for the LTC1661 low.
2. Set SPI0DR equal to the first byte we want to send ('data1').
3. Wait for SPIF = 1.
4. 'Read' and discard the incoming word from SPI0DR.
5. Repeat 2-4 for 'data2'. We recommend you just repeat the code and do not use a loop ('unroll loop' optimization).
6. De-assert SS back to idle (high).

We can now implement our protocol layer using the above as a template. We are also able to implement a basic application layer. From the LTC1661 word format, we have the following:

1. **Data bits, $D_{9..0}$** – These set the output of the DAC according to the following formula:

(Datasheet, p. 8):

$$V_{OUT} = \frac{[D_{9..0}]}{1024} V_{REF}$$

In our case, $V_{REF} = 5V$.

2. **Control Code, $A_{3..0}$** – The DAC chip implements a number of functions, including a low-power sleep mode. It also gives you the ability to set each channel (A or B) separately or simultaneously to the same value. Since we only want to set Channel A to a value, we can send a fixed control code:

(Datasheet, p. 9):

1	0	0	1	Load DAC A	Update Outputs	Wake	Load Input Reg A. Load DAC Regs with New Contents of Input Reg A and Existing Contents of Reg B. Outputs Update. Part Wakes Up
---	---	---	---	------------	----------------	------	--

Note that two commands on p. 9 load DACA, but only this command makes the analog output change (the other waits for a separate ‘update output’ command, $A_{3...0} = [1000]$).

Thus, we can implement a basic application layer as follows:

DAC_SetOutputA(unsigned int output)

- Calls SPI_Send(data1, data2) such that:
 - $data1_{7..4} = [1001]$ (control code)
 - $data1_{3..0} = output_{9..6}$
 - $data2_{7..2} = output_{5..0}$
 - $data2_{1..0} = XX$ (anything, not used)

In a more complete implementation, one could also write a function, say SetOutput(float), which calculates the integer value needed to achieve a certain voltage level before calling DAC_SetOutputA. This is unnecessary in our program, however, as we will move this functionality outside of our main program loop – we want to send values to the DAC as fast as possible, and we know that floating point calculations take a significant amount of time and thus should not be done in the main ‘send’ loop.

Part 2 – Protocol and Application Layers	Complete?
1. Open ‘Lab8_2.mcp’ in CodeWarrior. This is a template for you to implement the protocol and application layers.	<input type="checkbox"/>
2. Copy over your code for InitializeSPI() and SPI_Send() from Part 1. Modify your code for SPI_Send() as described on the prior page and by the comments in the template – it should now send two bytes (data1, data2) in direct succession without de-asserting SS in between bytes.	<input type="checkbox"/>
3. Implement DAC_SetOutputA() according to the description above and the provided comments.	<input type="checkbox"/>
4. Run and debug the program. Start by using the oscilloscope in the same setup as before to verify that SPI communications are occurring, and that the data being sent is now a complete 2 bytes (count the SCLK pulses while SS is low).	<input type="checkbox"/>
5. Next, remove all but one scope probe and adjust the V/div lower, to around 1V/div. Move the waveform for this channel down the screen so that at least 5 divisions are visible above the time. Be sure the scope is set to trigger on this channel (rising or falling is fine, ~2.5V level). Connect this channel to ‘DACA’ on the pin headers (top-center above the breadboard area). Be sure to leave the scope ground clip connected as well.	<input type="checkbox"/>
6. With the program running, you should see something close to a sine wave on the screen. It will appear distorted, however, as we are approximating it with only 8 ‘key’ points (in essence, only 3 bits of precision – see the main loop for exact values).	<input type="checkbox"/>
7. Copy your code for DAC_SetOutputA() into the deliverable document as Q4 . Attach a capture of the DACA waveform as Q5 . Use the scope measurement feature to measure the frequency of the sine wave on DACA and record this as Q6 .	<input type="checkbox"/>

Implementation – Function Generator Application

We are now going to load our application code which implements the high-level goals described in the overview at the start of this lab manual.

Note that this project provides two improved libraries, one for the LCD and one for the keypad, with added functionality and quality-of-life improvements. You may wish to reuse these in your project if you are coding for this platform.

The following is a brief description of some key points of the code. The main loop of the program performs only two functions:

- If the PH0 or PH1 buttons are pressed, it uses the LCD and Keypad to request a new amplitude and frequency from the user, respectively. Note that the PH0/1 DIP switches must be in the ‘up’ position for these buttons to operate.
 - When entering a value using the new keypad functions, value limits are imposed: for example, if the user enters 10000 for the amplitude, it will automatically be limited to 5000, the maximum output.
 - The key ‘A’ acts as a ‘backspace’ allowing you to correct errors in value entry. The key ‘D’ commits the value.
- The main loop endlessly sends values from the buffer ‘**lookupTable[512]**’ over SPI to the DAC. However, to achieve the full range of frequencies requested with the best possible precision, we make some small modifications:
 - At lower frequencies ($f < 313 \text{ Hz}$), the size of this buffer normally determines the quality/bit-precision of the generated waveform. However, large amounts of memory are needed for this buffer as the time scale grows. Thus, to achieve very low frequencies we instead fix the buffer size and insert a **scaled delay** between each byte sent which ‘stretches out’ the generated waveform to the desired frequency, at the cost of accuracy. This effect is controlled by the variable ‘**addedDelay**’ set when the user enters a new frequency. The delay is implemented by the function ‘**delay_us**’, which is made **inline** to minimize call overhead.
 - At higher frequencies ($f \geq 313 \text{ Hz}$), the SPI bus is a bottleneck and we may not be able to send all 512 values of the buffer within one output cycle. Instead, we send only a limited portion of the buffer. This effect is controlled by the variable **lookupTableDepth**, set when the user enters a new frequency.

Aside from the SPI code which you provide, the only other new function is **calculateLookupTable()**, which implements the calculation of the actual buffer/DAC values. Its function can be described as:

INPUT: $f = \text{Frequency}$ (1 ... 20000 Hz), $a = \text{Amplitude}$ (0 ... 5000 mV)

OUTPUT: 16-bit unsigned integers, range 0 ... 1023

$$\text{lookupTable}[i] = \left\lceil \frac{255(a)}{2500} \right\rceil \sin \left\lceil \frac{2\pi(i)}{\text{lookupTableDepth}} \right\rceil + 512$$

Part 3 – Function Generator Optimization

Complete?

1. Open 'Lab8_3.mcp' in CodeWarrior. This is code is mostly complete, with the exception of the SPI code you created in previous parts. Begin by copying over your code from Part 2 for InitializeSPI(), SPI_Send(), and DAC_SetOutputA().
2. With a scope probe attached to DACA (and ground lead) run the program. If all SPI code is correct, you should see a $5 V_{p-p}$ sine wave. Try out the program by using PH0 and PH1 to set different values for amplitude and frequency.
3. Set the frequency to 1 KHz and amplitude to $5 V_{p-p}$. Use the scope to measure the actual frequency at the output and record it in the deliverable as Q7.
4. The frequency you measure indicates if your SPI code is faster, slower, or the same as our reference code (as the time-base is scaled based on our implementation). A higher frequency indicates your code is faster, a lower frequency indicates slower code. Note that there is a margin of error of a few 10's of Hz here. Your final task is to **optimize** the program to achieve an actual output frequency $f > 1 \text{ KHz}$ when the program is set to 1 KHz. A few notes:
 - a. You may modify any part of the program or change its structure **except** for the calculateLookupTable function or related variables (in other words, you cannot just artificially increase the target frequency or change the scaling. You must make a meaningful optimization which addresses the SPI/output loop bottleneck and is measurable on the scope.
 - b. The modified code must still be able to change amplitude and frequency of the output.

For reference, our reference code when lightly optimized achieved $f = 1.1 \text{ KHz}$, and $f = 1.25 \text{ KHz}$ when heavily optimized. Describe your modifications and record relevant code snippets for Q8. Record your final frequency achieved in Q9. Note that full marks can still be achieved if you do not break the 1 KHz barrier, so long as you achieve some measurable optimization over your original code (Q7).

*** End of Lab 8 ***

☐☐☐☐

MIE438 – Microprocessors and Embedded Microcontrollers

Laboratory Experiment 8

Deliverable

Department of Mechanical and Industrial Engineering

University of Toronto

NOTE: For all waveform captures, you may upload them using links provided on the course page instead of attaching them to the PDF.

Last Name, First Name	Student Number

Q1. Code for InitializeSPI()

Q2. Code for SPI_Send()

Q3.

Attach or upload SPI waveform capture

Q4. Code for DAC_SetOutputA ()

Q6.

**Frequency of sine wave
(theoretical maximum)**

Q7.

**Frequency of sine wave
(base code, non-optimized)**

Q8. Describe your optimizations and copy/paste relevant code here:

Q9.

	Frequency of sine wave (optimized code)
--	--