Lab #3: Non-Volatile Memory and Analog Inputs

ECE-121 Introduction to Microcontrollers

University of California Santa Cruz

Additional Required Parts

You will again be using the USB connection between your Uno32 kit and the *ECE121 Console* application on the lab PCs. In addition to the provided parts you will be using a signal generator (sometimes called an *arb*) to inject sine waves into your micro.

You will need a resistor and capacitor to create a low-pass *anti-aliasing* filter to ensure that you are working on the fundamental frequencies involved and not higher frequency signals being mapped down by the sampling process.

Introduction

In the previous lab, you developed a series of input and output functions based on pulse width modulation (PWM) and interfaced with a ping sensor and an R/C servo. You used the SPI interface bus to communicate with a magnetic encoder, and demonstrated some basic reading and writing using the SPI bus.

In this lab, we will be learning to use the inter-interface circuit (I2C) bus with and using it to communicate with an onboard non-volatile memory chip. Non-volatile memory (NVM) is especially important in embedded systems because it survives a restart and is often used to store critical data that enables the system to come up in a useful manner.

Additionally, we will be exploring the analog to digital conversion (ADC) subsystem that can translate a voltage level into the microcontroller to a number which can then be used in additional computations. The ADC is one of the primary microcontroller peripherals that is commonly used in embedded systems.

Lastly, you will be using digital filtering techniques (specifically the finite impulse response, or FIR) to filter your incoming signals. You will NOT be expected to design these filters, rather you will simply learn to use them and call them in a correct fashion.[1]

As with the previous labs, this lab will divide itself into several parts that will be solved sequentially. To borrow Blue Origin's motto: *Gradatim Ferociter*,[2] you will be developing and testing each required subpart before integrating them into a coherent whole.

[1] If you are interested in digital filtering, you should consider the control sequence and the DSP courses available to you.

[2] Meaning "ferociously incremental" in Latin

Info: The strategy of incremental development is one which we will continually emphasize and reinforce through this and many other classes in your engineering education. Much work has been done analyzing project failures, and the most common problem is that systems integration took too long or did not work because the component parts had not been tested together. While at first glance, incremental development appears to be slow, it is–in fact–the fastest way to accomplish the project. This is especially true as complexity grows and more people are involved. Build (code) a little, test a little, repeat. Break things up into smaller, manageable pieces, and get those working and tested. We will be repeating this *ad nauseum*.

Again, this class will be structured in a very similar manner: (i) generate library code, (ii) write a test harness that tests that library, and (iii) write an application using the libraries to do the lab.

For this lab, we will be generating three different libraries:

1. A non-volatile memory interface;
2. An analog to digital conversion interface; and
3. An FIR filtering library.

See the header files in the class folder on the lab computers for more details on the underlying implementation that you are expected to complete. As with the other labs, you will be writing included test harnesses for each library, and as well you will be writing an application using these libraries.

1 Non-Volatile Memory (NVM)

In a computer (of any size) there are going to be several distinct memory banks that operate in different ways. Each, of course, has their own advantages and disadvantages that lead them to be used in the specific ways envisaged by the designers of said computer systems. Very often these have to do with speed, ease of use, and density. Wikipedia has a fairly understandable explanation of non-volatile memory.

The terms RAM and ROM are often used, but somewhat misleading. Both RAM and ROM are randomly accessible, however ROM is non-volatile, meaning that the contents of the memory persist after resetting or power-cycling. In your laptop, you will have RAM (volatile) and either a hard drive or SSD (non-volatile). Data on your hard drive/SSD remains after you turn off the laptop and restart it. Data in your RAM does not.

In embedded systems, typically the non-volatile memory is FLASH, and the volatile memory is conventional (though limited) RAM. Note that in most modern embedded systems, the non-volatile memory has both read and write access.

1.1 Memory / NVM Technologies

There are many memory technologies that have been used over the years, and many variants are still in use today. This appnote from Maxim has a nice table of advantages and disadvantages of the different types. In brief (for your edification):

SRAM: Static Random Access Memory, is a bank of flip-flops that are stable, require no clock or refreshing, and memory is retained as long as power is applied. SRAMs are extremely fast, have a parallel access structure,

and are typically used for high speed low-density applications (e.g.: cache). The disadvantage of the SRAM is that it requires 4-6 transistors and thus a relatively large memory cell on the silicon.

DRAM: Dynamic Random Access Memory, is similar to SRAM, but stores the data is a small capacitor, thus requiring much more current to maintain the memory. The memory cell is much smaller, allowing larger and less expensive memory. DRAMs must be constantly refreshed, and require sophisticated interface circuitry.

NV SRAM: Nonvolatile Static Random Access Memory, is a package which contains SRAM, a nonvolatile memory controller, and a battery all integrated together. The controller switches between supplied external power and the internal battery power transparently, so the SRAM bank never loses its data. These are sometimes called battery backed SRAM.

ROM: Mask Read Only Memory, is the most durable form of memory, in that it is "burned in" at the factory. That is, the memory is hard-coded using a mast at the point of fabrication. These are extremely cheap in huge quantities, but any errors are very expensive to fix (you need to rerun the fab).

EPROM: Electrically Programable Read Only Memory, is like the masked ROM, but can be programmed using a special programmer. The chip is then transferred to the target device. EPROMs come in two basic varieties, the *one time programmable* (OTP) and the reprogrammable version. They are, in fact, identical to each other, except that the reprogrammable kind has a quartz window in the top of the chip allowing the chip to be erased using UV. The OTP simply has no window.

In the early embedded systems days, waiting for the EPROM to erase in the UV, then loading it into the programmer, then loading your code onto it, then transferring it to the device and testing made debugging quite a bit more challenging than it is today. Once the code was working, you would place a piece of electrical tape over the window to make sure you didn't accidentally erase it.[3]

EEPROM: Electrically Eraseable/Programable Read Only Memory, is an EPROM with the ability to be erased and reprogrammed electrically, making it much more practical because it can now be used directly in the circuit. EEPROMs have limitations in the number of read/write cycles, and are often quite slow in access times (compared to S/DRAM).

FLASH: Block Accessible EEPROM, is now the common nonvolatile memory standard. It is electrically erasable and reprogrammable (like the EEPROM), but erasing is limited to larger blocks rather than single bytes. Read access is still byte or word based. Originally floating gate structures developed in the 1980's, modern versions are NAND gate based and usually have very large storage capacities. While each block has the same limited read/write access as the EEPROMs, the controller will often keep track of access and switch to other blocks, extending the life of the FLASH.

1.2 Addressing Modes

As pointed out in Sec. 1.1, there are quite a few different nonvolatile memory options, and each has its own specific addressing requirements. What all memories will share in common is that you will need to differentiate between the *address* and the *data*.

In order to be able to gain access to (for example) 256K of data space, you will need to be able to address each data object individually, thus requiring 18 bits. If you are addressing 32K objects, you will need 15 bits.

[3] This is really historical, you are unlikely to ever see an erasable EPROM unless you are digging through very old hardware.

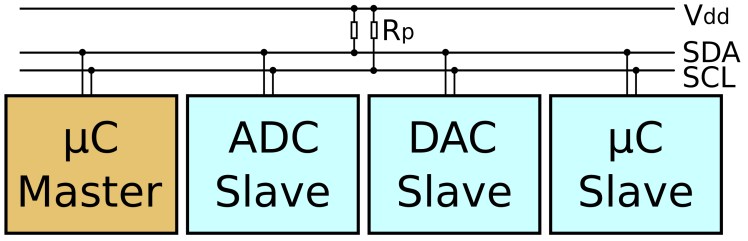


Figure 1: Typical I2C bus configuration

Info: Many of the EEPROMs specify the storage capacity in *bits* rather than *bytes*. This can be confusing as you will need to know how large the bank (object size) is to determine the address space. For example, a 256Kbit device with 8 bits data is in fact 32K x 8, and thus requires only 15bits of address to fully access the device.

Depending on the device, the data object might be 8 bits, 16 bits, or 32 bits stored at each address. This is something you will need to know in order to determine how to access the memory.

Note that most EEPROM / FLASH devices offer a block read or write capability. That is, once the first address has been communicated to the device, subsequent reads (or writes) automatically increment the address until the block is read (or written). This can require aligning to the beginning of the block, and is device dependent (see Sec. 3 for details on our particular chip).

2 Inter-Interface Circuit (I2C) Bus

The inter-interface circuit (or I2C) bus is a standard developed by Phillips (now NXP) in the early 1980's. It has undergone several revisions, and is now an established standard that is very common to most microcontrollers. Unlike the SPI bus in the RC Servo Lab, the I2C bus protocol is much more complicated, and more precisely specified. This makes getting it to work more complicated, however it is well worth the effort as many modern devices rely on I2C to connect to the host microcontroller (e.g.: sensors, cameras, etc.).

There are quite a few references on I2C, and the ones we recommend are the Family Reference Manual Section 24 on I2C, the Wikipedia article on I2C, and the I2C appnote from Texas Instruments. There are many many others out there, but these should give you a sufficient understanding of I2C.

I2C is a multi-master duplex synchronous communication protocol. The physical layer consists of two open-collector bi-directional lines (serial clock—SCL, and serial data—SDA). The most common clock speeds for I2C are 100kbits/s and 400kbits/s (compare this to SPI clock speeds in the MHz range to note that I2C is somewhat slow). A typical setup for I2C is shown in Fig. 1.

Unlike the SPI bus, I2C has no "modes," rather the idle state of clock and data are both defined by the protocol, as well as all of the sampling and transaction details (see Sec. 2.1 and Sec. 2.3 for more details).

Each and every I2C device has a unique 7-bit address (meaning that a maximum of 127 devices can be hooked together). There is a seldom used 10-bit address extension to increase this number, but it is very uncommon. The addresses are often hard coded into the device. Some I2C devices include jumpers or other mechanism to choose between a limited set of addresses. If you find that you need a different address than provided by the

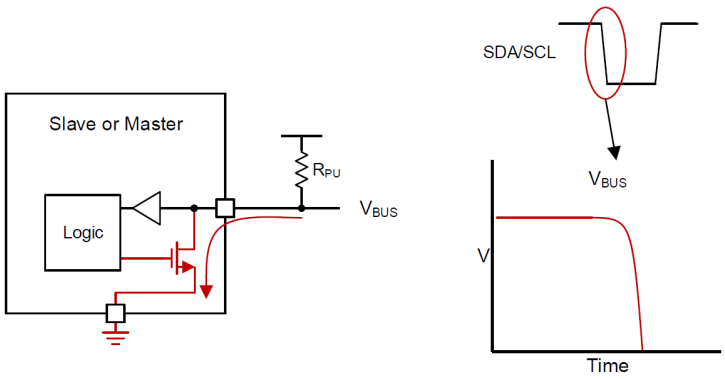


Figure 2: I2C Open Drain Interface pulling down line

device, the manufacturer will either sell you a programmer that can alter the address, or charge you a (very large) fee to change them at the factory.

In practice, the vast majority of I2C busses you will encounter will have one master and a limited number (very much less than 127) of slave devices on the bus.[4] I2C is **NOT** the most efficient multi-master bus, and most I2C busses have a single master on them.

2.1 I2C Signaling

As stated in Sec. 2, the I2C bus is much more precisely defined than the SPI bus. It is more complex, and requires more care to instantiate. However, it is an extremely popular standard, and you will need to know how to use it in your embedded systems careers (or frankly even on your capstone design).

There are two bi-directional lines that make up the I2C bus: the clock line (SCL) and the data line (SDA). Both lines **MUST** be pulled up to V_{cc} for the bus to work. Each device (master or slave) can only pull down the line, not drive it high. That is, both SDA and SCL are open-collector at every device. The open-collector drive of SDA and SCL are required to implement a multi-master system.

Both SDA and SCL idle high (that is, no device holds either line low during times of no activity). Since they are pulled up to V_{cc} using pullup resistors, they will "float" high. When a device actively pulls down the line, forcing a transition from high to low, it causes a falling edge on the line (see Fig. 2). Since no one else on the bus can pull the line high, the line will fall when any device asserts its drive.

Likewise, when the device is done asserting the line (pulling it low), then the device releases the line and the pullup resistor brings the voltage back up to V_{cc}. This can be seen in Fig. 3. In both cases, asserting the line or releasing it, there is a time constant associated with how fast the pullup resistor affects the line voltage. This is one of the primary reasons that I2C uses slower signaling speeds.

An I2C transaction can only begin when the bus is *idle*, which is that both SCL and SDA are high following a *stop* condition (or for a minimum of a certain number of clock times) to ensure that an I2C transaction is not initiated during the middle of an ongoing transaction.

[4]The I2C protocol does allow for multiple masters on the same bus, and thus each master will attempt to drive the bus and some form of arbitration needs to happen. In the I2C bus, each master reads the SDA line while it is sending a 1, and if it reads back 0 then the transaction is aborted and the master assumes it has lost arbitration and control of the bus. This is done automatically on the PIC32, and will generate an error flag (and interrupt) in the case of a bus collision.

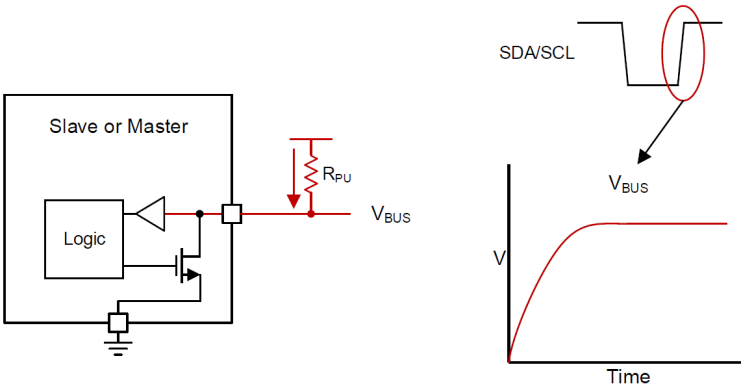


Figure 3: I2C Open Drain Interface releasing the line

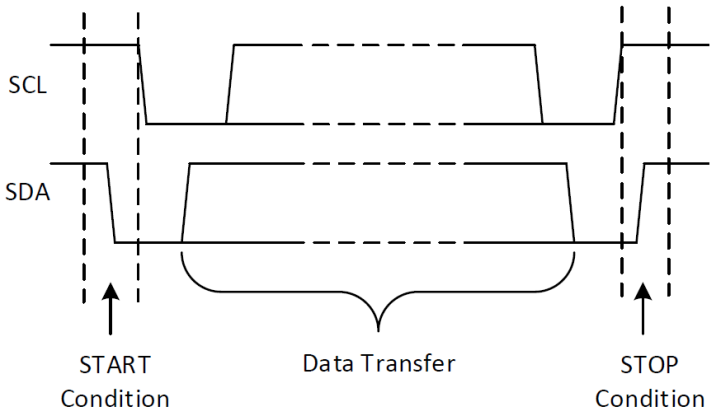


Figure 4: I2C Start and Stop conditions

I2C transactions start by the master sending a *start* which is to pull SDA down while leaving SCL high. Transactions are ended by the master sending a *stop* which is to release SDA after SCL is already high. If the transaction is multi-byte and would require a *stop* and a *start* immediately afterwards, this can be replaced by a *repeated start*, which looks just like a start, but happens before the bus is idle.

In between the start and stop conditions, data is written one byte at a time, MSB first. Data on SDA is only allowed to change while SCL is low, as changes in SDA while the SCL is high are interpreted as start/stop commands. SDA is read when SCL is high, and the data is transferred in/out on each clock pulse. The data on SDA can be the device address, and internal register address, or data sent from the master to the slave, or from the slave to the master. See Fig. 4 for an illustration of the bus commands.

Each byte of data transferred must be acknowledged (ACK) to be considered valid. The ACK bit is sent from the receiver of the data by pulling SDA low during the ACK/NACK clock cycle. This is why it is important that the SDA line be both bi-directional and open-collector. For example, when the master sends an address out on the I2C bus, the slave with that corresponding address must pull SDA low during the 9th SCL cycle.

The NACK (or non-ACK) is when the SDA line is high during the ACK/NACK clock cycle, and generally indicates that the transaction did not get through. There are other conditions that can NACK, and these are detailed in the references. Note that only the SDA line has this transfer of control from master to slave; the SCL line is always controlled by the master (however the master can change, unlike in SPI).

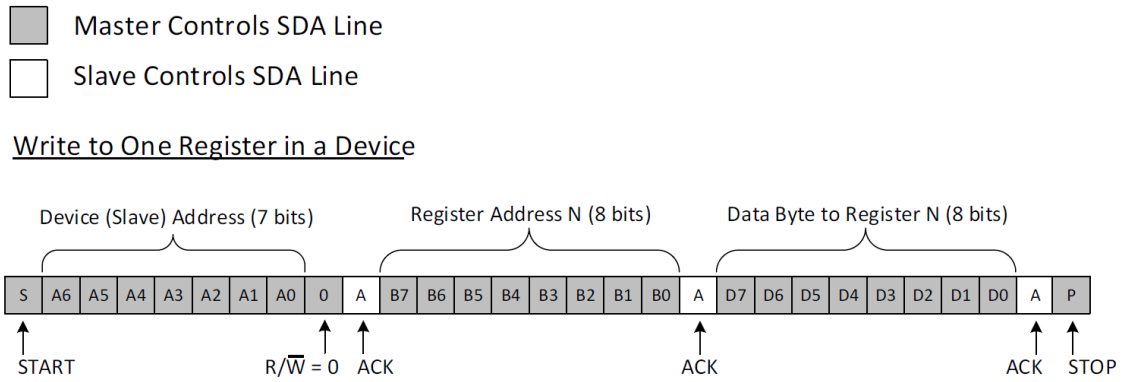


Figure 5: I2C Single Register Data Write

There is an exception to this, which is known as *clock stretching*. If the slave device needs more time to get its data ready, it can hold the SCL low until it is ready, then releasing it back to the master. The master will only start to strobe SCL when it writes a "1" on SCL and reads a "1" on the line (if it is being held at "0" then someone else is stretching the clock).

2.2 I2C Data Transactions

Sec. 2.1 detailed the physical layer of the I2C bus, and the differences between a start and stop, when and how data is transferred, and the acknowledgement (ACK) at the end of the byte. These low-level signaling specifications are simply to transfer a single byte between master and slave. The actual data transaction is more complicated.

First, the address of the slave must be encoded in the first byte (actually 7 bits) of the I2C transaction. The eight bit is the R/$\overline{\text{W}}$ which indicates a read from (R/$\overline{\text{W}} = 1$) or a write to (R/$\overline{\text{W}} = 0$) the slave device.

Data is sent to and read from slave devices through the use of registers on the slave device. Registers are specific locations in the slave memory which contain information (e.g.: configuration infortmation, sampled data to send back to the master, etc.). The master must write information into these registers in order to instruct the slave device.

Note that not all I2C devices have registers; some are simple devices and contain only a single register, which may be written or read simply by sending the device address (as the device only has one register).

In the case of writing to a specific register, the sequence of commands are:

1. Bus Idle, send START
2. Send 7-bit slave address
3. Send R/$\overline{\text{W}} = 0$
4. Receive ACK from slave
5. Send 8-bit slave register address
6. Receive ACK from slave
7. Send 8-bit data to store into register
8. Receive ACK from slave
9. Send STOP

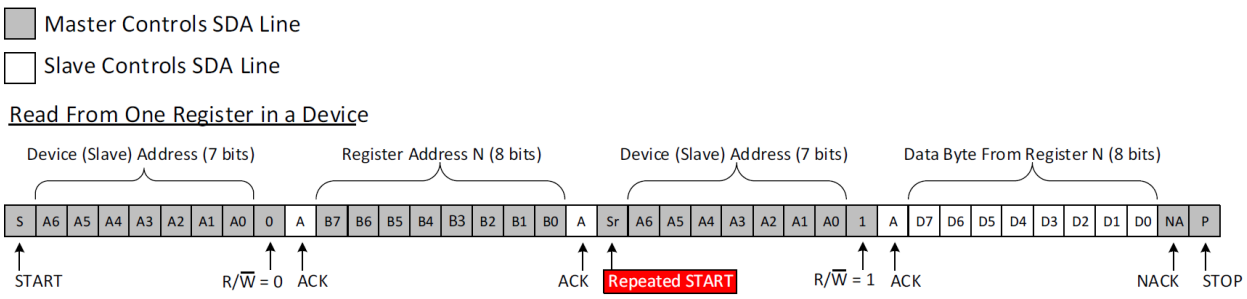


Figure 6: I2C Single Register Data Read

Note that during this transaction, the master controls SCL and SDA *except* for the ACKs in which SDA is driven by the slave. This is depicted graphically in Fig. 5.

In order for the master to read from the slave register, the sequence is typically more complicated. In brief, the master must send the slave address and slave register that is desired then send a repeated start and the slave address again, at which point the master drives the clock but the slave drives the data.

Specifically, the usual steps required are:

1. Bus Idle, send START
2. Send 7-bit slave address
3. Send $\mathrm{R}/\overline{\mathrm{W}} = 0$
4. Receive ACK from slave
5. Send 8-bit slave register address
6. Receive ACK from slave
7. Send REPEATED START
8. Send 7-bit slave address
9. Send $\mathrm{R}/\overline{\mathrm{W}} = 1$
10. Receive ACK from slave
11. Drive SCL while slave drives SDA line to receive 8 bits
12. Masters sends NACK
13. Send STOP

During the dat transfer from slave to master, the master drives the clock (SCL) and the slave drives the data line (SDA). The master NACKs the data received to indicate that no more data is requested, and then sends a STOP to terminate the transaction. This is illustrated graphically in Fig. 6, again indicating when the master is in control of SDA and when the slave is in control of SDA.

2.3 I2C State Machine

Given the complexity of the typical I2C transaction, it is easy to see that it should be implemented in the underlying hardware as a state machine.

The state machine implementation of the I2C bus is, in a sense, hierarchical. There is a low level state machine that translates symbols such as *START*, *STOP*, and the actual data transmissions into the correct bit sequences on the SCL and SDA lines and detailed in Sec. 2.1. This would be in charge of the “bit-banging” and unless

you are attempting to implement I2C in software on a device that does not have the hardware, you will never need to do or see this.

The more important state machine is the one that handles the transaction from *START* to *STOP* with all of the steps in between. If you study Sec. 2.2, you can see that, for example, to write a byte to a specific register on a device requires a detailed flow of data in the correct fashion. Note that if the ACK does not come in the correct place, then the transaction would be aborted immediately and the bus allowed to return to idle (no STOP sent).

Prelab Part 1 I2C Transaction State Machine

You are going to diagram out a state machine for writing to a specific two-byte register (see Sec. 3.1), and another for reading from a specified register:

1. Assume you have START, STOP, IDLE, ACK, NACK as events or actions
2. Assume you have the device address, register address, and data to be inserted
3. Assume that yours is the only Master on the bus, and there is only one device
4. Be sure to recover (abort transaction) if the ACK/NACK fails

You may ignore any arbitration issues. Keep your state machine clean, and use the event/action paradigm for the state machine. The two state machines you produce will be similar in many ways.

In this lab, all of your I2C transactions will be blocking. In the future, when you did not want to block but still required I2C, you would have to build your own transaction state machine that was called by the I2C interrupt to process the next part of the transaction. This would require detailed knowledge of the specific device you are talking to, and how it needed to be interfaced with.

I2C is a complicated bus, and using it in an interrupt driven fashion is beyond the scope of this lab (and this class). However, we feel strongly that once you have finished all of the labs in this class, you will be very well equipped to write an interrupt driven I2C library for whatever specific device you are using.

3 EEPROM

At long last, we get to the I2C device we are using in this lab: the Microchip 24LC256 Serial EEPROM. The datasheet for this device is on the CANVAS page, and you should spend some time familiarizing yourself with the data sheet.

The EEPROM has a capacity of 256K*bits*, thus it is actually a 32K*byte* or $32K \times 8$ device. It has a 64 byte page buffer, over a million read/write cycles, and can complete a full page write in 5 msec. This is a low-cost device that is used in millions of devices to ensure that data required after a reset or power-failure is available to the device.

While it can be written to or read from a single byte at a time, in fact, it is operating on 64-byte pages, and each read or write is a cycle for the entire page. The 24LC256 can be hooked up to other 24LC256's in parallel to create a larger memory space. On the I/O Shield, there is only one EEPROM, and the A0–A2 lines are all tied to ground. That is, the 7-bit address of the EEPROM on the I/O Shield is `0x1010000`.

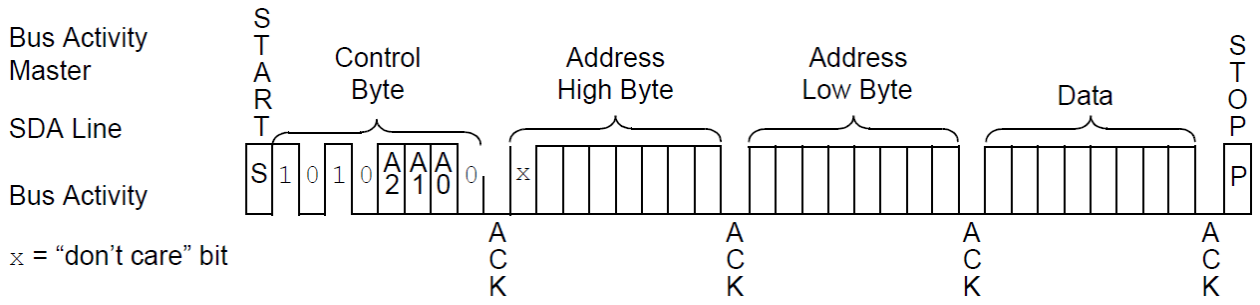


Figure 7: EEPROM Random Access Single Byte Write

One detail from the datasheet that should be especially noted is that the EEPROM uses a 64-byte internal buffer that you write the data to, which then writes the data to the actual EEPROM memory. That internal buffer is keyed to the *current* page. If you start out unaligned (off of the page boundary address) and write 64 bytes the data will loop around and overwrite the beginning of the page. That is, the 64-byte internal buffer acts like a circular buffer.

The entire page write (and all writes are page writes) takes a maximum of 5 msec. You can ensure that the data has been written by waiting 6 msec, or you can poll the device over I2C to see if it is done yet (which might be significantly faster than 5 msec). This will be discussed further in Sec. 3.3.

3.1 EEPROM Random Access Single Byte Write

The EEPROM has a few different access modes which give the user of the device great flexibility when using it in their embedded system. The EEPROM has a random access single byte write, a random access page write, and an internal address pointer incremental byte write.

The internal address pointer incremental byte write operation relies on the internal memory pointer that the chip maintains (and increments by 1 every read or write operation). The internal pointer is set to one greater than the last address manipulated. In this lab, we will not use this mode of access.

To write a single byte to a specified address within the chip (any address can be specified, thus the moniker of random access), the process is similar to the one outlined in Sec. 2.2. The only caveat is that the address specifier in the I2C transaction is two bytes long. Note that to get to 32K locations, you need 15 bits, so that the upper most bit of the 16-bit memory address is ignored.

The sequence required to write a single byte to a specified address is:

1. Send START
2. Send device address
3. Send write flag: $\mathrm{R}/\overline{\mathrm{W}} = 0$
4. Receive ACK from device
5. Send memory address high byte
6. Receive ACK from device
7. Send memory address low byte
8. Receive ACK from device
9. Send data to be stored in address location
10. Receive ACK from device
11. Send STOP

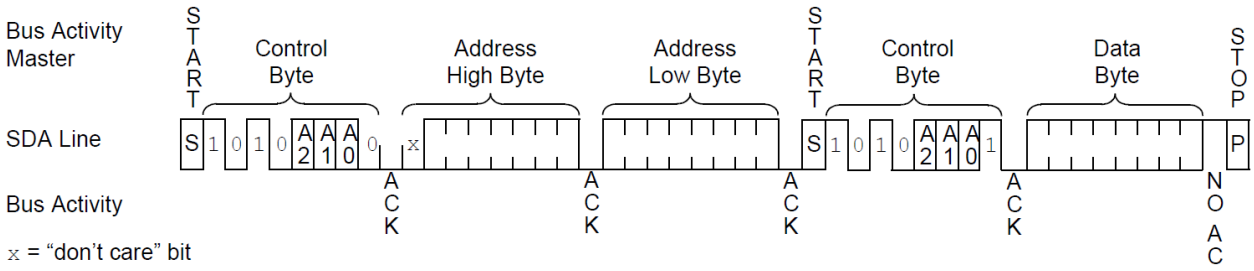


Figure 8: EEPROM Random Access Single Byte Read

This is illustrated graphically in Fig. 7. Again, the major difference here between the generic write outlined in Sec. 2.2 and the above is that the "register" is two bytes wide, and thus needs two transactions (each with an ACK). Should any ACK fail, the entire write transaction is aborted.

Warning: You are working with 16 bit addresses internally to the EEPROM, which means that Endianess is a concern. Make sure that you know which byte is the "high" byte of the address and which one is the "low" byte. This will also mean that you will need to be careful when messages coming from the console include an address.

As you can see from the above sequence (and from the work in Sec. 2.3, writing a single random access byte can be very easily encoded into a state machine (or really a sequence of steps). The big caveat that needs to be reemphasized is that at any point failure to receive an ACK should cause the entire transmission to be aborted (and an indication of the fact that the I2C transaction failed).

Prelab Part 2 I2C / EEPROM single byte write

What registers and sequence do you need to write a single byte to a known address (16bits) in the EEPROM?

1. Assume you have START, STOP, IDLE, ACK, NACK as events or actions
2. Assume you have the device address, register address, and data to be inserted
3. Assume that yours is the only Master on the bus, and there is only one device
4. Be sure to recover (abort transaction) if the ACK/NACK fails

You may use a flowchart, pseudo-code, or state machine drawing. List all registers you need; the special structs `I2C1CONbits.XXX` and `I2C1STATbits.XXX` will be very useful here. The more detailed this is, the easier it will be to implement the lab.

3.2 EEPROM Random Access Single Byte Read

The single random access read is very similar to the single random access write, except that a few additional steps are required. The sequence below (also illustrated in Fig. 8) specifies the single byte read operation:

1. Send START

2. Send device address
3. Send write flag: R/$\overline{\text{W}} = 0$
4. Receive ACK from device
5. Send memory address high byte
6. Receive ACK from device
7. Send memory address low byte
8. Receive ACK from device

---

9. Send REPEATED START
10. Send device address
11. Send read flag: R/$\overline{\text{W}} = 1$
12. Receive ACK from device
13. Receive data from device
14. send NACK to the device
15. Send STOP

In the list above, everything above the line (steps 1-8) is the same as for writing a byte to the EEPROM, the differences come with the repeated start, and then a resending of the device address (control byte) with the read bit set high. For the duration of the data transfer, the PIC32 is generating the clock on SCL, and the EEPROM is driving the SDA line. This is standard for a read operation. Also note that the NACK is crucial, as it indicates to the device that no more data is requested. This will become clear in the block reads and writes.

Prelab Part 3 I2C / EEPROM single byte read

What registers and sequence do you need to read a single byte from a known address (16bits) in the EEPROM?

1. Assume you have START, STOP, IDLE, ACK, NACK as events or actions
2. Assume you have the device address, register address, and data to be inserted
3. Assume that yours is the only Master on the bus, and there is only one device
4. Be sure to recover (abort transaction) if the ACK/NACK fails

You may use a flowchart, pseudo-code, or state machine drawing. List all registers you need; the special structs `I2C1CONbits.XXX` and `I2C1STATbits.XXX` will be very useful here. Again, the more detailed this is, the easier it will be to implement the lab.

3.3 EEPROM Random Access Page Write

As noted in the EEPROM data sheet (and above is Sec. 3), the EEPROM has a 64 byte buffer that it uses for reads and writes. Again, it is important to note the two peculiarities with the device.

First, all reads and writes are block (or *page*) reads and writes. This is true even when you are reading or writing single bytes. The device reads and writes the entire page (thus you don't save anything on the life cycles by using single reads/writes).

Secondly, the 64 byte page buffer acts as a circular buffer in the sense that if the beginning address is not correctly aligned to a page boundary, then the data that goes off the end of the page will get wrapped around

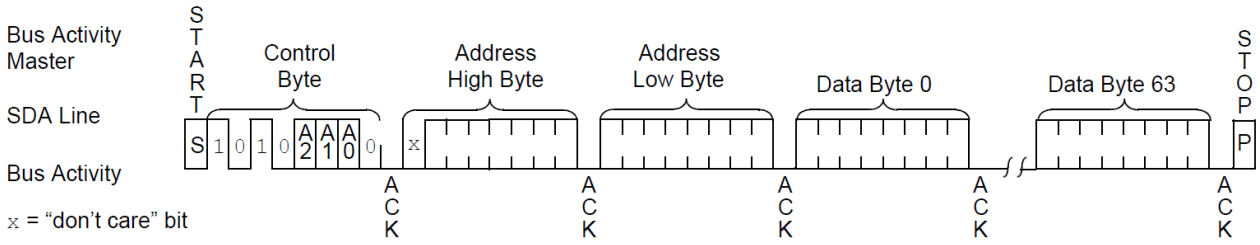


Figure 9: EEPROM Random Access Block Write

to the data at the beginning. It is strongly suggested that you avoid this by doing block read and writes *only* on page boundaries.

Warning: To ensure that you are aligned strictly onto page boundaries when doing block reads/writes, the function prototypes in the `NonVolatileMemory.h` header files `NonVolatileMemory_WritePage` and `NonVolatileMemory_ReadPage` both use a *page* argument for address, rather than a specific address. Pages are 64 byte blocks (hence to convert from page to address, multiply by 64 or left shift by 6).

The steps to doing a page write to the EEPROM are *identical* to doing a single byte write (see Sec. 3.1) up to step 10, at which point the you have written the first byte at the designated address and are ready to write the next 63 bytes.

Steps 9-10 are repeated (Send data, receive ACK) until all 64 bytes have been accepted, and then the final STOP command is sent. Internally, you are writing to the page buffer, and the internal address pointer is being incremented automatically. Once the stop command is sent, the EEPROM will transfer the contents of the page buffer to non volatile storage. This process can take up to a maximum of 5msec. The entire page write process is shown in Fig. 9.

If you wish to write faster than the 5msec block write limitation, then you need to poll the device to see if the data transfer has finished. This is done by repeatedly sending a start, the device address with a write command, and waiting for an ACK. If the device does not ACK, then it is still transferring the data. Or you can wait 6msec and not worry about it.[5]

3.4 EEPROM Random Access Page Read

The last EEPROM operation you will need to complete is the page *read* command. As with the page write in Sec. 3.3 above, the page read shares almost all of its steps with the random access single byte read.

Examining Sec. 8, steps 1-13 are identical. However, if reading out more data (e.g.: more than a single byte) then rather than send a NACK in step 14, the master sends an ACK. The cycle of receive data, send ACK is repeated until the desired number of bytes have been read in, at which point the master sends a NACK and a STOP completing the transaction. This is shown in Fig. 8.

Note that this works because the EEPROM keeps an internal pointer to the address that is automatically incremented by 1 every read or write. Unlike the page write, the page read does not loop back around the

[5]This idea that you are polling the device to see when it is ready is really only required if you are trying to maximize the write throughput of the device. Most applications will not need this level of timing optimization.

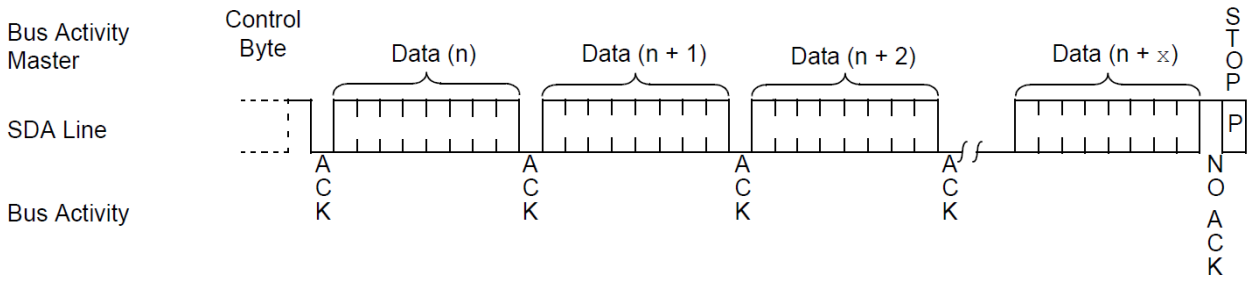


Figure 10: EEPROM Random Access Block Read

page boundary. Rather, it keeps going until the last address is read out of the device. That it, it is possible to start at address 0 and read out the entire chip in one single (very long) I2C transaction.

Note here that the page read is not really a *page* read at all, but rather a *sequential* read. This has to do with the fact that reading is easier than writing on the EEPROM, such that the reading out does not limit the life-cycle of the device. Thus the read does not have the same limitations as the block write and the page buffer. The means that you can keep reading as far as you like.

4 NonVolatileMemory (NVM) Software Module

With all of the detailed information in Sec. 2 and Sec. 3, you are now ready to implement the NVM software module. As with previous labs, there will be some work building up the elements of the library, and then integrating that library with a test harness that will be tested using the *ECE121 Console*.

4.1 NVM Initialization

Initializing the NonVolatileMemory module is very simple, as you only need to set up the I2C subsystem to enable transmission and reception. The main thing that this involves is setting the correct clock frequency using the I2C baud rate generator. Details of how to set it up are in the Family Reference Manual Section 24.4.4.

You should set your software to use I2C1 (there are two available on the PIC32), and set the clock frequency to 100KHz. Note that it is good safety precautions to disable the subsystem *before* you change one of the configuration settings and then re-enable it.

Once you have the I2C1 subsystem enabled and at the correct clock speed. Send out data on the bus and capture it on an oscilloscope or logic analyzer (this means sending a START, and an address, and a R/$\overline{\text{W}}$ flag). Include this image in your lab write-up. Note that you will error out because unless you choose the right address, there is no device to send back an ACK.

Once you have verified on the oscilloscope or logic analyzer that you can generate the clock at the correct frequency and send an I2C address on the bus, have your code output the EEPROM address with a write flag and verify that you get at ACK back. This is a quick indication that your EEPROM and I2C bus are working.

Note that we do **NOT** expect your I2C transactions to be non-blocking (interrupt driven). Rather, we implicitly expect them to block. For example: sending a *START* entails setting `I2C1CONbits.SEN` to 1, and then testing

`I2C1CONbits.SEN` inside a while loop until it no longer is 1. This is because the I2CxCON register SEN bit is automatically cleared once the START is sent by the hardware.

4.2 NVM Data Write

Incrementally, the next two tasks to complete are the single byte write and the page write, as they are both very similar. Once you have the single byte write working, the page write is relatively easy to implement. Details of single byte write and block write are found in Sec. 3.1 and Sec. 3.3 respectively.

4.3 NVM Data Read

The last data transactions you need to complete are the single byte read, and the page read functions. Again, just as the write byte and write page are very similar, so are the read byte and read page functions. Details of how to read a byte and a block are found in Sec. 3.2 and Sec. 3.4 respectively.

4.4 NVM Test Harness

With Sec. 4.1 to Sec. 4.3 completed, you are ready to integrate them into your full test harness. The test harness is straight forward, and takes full advantage of the *ECE121 Console* capabilities.

The test harness consists of reading and writing single bytes, and reading and writing pages. The *ECE121 Console* will generate a random address, and a random value which can be written and read back. The *ECE121 Console* interacts with your software through protocol messages (as always).

For reading a byte, the protocol will send an `ID_NVM_READ_BYTE` with an integer address as the payload, and the micro will respond with an `ID_NVM_READ_BYTE_RESP` message with a single byte (what was read out of the memory) in the payload.

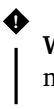
For writing a byte, the protocol will send an `ID_NVM_WRITE_BYTE` message with an integer address followed by a single byte of data as the payload. The micro should write the value into the EEPROM at that address, and respond with an `ID_NVM_WRITE_BYTE_ACK` once it is done.

Obviously the way to test your module is to write data to a random location and then read it back to ensure that it was recorded. Using the *ECE121 Console* you should *first* read the random location, then write it, then read it back to see if it has changed (and still matches). You should be able to turn off your micro and reset it and still read back the correct value (as the EEPROM is nonvolatile).

For reading a page, the protocol will send an `ID_NVM_READ_PAGE` with an integer address for the payload. The micro should respond by reading the 64 bytes located at the address and replying with an `ID_NVM_READ_PAGE_RESP` with the 64 byte payload of the page that was read.

Likewise for writing a page, the protocol will send an `ID_NVM_WRITE_PAGE` with a payload consisting of an integer address and 64 bytes of data. The micro should write the page at the specified address with the data included in the payload, and then respond with an `ID_NVM_WRITE_PAGE_RESP`.

Again, to test everything: (i) generate a random address, (ii) read the page, (iii) write the page with random data, and (iv) read the page back. Power cycle the micro and reread the page and make sure you still get a match.



Warning: Be careful with the endian-ness of the address and data, as the protocol is Big-endian, and the micro is Little-endian. You will need to do some byte swapping in order to get it right.

As usual, you should include an `ID_DEBUG` message with the date and time of compilation (and some message about what it is you are doing) to begin. Again, this test harness needs to be (as always) contained within your `NonVolatileMemory.h/c` module, and turned on using a preprocessor macro at the project level.

In summary, your test harness should:

1. Send an ID_DEBUG message that includes compilation date and time
2. Respond correctly to protocol `ID_NVM_READ_BYTE` with `ID_NVM_READ_BYTE_RESP`
3. Respond correctly to protocol `ID_NVM_WRITE_BYTE` with `ID_NVM_WRITE_BYTE_RESP`
4. Respond correctly to protocol `ID_NVM_READ_PAGE` with `ID_NVM_READ_PAGE_RESP`
5. Respond correctly to protocol `ID_NVM_WRITE_PAGE` with `ID_NVM_WRITE_PAGE_RESP`
6. Verify that the data written into the EEPROM survives a power cycle

Demonstrate your test harness to the TAs using the Lab3, "Non Volatile Memory" tab on the *ECE121 Console*. Write it up in your lab report using any appropriate diagrams to make it clear how (and why) you did things the way you did.

5 Analog to Digital Conversion

It is very common for a microcontroller to sense something in the environment, and then react by driving some output to change something else in the environment. An easy example would be a thermostat that is used to hold the temperature of a room constant (roughly).

The world, however, is an analog place where everything varies continuously in both value and time. The microcontroller, however, can only operate on binary values, and only at discrete times due to clock cycles.

To bring the analog world into the microcontroller, we need a bridge between the analog world and the digital world: the Analog to Digital Converter (known as A/D, ADC, A2D, or AD).[6] Fundamentally, the A/D samples an analog signal (usually a voltage) and converts it to a binary number that represents that analog signal.

Analog to digital conversion is a multi-step process: (i) voltage on the appropriate pin must be routed to an internal difference amplifier, (ii) the difference amplifier outputs the difference between the pin voltage and a reference voltage, (iii) the voltage difference is used to charge up an internal capacitor, (iv) the internal capacitor is disconnected from the pin, (v) and the A/D converts the voltage on the capacitor into a binary number.

The PIC32 implements the most common type of A/D, which is a sample and hold followed by a successive approximation register (see Fig. 11). This combination is a reasonable tradeoff between speed and accuracy. Review the Family Reference Manual Section 17 on the Analog to Digital Converter. There is also a nice appnote from Microchip on stand alone ADCs that has lots of good information.

[6] We will visit the digital to analog conversion later in the class.

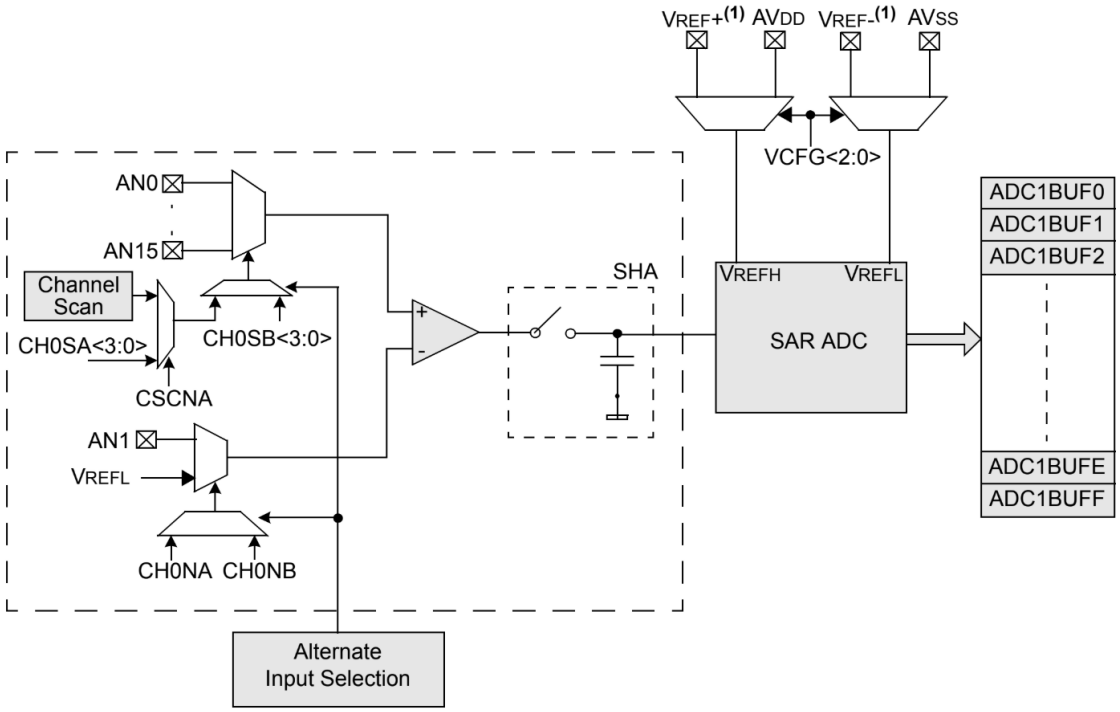


Figure 11: Analog to Digital Conversion Block Diagram

5.1 Sample and Hold (SAH)

In order for the ADC to being a conversion, it must have a stable voltage to work with. This is done by connecting the desired external pin (and thus desired analog voltage) to an amplifier that charges up an internal capacitor. Once the capacitor is charged, the input is disconnected and the ADC can convert the voltage. This is known as sample and hold (SAH).

The internal capacitor is quite small (approximately 4.4pF), and it takes some time for the amplifier to drive it to a stable value. If using the normal fixed voltage references, then it takes approximately 200nS to charge up the capacitor. This is known as the *aquisition time*.

5.2 Successive Approximation Register (SAR)

Once there is a stable voltage on the internal SAH register and it has been disconnected from the input, the *conversion* can take place. The conversion process is where a "code" matching the SAH voltage is generated. Note that there are several methods of converting the analog voltage to its appropriate code. For more information see Appendix A.

The most common type of conversion is the successive approximation register (SAR), which essentially keeps guessing at a value and comparing it to the value held in the SAH register. SARs are popular because they are fast (up to 1Msps) and reasonably high resolution (up to 16bits). They also work in fixed time regardless of the input voltage, and can be easily reset to convert the next value.

# of Bits	2^n	LSB (FS = 1V)	Resolution (%)	Resolution (ppm)	Resolution (dB)
8	256	3.91 mV	0.391	3910	48.16
10	1024	977 µV	0.0977	977	60.21
12	4096	244 µV	0.0244	244	72.25
14	16384	61 µV	0.0061	61	84.29
16	65536	15.3 µV	0.00153	15.3	96.33
18	262144	3.81 µV	0.000381	3.81	108.37
20	1048576	954 nV	9.54E-05	0.954	120.41
22	4194304	238 nV	2.38E-05	0.238	132.45
24	16777216	59.5 nV	5.95E-06	0.0595	144.49

Figure 12: Analog to Digital Conversion Bit Resolution Table

Fundamentally, the SAR works like a scale with a set of binary weights (1/2, 1/4, 1/8, ...). Adding up all of the binary weights gives you 1 unit. Imagine you are given a set of binary weights (each successive weight is half of the previous one), a scale, and an unknown object whose weight you must determine.

The algorithm is simple, place the largest weight you still have on the scale and see if it is lighter or heavier than the object. If it is lighter, leave the weight on the scale. If it is heavier, remove the weight. Move onto the next weight and repeat.

This is how the SAR works, except that instead of weight you have 1/2 the full scale voltage ($\frac{1}{2}V_{ss}$), $\frac{1}{4}V_{ss}$, and all the way down. First test against $\frac{1}{2}V_{ss}$ using a comparator, if it is greater, record a 1 in the MSB and leave it connected. If it is less then record a 0 and disconnect it. Keep repeating until you reach your LSB, and read out the binary representation of your value.

Each bit in the SAR gets you one more "half step" closer to the exact answer. While there will always be some small residual error (depending on the bit depth of the ADC), it will get very close.

5.3 ADC Bit Depth

ADCs have a range of conversion speeds and resolution. The resolution is usually referred to as *bit depth* and ranges anywhere from 8-bits to 16, or even 24-bit resolution.

Unlike in conventional binary, the conversion code is actually a fraction of full scale voltage. That is, for a 10-bit conversion, 0 corresponds to 0 voltage, and 1023 corresponds to 3.3V (assuming Vref is 3.3V).

As pointed out in Sec. 5.2 above, the binary code is really a fractional representation. The MSB is $1/2$, then $1/4$, all the way until $1/2^n$ for an n-bit ADC. This is best seen in the table from the Microchip appnote shown in Fig. 12.

As can be seen, by the time you get to 16-bits, the LSB resolution is $15.3\mu V$ on a $1V$ full scale. The table helpfully shows you the resolution in dB as well as ppm. The Uno32 has a 10-bit ADC, which means it can resolve to slightly better than 0.1% resolution, or an LSB of 3.3mV when operating at a 3.3V full scale.[7]

[7] If you need higher resolution, there are several ADC peripheral add on chips that are basically dedicated subsystems that talk either SPI or I2C. A 24-bit dedicated ADC from Burr-Brown is approximately $5.

5.4 ADC voltage references

Looking back at the block diagram of the ADC system shown in Fig. 11, quite a few of the SFRs are dedicated to choosing the voltage references.

Nominally the low voltage reference is 0V and the high voltage reference is 3.3V (as set by the AV_{DD} and AV_{SS} pins). This means that `0x000` on the ADC corresponds to 0V and `0x3FF` corresponds to 3.3V. However, you might want to increase your resolution if you knew that your signal was always within 1-2V.

To do this, you would set V_{REF-} to 1.0V and V_{REF+} to 2.0V. This increases your native resolution (3.3mV) to $(2-1)/1024 = 1mV$. The V_{REF-} and V_{REF+} must always be between 0 and 3.3V. Note that it is fairly rare to not use the main rails as your reference voltage, but the micro includes the capability.

5.5 ADC Conversion Process / ADC Pin Multiplexing

The ADC subsystem can only convert a single voltage on the SAH capacitor (Sec. 5.1), and has an upper limit of approximately 1 million samples per second. However, you usually want to sample more than one analog signal; thus you need to multiplex pins into the SAH to sample multiple signals.

The PIC32 includes two multiplexers to determine which pin (AN0 - AN15) is connected to the SAH. The ADC can also be placed into *differential* mode, where the SAH is driven by the *difference* between the chosen pin and AN1. Note that to do this, you would need to ensure that the pin is always greater than AN1; that is, ANx - AN1 > 0.

The ADC system can be set up to automatically change the mux each sample so that you ripple through each of the desired ANx pins and sample them. Thus making it possible to automatically sample several pins.

As previously stated, the SAH capacitor needs time to charge up, and then the SAR register needs time to estimate the value of the voltage on the SAH capacitor. As detailed above in Sec. 5.2, the SAR sets the first bit on the internal DAC and checks the output of the comparator. If it is above, the first bit is set to 1, if it is below the first bit is set to 0. Then the first bit is set and the *second* bit of the DAC is set and the process repeated until all 10 bits have been set to either a 1 or a 0.

Thus, the entire process for a single channel ADC conversion is to first charge up the SAH capacitor, then disconnect it form the input pin, then go through the SAR steps to match the voltage on the pin and copy the results from the SAR to the ADC buffer.

The ADC can be set to trigger automatically, or forced to start a sample either directly or via Timer3. The way we will be using it in this lab is to set a continuous scan through all of the desired channels at a constant rate.

5.6 ADC Timing

As stated in Sec. 5.5, the conversion process consists of charging up the SAH capacitor and then completing the SAR sampling to conversion process. When using the voltage rails for V_{REF-} and V_{REF+}, the minimum sample time is 200nsec. The conversion time is controlled by the ADC conversion clock, known as T_{AD}.

According to the Family Reference Manual, T_{AD} must be greater than 83.33ns to allow enough time for a bit to be converted. The full conversion takes 12 T_{AD} times. T_{AD} is set from the peripheral bus clock—which for

our processor is running at 40MHz (or 25ns per tick)—and is set using the ADC conversion clock select bits and is $2 \times (ADCS + 1) \times T_{PB}$. Since $ADCS$ is 8 bits then T_{AD} ranges from 50ns to $12.8\mu s$.

Thus the fastest possible sampling rate using the ADC subsystem on our system would be if we set T_{AD} to 50ns. Thus we would need $(12+2) \times 50 = 700$ns for sampling and conversion (2 T_{AD} for sampling > 83.33ns, and 12 for conversion). Thus, if scanning a single channel, the maximum sampling rate would be 1.4Msps. A similar operation could be done to determine the slowest sample rate.

Remember that in scan mode, you will divide your sample rate by the number of analog inputs you are scanning through. Thus if you have 10 ADC channels turned on, your 1.4MHz sample rate would turn into 140KHz for each channel.

6 Filtering

When dealing with analog signals, you will almost always have to apply some basic signal conditioning either before or after you digitize the signal using your ADC. Sometimes this will be simple amplification if the signal is too small. If the signal is too big, you might need some attenuation (less common).

All signals have some *frequency* content, that is there are parts of the signal at different frequencies.[8] Very often, you are going to want to amplify some frequency of interest, and attenuate other frequencies (which are noise or nuisance signals).

Filtering is a large subject, and there are lots of different ways to approach it. The Wikipedia article on filtering provides a decent place to start. This lab manual will only scratch the surface in the most cursory of ways; we expect you to be *reminded* of what you have previously learned about filtering.[9] There are many excellent references on filtering, for example this TI appnote on filtering basics.

From the appnote:

> In circuit theory, a filter is an electrical network that alters the amplitude and/or phase characteristics of a signal with respect to frequency. Ideally, a filter will not add new frequencies to the input signal, nor will it change the component frequencies of that signal, but it will change the relative amplitudes of the various frequency components and/or their phase relationships. Filters are often used in electronic systems to emphasize signals in certain frequency ranges and reject signals in other frequency ranges. Such a filter has a gain which is dependent on signal frequency.

In this lab (and this class) we will only be looking at the *amplitude* characteristics of the filters; the phase relationship will be of no concern to us.[10]

6.1 General Analog Filtering

Analog filtering is the process of filtering the incoming signal *before* digitizing the signal using specialized electronic circuits. These can range from a very simple resistor/capacitor network, to quite complicated circuits with several operational amplifiers (OpAmps) set up to generate the desired results.

[8] This comes directly from Fourier Analysis, as any signal can be broken down into a sum of higher and higher frequency sine waves.

[9] The math that describes filters is complex, with the complex part of the result set to $j\omega$ as a function of frequency. Several of the controls classes go into this in far more detail.

[10] Phase distortion of filtering is a very important topic in audio work, but beyond the scope of this class.

Filters with only passive components (resistors, capacitors, inductors) are known as passive or RC filters, whereas ones with OpAmps involved are known as "active" filters. There are many reasons to use one over the other, but the most common is that it is difficult to achieve a large attenuation with a passive filter network.[11]

Filters are characterized by a *transfer function* which is used to mathematically describe the filter. The transfer function is a mapping of input to output of the filter, and is usually a function of frequency. Every filter has a characteristic frequency, often called the break frequency or cutoff frequency.

6.2 Low Pass Filtering

The most common filter used in almost all applications is the low-pass filter. This is because most sensors have high frequency noise on them that needs to be removed in order to get a better handle on the true signal.

A low-pass filter allows frequencies below its cutoff to pass and attenuates frequencies above its cutoff frequency. The cutoff frequency, f_0, is often referred to as the *break* frequency. On a log-log plot of log(magnitude) vs log(frequency), the low-pass filter appears to be a flat line at unity gain (0dB) between low frequencies and f_0, and then slopes downwards; the actual slope will depend on the filter topology and order (though a simple RC low-pass will have 10dB/decade slope).

A simple passive low-pass filter is a resistor with a capacitor to ground. The lowest of the low-pass filters is the "average" which simply allows the average to pass and all frequency content is smashed to 0.[12]

6.3 High Pass Filtering

The high pass filter blocks frequencies below its cutoff frequency, and allows frequencies above to pass. Again, on a log-log plot, it will appear to be a line sloping upwards (equivalent to the low-pass downwards slope) until f_0 and then it will be flat for all frequencies above.

The high pass filter is typically used to remove low-frequency noise from a signal, such as biases (which are DC offsets). Again, a simple high-pass filter can be constructed from a capacitor in series with the signal, and a resistor to ground (or some other handy reference).

6.4 Digital Filtering

When designing filters for specific applications, as the filtering demands increase, you will find yourself looking up Butterworth, Chevychev, and Bessel filters of larger and larger orders to achieve the desired filtering. These start to get complicated and difficult to build.

An alternative is to digitize the signal using your ADC, and then manipulate it within the microcontroller. The Wikipedia article on digital filtering again has a decent general treatment. There are many different algorithms that you can imagine running inside the micro that would have the desired result. For example, you might consider running an n-point moving average. That is, average the last n points and that is your signal.

[11] This is, of course, over-simplified. Proper filter design is highly dependent on the needs of the application, and the characteristics of the incoming signal. Often you will have a mix of passive, active, and digital filters to achieve the desired response.

[12] This is not exactly true in practice as you need a finite number of points to average, so it is only the average of those points.

You can quickly see that the n-point moving average acts as a low-pass filter, and is not complicated to implement. There are many others possibilities, and signal manipulation once digitized is a field unto itself.[13]

Anti-Aliasing Filter: There is one thing to be aware of when bringing in analog signals into the digital domain—aliasing. According to Shannon's sampling theorem, the best you can ever reconstruct is a sine wave at half the sampling frequency (also called the *Nyquist* frequency). Any frequency content in the signal above the Nyquist frequency will appear as a lower frequency signal within the lower frequency bound. These "phantom" lower frequencies are indistinguishable from real signals once they have been sampled. They are called *aliases*.

This frequency folding around the nyquist is a function of the mixing of the sampling with the signal, and is used in communications and RF to great advantage. However when dealing with an incoming signal to be digitized, it is a nuisance.

Almost all sampled systems have an anti-aliasing filter that sits between the incoming signal and the ADC; it is a passive low-pass, with a cutoff frequency set slightly below the nyquist to attenuate all higher frequency signals.

6.5 Finite Impulse Response (FIR)

The finite impulse response (FIR) filter is a special case of the digital filter, and can be designed to be high-pass, low-pass, or almost any other map of gain to frequency. The FIR filter consists of a number of "taps" or "lags" and a corresponding weight to each one.

The incoming signal is digitized, and then each sample is multiplied by its corresponding weight, and then summed across the taps. The FIR filter is nominally a simple array of numbers to multiply the incoming signal (and its history). The Wikipedia article on FIR filters is somewhat informative, if limited.

For example, a 4-tap filter with weights [4 3 2 1] will multiply the current sample by 4, the previous sample by 3, the one before that by 2, and add to it the sample 3 sample times ago and divide the entire sum by 10 ($4 + 3 + 2 + 1 = 10$). Another example of an FIR filter would be the n-point moving average. Here, all the weights are $1/n$ or all of the weights are 1 and at the end you divide by n.[14]

$$FIR = \frac{1}{\sum w_i} \sum w_i x_{-i}$$

where w_i are the FIR weights, and x_0 is the current sample, x_{-1} is the previous sample, etc. You can see that you need to store the same number of data samples as you have taps on your FIR filter.

The FIR filters have two very nice qualities. The first is that they are always stable, meaning a bounded input will produce a bounded output. The second is that there are excellent methods for designed FIR filters to produce the desired results.

The design of an FIR filter is beyond the scope of this class. We will be handing them to you. But you will need to do the multiplication and division to get the output of the FIR filters in your code.

[13] The DSP class is not a bad place to start.

[14] This requirement to divide by the sum of the weights at the last step leads to two common implementations of the FIR filter; the *normalized* FIR filter where all of the weights sum to 1, or where the weights sum to a power of 2. The first requires floating point or fractional math, which can be computationally expensive. The second uses integer math and requires only a right shift at the last step.

7 ADC / Filtering Module

Finally, we get to the part where you will actually write some code and set up the ADC peripheral. The test harness consists of sampling the requested channel (A0–A3), and applying the filter to it, and reporting back both the raw and the filtered results. You will be using the *ECE121 Console* to test your `ADCFilter` module.

7.1 ADC Init

The first function required in the `ADCFilter` module is the `ADCFilter_Init()` function. There are quite a few things to set up in the filtering section, as well as the ADC subsystem init as well.

You will need to be able to handle sampling and filtering multiple ADC channels; this means you will need to store filters for each ADC channel, and you will also need to store a number of samples for each ADC channel. The number you will need is defined in the header file as `FILTERLENGTH`. The easiest way to implement this is using 2-D arrays, with one dimension being the number of ADC channels, and the other being the number of taps on the filter. These arrays should be *signed shorts* and should also be static (module level).

These 2-D static short arrays (for the filters and for the data) are going to be used to apply the FIR filter. The data 2-D array acts as a circular buffer, with new data being inserted at the next spot (and rolling back around once it goes off the end). The only difference is that you always use the entire array, so you only need to know where the latest data is.

The first step of the init function will be to initialize the filters to some known values (here all ones or all zeros both have utility). Since the FIR filters we provide all sum to 2^{15}, you can distribute that sum through the filter weights as an alternative.

To set up the ADC you want to choose the following:

1. Connect the desired pins to the ADC (`AD1PCFGbits.XXX`)
2. Add those same pins to the scanner (`AD1CSSLbits.XXX`)
3. Set the ADC to auto-sample, auto-convert, and 16 bit unsigned mode
4. Set the ADC to use internal voltage references, scan mode, and to interrupt after each *set* of measurements
5. Set the ADC to use 16 bit buffers, set T_{AD} to use the peripheral bus clock
6. Set the ADC T_{AD} to 348 T_{PB}
7. Set the sample time to 16 T_{AD}
8. Turn the ADC on, enable the ADC interrupt

Note that the uC32 analog input pins, A0–A3, are mapped to AN2 through AN10 in a non-linear fashion, and remember to adjust your code accordingly. This is documented in the uC32 manual.

As in the other labs, the ADC interrupt will need to be setup. For this you will need the following code:

```
void __ISR(_ADC_VECTOR) ADCIntHandler(void) {
```

The interrupt is fairly simple, you will need to first clear the interrupt flag, then copy the data contained within the ADC buffer (for each channel) into its appropriate 2-D array. You will need to index into the array such that you keep only the last `FILTERLENGTH` samples for each channel.

Once you have gotten the initialization setup, you will need to get the ISR working and verify that you are filling your data array in the correct manner. This can be accomplished by injecting test values of ground or 3.3V into the pins (A0 is connected to the pot, so it is easy to verify).

The `ADCFilter_RawReading` function returns the latest ADC conversion on the corresponding pin (0-3). This should simply return the data array entry for the current index and pin.

Prelab Part 4 ADC Filtering Setup

You are going to be setting up the ADC subsystem, to the specifications stated above. Detail the setup.

1. What is the conversion rate implied by the specs (assume PBK 40MHz)
2. Which pins are enabled for scan mode
3. Pseudo-code both the data and filter 2-D matrices, and how you will access them

You may use a flowchart, pseudo-code, or state machine drawing. List all registers you need; the special structs `AD1PCFGbits.XXX`, `AD1CSSLbits.XXX`, and others will be very useful here. Again, the more detailed this is, the easier it will be to implement the lab.

7.2 Filtering with FIR filters

Once the ADC init is completed and the 2-D arrays have been properly constructed, you should next tackle applying the FIR filters. The function is `ADCFilter_ApplyFilter` and it takes a pointer to the filter array, a pointer to the data array, and an index indicating where the start point of the data array is.

The reason that you need the start point of the data is that while the filter coefficients are in order $0 \rightarrow$`FILTERLENGH`, the data array is not going to necessarily have its first point at the [0] index. Thus the data array and the filter array will be offset from each other. This means modulo arithmetic just like when using the circular buffer.

The FIR filter is applied by multiplying the first weight (filter[0]) by the current measurement (values[startIndex]), and then incrementing both and doing it again. Once you have iterated through the filter and data, divide the entire sum by 2^{15}.[15]

Warning: The intermediate variable you put the FIR sum into must be an *int*. If you use a short, you will overflow/underflow. You will need to use the modulo (%) operator or have an if statement in order to wrap the values index around. Be careful here as the filter index and the value index will be different when you multiply them.

In order to apply the FIR filter, you will need the weights of the FIR filter; this is accomplished using the `ADCFilter_SetWeights` function. This function indicates which pin (0-3) and a pointer to an array of weights (signed shorts). This structure allows that each pin can have a different filter associated with it. Again, all you are doing here is filling in your 2-D array of filter weights.

The final function to be implemented, `ADCFilter_FilteredReading` simply applies the FIR filter to the data collected on the appropriate channel. This simply applies the weights that you set and runs it through

[15] This should be done with a shift operation; if you do it by division there will be wailing, gnashing of teeth, and rending of cloth.

the ADCFilter_ApplyFilter function. Note you will need to be somewhat clever about the arguments to ADCFilter_ApplyFilter, as you will have to isolate the correct part of the 2-D array to pass in.

Prelab Part 5 FIR Filtering

In order to apply the FIR filter, you will have an offset between the index of the filter weight array and the data array?

1. Pseudo-code out the ADCFilter_ApplyFilter function
2. How are you going to pass in only part of the 2-D array to the ADCFilter_ApplyFilter function?

Again, the more detailed this is, the easier it will be to implement the lab.

7.3 ADCFilter Module Test Harness

Finally, we come to the test harness, which is intended to test all of the functionality of the module. As always, the test harness must be included in the ADCFilter.c file, and be conditionally compiled using a project configuratoin flag. At startup, your test harness will issue an ID_DEBUG message with the data and time of compilation, and some indication of what test harness is being run.

The test harness is run from the ECE121 Console under the Lab3 -> ADCFiltering tab. Changing the channel in the pulldown menu and clicking on the "Change Channel" button with send the protocol message.

The protocol message ID_ADC_SELECT_CHANNEL instructs the microcontroller to switch ADC channels. Note that the ADC is always sampling all 4 channels, the message simply tells the microcontroller which one to report back. The microcontroller should reply with an ID_ADC_SELECT_CHANNEL_RESP that reports back the current channel.

The two buttons, "Load Low Pass" and "Load High Pass" generate a protocol message ID_ADC_FILTER_VALUES with the 32 shorts to be loaded into the filter of the current channel. Be a bit careful here, as you will need to deal with the endian-ness of the shorts. Note here that each channel has a single FIR filter associated with it. The buttons allow you to change the filter.

Once the filter values have been decoded and updated into the correct column of the 2-D filter array, the microcontroller should answer back with an ID_ADC_FILTER_VALUES_RESP message indicating the channel that had its filter coefficients updated.

Using the free running timer module to set a 100Hz report rate (that is every 10ms), your microcontroller should send a ID_ADC_READING to the *ECE121 Console* with the raw and filtered readings of the currently selected channel. These will be displayed and the console will calculate absolute values and peak-to-peak of the reported channel.

In summary, your test harness should:

1. Send an ID_DEBUG message that includes compilation date and time
2. Respond correctly to protocol ID_ADC_SELECT_CHANNEL and sends back an ID_ADC_SELECT_CHANNEL_RESP
3. Respond correctly to protocol ID_ADC_FILTER_VALUES and sends back an ID_ADC_FILTER_VALUES_RESP

4. Every 10ms report back an `ID_ADC_READING` with the raw and filtered readings of the current channel

In order to test your ADCFilter module, you will need to use a signal generator to generate a 25Hz and 450Hz sine wave to be injected into your analog pins, these sine waves should range from 0-3.3V. When using the low-pass filter, the 25Hz signal should come through more or less unchanged, and the 450Hz one should be very much attenuated. The opposite is true for the high-pass filter.

Show your ADCFilter test harness to the TAs for checkoff. Write up in your lab report all of the implementation details. Use flow charts, diagrams, pseudo-code, or any other items you need to explain your code in sufficient detail such that any other student in the class would understand how you implemented the module.

8 NVM and ADC Lab Application

As with the previous labs, you will now use your libraries in an application that will demonstrate that you have mastered the required peripherals. This application is a bit more involved than the ones on previous labs to correspond to your increasing skills in microcontroller programming.

As always, begin your application with an `ID_DEBUG` message with the date and time of compilation, and which application you are running. At startup, the 8 filters (2 for each channel) are assumed to be already stored in the NVM. When filter coefficients are updated, they are to be stored back into the NVM at the appropriate page location.

The application interacts with the *ECE121 Console*, and is driven by the 4 slider switches (SW1 to SW4). The first two (SW1 and SW2) are used to select the channel (both down = channel 0, both up = channel 3). SW3 is used to choose which of the two filters to run (switch up = second filter), and the last switch, SW4, chooses between absolute value or peak-to-peak value display on the LEDs (switch up is peak-to-peak).

In short, you choose the channel, set which filter to use (high pass or low pass), and display either the peak-to-peak on the LED bank, or the absolute value. The filter is loaded from the NVM for the appropriate channel and filter selection.

Use the free running timer to update at 100Hz (every 10ms), and check the status of the switches to see if the channel or filter selection has changed. If there has been a change, the microcontroller should send a protocol `ID_LAB3_CHANNEL_FILTER` with a single char payload whose upper four bits are the channel number and the lower 4 are the filter selection.

The microcontroller should respond to a `ID_ADC_FILTER_VALUES` and send back an `ID_ADC_FILTER_VALUES_RESP`, loading the sent filter coefficients into the channel and filter specified by the state of the switches. The new filter coefficients should also be written to the NVM at the correct page (again, corresponding to the state of the switches).

Every 10ms report back an `ID_ADC_READING` with the raw and filtered readings of the current channel, with the channel and filter specified by the switches.

For the display, if in peak-to-peak mode, you should take the minimum and maximum of the filtered readings over some window of measurements (63 to 127 samples work well) and use that to set a bar graph on the LEDS. Note that you will need to scale your min and max measurements such that your full range lights up all of the LEDs.

If in absolute value mode, the bar graph should simply display the absolute value of the current filtered reading. That is, if the filtered reading is less than 0, then you should multiply it by -1. Again, you will need to scale your number so that you light up the segments appropriately.

In summary, the Lab3 application (as its own `main.c` file) needs to:

1. Send an ID_DEBUG message that includes compilation date and time
2. Load the existing filter coefficients in from NVM at startup
3. Set the ADC channel based on SW1 and SW2 (channel 0 - 3)
4. Use high-pass or low-pass filter based on SW3
5. Display either peak-to-peak or absolute value filtered response (selected by SW4) on LED as a bar graph
6. Report back an `ID_LAB3_CHANNEL_FILTER` when switches are changed (except SW4)
7. Respond correctly to protocol `ID_ADC_FILTER_VALUES` and send back an `ID_ADC_FILTER_VALUES_RESP`
8. Store new filter coefficients in the NVM
9. Every 10ms report back an `ID_ADC_READING` with the raw and filtered readings of the current channel

Show this application to the TAs for a checkoff. Write up the lab report detailing how you implemented everything; include flow charts and pseudo-code where appropriate.

A Analog to Digital Conversion Methods

If time allows, this section will be completed. If the section is incomplete and you are interested, go see Max Dunne during office hours and ask him.

Acknowledgements

- Figure 1 courtesy of Wikipedia
- Figure 2 courtesy of Texas Instruments
- Figure 3 courtesy of Texas Instruments
- Figure 4 courtesy of Texas Instruments
- Figure 5 courtesy of Texas Instruments
- [TODO: Finish this later]