Quadrature Encoders

February 18, 2004 Version 0.1

By Daniel Katanski, Programming Mentor, Team 240

Introduction

This document introduces you to the concepts of quadratuer encoders, their use, how to select them, and most importantly how to use them on your robot. Actual code chunks are included to help you with the more complex interrupt-driven aspects of using encoders.

Additional discussions will provide you with things to consider when designing and programming your robot. (Consider this a ‘core dump’ of thoughts about encoders.)

Read this document to get a general understanding of how encoders work. However, when you begin to implement quadrature encoders I strongly suggest that your read the “Interrupts for Dummies” white paper which will bring you up to speed about how the interrupts actually work so that you can more easily see how the included code functions. The logic required to do a quality implementation should use interrupts, as opposed to polling the encoders for changes.

NOTE: This is an incomplete document. More information may be added as time allows. However, if I wait any longer the robot build season will be over and this document will help no one until next year.

I want to acknowledge the help of E. J. Meyer, Captain of Team 240, who pair-programmed with me to write and test the encoder state machines used in the Interrupt Service Routine code. Mr. Kevin Watson’s timer example code is an excellent base from which to work. Also, Doug Hilton and Tom Radcliffe for giving the document a quick look over.

Comments and constructive criticism to improve this document and help others is joyfully welcomed by email at dan@provide.net. Good luck.

Encoders

Have you ever wondered how you can determine how far your robot has moved? Well that is the job of an "encoder".

An encoder is a device that somehow encodes information so that your computer processor and program can interpret them. If you want to count rotations of a wheel with a tab that pushes a switch every rotation you have just encoded information. This is a form of a mechanical encoder.

There are also magnetic and optical encoders in common use. Most automotive distributors now use a solid state form of the old mechanical "points" which uses the "Hall effect", where the magnetic properties of a metal-toothed wheel rotate past this encoder. Optical encoders somehow use light to switch on and off. There are also rotary and linear encoders.

Which type of sensor you use depends on your application. However, in this competition you might want to lean toward optical encoders due to their speed and reliability, as opposed to a mechanical encoder whose mechanical components may eventually deteriorate.

Quadrature encoders

What is a quadrature encoder? A quadrature encoder is simply an encoder. However, the word "quadrature" has the root of "quad" standing for four. So a quadrature encoder is a device which has four states or positions.

A regular encoder has two states which are "off" and "on", or the popular binary values 0 and 1. A quadrature encoder has four states. If you are again thinking in binary you immediately will realize that to represent four states you need at least two bits – and you would be right. Quadrature encoders use two switches, called the "A" phase and the "B" phase.

Essentially quadrature encoders supply counts to you. You will receive the signals of phases A and B in some pattern like 00, 01, 10, 11 (or more colloquially know as 0, 1, 2, 3, if you are thinking like a state machine, which I hope you are right about now).

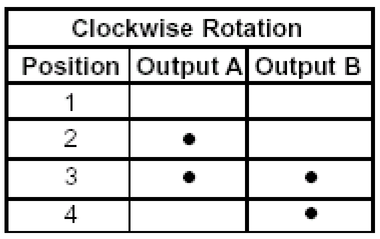
The most interesting aspect of a quadrature encoder is that it can tell you direction! So let's say that the pattern 0, 1, 2, 3, 0, 1, 2,… is forward. Then the pattern 0, 3, 2, 1, 0, 3, 2,… is backward! If you keep track of these state changes, you can track distance moved. And it works too. Does this sound a little complicated? Then read on…

By the way if you think that this technology is not in common use… then take apart your mechanical mouse that is probably in your hand this very moment (at your own risk, of course). Go ahead, open it up by twisting the cover around the ball, and then take out the ball. Look inside and you will see two shafts that the ball rolls against at right angles to each other. Then look to the ends of each shaft and you may see the encoder, and, depending on the type of mouse, you may also see an encoding wheel. A more detailed examination should be done on a broken and unrepairable mouse (or else yours might become one if you are not careful).

Mechanically, the shift in encoder phases can be accomplished in optical encoders by having two "mask wheels" through which light shines. Or there can be two light sensors mounted around one mask wheel at an angle that creates the phase shift.

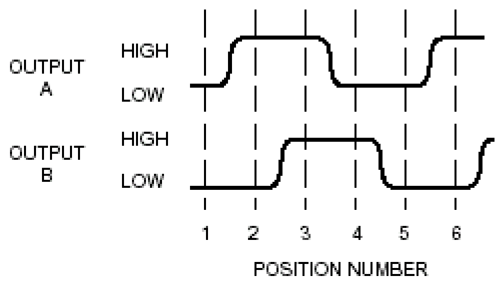
How do quadrature encoders work?

Let’s look at a quadrature encoder in a bit more detail showing you how the encoder actually works. The following image shows the basic information about an encoder (thanks Grayhill):



Clockwise Rotation		
Position	Output A	Output B
1		
2	●	
3	●	●
4		●

● Indicates logic high; blank indicates logic low. Code repeats every 4 positions.



This diagram shows two important things. The first is the binary pattern of bits for phases (or outputs) A and B in each of the four positions.

Note that the table and graph shows 1, 2, 3, 4. I’m certain that you can translate that to 0, 1, 2, 3, which is what you will see in the computer if you look at the two bits as a binary number. Any good computer scientist knows that zero is the first number!

The graph shows the secret behind the quadrature encoder - it is the simply the shift or offset between the two phases. As you look at the numbers 1 to 4 (5 and 6 are a repeat of 1 and 2) you will see that between states 1 and 2 phase A changes. Then from 2 to 3 phase B changes. And so on. Each output change tell you that movement has occurred and its direction.

Quadrature theorem: For any state change in a quadrature encoder state machine for a given direction there is one and only one resulting state change.

This is important because it affects how the software is written to track the state changes. This statement implies that there is only one possible thing to do give a current state and a specific change of state. This impacts what must be done for each interrupt. All your ISR should need to do is increment the distance counter either positive or negative, and change to the new state.

I have seen quadrature logic that is more complex than necessary. That is, the Interrupt Service Routines (ISR) have more code than necessary. This is important because the ISRs must be fast enough to service all the state changes (interrupts) that are generated by the quadrature encoders and not lose them because the precious interrupt has not finished. Depending on your design that could be 1 to 6 thousand interrupts per second.

Design considerations

Designing the use of an encoder into a robot is a multi step process. You will need to:

1. Choose a desired "resolution" to let you accomplish your missions.
2. Given the motor "unloaded speed", compute the robot’s maximum speed. Also measure the observed maximum speed if possible.
3. Compute the number of encoder revolutions per wheel-rotation, given the selected mounting location.
4. Select an encoder
5. Compute the robot’s actual resolution.

There are many factors to consider when designing encoders into your robot. Two factors are at the top of the list, they are:

- Protecting the encoder – mount the encoder in a safe place where it can not be hit by another robot or hit by something like the side of a platform should a wheel slip off. This is relatively easy.

- Desired resolution – which implies the number of interrupts per second. This requires some initial assumptions, detail thought and math.

Where you mount the encoder matters. This affects how the encoder is protected and how fast it spins (i.e., its resolution). That is, for a gear-driven power train, do you want to mount the encoder on the shaft of the motor, an intermediate shaft, or your drive wheel axle? Alternatively, for chain- or belt-driven power train you can add an idler pulley or sprocket to turn the rotary encoder.

Those with a transmission should mount the encoder to the drive-wheel side of the gear changer. This way an encoder signal always means the same distance traveled.

Once you have the encoder location selected you have to do some math. Find the motor’s maximum RPMs without load from the specification sheet. Also if possible measure how fast the robot travels a given distance, then using known gear ratios and drive wheel circumference compute the motor shaft’s RPMs under load for the robot’s maximum speed. (Most teams are disappointed.) The difference in speed is the effect of friction within your drive train. You should have both numbers for a sanity check.

Next, identify the resolution that you want the robot to have. Its OK to be unreasonable. I suggest that you try a number of different resolutions when all is said and done.

For example, choose the resolution to be 1/8”. Then compute how many increments of the selected resolution are required for the drive wheel to rotate once. Given a 28” circumference divided by 0.125” resolution we got 224 increments or "state changes" or interrupts per drive wheel revolution.

At the robot’s maximum theoretical and observed speed how many times did the wheel rotate per second? Our computed speed was 5.1 and 4.3 drive wheel revolutions per second, receptively.

Multiply the number of interrupts per drive wheel revolution by the drive wheel rotations per second to get the number of interrupts per second. Believe it or not, numbers like 1,000 to 2,000 interrupts per second are practical. From our observed wheel speed of 4.3 rev./sec. * 224 interrupts/rev. = 936 interrupts/sec.

Now you begin selecting which encoder to use an optical encoder if possible. Encoders are described by the number of positions, cycles or phase changes per revolution of the encoder. They are basically all the same things. As a rule, the fewer phase changes per revolution, the less you will pay.

We started by asking where and how you wanted to mount the encoder. Knowing how you want to mount the encoder, compute the resulting number of encoder revolutions per wheel revolution.

Next take the theoretical and observed target number of interrupts per second and divide it by the number of encoder revolutions per second to determine the number of encoder interrupts per revolution needed, (interrupts / sec.) / (rot./sec.) = (interrupts/rot.)

Now comes the judgement part of the design. Look at the catalogues to find an encoder that comes close to the desired state changes per revolution. Some encoder makers seem to like using powers of 2 so numbers like 8, 16, 32, 64, 128 and 256 are popular. But number like 24, 25, 50, 75 or 100 are common as well. It all depends on the manufacturer.

Now compute your "real" resolution. Given the number of state changes per rotation of the encoder selected times the number of revolutions of the encoder per drive wheel rotation, compute the number of state changes per drive wheel rotation, and finally your real resolution. Also compute the number of interrupts per second given the selected encoder.

In this design process you have to iterate your design multiple times to find the best fit. It is most important to be certain that the number of interrupts per second is reasonable with the motors under load. The unloaded state just makes certain that you don’t damage the encoder by spinning it too fast should your drive wheel lose grip. But realize that if the drive wheel ever loses its grip, your positional information might be unreliable unless you can show how to calibrate the robot’s position - more on this later; see ramp-up and ramp-down functions.

Encoder Options

A ‘detent’ is a mechanical stop that feels like a click upon every state change to make the encoder hold its position. If you are using the encoder to measure distance traveled on a robot the detent does not help you, and in fact might cause problems. When the encoder is spun fast enough the mechanical nature of the detent will eventually break possibly causing the entire encoder to fail.

Most rotational encoders mount through a hole like a toggle switch. Be cautious about the alignment. Poor alignment can cause side-loading on the encoder’s bushing, resulting in premature failure.

Some type of flexible shaft would help with the alignment and side-loading problem. Consider using 2 – 3” of ¼” clear flexible tubing. Make certain that it is still flexible.

Some encoders are small making wiring a challenge. Most encoders are mounted on a panel or directly to a circuit board. Sometimes a special cable designed to attach to the encoder can be acquired from the manufacturer. If available use it.

Some encoders can have a built-in push button switch. Do not order encoders with this feature – you just do not need it, and it would be one more thing that could cause the encoder to fail.

Other considerations

"Should I use one or two encoders on my robot?" This is an age-old question. But a better question is “In autonomous mode, does my robot go straight?” The answer is unequivocally and resoundingly “No!” This is because there is friction in the drive train, which may be different from side to side causing the robot to drift. Using two encoders, one on each side of the robot (and preferably on a drive wheel) you compute a "correction factor" to make the robot go much straighter.

Drive the robot ahead a foot or so. How far did each side go? On the side that went the fastest reduce the power slightly. After a few tests you can make it an initial power setting and do one more to adjust for differing environment and wear, and get your team on the straight and narrow path of robotics righteousness.

Turning can be accomplished most accurately by locking one wheel and rotating the other wheel. When you power both wheels in opposite direction your absolute position is not very accurate. This is because the robot’s center does not stay in the same place. Due to friction and motor strength the ending position waders in an unpredictable way every move. The driver can compensate, but the autonomous program can not.

This might help you go straight a measured distance or while turning, but line- or wall-following is another story. Trying to know your absolute position when line-following is not practical even with using the encoders because the robot is swaying back and forth in a random pattern while it is watching the line. Every turn adds error to your known position.

There is error inherent in every mechanical system. That is, every move that the robot makes has some amount of error. This error is sometimes on the positive side and sometimes on the negative side. Frequently it cancels itself out - but this is not reliable. This can explain the apparent error that you see every time you use the autonomous programs. Rest assured that it is not you, or the robot… it is physics.

A team created a charting program to track where the robot was, based upon encoder information. Great idea, but how accurate is it really?

Robot navigation

In autonomous mode there are three basic types of navigation, they are:

- Dead-reckoning navigation
- Sensor navigation
- Hybrid navigation

Dead-reckoning navigation is moving the robot in specified directions without any outside sensor input. Its advantages are that it can be very fast because it does not require time to think or adjust or correct its position. The disadvantage is that this method of navigation is not adaptable to changes in its environment. The robot is blind and dumb.

Sensor navigation uses sensors to guide the robot's movement. An example of this is line following. The advantage is you always know where the robot is relative to clues in the environment – assuming that there are clues to guide the sensors. The disadvantages are that this method of navigation is slow and can be confused. Sensor navigation uses feedback to guide its small movements.

Hybrid navigation takes the best of both dead reckoning and sensor navigation as needed. For example, follow a line for a few feet to align the robot, then move at high speed toward the target, and finally line follow the last few feet. Do what you have to do to get the job done. Use sensors when reliable and useful. The advantages are that the program can adapt to its surroundings which still maintaining its speed. The disadvantage is that this more complex to program. But in doing so you are teaching the robot to think. Get use to this – autonomous mode is going to be with us for a long time to come.

There is a type program called Copy Cat that can record you robot's movement as you drive and be able to replay them on command (like during autonomous mode). Interesting concept. Let's look a bit deeper in to pluses and minuses of using this technique. When you are recording the robot's movements you are using your own senses to guide the robot. Your vision and reasoning skills are very powerful tools.

However, when you play back the recorded moves you are simply using dead reckoning navigation. That is, the robot is not using any information from the world around it. This may be adequate, however, variations in the field layout, slipperiness of the carpet, or other variables can render the recorded navigational moves ineffective.

The desired navigation technique is the hybrid method, where you take advantage of the best of both the dead reckoning and sensor navigation. How you choose to use them adds intelligence to your robot.

The problems of navigation are not unique. The LEGO robots have similar problems, except all of their programs are autonomous. Everyone has repeatability problems. But there is a way to handle them…

Recalibration and feedback

“All I ask is a tall ship and a star to steer her by.” John Masefield's "Sea Fever"

As the robot moves it accumulates errors from each move. Frequently the errors cancel each other out. However, equally likely they don’t - resulting in poor repeatability. It is unavoidable. It can be somewhat minimized but it is still there and will cause you frustration.

Some sources of errors are:

- Friction or slippage
- Surface contamination and dirt
- Momentum (not pausing) during maneuvers
- Hitting obstacles
- Battery strength
- Weight change
- Time
- Earth or table quake (Tilt)

A solution to resolve the repeatability problems is to use hybrid navigation with "recalibration".

I make a distinction between recalibration and feedback. With **feedback** you get information about your environment, but some error still remains. It may keep you on track or can mislead you.

The following are forms of feedback:

- Light sensors
- Touch sensors
- Rotation sensors

- Distance sensors
- Timers
- Hitting the wall
- IR Beacon
- Current sensors

When you **recalibrate** you reset you positional information to a new known starting position so that you make new assumptions without considering past accumulated error. Recalibration lets you ignore accumulated errors and lets you start over again with the sensors.

Forms of recalibration are:

- Bump into a wall (motor will stall)
- Bump into a wall with a touch sensor
- Look for a line

To effectively use recalibration you must intentionally design it into your robot movements and program. Like using hybrid navigation, it requires thought and planning.

The great debate…Speed Vs. Accuracy. They say, “You can’t have both”. However, you may be able to maintain speed if you are able to recalibrate periodically.

But be aware that speed implies momentum, which creates errors. As long as the accumulated error does not become so large that the robot is unable to perform its recalibration maneuver. Go as fast as you can because with the recalibration all previous errors are discarded.

Absolute versus incremental movements

When moving the robot around the field you can use "absolute" or "incremental" movements.

Using absolute movement you tell the robot it’s new destination position relative to an absolute coordinate system where zero is most likely the initial starting position. This sounds great, however, using two encoders it is difficult to actually identify exactly where the robot is at. If you can figure out how to use absolute coordinates you can account for accumulated errors. It is more complex to implement.

Using incremental moves, you essentially zero out the distance sensor before each move. Your current position becomes the new zero point relative to the next move. It’s a lot like dead-reckoning. However, using sensor feedback and recalibration you can actually create intelligent robots.

How do you know your exact position using encoders? That is a good question. If all of the robot’s movements are assumed to be straight or clean turns with one wheel locked, you can use various trigonometric functions to compute the position with reasonable accuracy.

However, when using line following algorithms it gets more difficult. This is because the robot may actually zigzag along the line or move in curved trajectories. This is very difficult to compute the robot's position based upon encoder position. This is more so the case if the power to the two motors is set uneven to get the robot to drift in a direction rather than making an discrete turn.

Imagine telling the robot to smoothly move in a six-foot diameter circle by adjusting the motor power, and stopping in the exact starting position (assuming it can do that). Now you look at the encoder values. The inner encoder seems to have moved in a five-foot circle while the outer encoder moved in a seven-foot circle. And to boot you find yourself in the exact starting position. I'm not certain that we have the tools to track the robot with adequate accuracy at the moment.

Ramp up functions and ramp down functions

Quadrature encoders can be very useful none the less, as long as the wheels do not spin or slide. Once they do its time to recalibrate. This is very critical in autonomous mode.

The way to minimize the possibility of wheel spinning or sliding is to use "ramp-up" and "ramp-down" motor speed functions. If you turn the motors on full power they might spin. Conversely, if you turn them off the robot will coast in accordance with its momentum and friction, most likely missing the desired ending point. But do not despair, we programmers have our ways of making ze robot cooperate.

A **ramp-up function** changes the motor power setting based upon time. That is, you tell the ramp-up function to set the motor to its maximum speed in one second. So every time the function is called it recalculates the appropriate motor power based upon the amount of time since it first turned the motor on.

A ramp-up function can change the motors in a set pattern. A linear ramp up is easy to implement, but other methods can be used. The following is an example of a linear ramp-up function.

```
// Function:  Ramp Up
// Purpose:   Return the power setting using a linear ramp up strategy.
// Code:      rampup (max, start, ramptime)
// Parameters:
//   max is the maximum speed to ramp up to (0 to 254, 127 mid)
//   start is the starting time for this move.
//   ramptime is the time to get to max speed.

unsigned char rampup (int max, unsigned long start, unsigned long ramptime) {
       int x, t;

       // Do ramping for only ‘ramptime’ hundredths of seconds.
       if(get_Clock () < (start + ramptime)) {

              x = (max - 127);    // Range to ramp through.
              t = Clock - start;  // How far through ramp time.
              t = t * 100;        // Scale time for integer math.
              t = t / ramptime;   // Percent of time through rampup.
```

```
                return  127 + ((x * t) / 100);
        }
        else {
                // Past ramp up time, use maximum power setting.
                return max;
        }
} // rampup
```

The four lines of computations in the above code can be optimized to something like the following that can result in more optimized code by the compiler.

```
return 127 + ((max - 127 * (((Clock – start) * 100) / ramptime) / 100);
```

A ramp-down function is used as a brake to slow the robot down in a controlled manner. Reversing the motor's direction with just a little bit power for a set amount of time to make it act as a brake can do this. Not enough power and the robot coasts. Too much power and the robot will skid. Not enough braking time and you may continue to have undesired momentum into your next maneuver. Too much braking time and the robot may backup. The amount of time depends on the moment and speed. You should have gotten the idea that it's a balancing act.

The following code is an example of a ramp-down function

```
// Function:  Ramp Down
// Code:      rampdown (max, start, ramptime)
// Purpose:   Return the power setting using a linear ramp down strategy.
// Parameters:
//   max is the maximum speed to ramp down from (0 to 254, 127 mid)
//   start is the starting time for the deceleration
//   ramptime is the time to get to lower speed

unsigned char rampdown(int max, unsigned long start, unsigned long ramptime){
        int x, t;

        if(get_Clock () < (start + ramptime)) {
                x = (max - 127);    // Range to ramp through.
                t = Clock - start;  // How far through ramp time.
                t = t * 100;        // Scale time for integer math.
                t = t / ramptime;   // Percent of time through rampup.

                return  127 - ((x * t) / 100);
        }
        else {
                // After ramp down time motors should be unpowered.
                return 127;
        } // if
} // rampdown
```

This ramp-down function has a problem. It assumes that you know what your current velocity is. That is, it assumes that the variable 'max' is the current robot speed. But what is the robot did not travel far enough to get up this speed? You could be braking too long and be wasting time and perhaps moving in the wrong direction. The next section discusses how to determine the robot's speed in 1/100 of second intervals.

Determining robot speed from the encoder

The robot's speed can be calculated by computing the distance traveled in a given amount of time (distance = rate times time). For example, count the interrupts from one 1/100 of a second clock tick to the next. This number can be very useful by the ramp-down function. If you are using two encoders then consider averaging the distance they traveled as the average robot speed. For example the following code can be placed in the 'timer' Interrupt Service Routine:

```
If (Clock != OldClock) {          // Has it been another 1/100 sec. already?
       Clock = OldClock;          // Yes.

       AverageSpeed = (LeftEndDistance – LeftStartDistance) +
                      (RightEndDistance – RightStartDistance);
                                  // Distance traveled by both encoders.
       AverageSpeed *= 125;       // 1/1,000" traveled per interrupt.
       AverageSpeed \= 2;         // Make it the average distance.
       AverageSpeed *= 100;       // Multiply by 100 to
                                  // get 1/1,000 in. per second.

       LeftStartDistance = LeftEndDistance;    // Reset starting distance.
       RightStartDistance = RightEndDistance
}
```

Earlier in this document, you saw how you compute the robot's resolution. Multiply the average distance traveled *times* the resolution *times* 100 to get the number in "inches per second". Try to do it in all integer arithmetic giving a value of a number of 1/1,000 of an inch traveled in one second. This code assumes that the robot has 0.125 inch encoder to wheel resolution.

This adds a bit of work to the timer Interrupt Service Routine but the information is very useful. For good information about implementing timers see Kevin Watson's examples at www.kevin.org/frc.

Encoder Programming

The programming necessary to implement the quadrature encoder is just the initialization code and the Interrupt Service Routine. Other than these pieces of code all you need to do is use the distance values in your application.

Before delving into this section, be certain to read the white paper Interrupts for Dummies. It will make what you see much more understandable.

The following code assumes that two encoders are used that are connected as digital i/o pins3 – 6 (interrupts 4 – 7) which are ganged together as one interrupt by the processor's architecture.

The first piece of code to look at are the variables used to track the encoder values.

```
// Pins 3 - 6, Interrupt 4 - 7, Quadrature encoders.

volatile static unsigned char Old_Port_B = 0xFF;
             // Saved copy of port b to identify interrupts 4 - 7

volatile  unsigned char LeftWheelState = 0;
volatile  long          LeftWheelDistance = 0;

volatile  unsigned char RightWheelState = 0;
volatile  long          RightWheelDistance = 0;
```

The next piece of code is the actual initialization code for the Interrupt Service Routine:

```
// Wheel encoder distance sensors.
// ==============================

// Left and Right Wheel Encoder user interface routines.

// InitializeEncoders by setting up the interrupts and setting
// the state of the state machines.

void Initialize_WheelEncoders () {
       // Set Pin 3 and 4 to be the Left  wheel encoder inputs for
       // phase A and B, respectively.

       // Set Pin 5 and 6 to be the Right wheel encoder inputs for
       // phase A and B, respectively.

       // Step 1.  Set pint to be inputs.
       // Same as coding: digital_io_03 = INPUT;
       TRISBbits.TRISB3 = INPUT;
       TRISBbits.TRISB4 = INPUT;
       TRISBbits.TRISB5 = INPUT;

       TRISBbits.TRISB6 = INPUT;


       // Step 2.  Set pins 3 - 6 (interrupts 4 - 7) as
       //low priority.

       // This is done by the default program, but do it again
       // to be certain.

       INTCON2bits.RBIP = 0;

       // Step 3. Set edge select.  Not needed for interrupts 4-7.

       // Step 4. Clear the interrupt flag
       INTCONbits.RBIF = 0;

       // Step 5.  Enable the interrupts.
       INTCONbits.RBIE = 1;


       // Initialize the distance moved variables.
       LeftWheelDistance = 0;
       RightWheelDistance = 0;

       // Set the initial state of the left wheel encoder
```

```
        // state machines based upon the current encoder position.
        if ((digital_io_03 == 0) && (digital_io_04 == 0))
              LeftWheelState = 0;
        else if ((digital_io_03 == 1) && (digital_io_04 == 0))
              LeftWheelState = 1;
        else if ((digital_io_03 == 1) && (digital_io_04 == 1))
              LeftWheelState = 2;
        else
              LeftWheelState = 3;

        // Set the initial state of the right wheel encoder
        // state machines based upon the current encoder position.

        if ((digital_io_05 == 0) && (digital_io_06 == 0))

              RightWheelState = 0;

        else if ((digital_io_05 == 1) && (digital_io_06 == 0))

              RightWheelState = 1;

        else if ((digital_io_05 == 1) && (digital_io_06 == 1))
              RightWheelState = 2;
        else
              RightWheelState = 3;

  } // Initialize_WheelEncoders ()
```

The following code is a portion of the Interrupt Service Routine:

```
  // external interrupts 4 through 7?
  else if (INTCONbits.RBIF) {
        // Remove the "mismatch condition" by reading port b.

        Port_B = PORTB;
        // Clear interrupt flag so next interrupt can set it flag.
        INTCONbits.RBIF = 0;
        // Determine which bits have changed.

        Port_B_Delta = Port_B ^ Old_Port_B;
        // Save a copy of port b for next time around.
        Old_Port_B = Port_B;

        // Did external interrupt 4 change state?
        if(Port_B_Delta & 0x10) {
              // Left wheel encoder Phase A interrupts.
              switch (LeftWheelState) {
                    case 0:                          // A=0, B=0
                          LeftWheelState = 1;
                          ++LeftWheelDistance;       // Increment
                          break;

                    case 1:                          // A=1, B=0
                          LeftWheelState = 0;
                          --LeftWheelDistance;
                          break;

                    case 2:                          // A=1, B=1

                          LeftWheelState = 3;
                          ++LeftWheelDistance;
```

```
                        break;

                case 3:                                 // A=0, B=1
                        LeftWheelState = 2;
                        --LeftWheelDistance;
                        break;
        } // switch (LeftWheelState)
}

// Did external interrupt 5 change state?
if(Port_B_Delta & 0x20) {
        // Left wheel encoder Phase B interrupts.
        switch (LeftWheelState) {
                case 0:                                 // A=0, B=0
                        LeftWheelState = 3;
                        --LeftWheelDistance;
                        break;

                case 1:                                 // A=1, B=0
                        LeftWheelState = 2;
                        ++LeftWheelDistance;
                        break;

                case 2:                                 // A=1, B=1
                        LeftWheelState = 1;
                        --LeftWheelDistance;
                        break;

                case 3:                                 // A=0, B=1
                        LeftWheelState = 0;
                        ++LeftWheelDistance;
                        break;
        } // switch (LeftWheelState)
}

// Did external interrupt 6 change state?
if(Port_B_Delta & 0x40) {
        // Note that the right wheel state machine increments and
        // decrements in the opposite direction as does the left.

        // Right wheel encoder Phase A interrupts.
        switch (RightWheelState) {
                case 0:                                 // A=0, B=0
                        RightWheelState = 1;
                        --RightWheelDistance;
                        break;

                case 1:                                 // A=1, B=0
                        RightWheelState = 0;
                        ++RightWheelDistance;
                        break;

                case 2:                                 // A=1, B=1
                        RightWheelState = 3;
                        --RightWheelDistance;
                        break;

                case 3:                                 // A=0, B=1
                        RightWheelState = 2;
                        ++RightWheelDistance;
                        break;
        } // switch (RightWheelState)
        }

        // Did external interrupt 7 change state?
        if(Port_B_Delta & 0x80) {
                // Left wheel encoder Phase B interrupts.
                switch (RightWheelState) {
                        case 0:                         // A=0, B=0
                                RightWheelState = 3;
                                ++RightWheelDistance;
                                break;

                        case 1:                         // A=1, B=0
                                RightWheelState = 2;
                                --RightWheelDistance;
                                break;

                        case 2:                         // A=1, B=1
                                RightWheelState = 1;
                                ++RightWheelDistance;
                                break;

                        case 3:                         // A=0, B=1
                                RightWheelState = 0;
                                --RightWheelDistance;
                                break;
                } // switch (RightWheelState)
        } // if
} // else if (INTCONbits.RBIF)



/*****************************************************************
 * Explanation for the quadrature encoder state machines.
 *
 * Given a current state and an encoder interrupt:
 *
 *  State 0                         State 1
 *  A=0, B=0                        A=1, B=0
 *
 *  if A==1 then 1  -->             if A==0 then 0 <--   Phase A
 *  if B==1 then 3  -|              if B==1 then 2  -|   Phase B
 *                   |                               |
 *                   v                               v
 *
 *  State 3                         State 2
 *  A=0, B=1         ^              A=1, B=1         ^
 *                   |                               |
 *  if B==1 then 0  -|              if B==1 then 1  -|
 *  if A==1 then 2  -->             if A==0 then 3 <--
 *
 * For a single encoder the above is implemented as two state machines.
 * One for each of the A and B encoder phases.
 *
 * When an interrupt happens on either the A or B phases there
 * is only one state change that makes sense as shown in the
 * above diagram.
 *
 * Having two state machines per encoder may look like more code,
 * but isn't because there are no conditional statements. And it's fast!
 */
```

The left wheel encoder has a state machine for the both the A and B phases that work together. That is they alternate execution just like in the encoders the A and B phases alternately change states.

Notice that the increments and decrements for the left wheel and right wheel are reversed. This is because the encoders are turning in opposite directions.

This last piece of code is the property functions. Use these so that the programmers do not touch the variables used in the ISRs. It also allows you to remap or redesign the implementation without changing the application portion of the program. It's just a good programming practice.

```
// Disable Encoders
void Disable_WheelEncoders () {
      INTCONbits.RBIE = 0;      // Disable the interrupts.
}                               // This effects interrupts 4 - 7.


long get_LeftWheelDistance () {
      return LeftWheelDistance;
}

long get_RightWheelDistance () {
      return RightWheelDistance;
}

// The “set” wheel distance functions allow the programmer to
// work in "incrental mode" as instead of "absolute mode".
void set_LeftWheelDistance (long d) {
      LeftWheelDistance = d;
}

void set_RightWheelDistance (long d) {
      RightWheelDistance =d;
}
```

Note that disabling the quadrature encoder interrupts after autonomous mode is a very good idea. Why waste the processors time servicing interrupts that the robot will not use again until the next autonomous mode.

Conclusion

Your conclusion should be that quadrature encoders are a useful tool. But they are not the answer to all of the autonomous mode navigational problems.