

Lab #2: R/C Servo, SPI, and Pulse Width I/O

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 that you will be using, you will also need to power the RC servo using 5V:

1. 7805 Fixed 5V Regulator
2. Alternately, 5V power supply
3. Hookup wire, jumpers, etc.

Introduction

In the last lab, you developed a communications protocol that can send messages to and receive messages from the ECE121 Console. In this lab, we are going to be using the protocol (and expanding your use of protocol payloads) to control a real physical device: the R/C servo.

This lab will utilize three new devices, an encoder, a ping sensor, and the R/C servo. The ping sensor and the R/C servo are both based on a pulse width control. That is, the high time of the pulse determines the value you read from the ping sensor, and the command you send to the servo. This is sometimes also called Pulse Width Modulation (PWM), and is one of the fundamental ways in which a microcontroller interfaces to the external environment.

An R/C Servo (or servo) is a small actuator that is mass produced and very inexpensive. That is, they are cheap (and crappy). The R/C is for “Remote Control” as they have traditionally been used in remote controlled toy planes and cars. They have had a resurgence in applications related to robotics. See Sec. 5 for a more complete introduction to servos.

Internally to the R/C servo is a (surprisingly) small DC motor, a compound gear train, and a feedback potentiometer that gives the feedback signal to the controller. The controller drives the motor to the desired position (as determined by the potentiometer), and will continue to do so until the position is reached. They range in price from <\$5 to over \$600 depending on torque, size, and speed specifications. The ones we use will be at the lower end of the cost range.

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*,¹ you will be developing and testing each required subpart before integrating them into a coherent whole.

¹Meaning “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 four different libraries:

1. A free running timer (for timing events);
2. A ping sensor library (using input capture);
3. An encoder library to read the encoder sensor; and
4. An R/C Servo library that can command the servo to a desired position.

See the header files on the ECE121 Folder on the lab computers for more details on the underlying implementation that you are expected to complete.² As with the other labs, you will be writing included test harnesses for each library, and finally you will be writing an application using these libraries.

1 Timers

One of the most fundamentally important things that a microcontroller can do is to time events. This includes triggering things at certain predefined times, but also to time the elapsed time or duration of other events.

It is strongly suggested that you carefully review the Family Reference Manual chapter on Timers. There is also a decent introduction here, though the included code is for a different microcontroller.

In principle, a timer is a simple counter. Its advantage is that the input clock and operation of the timer is independent of the program execution (that is, it is done entirely in hardware). The deterministic clock makes it possible to measure time by counting the elapsed cycles and taking the input frequency of the timer into account.

Fundamentally, a timer is a subsystem within the microcontroller that functions as a counter.³ There are many modes in which the counter can be triggered. When used as a timer, the counter is strobed using the system clock (possibly prescaled) to drive the counter.

When the counter (timer) counts up to its maximum range, the next increment of time causes it to roll over, resetting to 0, and optionally generating an interrupt at that time. The timer subsystem can also be reloaded with a period match that will cause it to roll over on reaching that period, rather than the full timer count (note that you could achieve the same thing by reloading the timer with a new value on roll over, different micros give you different options to achieve a specific rollover).

²If you are running on your own computers, then you need to remember to run the CodeUpdater to grab the latest versions.

³If you remember your CMPE12/L class, you will recall that you can make a ripple counter from a chain of flip-flops that are connected sequentially. The ones you are dealing with are, of course, more complicated than that, but conceptually they are the same.

The resolution of the timer will be dependent on the clock pre-scaler that is used, and the system or peripheral bus clock. Faster tick rates (smaller pre-scaler) will give a very high resolution, but a low elapsed time between rollovers. Longer time rollover times can be achieved with a large pre-scaler, but at the cost of less resolution per tick of the timer.

Note that you can make an arbitrarily large timer by using the interrupt on rollover to increment another variable to count integer rollovers (See Sec. 1.1 below for more details on this). Because the timer count is always an unsigned value, subtracting the previous time from the current time to generate elapsed time will always give the right value even if a rollover has occurred between the two events (if you can have multiple rollovers, you will need to account for them).

1.1 Free Running Timers

As stated above, you will often need to time events to determine how much time has elapsed or the duration of some event. There are some specific hardware subsystems that you can use (more on this later), they are limited in how many you have available to you.

Often, you will need a generic sense of time that you can use in your code, and the scale has to be appropriate to the application. Again, timers are limited resources, and you will run out of them quickly, so using a single one as a master clock that you can refer to is quite useful.

This is usually called a free running timer (FRT), and as the name implies, it simply counts up continuously from reset. The basic idea will be to set up the timer to tick at some relatively high rate, and have it roll over at a convenient period.



Info: On the PIC32 there are two (really three) kinds of timers: “A” and “B.” Both sets of timers are 16bits wide. The main difference is two “B” timers can be ganged together to make a 32bit wide timer (at the cost of using up two timers). The other big difference (especially for you) is that “B” timers have a larger range of pre-scalers than the “A” timers do. See the Datasheet and FRM for more info.

On this version of the PIC32, only Timer 1 is an “A” timer, Timers 2-5 are all “B” type. There is an addition core timer (accessible only through assembly instructions), but this is not commonly used.

The requirements for your free running timer (outlined in `FreeRunningTimer.h`) are:

1. Use the Timer5 subsystem.⁴
2. Timer should roll over every 1 millisecond.
3. Timer should be configured (pre-scaler) to have a tick time that is an integer division of microseconds.
4. millisecond and microsecond counts should be module level unsigned integers
5. Create wrapper functions to report out the milliseconds and microseconds from the module.

The reason for having the timer set (using the pre-scaler) to have a tick time that is an integer division is such that you can have an accurate microsecond count. For example, if every tick is $\frac{1}{6}\mu s$ then every six ticks equals one microsecond. That is, you want $1\mu s$ to be an integer number of ticks.

⁴There are five timers available on the Uno32, you could use any of them for the free running timer, but some of the timers are slaved to other hardware functions that you will use later. Rather than having you re-implement the free running timer on a different timer, we are opting to just tell you which one to use such that it won't be a problem further on.

Note that the microsecond count and the millisecond count are both free running unsigned integers. Thus the microseconds will roll over every 1.19 hours, and the milliseconds will roll over every 49.7 days. If you were to need longer duration than that, you would need to use long ints (uint_64t) rather than ints.

If the timers is configured to overflow every 1 millisecond, then the interrupt is fairly simple; on rollover, increment the millisecond timer by 1, and the microsecond timer by 1000. The wrapper functions to return milliseconds and microseconds are also quite simple. The `FreeRunningTimer_GetMilliSeconds()` function simply returns the internal module variable. The `FreeRunningTimer_GetMicroSeconds(void)` is slightly more complicated, requiring you to return the sum of the current timer count (divided appropriately) and the module variable.

As in the protocol lab, finding the correct interrupt syntax for the compiler can be a bit of a challenge, so we give it to you here:⁵

```
void __ISR(_TIMER_5_VECTOR, ipl3auto) Timer5IntHandler(void) {
```

Prelab Part 1 (Free Running Timer)

In your prelab, document everything you will need to set up Timer 5 such that you can hit the target specifications:

- Timer5 rolls over every 1 millisecond
- Timer5 ticks are an integer divisor of microseconds
- FreeRunningTimer is a 32bit module level variable

This should include all appropriate registers, and what values you will put into said registers. Show pseudo-code or diagram out your thoughts on how to implement for the FRT module, and show any calculations done for Timer 5 setup.

A good debugging technique here is to toggle an I/O output pin in the interrupt and check on the oscilloscope that you are getting the correct frequency. You can also toggle an LED just to ensure you are getting into the interrupt (always a good first step) but you won't be able to tell the timing from the LED.

1.2 Elapsed Time and Duration

In Sec. 1.1, you have coded up the free running timer module such that you can get the elapsed milliseconds and microseconds since the microcontroller was reset.

If we are using the FRT to trigger some timed event, we need to know how to evaluate elapsed time or to compute the future readings at some desired time interval. Remember that we have 2 independent (but synchronized) elapsed time interval counters which are retrieved using the functions provided in the module header: `FreeRunningTimer_GetMilliSeconds()` and `FreeRunningTimer_GetMicroSeconds(void)`.

The first important detail to note is that both of these are unsigned integers, which have a range of 0 to 4,294,967,295. This gives a rollover of 1.19 hours and 47.9 day. However, since they are unsigned integers,

⁵Again, if you cannot see the predefined names in the MPLAB-X compiler (using the CTRL-space keys) then you need to `#include sys/attribs.h`.

the computation of *current-previous* will yield the correct result even if a (single) rollover has occurred. Thus if you want to time an event that is sooner than 1.19 hours in duration, and you want to know its elapsed time in microseconds, you would record the microsecond counter when the event started, and then record it again once the event ends. The elapsed time is: $\Delta T = T_{current} - T_{prev}$.

Likewise, if you only need millisecond level precision, you would do the same thing, except that you would use the millisecond counter rather than the microsecond one.

Another use for the FRT is to schedule some action for a future time. In this case, you need to be able to calculate the value of the FRT at some future time. The way to do this is to calculate the ΔT you want (in either milliseconds or microseconds as appropriate) and add it to the current counter value.

For example, let's say you wanted to trigger some event precisely 1.5msec into the future (or $1500\mu s$). You would read the microsecond counter, add 1500 to it, and set that as your target.



Warning: When testing if you have reached your target, it is very important to use the \geq test rather than the $==$ test. This is because you *might* miss the exact microsecond when testing, and you want to trigger as soon as you can after the time has elapsed.

How would you handle an event that you wanted to time to microsecond level timing, but lasted more than 1.19 hours (i.e.: the counter rolls over twice)? The answer is to test the millisecond counter AND the microsecond counter. That is, you want the correct number of milliseconds and the correct number of microseconds. A caveat here is that if you are looking at the milliseconds, you are really only interested in the 1000 microseconds that make up each millisecond. That is, you are going to use the modulo ($\%1000$) into the comparison to enforce that the count is within the integer millisecond.

For example, to set the target for $5,214,635,245 \mu s$ (1.45 hours) into the future, you would use the code as shown in Listing 1. Of course, for it to work, you have to be checking the conditional at least every millisecond.⁶

Listing 1: Elapsed Time example

```
unsigned int targetMilli, targetMicro;
targetMilli = FreeRunningTimer_GetMilliSeconds() + (5214635245 / 1000);
targetMicro = (FreeRunningTimer_GetMicroSeconds()%1000) +
               (5214635245 % 1000);
...
if ((FreeRunningTimer_GetMilliSeconds() >= targetMilli) &&
    (FreeRunningTimer_GetMicroSeconds()%1000 >= targetMicro)) {
    // do the thing
}
```

1.3 FRT Test Harness

While you have been coding up the FRT, you should have been writing your test harness along the way (you have been, right?). The requirements of the test harness are that you exercise the FRT module to show that it

⁶Note that this is a bit of a ridiculous problem, but it would show up if you had 16bit integers rather than 32bit integers (which on other platforms you might). If you find yourself in a situation where you need microsecond level timing over a span of more than an hour, you are in a very strange place, indeed.

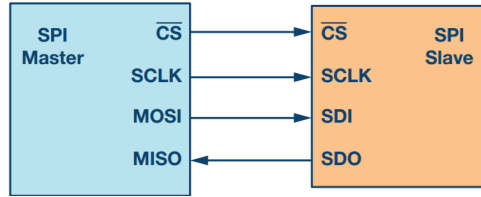


Figure 1: Typical SPI configuration

works. You will be using your protocol software library to interface with the *ECE121 Console* to demonstrate that you are able to see the output.

You will also be strobing a pin (and LED) so that you can check your timing with the oscilloscope.

Because you used the conditional compilation for the Protocol test harness, and the `#define` was in the project configurations, you can directly include `Protocol.h/.c` in your new `FreeRunningTimer` project with no alterations; it will all just work. This is why we force you to do it this way.

The requirements for the FRT test harness are:

1. Output an `ID_DEBUG` that includes the module name, date and time of compilation
2. Set your code to print out every 2 seconds (2000 msec)
3. Flash an LED and toggle an external I/O pin at 0.5Hz
4. Output an `ID_DEBUG` that includes the millisecond and microsecond count

Again, the test harness must be embedded into the `FreeRunningTimer.c` file, and use conditional compilation from the preprocessor to run it. Use the MPLAB-X configurations such that only inside the `FreeRunningTimer` project does it compile the test harness.

Optional: if you want to be more elaborate (and have a better test harness), you can also monitor one of the I/O pins (set as input) and report the duration of the high time in microseconds (using an `ID_DEBUG` message). Only output the `ID_DEBUG` message at some fixed rate (1Hz). Use a signal generator to drive your input pin and vary the frequency; see if you are getting correct values back. Note that it is likely that the oscillator on the microcontroller is not exactly the same as the source for the signal generator, but they should be quite close.⁷

2 SPI: Serial Peripheral Interface

Serial peripheral interface (SPI) is one of the most widely used interfaces between microcontroller and peripheral ICs such as sensors, ADCs, DACs, shift registers, SRAM, and others; it was originally developed by Motorola in the early 1970's. An excellent reference on the SPI bus is the *ApNote* from Analog Devices; you are strongly recommended to read it before attempting to do this lab.

SPI is a widely used bus interface because it allows a high data rate transfer between devices, and is rather versatile in how you can configure it. While it is a *de facto* standard, it does not have any formal standards;

⁷This gets into a squishy area of dealing with microcontrollers: what exactly is the *truth*? You can have your microcontroller output a square wave and then use an oscilloscope to determine how accurate your onboard clock is, but that depends on the oscilloscope being accurate. When was the last time it was calibrated? Who did so and what time base did they use? As you get farther and farther down the rabbit hole, you have to decide ultimately which time base to trust (**hint**: one of the best is the GPS PPS time).

Table 1. SPI Modes









Mode	Polarity (CPOL)	Phase (CPHA)
0		
1		
2		
3		

Figure 2: Table of SPI modes

thus each device requires looking at its datasheet to determine the exact configuration required. This can be difficult when communicating to multiple devices (especially from different manufacturers).

SPI is a synchronous, full duplex master-slave-based interface. The data from the master or the slave is synchronized on the rising or falling clock edge (see Sec. 2.1). Both master and slave transmit data at the same time. SPI consists of a four (unidirectional) wire bus, with a Master Clock (CLK), Master Out Slave In (MOSI), Master In Slave Out (MISO), and a Chip Select (\overline{CS}). Note that these are the standard names, but several manufacturers have different names for these same canonical lines. Also note that SPI requires a common ground (absolute voltage signaling).

The Master generates a clock, and shifts data out on the MOSI line, while the slave shifts its data out on the MISO line (see Fig. 1). At the end of clock strobes, data from the master has shifted to the slave, and data from the slave has shifted to the master. If there are more than one slave, then a slave select line is used to indicate which device is active (only *ONE* slave can have its \overline{CS} line low at a time). See the Family Reference Manual Section 23 on Serial Peripheral Interface for specific information on the PIC32 implementation.

SPI is often implemented internally as a simple shift register that clocks data out and back in. It is simple to set up, and quite fast when running. For short range serial communications, it is a very good choice. Like other peripherals available on the Uno32, the SPI subsystem is powerful, and fully configurable. For this lab, we will be using the SPI subsystem to configure the AMS encoder into pulse width mode (see Sec. 3.3).

2.1 SPI Modes

In a synchronous interface, the data must be stable on the clocking edges (rising, falling, or both). Additionally they must be stable before and after the edge as given by the specs of the chip. In a single edge system, a very good strategy is to change the data on the opposite edge; many early processors did exactly this. As long as the data is stable at the required edge, the data can change at almost any time. The SPI interface defines four transmission modes, defined by the clock phase and polarity (see Fig. 2). The PIC32 is capable of operating the SPI subsystem in any mode.

For the most part, the slave cannot be configured and can only operate in one mode (sometimes it can operate in up to two different modes), this it is incumbent on the master to configure to operate in the correct mode.

In SPI the master can select the clock polarity (CPOL) and clock phase (CPHA). Clock polarity indicates the clock idle (high or low), that is what is the clock before beginning an SPI transaction. Clock phase indicates which clock edge (rising or falling) is used to sample data and shift. These modes are defined as:

- Mode 0: Clock idle low, Data sampled on rising edge
- Mode 1: Clock idle low, Data sampled on falling edge
- Mode 2: Clock idle high, Data sampled on rising edge
- Mode 3: Clock idle high, Data sampled on falling edge

Note that clock idle is defined as the state of the clock line (high or low) when the slave select (\overline{CS}) is asserted. This slave select assertion also tells the slave device to get data ready. The Wikipedia article on SPI is informative, and includes some details about the specifics of the PIC32.

The AMS encoder (see datasheet) we will be using requires SPI mode 1 (CPOL = 0, CPHA = 1). Data transfer starts with the falling edge of the slave select (\overline{CS}); the encoder samples the MOSI line on the clock falling edge, and SPI commands are executed at the end of the frame (when \overline{CS} rises). The bit order is MSB first, and data is protected by parity (see Sec. 2.2).

2.2 Parity Bits

The AMS encoder requires a correct parity bit in order to accept any SPI message. The parity bit must be correct or the SPI transaction will be ignored by the AMS encoder.

A parity bit is a simple check that the bits are odd or even and is a common check used in serial communications. In the case of the AMS encoder, the parity bit is the *MSB*. Since this is not included in the SPI hardware, you will need to calculate the parity bit yourself and insert it into the MSB of the SPI shift register. The wikipedia article on parity is a bit light, but has an OK explanation. The AMS encoder requires *even* parity, and will ignore any SPI transaction that does not have the correct parity bit (see the AMS datasheet on CANVAS).

The algorithm for implementing the parity bit is straightforward and shown in pseudo-code in Algorithm 1.

Algorithm 1: Function to check parity

Input: in , integer to check parity

Result: p , parity bit, 0 or 1

```

 $p \leftarrow 0$  ;
for  $i \leftarrow 1$  to 0x8000 do
    if ( $in \ \& \ i$ ) then
        |  $p \leftarrow p + 1$ 
    end
     $i \leftarrow (i \ll 1)$  ;
end
 $p \leftarrow (p \% 2)$  ;
return ( $p$ ) ;

```

Insert Parity Bit into Payload: $in \leftarrow in | (p \ll 15)$;

Note that before sending *any* packet to the AMS encoder, you will need to insert the parity at the top of the transmission packet (as detailed in the last line of Alg. 1). Again, for more details look at the AMS datasheet provided.

Code up and test your parity function (it is a private function that should be inside `RotaryEncoder.c`). Hand calculate a few values and test the parity; see if your code produces the same output.

2.3 SPI Initialization and Testing

Before you can use the SPI subsystem, you must initialize it. This consists of first disabling the SPI subsystem, then configuring the SPI subsystem using the SPI control registers (again, you should be able to use the `_bits` version of the registers inside MPLAB-X, e.g.: `SPI2STATbits.XXX`). Set the slave select pin to an output, and set it high.⁸



Warning: You will need to designate an I/O pin as the slave select pin (using `#defines` inside your code to keep it clean). There is a hardware driven slave select provided by the SPI subsystem but you will not be using it, you will need to create your own slave select and control it.

At that point, re-enable the SPI subsystem. Note that for this lab we are **NOT** using the SPI interrupt; rather we will be blocking with the SPI for this lab (that will change in Lab 4). Also, because SPI 1 is multiplexed with the serial port, we will be using SPI 2.

The SPI init code resides inside your `RotaryEncoder_Init()` function.

In order to transmit (and receive) a value on the SPI bus, the transaction consists of:

1. Compute the parity of your packet (device specific).⁹
2. Insert the parity bit in the MSB location of your packet (again, device specific).
3. Set the Slave Select (\overline{CS}) output low.
4. Copy the packet into the appropriate SPI buffer (`SPIxBUF`).

The SPI subsystem will drive the clock as soon as it has data in the SPI buffer. When the transaction is ended, the data sent by the slave device will be in the SPI buffer (as bits are shifted in and out as the clock is strobed). You can determine that the transaction is over by reading the SPI status register. Once the transaction is done, raise the Slave Select line.

In lab, write the code that initializes the SPI2 subsystem to Master, 5MHz, mode 1, and 16 bit transfer at a time. Hook a scope up to the MOSI and SCLK lines. Transmit various short ints (e.g.: `0xAA`, `0x55`, `0xF0`, `0x0F`, etc) that have bit patterns that you can recognize, and see if you can pick them up on the scope.¹⁰ Include an image of the scope trace in your lab report.

Either ground or tie MISO to 3.3V, and inspect the `SPI2BUF` register after the transaction; you should get either `0x00` or `0xFF`. It is a bit too hard to try to get the value to change during the very short time a transaction takes, so just keep MISO constant. Once you have done this and verified that you get `0x00` or `0xFF` as appropriate, you can hook MOSI to MISO and conduct a *loopback* test. That is that `SPI2BUF` should have the same value in it that you sent.

⁸Many devices do not need \overline{CS} and it can be tied low. Note that this is **NOT** the case for the AMS encoder—you will need to drive the \overline{CS} line actively to talk to it.

⁹Not all devices require parity on the SPI transaction. The AMS encoder does, and uses *even* parity.

¹⁰This is where a logic analyzer comes in really handy.

This set of tests should verify that the SPI subsystem is configured correctly and you are able to transmit and receive data through the SPI. Make sure that you include an image of the oscilloscope trace in your lab report (if you can get your hands on a logic analyzer, that is acceptable as well).

Prelab Part 2 SPI initialization

How are you going to configure the SPI subsystem (SPI 2)? List all registers that you will use to configure the SPI to:

1. Clock Speed 5MHz
2. SPI Mode 1 (Clock idle low, Data sampled on falling edge)
3. 16 bit mode
4. PIC32 is Master

Also include information about how you will determine that the SPI transaction is over (again, listing register and values to test).

3 Encoders

Background: Encoders come in two basic varieties, the incremental and the absolute, and are both used to measure angular displacement. Incremental gives a relative angle from the past measurement, and absolute from a fixed angular reference.

Note that the vast majority of encoders are incremental, but give no sense of direction (simply a pulse train whose frequency corresponds to speed). These are used everywhere (e.g.: wheel speed sensors on most modern vehicles), and assume that only speed is important (or that you have another method for detecting direction of rotation).

When direction (CW or CCW) and angle are both important, then a subset of the incremental encoder, the Quadrature Phase Encoder is used.

Quadrature Phase Encoders are used to measure angular displacement. Wikipedia has a decent reference on incremental encoders, and TI has an appnote that is quite thorough. Quadrature Phase Encoders are used in a wide variety of applications, from joint angles in robotics and keeping track of wheel rotation for navigation (odometry), to keeping track of axes on CNC machines and tracking tank turret angles. They are also used to drive the commutation of BLDC motors, in order to keep track of the rotor (Permanent Magnet) position.

They are called Quadrature Phase Encoders because the signal consists of two square waves that are 90° out of phase with each other. By comparing the state of one of the square waves when the other is either rising or falling, you can generate a direction in addition to a count (or accumulated angle). Quadrature Phase Encoder's are also called incremental encoders, because they give you no indication of absolute angle, rather only accumulated angle from when you started measuring them. Some Quadrature Phase Encoder's have an index mark that pulses once per revolution to give you a reference angle.

The phases of the encoder are usually designated "A" and "B" and if there is an index mark, it is customarily called "Z" or "I." A typical trace of the outputs is shown above; it can be seen that if "B" is high when "A" is on a rising edge, the encoder is going in one direction, and if "B" is low when "A" is on a rising edge, then the

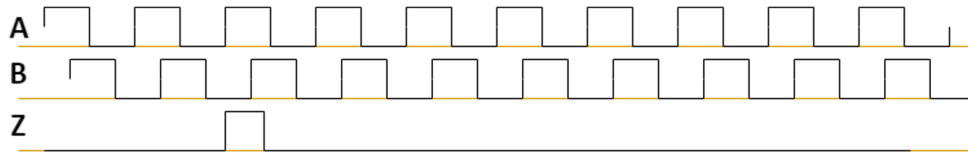


Figure 3: ABI Quadrature Incremental Encoder Pulse Train

encoder is going in the other direction. The beauty of the incremental encoder is that there is no noise, and the signal does not drift. This makes it an excellent sensor for angles and rotation rates.

Ignoring the index pulse, “Z,” the counting up and down can be formulated as a simple state machine from the values of “A” and “B” and triggered any time there is a change in either. Some microcontrollers possess a QEI subsystem that implements the state machine in hardware, but our variant of the Uno32 does not.

Absolute Encoder gives you an absolute position at any time (including startup). Typically they have lower resolution than quadrature encoders, and are also more expensive.¹¹ The Wikipedia article has a decent reference on absolute encoders. Absolute encoders are used everywhere that you need to know the precise position at turn-on; where sweeping the angle to find the index mark would possibly damage the mechanism.

The encoder that we are using, the Austria Microsystems AS5706D, is an unusual device that can be both an absolute or incremental encoder depending on what mode you set it in. It is a “system on a chip” type device that uses a magnet to sense the angle using hall effect sensors. It is relatively high resolution, and quite inexpensive. It has many modes that can be used to read the sensor depending on the application.

For this lab, you will be configuring the AMS encoder to output its absolute angular position using a pulse width modulation (PWM). That is, the device will output a pulse whose high time is proportional to the rotation. To configure the AMS encoder into pulse width mode, you will need to communicate to it using the SPI protocol.

3.1 AMS Magnetic Encoder

The AMS AS5706D is a system on a chip that uses a set of hall effect sensors to detect the orientation of a (special) magnet above the chip. It is a surface mount device, and is generally used as a high precision encoder (14 bits absolute) for myriad applications.

From the AMS datasheet:

The AS5047D is a Hall-effect magnetic sensor using a CMOS lateral technology. The lateral Hall sensors convert the magnetic field component perpendicular to the surface of the chip into a voltage.

The signals from the Hall sensors are amplified and filtered by the analog front-end (AFE) before being converted by the analog-to-digital converter (ADC). The output of the ADC is processed by the hardwired CORDIC (coordinate rotating digital computer) block to compute the angle and magnitude of the magnetic vector. The intensity of the magnetic field (magnitude) is used by the

¹¹Both the lower resolution and the increased cost come from the typical construction of the encoders. Absolute (optical) encoders need many more photodetectors than the incremental ones do. However there are several technologies (including the magnetic one used in this lab) that are bringing the price down and the resolution of absolute encoders up.

automatic gain control (AGC) to adjust the amplification level for compensation of the temperature and magnetic field variations.

The internal 14-bit resolution is available by readout register via the SPI interface. The resolution on the ABI output can be programmed from 2048 to 32 steps per revolution. The Dynamic Angle Error Compensation block corrects the calculated angle regarding latency, by using a linear prediction calculation algorithm. At constant rotation speed the latency time is internally compensated by the AS5047D, reducing the dynamic angle error at the SPI, ABI and UVW outputs. The AS5047D allows to switch OFF the UVW output interface to display the absolute angle as PWM-encoded signal on the pin W.

At higher speeds, the interpolator fills in missing ABI pulses and generates the UVW signals with no loss of resolution. The non-volatile settings in the AS5047D can be programmed through the SPI interface without any dedicated programmer.

The AMS can be configured into many modes, depending on the application, and uses SPI for native communication and configuration. The sensing elements for the encoder are within the chip, on the stationary part of the device. The magnet rotates and provides a signal to the chip. This allows the magnet to be put on the other side of a membrane to seal in the encoder, allowing great flexibility when using it in harsh environments.

The AMS includes circuitry to compensate for small misalignments between the magnet and the sensing elements, as well as several self-test functions. In this lab, the AMS encoder will be read using SPI messages in a blocking fashion. You will use the SPI system to read out the angle and possibly also reading the error messages from the encoder using SPI.



Warning: Please read the datasheet for the AMS encoder carefully. The reply to SPI transactions is delayed until the *next* transaction. We are aware that in this lab, your SPI transactions will be blocking code. This will change later in the class in Lab 4.

The AMS encoder is quite capable, and can do much more than we are using it for in this lab. It remains a nice example of an encoder with multiple different ways to interface with it. For details on how to initialize the encoder, see Sec. 3.3.

Note that the pins for the encoder are labeled in such a way as to *match* the pinouts of the master (e.g.: MISO→MISO). Also, the supplied angle pins will cover the labels on the lower pins if you put it in the obvious way. If you insert the angle pins from underneath the board, this is not an issue (see TAs for how to do this).

3.2 Encoder SPI Transactions

In Sec. 2 you learned how SPI worked, and went as far as getting SPI to run and traced the output on the oscilloscope. Now, we will learn how to interface specifically with the AMS encoder.



Info: SPI does not have a specific standard that all manufacturers adhere to. Rather, it specifies a range of clock speeds, how the wiring is set up, and a choice of modes based on which clock edge to sample on. As such, each device will have its own specific implementation which the SPI master needs to accommodate (this can make interfacing with multiple devices from different manufacturers challenging)

The AMS encoder has a maximum SPI clock rate of 10MHz, can only act as an SPI slave, has a 16 bit data field, and operates in SPI Mode 1 (Clock idle low, Data sampled on falling edge). That is, data transfer begins on

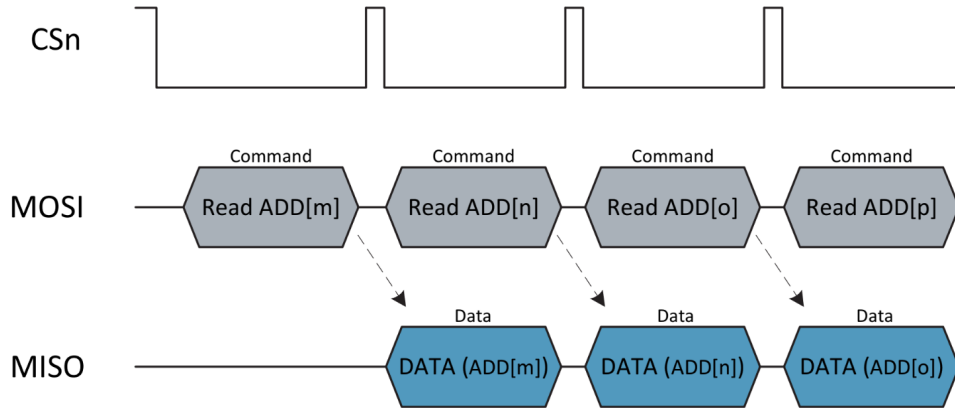


Figure 4: AMS encoder data read using SPI

the falling edge of \overline{CS} (clock idles low). The encoder samples the MOSI data on the falling edge of the clock, and every data frame must be ended by raising \overline{CS} . Note that \overline{CS} must be held high for a **minimum** of 350nS before the next frame.



Warning: The SPI transactions need to have the correct parity (see Sec. 1), as computed in Alg. 1. If you do not get the parity correct in the correct MSB, then the encoder will reject your commands.

The AMS encoder implements *command* and *data* frames, each of which are handled a bit differently. Command frames are constructed as $[\text{parity_bit} \text{ R/W_bit} \text{ Address (bits 13:0)}]$ (see Fig. 13 in the AMS datasheet). The parity bit (as detailed in Sec. 1) is inserted into the MSB (bit 15) and must be the *even* parity of bits [14:0]. Note that if the parity bit is wrong, the command will not be accepted.¹²

For the data frame, the AMS encoder calculates the parity bit and inserts it into the MSB of the data (it also inserts an error flag if the *previous* command had a parity error). The actual data is in bits [13:0] of the data frame.

Note here that *every* SPI transaction with the AMS encoder requires at least two SPI transactions; one for the address, and one for the data. This is due to the fact that the encoder cannot know what address you are sending, and thus cannot know what to send back until after it has received the SPI transaction. This is a bit subtle, and is somewhat easier to understand for the data read frames. We will go through two examples below.

3.2.1 AMS Diagnostic Message

Referring to Fig. 4, the encoder must receive a command with the address that is to be read, then *another* frame must be transmitted so that the encoder can answer with the actual data. Since the SPI master generates the clock, the slave (encoder) cannot send the data unless the master is transmitting. Hence the need for a second frame. Also note that the \overline{CS} must be strobed (raised and lowered) between frames.

¹²The parity error will change a flag in the error register, but realistically if you are getting the parity wrong, you won't be able to read the error register either.

For example, let us consider the DIAAGC register (0x3FFC); this register contains diagnostic codes indicating if the magnetic field strength is too low (magnet too far away), too high (magnet too close), overflow status, offset compensation status, and the value of the automatic gain control.

Assuming that your SPI subsystem is already set up as in Sec. 2.3, the following steps are required to read the DIAAGC register from the encoder:

1. Write an output low to \overline{CS}
2. Send SPI command: 0xFFFC (parity is 1, R/W is 1)
3. Write an output high to \overline{CS}
4. Discard SPI data that came in, wait 350nS
5. Write an output low to \overline{CS}
6. Send *NOP* command on SPI: 0xC000 (parity is 1, R/W is 1)
7. Write an output high to \overline{CS}
8. Read the diagnostic codes in the SPI data that was returned



Warning: Do **NOT** just blindly copy (and modify) these examples. Take the time to understand the transactions and how they are occurring. This will greatly improve your understanding and ability to use microcontrollers in the future.

The *NOP* command is special in that it requests no operation from the encoder. It must be read (hence the R/W of 1) and has an address of 0x0000. The *nop* is used to transmit something to the encoder so that it may respond with the requested data without giving the encoder an additional command (requested in the previous frame).

Note that any other data frame can be read in a similar manner. For example, the ANGLE register (0x3FFE) reports back the angle without the dynamic angle error compensation. This would be read the same way as the DIAAGC register, but you would send the initial SPI command of 0x7FFE (parity is 0, R/W is 1). All the rest would be the same.

When reading data, it is entirely up to you if you wish to check the parity bit, or simply discard the top two bits of the received data and assume the transaction was completely valid.¹³

3.2.2 AMS Command Message

Commands (writing) are inherently more difficult than the reads. This is for two main reasons, the first is that the transaction necessarily consists of one more than reads because you need both the address of the register you wish to write to, and the data you wish to write into that register. The second difficulty comes in verifying that the data has been successfully recorded by the device.

If you examine Fig. 5, you will immediately notice the extra transaction required. You will also see that the first response to the address is the *current contents* of that register. The new contents (hopefully echoing the data you sent) will be sent on the *third* transaction. Note that if the parity was wrong, the data you read back will have the EF flag (a high bit in position 14 of the reply).

¹³Obviously checking the parity bit is more robust than not, but for this lab you don't need to be that careful. However, if this encoder were being used in an application that was safety critical (e.g.: car steering) the you would implement the parity check.

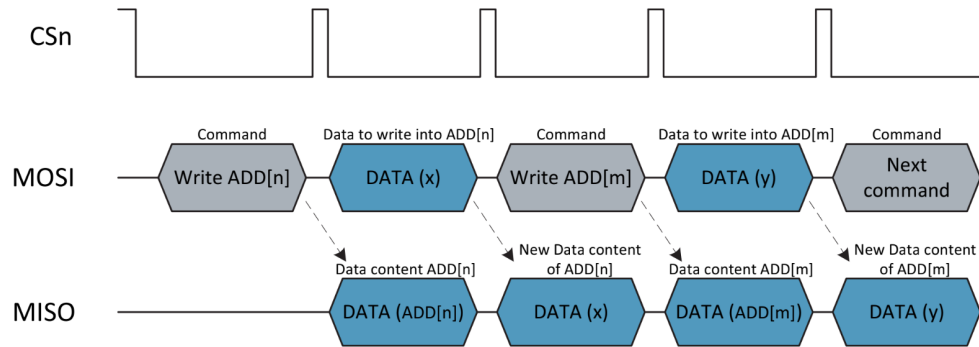


Figure 5: AMS encoder command write using SPI



Info: Note that there are two distinct ways to verify that the command was accepted: (i) Rely on the fact that the AMS encoder responds with the new data on the third transaction (as in Fig. 5); or (ii) After the write, resend the same address with a read request (1 in the R/W field) and read the data as if it were a normal data frame.

Thus, for example, if you wish to set up the AMS encoder as a quadrature phase encoder with 800 steps per revolution, with no hysteresis, then you would write to SETTINGS2 register (address 0x0019). Examine Fig. 29 of the datasheet, and convince yourself that the contents of SETTINGS2 should be set to 0x0070 (UVWPP = 000, HYS = 10, and ABIRES = 011).

To set this up, you would undertake the following transactions:

1. Write an output low to \overline{CS}
2. Send SPI command: 0x8019 (parity is 1, R/W is 0)
3. Write an output high to \overline{CS}
4. Discard SPI data that came in, wait 350nS
5. Write an output low to \overline{CS}
6. Send data for configuration on SPI: 0x8070 (parity is 1, R/W is 0)
7. Write an output high to \overline{CS}
8. Read the previous settings for SETTINGS2 in the SPI data that was returned
9. Wait 350nS, write an output low to \overline{CS}
10. Send NOP command on SPI: 0xC000 (parity is 1, R/W is 1)
11. Write an output high to \overline{CS}
12. Verify the SETTINGS2 was written correctly in the SPI data that was returned

Note that these settings will last until the AMS encoder is power cycled, and then it will return to its default settings (for the ABI interface, 2000 steps per revolution).



Warning: Do not, under any circumstances, attempt to permanently program your settings into the AMS encoder. It is very easy to do this wrong and the chip will not recover. Leave it with its factory defaults. This section is provided for completeness *ONLY*; we do not expect you to set the encoder to anything but its default modes.

3.3 Encoder Initialization

Given the rather detailed instructions in Sec. 3.2.1 and Sec. 3.2.2, you are now ready to initialize the encoder. In order to set up the encode you must configure the SPI subsystem (see Sec. 2.3). Note that you should be able to look at the datasheet and determine what settings you need to change to do this.

Use the SPI subsystem to read the encoder angle and verify that you are seeing roughly same angle as you are reading on the SPI (yes, we know you are going to block when reading the SPI bus). Once this is completely done you can optionally read the error diagnostics using the SPI and verify that you are able to read the correct errors when you remove the magnet. Do NOT even debate attempting to do this until you can reliably read the angle.

Prelab Part 3 Encoder initialization

How are you going to configure the AMS encoder to use (blocking) SPI transactions?:

1. What kind of errors can you detect (pseudo-code for parsing the diagnostic message)?
2. Hand calculate the entire SPI transaction (Hex is OK, binary is better)

As usual, the more detail you include here, the easier it will be for you to do it when the time comes.

You will need to code up the `RotaryEncoder_Init()` function, which needs to set up the SPI subsystem in order to be able to access the AMS encoder using *blocking* SPI transactions. This means that the `RotaryEncoder_Init()` function is called with the `ENCODER_BLOCKING_MODE` parameter in order to put it into that mode (and also for the test harness to know how it is accessing the encoder).

You can use the protocol messages `ID_ROTARY_ANGLE` and to report the angle (raw 14 bits) to the *ECE121 Console* application for testing. Note that you can always use the `ID_DEBUG` and `sprintf()` to print anything you like to the console.

3.4 Encoder Test Harness

The encoder test harness consists of configuring the encoder for SPI (blocking) angle reads, and then to continually read the encoder position, and report back the encoder value. This is done with the `ID_ROTARY_ANGLE` message in the protocol. The *ECE121 Console* includes a graphical wheel what shows the encoder angle graphically as well as reading out the actual number.

Before you enter your `while(1)` loop, you should read the SPI angle message and report back the angle using an `ID_DEBUG` message. Once you have done this, enter your `while(1)` loop and report back your position (based on blocking SPI read) periodically (100Hz).¹⁴

Note that in order to do a blocking encoder read, you will have to set up an SPI read command (correctly formatted for the AMS ANGLE register), and then write it to the SPI2BUF. When the SPI receive buffer is full (check by polling), then the SPI buffer contains the *previous* angle. Each time you send another angle read command to the AMS, it responds with the previous angle.

¹⁴You should be using NOP delay loops in your test harness, rather than the free running timer.

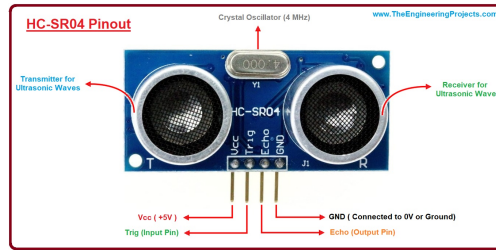


Figure 6: HC-SR04 Ultrasonic Ranging Sensor

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 `RotaryEncoder.h/c` module, and turned on using a preprocessor macro.

Demonstrate your test harness to the TAs using the Lab2 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.

4 Time-of-Flight Sensors

The second sensor used in this lab is a time-of-flight ranging sensor called a “ping” sensor (see Fig. 6). Time of flight sensors work by injecting some form of energy into the environment and measuring the time of flight between the emitter and the reflection (echo) from the object. The kind of energy and the method of measuring the time of flight differentiate the various sensors, ranges, costs, and accuracies. These sensors are used widely in a variety of applications, from mm ranges (VCESLS) to 100s of kilometers (RADAR). Typically, light and sound (at various frequencies) are used as the source. For ranging, techniques range from simple timers, signal strength, frequency matching, and interferometric techniques. A decent background document (available at: https://en.wikibooks.org/wiki/Robotics/Sensors/Ranging_Sensors) shows many of these techniques and how they work.

Fundamentally, all time-of-flight sensors measure distance by multiplying the time between sending the pulse and measuring the return by the speed of flight in the medium. That is: $d = 0.5 \times v \times \Delta t$ (the 0.5 is to account for the fact that the echo covers the distance there and back. The specific range sensor in this lab uses ultrasound ($\sim 40\text{KHz}$) way above human hearing, and uses a chirp function (both frequency and amplitude modulated) to get better discrimination on the return timing. They are low cost (you are perhaps sensing a theme here), and generally work quite well.

4.1 HC-SRO4 Ping Sensor

The HC-SRO4 Ping Sensor is a low-cost embedded device with a discrete transmitter and receiver operating at the ultrasonic frequency range. Internally it uses a set of ultrasonic pulses, modulated both in frequency and amplitude. Looking at the timing diagram in the datasheet¹⁵, you can see there are several things happening that lend themselves to using the timer and the input capture subsystem of the Uno32 to get it to work. The first thing to note is that the trigger input needs to be raised for at least $10\mu\text{s}$ and then lowered again. Following this, the module will transmit a burst of ultrasonic pulses, and will generate an output pulse whose high time is proportional to the distance (they specify the time of flight as 340m/s). The datasheet also specifies the

¹⁵https://cdn.sparkfun.com/assets/b/3/0/b/a/DGCH-RED_datasheet.pdf

maximum range of 400cm (corresponding to $11765\mu s$), and a delay of at least 60ms between triggers (or a max repetition rate of 16.667Hz). Note as well that the output pulse may not come at all if the device cannot generate a valid echo timing.

Thus, your code must (roughly): raise the trigger line and lower it $10\mu s$ later, set yourself up to 60ms later start the ping cycle again. On the input capture interrupt, read the input capture buffer, and use this to calculate time of flight. Your user code will convert time of flight to distance.¹⁶ You will be implementing this in non-blocking, interrupt driven code within the PingSensor.h/.c module.¹⁷ Note that the way we are running the sensor, it is always running and the distance you are reading is the last one you read (held in a module level variable). We are going to specify that you use Timer4 to run the trigger using the interrupt service routine prototype below.

```
void __ISR(_TIMER_4_VECTOR) Timer4IntHandler(void)
```

Prelab Part 4 Ping Sensor Code

How are you going to trigger and capture the echo on the ping sensor:

1. Which pins will you use to trigger the device, and how will you ensure at least $10\mu s$ high time?
2. Describe (in detail) how you will set up Timer4 to give you a repeating 60ms pulse.
3. Diagram out the code and how it will run (in terms of init, then ongoing sampling).
4. Set up the init function to get everything ready for sampling the ping sensor (pseudo-code).

As usual, the more detail you include here, the easier it will be for you to do it when the time comes.

Thus, at this point you should be able to write the PingSensor_Init() function to set up your I/O pins accordingly, set up Timer4 to interrupt every 60ms, and use the Timer4 ISR to raise the trigger pin for a minimum of $10\mu s$. You should be able to verify that your code is correctly sending the trigger pin using a logic analyzer or oscilloscope. Once you have verified this, hook up the actual ping sensor. You can monitor the echo pin (again with either the logic analyzer or scope) and see it vary as you move a target closer and farther from the sensor.

Note that you should be able to calculate the range from the high time of the echo pin by hand and verify that it makes sense. One method to get consistent returns is to point the sensor at a wall and move the sensor closer or farther from the wall.

4.2 Input Capture

To accurately measure the pulse width generated by the HC-SRO4 ping sensor, we are going to use the input capture subsystem of the Uno32. Input capture works by automatically copying the clock to a capture register when an external event occurs (in this case, a certain pin changing state).

The Family Reference Manual Section 15 on Input Capture details the operation and registers involved (see Fig. 7). There are relatively few input capture pins available; in this lab, we will be using IC3 which can be slaved to either Timer2 or Timer3. For the lab and this manual we will assume it is slaved to Timer2.

¹⁶Any of you who decide to do a floating point multiply by 0.5 or floating point divide by 2.0 will immediately fail the class. And we will go back and retroactively fail you for CE13 as well. Remain in integer math as absolutely long as possible and don't use floats for simple calculations.

¹⁷Again, read the comments of the header file carefully as many of the specifications are there.

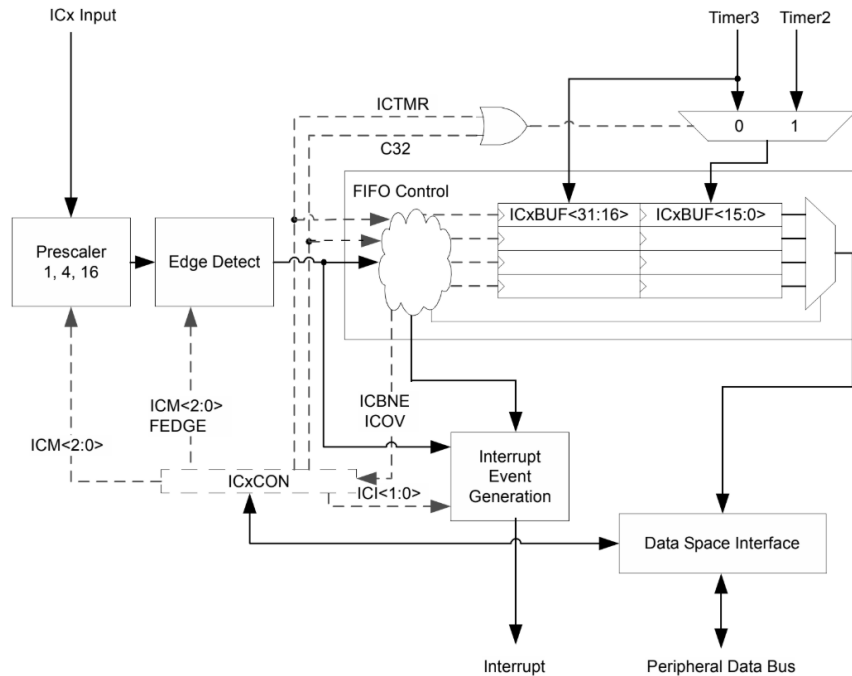


Figure 7: Input Capture Subsystem Block Diagram

Using Fig. 7 as a reference, when the IC3 pin changes state, assuming that FEDGE is set to capture that edge, then Timer2 (TMR2) will be copied into the IC3BUF (the FIFO control makes sure that the data is appended rather than overwritten). The value can be read in the interrupt by reading IC3BUF.

Timer 2 is configured to count up and roll over at some fixed interval and tick at an appropriate rate using the pre-scalers. The IC3 is set to trigger on a every edge and will copy the value of the timer 2 register to a buffer when it triggers. In order to compute an elapsed time, you will need to record the previous (upgoing) trigger and the current (downgoing) trigger and subtract.



Warning: Make sure that the upgoing and downgoing measurements from the IC3BUF are *unsigned short* variables. Make sure that the resulting difference is **ALSO** an unsigned short (no intermediate ints). Lastly, there is a silicon errata that indicates that IC3BUF needs to have its upper 16bits are masked off (e.g.: `upgoing = (0xFFFF & IC3BUF);`). That is, IC3BUF is a full 32 bits, even though you only want the lower 16, and this is a hardware error.

First, to finish the `PingSensor_Init()` function, you will need to add the initialization and setup of the input capture subsystem. Thus, to your previous init function, add in:

1. Configure Timer2 (as much resolution as you can get with > 11.68msec rollover)
2. Configure the IC3 subsystem (trigger on every edge)
3. Configure the IC3 subsystem to first trigger on a rising edge
4. Configure the IC3 subsystem to interrupt on every edge
5. Turn on the IC3 interrupt priority level (required)
6. Enable the IC3 interrupt
7. Turn on Timer2

After getting the complete initialization working and tested, you should write the IC3 interrupt service routine and make sure that it is working (you can test it with a signal generator before trying it out with the actual ping sensor).

As with all previous peripherals, step into the complexity through incremental development. First simply blink an LED and toggle an I/O pin so you can hook up an oscilloscope and see that you get the same waveform.

As long as the interval is less than the Timer2 period, then the subtraction will give you the correct pulse width (in ticks). To convert that to distance you will need to do the appropriate math. This will be done inside the interrupt service routine:

```
void __ISR(_INPUT_CAPTURE_3_VECTOR) __IC3Interrupt(void) {
```

Make sure that your ISR can generate the correct time interval for different signals fed into the IC3 pin. Note here that the Input Capture subsystem has some issues when the signals get very short (this is due to the fact that the system cannot get through the ISR before the next signal hits). This will not affect you with the ping sensor, but it might happen in the future with different sensors.

Though unlikely, you might miss some pulses (and get out of polarity) with your ISR, and you will need to account for that. The way to do this is by testing if, in fact, you have the correct polarity on the IC3 pin. Briefly, if you think the measurement is a rising edge, then the pin should be high (read it and check). If that is correct, record the time into your “upgoing” variable. The next edge should be a falling edge. If the pin is indeed down when you are in the interrupt, then record the downgoing time and $\Delta T = \text{downtime} - \text{uptime}$.

If things fail (that is, you think it was a falling edge but the pin is high), you need to reset the computation so that it will recover on the next time. Also, be sure to clear the IC3BUF by continuously reading it until the buffer is empty (this will prevent a buffer overflow that will halt the interrupt).

Prelab Part 5 Input Capture initialization

How are you going to configure the Input Capture subsystem (IC3)? List all registers that you will use to configure the IC3 to:

1. Use Timer2, highest resolution possible with >11.68 msec rollover
2. IC3 uses Timer2 as its source
3. Trigger interrupt on every edge
4. Begin first trigger on rising edge
5. Priority level for IC3 cannot be 0

Every register you use should be called out.

Your code should be able to compute the distance reported from the ping sensor, using the IC3 interrupt and reporting the value back to a module level variable. Note that this is being handled entirely with ISR's (Timer4 to trigger, IC3 to read the echo), and simply runs continuously without further input from your code.



Figure 8: Typical RC Servo

4.3 Ping Sensor Test Harness

The ping sensor test harness consists of configuring the ping sensor trigger and input capture, and then to continually read the echo timing, and report back the distance value. This is done with the `ID_PING_DISTANCE` message in the protocol. The *ECE121 Console* includes a field for reading out the distance from the ping sensor.

Before you enter your `while(1)` loop, you should set up and start your ISR's and make sure to clear the Input Capture buffer. Once you have done this, enter your `while(1)` loop and report back your distance (based on the input capture read) periodically ($\sim 10\text{Hz}$).¹⁸

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 `PingSensor.h/c` module, and turned on using a preprocessor macro.

Demonstrate your test harness to the TAs using the Lab2 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.



Warning: Because the ping sensor works by reflecting sound waves back, it works better with a reasonably large flat target (for example, pointing it at a wall). If you are going to be moving something in front of the sensor to get distance, use something large and flat (e.g., a textbook would work well).

5 R/C Servo

An R/C servo is a small actuator that comes in a number of “standard” sizes and speeds. They are a small DC motor, compound gear train, feedback potentiometer, and a control circuit encased in a (usually) plastic case (see Fig. 8). The output shaft is splined, allowing various control horns to be attached. A simple but decent reference on RC servos can be found at http://www.societyofrobots.com/actuators_servos.shtml.

R/C servos are some of the lowest cost actuators you can find, largely because they are mass produced for toys. They are of generally low quality, both in terms of the gear train and the feedback, unless you are willing to spend a lot of money on them.

¹⁸You should be using NOP delay loops in your test harness, rather than the free running timer.

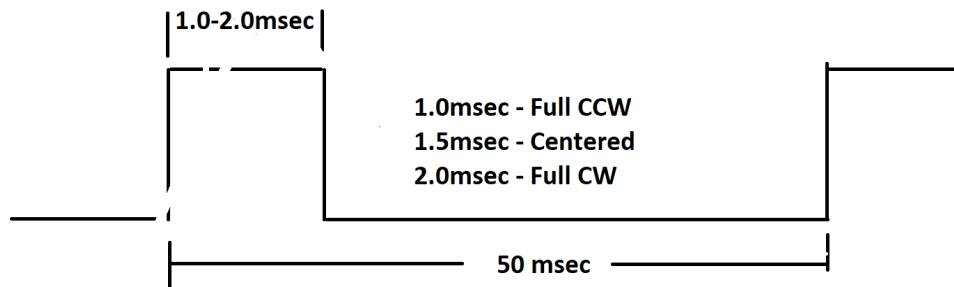


Figure 9: Pulse Width Command for RC Servo

Historically the servo is limited to a $\pm 60^\circ$ of rotation but most modern servos have a greater range.¹⁹ You might be able to get a bit more rotation out of them, but in general you don't want to overdrive them. For general health of the servos by staying within $1000 \rightarrow 2000\mu s$ on the pulse high time. Many servos, including the one in your lab kit, will go outside this range. Unfortunately most cheap servos do not have a datasheet and will not tell you their range.

There are both “analog” and “digital” servos, however they are driven in exactly the same way but have different internal drive mechanisms such that the digital one has a faster response (but at the cost of increased power consumption). From your perspective, the difference is irrelevant.²⁰

The servo has three wires; +5V power, ground, and a signal wire. There are some servos made that can take greater voltages, but unless you are certain, limit the voltage to 5V using either a power supply or a voltage regulator. Revisit Appendix B of Lab 0 on regulators if you are going to use a regulator.



Warning: The power rail on the servo must NOT exceed 5V. If you feed in higher voltage on the power you will permanently damage the servo. Do **NOT attempt to power this off the Uno, it pulls too much current for it.**

5.1 Signal Structure/PWM

The servo implements a closed loop control internally, driving the motor to the position commanded using the internal potentiometer to measure its actual position. Thus the servo will keep driving until it reaches the commanded position.

The signal wire uses a pulse width encoding to tell the servo the commanded position. The servo responds to the high time of the pulse: 1500 microseconds is centered, 1000 microseconds is full CCW, and 2000 microseconds is full CW rotation (see Fig. 9 for a drawing of the pulse width).

The pulse is repeated at 20Hz (or a period of 50 milliseconds). You will need to configure a timer to overflow every 50ms (it will have to be Timer 3 because you will be generating the high pulse using the output compare function, and Timer 2 is already used for the input capture).

¹⁹There are continuous rotation servo, and a wide variety of specialty ones, but all servos will have a limit.

²⁰There are some “digital” servos, such as the Dynamixels, that implement a proprietary serial protocol on the signal wire, however these are not the “R/C” digital servos you normally buy.

In order to use the output compare for driving the R/C servo, you will use Timer3 (we used Timer2 for the input capture in Sec. 4.2) and use the output compare function to drive the OC3 pin low when the correct time has elapsed. You will need to determine the appropriate pre-scaler and period match register (PR3) such that Timer3 rolls over precisely every 50msec (20Hz). You will need to configure the output compare subsystem to hold the OC3 pin high for the duration, pulse continuously, and set it to the correct value that will center the servo (e.g.: $1500\mu S$).

Prelab Part 6 Output Capture initialization

How are you going to configure the Output Compare subsystem (OC3)? List all registers that you will use to configure the OC3 to:

1. Use Timer3, rollover precisely every 50msec
2. OC3 holds pin high for duration
3. OC3 pulses continuously
4. Pulse width is set from module level variable on interrupt (you will need to convert from μS to Timer3 ticks)

Every register you use should be called out; anything special (e.g.: module variables) should also be documented.

On receiving a protocol message `ID_COMMAND_SERVO_PULSE`, you will need to set a module level variable inside your `RCServo.h/c` module for the pulse uptime. Note that you will have to scale this from μ seconds to ticks inside your module. The OC3 interrupt will change the value of the uptime register to the new one sent from the *ECE121 Console* (as stored in your module variable). Since the change can only occur at the end of the pulse (during the OC ISR) then you can never get a malformed pulse.

As good coding practice, you should only change your module variable keeping track of the pulse width if the inbound `ID_COMMAND_SERVO_PULSE` is different from the existing one. Again, be careful as this is an unsigned int and you must account for the Big-Endianess of the protocol.

Upon getting a changed pulse width, your microcontroller should reply with an `ID_SERVO_RESPONSE` with the new pulse width (limited to a reasonable range).



Warning: You should limit the pulse width to between a minimum and maximum limit (as defined in the `RCServo.h` file). Note that these limits might damage other servos, but work for the ones in your lab kit.

As before the interrupt you will need is:

```
void __ISR(_OUTPUT_COMPARE_3_VECTOR) __OC3Interrupt(void) {
```

5.3 RC Servo Test Harness

With all of the functions and interrupts in the `RCServo` module, you are ready to implement the test harness. The test harness is a simple loop that reads the incoming `ID_COMMAND_SERVO_PULSE` and scales it appropriately, sets the OC3 to drive the servo to that location. If the pulse is different than the previous pulse then the

micro should reply with an `ID_SERVO_RESPONSE` with the new pulse width (and should clip the pulse width to between the minimum and maximum limits).

This should loop forever.

You should hook up the OC3 output to a scope to time the pulse and see if it matches your command (before hooking it up to the actual servo).



Warning: Remember, the RC servo must be powered by a 5V rail, either through a regulator or through a separate output on the power supply. Do **NOT** exceed 5V on the power of the servo. Under no circumstances should you ever power a servo directly from your micro.

Use the *ECE121 Console* to drive the servo under the Lab2 tab, RC Servo Control. You should be able to command your servo and have it move back and forth, either using the `-/0/+` buttons, or by sliding the slider.

Show your RCServo 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.

6 Lab 2 Application

Finally, we come to the Lab 2 Application. This application should be coded in the `Lab2Application.c` file in your `Projects` \mapsto `Lab2` directory.

With the four modules you have written for this lab, `FreeRunningTimer`, `RotaryEncoder`, `PingSensor`, and `RCServo`, you now have all of the required tools with which to write the application.

The application is relatively simple, and uses all four modules to implement the RC servo application. The requirements are:

1. Send an `ID_DEBUG` message that includes compilation date and time
2. Read the Encoder via SPI
3. Read Ping Sensor via Input Capture
4. Drive the RC Servo using the encoder value (appropriately scaled and limited)
5. Drive the RC Servo using the ping sensor distance (appropriately scaled and limited)
6. Switch between the inputs (encoder/ping) based on the `ID_LAB2_INPUT_SELECT` message
7. Use the `FreeRunningTimer` to report encoder position and status using `ID_LAB2_ANGLE_REPORT` at 20Hz

There are two new protocol messages, `ID_LAB2_INPUT_SELECT` and `ID_LAB2_ANGLE_REPORT`, that needs to be implemented for the application. `ID_LAB2_INPUT_SELECT` is used to select the input between the encoder and the ping sensor. The application will send an `ID_LAB2_INPUT_SELECT` when the appropriate button is pressed.

Your micro must decode the message and switch to the appropriate input. Regardless of input, you micro will send an `ID_LAB2_ANGLE_REPORT` to the application. The data field for `ID_LAB2_ANGLE_REPORT` consists of a *signed int* that is millidegrees (degrees \times 1000), and an unsigned char that has 3 bits that represent status: (i) Bit 0 is angle value out of bounds high; (ii) Bit 1 is angle value within range; and (iii) Bit 2 is angle value out of bounds low.

That is, the status tells you if the input is trying to drive the servo too high, or too low or within range. Be careful as you will need a union to store the 5 bytes (4 for the int, 1 for the char). Remember that the int needs to be in the correct *Endianness* for the protocol.

Note that when the ping sensor is the input, the minimum distance (corresponding to the minimum or most negative angle on the RC Servo) should be set to 25cm. The maximum distance should be set to 125cm (corresponding to the largest or most positive angle on the RC servo).²¹ Think of the RC Servo as connected to a dial indicator that maps the distance from the ping sensor reading to some other scale; for example you might have the ping sensor mounted in a tank and the dial indicator gives you a full to empty reading.

Unlike the encoder, the ping sensor distance is a *conversion* mapping, rather than a *direct* mapping; you are converting distance to angle.²² That means you have to figure out scaling and appropriate limits. Linear scaling works well for this application; you get to calculate the corresponding angle change for a 1cm change in distance.²³

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.

Acknowledgements

- Figure 1 courtesy of Analog Devices Corporation
- Figure 2 courtesy of Analog Devices Corporation
- Figure 4 courtesy of Austria Microsystems Corporation
- Figure 6 courtesy of www.theengineeringprojects.com
- Figure 7 courtesy of Analog Devices Corporation
- Figure 8 courtesy of Absima Corporation
- Figure 10 courtesy of Microchip Corporation

²¹Typically these are $\pm 60^\circ$ for a normal servo

²²Just to be pedantic, both the encoder and ping sensor are digital sensors that return `ints` to the microcontroller. In the case of the encoder, those `ints` represent angle, and the output of the RC servo is also an angle—thus you might need some offset and scaling factors, but there is no unit conversion involved. For the case of the ping sensor, the `int` represents distance, thus you need a conversion constant that translates distance to angle on the RC servo.

²³If we choose the min/max angles as $\pm 45^\circ$ then the scaling becomes: $\theta = -45 + 90 \times \left[\frac{d-0.25}{1.0} \right]$ where θ is the angle commanded in degrees and d is the distance from the ping sensor in meters.