EE2A

Wire-Following Sensor and Associated Signal Processing

1.0 Aims

(i) To design, implement and test a wire-following sensor to be used within your integrated mechatronic project.

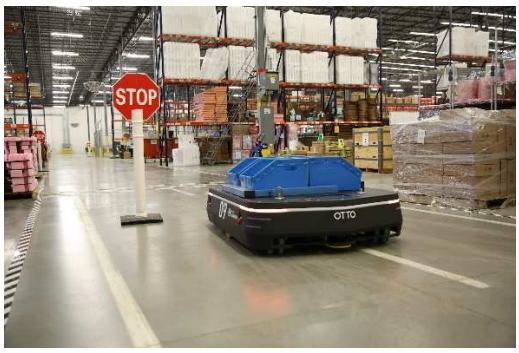
(ii) To implement a data collection 'snapshot' mode to download raw digitised data to a Matlab environment.

(iii) To implement the required processing algorithms in Matlab.

(iv) To migrate the required processing algorithms from Matlab to the C environment running on a PIC 18F processor.

(v) To familiarise yourself with unsigned integer arithmetic, signed integer arithmetic, the use of the hardware multiplier capability and the use of circular buffers in a signal processing environment.

Many autonomous vehicles follow a current-carrying wire in order to navigate. Examples include automated warehouse fork lift trucks, factory transport systems and robot lawnmowers.



Warehouse fork lift truck Factory transport Lawnmower

The sensors used to detect the wire need to know which side of the wire they are on and how far from the wire they are. There are numerous patents awarded in this field ranging from DC current-carrying wires to the use of direct-sequence spread-spectrum modulation (Bosch). For the purposes of this experiment, two phase-locked sinusoids will be transmitted and then processed within the receiver..

2.0 Learning Outcomes

Deliverables:

i) A working system.

ii) A well-structured and -commented PIC program that interfaces to a PC via an FTDI interface module and transmits both raw and processed sensor data according to the specification given in Section 3.1.

iii) A log book containing any preparatory work undertaken outside the lab, design decisions, test procedures and a discussion of the issues involved.

iv) Ability to generate your own mark scheme.

What will be assessed:

i) The program source code, with particular emphasis on adequate comments, program layout, elegance and interrupt handlers.

ii) The completeness and accuracy of the preparatory work, with particular emphasis on the relationship between the design of the program and its implementation.

iii) The adequacy of the log book (lab books not written up during the experiment will incur penalty marks).

3.0 Activities

3.1 Specification

3.1.1 PC Collection of a Snapshot of Raw Data via USB interface

A system is to be designed to collect 1024 samples of 10-bit data from the analogue-to-digital converter. This data should be transferred to the PC via the USB interface and displayed within the Matlab environment. A rudimentary script that may be used as a starting point is enclosed on Canvas. A suitable sensor coil should be used to detect the signal from a signal generator outputting a 1 kHz sine wave.

3.1.2 Matlab implementation of the required processing algorithms

A Matlab script is to be designed and implemented that extracts the 1 kHz and 2 kHz signals from the sensed signal. A 2 kHz phase reference should be derived by squaring the 1 kHz signal. The direction if the magnetic field should be derived from the complex multiplication of the two 2 kHz components. The Matlab script should model the flow of the final C program as closely as possible.

3.1.3 Migration the required processing algorithms from Matlab to the C environment running on a PIC 18F processor

The algorithms developed and tested in the Matlab environment should be migrated to run in C on the PIC 18F 27K40.

3.1.4 Final System Demonstration

Demonstrate a complete working system with the result displayed as a simple set of text characters on the PC.

That's it – now do it!

3.2 Preparatory Work

The following sections aim to guide you through some of the design processes.

3.2.1 Hardware

PW1. Design and construct a current-limiting and low-pass filter circuit, as shown in Figure 1. It should be assumed that the peak current to be drawn from the signal generator is 10 mA and that the peak voltage in 4.096 V. The tracking wire is assumed to be a short circuit (0 Ohm). The signal generator produces 1 kHz and 2 kHz frequency components at a sampling rate of 32 kHz. The cut-off frequency of the low-pass filter is to be set at a frequency of between 4 kHz and 10 kHz. Optimise the values of R1 and R2 to provide optimal current flow in the tracking wire, whilst the circuit still acts as a low-pass filter.

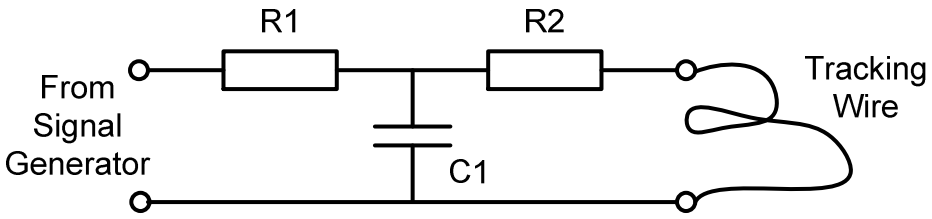


Figure 1: Required current limiting and low-pass filter

PW2. Design and construct sense coil conditioning circuit to provide a 0.512 V bias to the sense coils used. The bias voltage must be adequately decoupled such that minimal noise or co-coil interference leaks into adjacent ADC channels.

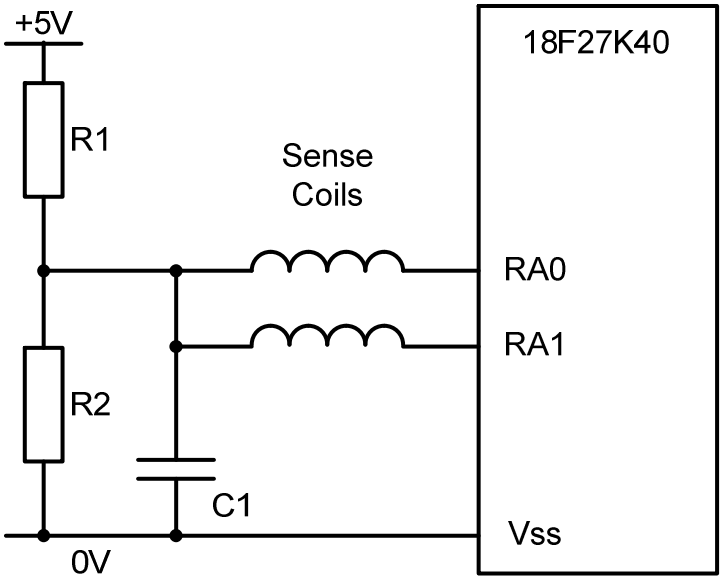


Figure 2: Required analogue signal conditioning for sense coils

3.2.2 Software

PW3. Write a Matlab script to automatically populate four look-up tables for the local oscillator (sin_1kHz, cos_1kHz, sin_2kHz, cos_2kHz). The data should be generated in two's complement format in any *N.M* and scaled value.

PW4. To be updated as the first groups hit problems.

PW5. To be updated as the first groups hit problems.

PW6. To be updated as the first groups hit problems.

Appendix 1

Signal Processing

Introduction

Your integrated mechatronic project requires you to design a vehicle to follow a hidden wire carrying a peak current of 10 mA – the RMS current will be significantly smaller than this. A sensor is required that determines which side of the wire the vehicle is on and the displacement from the wire.

The current flowing in a wire generates a magnetic field as shown in Figure A1.1.

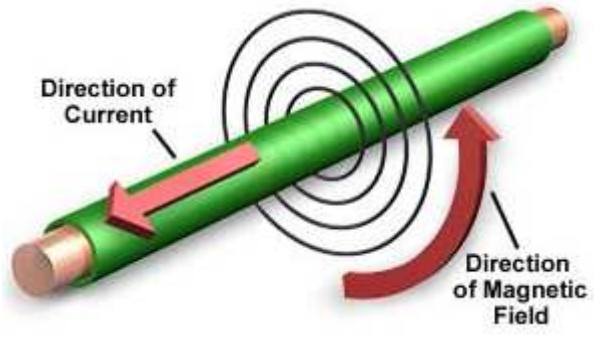


Figure A1.1: Reproduced from http://micro.magnet.fsu.edu/electromag/electricity/generators/

So if the magnetic field lines are travelling upwards, the vehicle is to the right of the wire, whereas if the magnetic flux lies are travelling downwards, the vehicle is to the left of the wire. This assumes that a DC current is flowing in the wire and a compass. However, for the integrated mechatronic project an AC field is specified so the flux lines reverse twice-per-cycle (a compass will always read zero in this situation).

Thus a DC signal needs to be synthesised when only AC signals are available. Remembering that $\cos(\omega_1 t)\cos(\omega_2 t) = \frac{\cos(\omega_1 t - \omega_2 t) + \cos(\omega_1 t + \omega_2 t)}{2}$, this representing the multiplications of two sinusoidal signals of frequency $\omega_1 t$ and $\omega_2 t$. Now if the two frequencies applied to the multiplier are the same, then $\cos(\omega_1 t)\cos(\omega_1 t) = \frac{1 + \cos(2\omega_1 t)}{2}$. The first term in the result is a constant, or DC value, whilst the second term is at twice the frequency and will probably be discarded. These ideas may be applied to a series of 1950s style functional blocks, as shown in Figure A1.2.

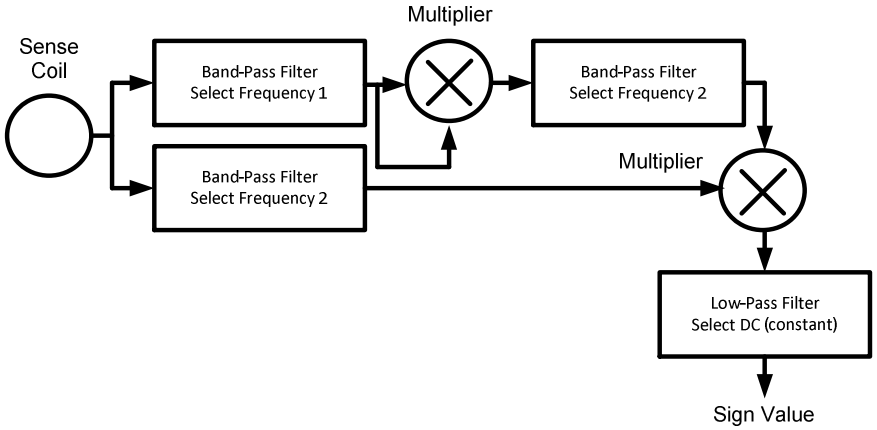


Figure A1.2: Signal flow block diagram (1950s approach)

The sense coil detects both the 1 kHz signal ($\omega_1 t$) and the 2 kHz signal ($\omega_2 t$). The upper horizontal flow-chain filters the 1 kHz signal using a band-pass filter to reject as much interference as possible. The output of this filter is fed into both inputs of a multiplier, effectively squaring the signal. The output of the multiplier will now contain both a 2 kHz component and a 0 kHz (DC) component. The 2 kHz component is selected using another band-pass filter.

The lower horizontal flow-chain filters the 2 kHz signal using a band-pass filter to reject as much interference as possible. The output of this filter is fed into one input of a multiplier, the other fed by the upper horizontal flow-chain. The output of this second multiplier will now contain both a 4 kHz component and a 0 kHz (DC) component. The 0 kHz component is selected with a low-pass filter, the sign of this signal indicating which side of the wire the vehicle is on.

In a modern signal processing, filtering is achieved by shifting the input frequency using a multiplier to near zero (called baseband) and then using a low-pass filter. This process is shown in Figure A1.3 and is known as a quadrature demodulator, or an I/Q demodulator.

The input signal is connected to two multipliers. The first multiplier is also connected to a sinusoidal oscillator, often known as the local oscillator. Mathematically, we might describe this as $\cos(\omega t)$, where ω is the frequency of the input signal that is to be examined. The second multiplier is also connected to a ninety degree phase-shifted version of the sinusoidal oscillator. Mathematically, we might describe this as $-\sin(\omega t)$, where ω is the frequency of the input signal that is to be examined. The output of these multipliers are low-pass filtered, this being an operation very similar to 'smoothing' or integration. Thus the operations shown in Figure A1.3 can be described as two equations:

$$I(t) = \int s(t)\cos(\omega t) \quad dt \qquad \text{and}$$

$$Q(t) = \int -s(t)\sin(\omega t) \quad dt$$

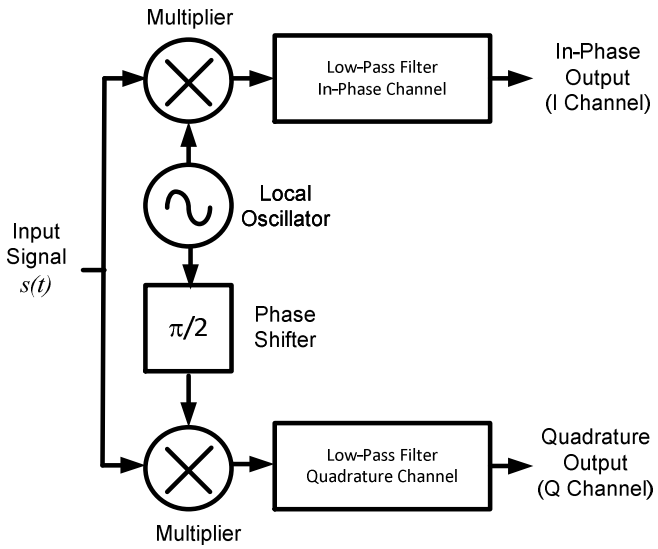


Figure A1.3: Quadrature Demodulator

All modern signal processing tends to carry around two channels and this process is very conveniently described using complex numbers – the real component is the in-phase signal and the imaginary component carries the quadrature signal. In first year mathematics complex numbers lectures a description for the two channels of the local oscillator was given as:

$$\cos(\omega t) - j\sin(\omega t) = \exp(-j\omega t)$$

Thus the two output signals could be combined into a single complex number, $S(\omega)$, resulting in a single equation for the whole of Figure A1.3.

$$S(\omega) = \int s(t)\,(\exp(-j\omega t) \quad dt$$

This is called a Fourier Transform.

In a practical implementation the filter is likely to be implemented using a moving-average filter (by summing over a circular buffer). Thus the actual implementation is described by:

$$S(\omega) = \int_{-\tau/2}^{+\tau/2} s(t)\,(\exp(-j\omega t) \quad dt$$

The frequency domain response corresponds to:

$$S(\omega) = S_0\ \tau \frac{\sin\left(\pi(f - f_1)/2\right)}{\left(\pi(f - f_1)/2\right)}$$

where S_0 is the amplitude of the 1 kHz signal (f_1) and τ is the equivalent integration time corresponding to the time duration of the circular buffer. Although simple, the moving average filter results in a sinc function in the frequency domain with the result that the first sidelobe of the response in the stop-band is only suppressed by 13 dB. To improve this, a time-domain impulse response, *h(t)*, is convolved with the input signal

$$S(\omega) = \int_{-\tau/2}^{+\tau/2} h(t)\ s(t)\ (\exp(-j\omega t)\ \ dt$$

The suppression of the sidelobes in the stop-band of the filter response may now be significantly improved. However, a large number of additional multiplication operations are now required.

Merging the I/Q demodulators of Figure A1.3 with the required signal flow chain of Figure A1.2 results in a signal processing flow diagram shown in Figure A1.4.

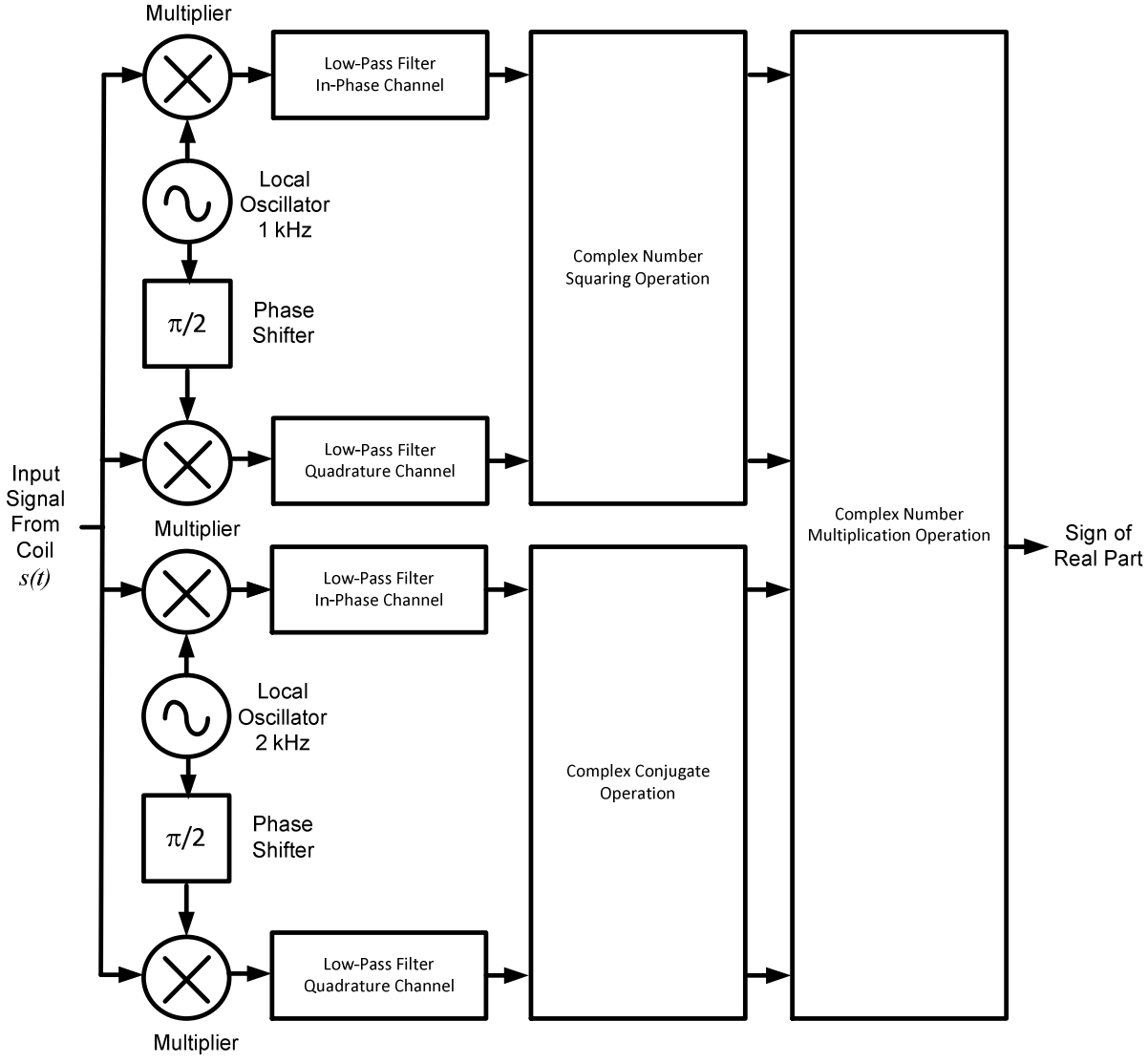


Figure A1.4: Signal processing flow diagram

Technical Note 1

Number Formats

In order to implement the signal processing functions efficiently, it is necessary to understand the implications of working with fixed-point number systems.

A language like Matlab operates with double precision arithmetic and so the programmer rarely has to consider the limitations of the processor. With a device like a PIC 18F27K40 the programmer has to be acutely aware of the number system being used. As an example, let us consider the number systems that might be encountered as the signal progresses through a chain illustrated in Figure TN1.1.

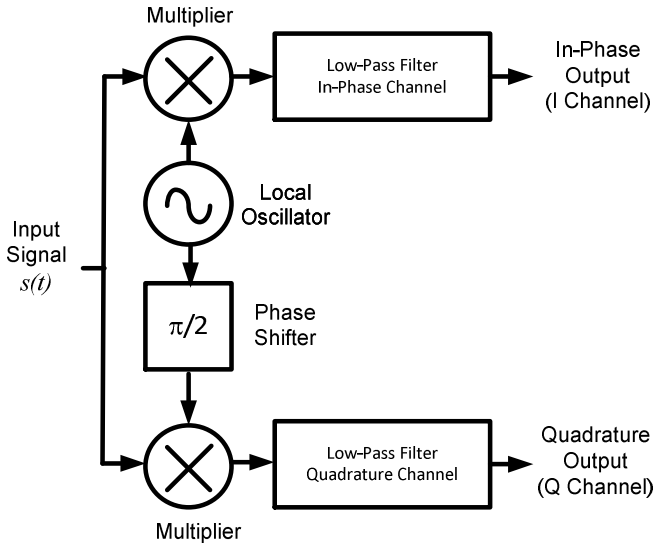


Figure TN1.1: Quadrature Demodulator

The input signal, *s(t)*, will be obtained from the on-board analogue-to-digital converter. This device uses an offset binary format. Assuming the sample has been transferred to an unsigned integer such as:

```
int16 Chan1_Analogue_Sample;
```

If the reference voltage was initialise to 1.024 V, then an input value of 0 V will provide a value of 0x0000, an input value of 0.512 V will provide a value of 0x01FF and an input value of 1.024 V will provide a value of 0x03FF.

The first step is usually to convert an analogue-to-digital sample value into a signed (two's complement) value by an operation such as

```
Unsigned int16 Chan1_Signed_Value;
Chan1_Signed_Value = Chan1_Analogue_Sample – 0x0200;
```

With the addition of a hardware offset circuit, a very small negative input value should result in an output of 0xFFFF, the largest negative input value should result in an output of 0xE000 and the largest positive input value should result in an output of 0x01FF.

The next stage in the signal flow chain illustrated in figure TN1.1 is that of multiplying the signed input value by a signed sinusoidal value obtained from a look-up table for speed. The PIC 18F27K40 contains and 8x8 bit multiplier with a 16-bit output, see Section 12 of the data sheet. Table 12.1 of the data sheet lists the times taken for a variety of different formats. With a 64 MHz clock speed, a 8x8 signed multiply would take 375 ns and a 16x16 signed multiply would take 2.5 µs. Four multiplies will be required per input sample, using at least 10 µs of the available 31.25 µs of a 32 kHz sampled signal.

TABLE 12-1: PERFORMANCE COMPARISON FOR VARIOUS MULTIPLY OPERATIONS

Routine	Multiply Method	Program Memory (Words)	Cycles (Max)	Time			
				@ 64 MHz	@ 40 MHz	@ 10 MHz	@ 4 MHz
8x8 unsigned	Without hardware multiply	13	69	4.3 µs	6.9 µs	27.6 µs	69 µs
	Hardware multiply	1	1	62.5 ns	100 ns	400 ns	1 µs
8x8 signed	Without hardware multiply	33	91	5.7 µs	9.1 µs	36.4 µs	91 µs
	Hardware multiply	6	6	375 ns	600 ns	2.4 µs	6 µs
16x16 unsigned	Without hardware multiply	21	242	15.1 µs	24.2 µs	96.8 µs	242 µs
	Hardware multiply	28	28	1.8 µs	2.8 µs	11.2 µs	28 µs
16x16 signed	Without hardware multiply	52	254	15.9 µs	25.4 µs	102.6 µs	254 µs
	Hardware multiply	35	40	2.5 µs	4.0 µs	16.0 µs	40 µs

Figure TN1.2: Multiplication times reproduced from the data sheet

The summary at this stage is that a system with an analogue-to-digital converter configured to run at 8-bits should have significant spare processing time available (assuming 8-bits in the sine wave look-up table and an 8x8 signed multiply). A system with an analogue-to-digital converter configured to run at 10-bits should have some spare processing time available (assuming 16-bits in the sine wave look-up table and a 16x16 signed multiply).

The output of the multiplier will be placed in a circular buffer of *N* samples. The contents of this circular buffer will be summed on every input sample, executing a 'multiply-accumulate' function. The greater *N*, the narrower the bandwidth of the corresponding filter. So for *N*=32 with a sampling rate of 32 kHz a 1 kHz bandwidth filter is obtained – this is not very good with a 1 kHz input signal. For simplicity, *N* should be chosen as $2^{integer}$ such as 128, 256 or 512.

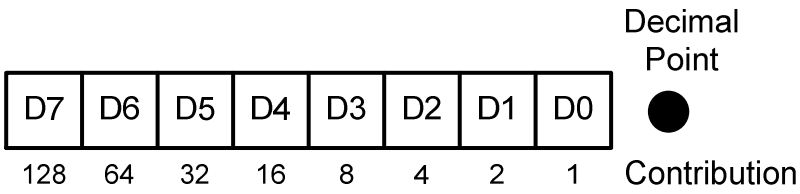
If two numbers are added together, then an additional bit is required to cope with a potential overflow. If 512 samples are added together then an additional nine bits are required to cope with a potential overflow. Obviously, the greater the number of bits required in the summation word, the slower the operation will become. The word-length allocation will look may be described by:

Description	Number of Bits
Analogue-to-digital converter word length	B_1
Local oscillator sine and cosine look-up table word length	B_2
Word length growth due to number of samples in the circular buffer (samples = 2^{B_3})	B_3
Total bits required without loss of accuracy	$B_1+B_2+B_3$

A first-guess error analysis budget might take B_1=10 (number of bits in the ADC), B_2=16 (just because a 16x16 multiplier is available and B_3=8 (because 256 samples is a nice round number and a 125 Hz filter bandwidth sounds like a reasonable compromise). Performing the calculation results in a requirement for a 34 bit word – exceeding that of an `int32` data type.

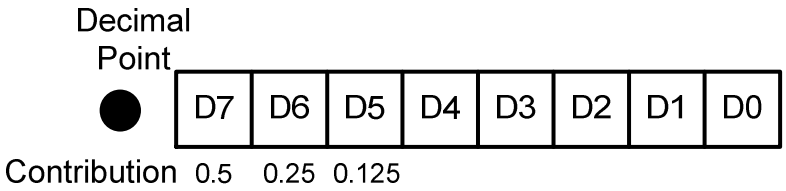
Decimal Point Position and Fractional Numbers

Basic undergraduate lectures will assume that an integer number has the decimal point to the right of the word, thus the bit contributions are 1, 2, 4, 8 etc.



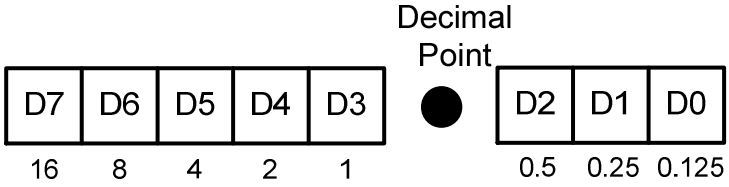
This is called an *N.0* format with *N* places to the left of the decimal point and zero places to the right of the decimal point.

Now there is no reason why you should not move the decimal point. Thus the same word with the decimal point assumed to be to the left of the word would be:



Thus the bit contributions are 0.5, 0.25, 0.125, 0.0625 etc. in decreasing order. This is called an *0.N* format with *0* places to the left of the decimal point and *N* places to the right of the decimal point.

The decimal point could also be placed in the middle of the word:



This is called an *N.M* format with *N* places to the left of the decimal point and *M* places to the right of the decimal point.

The interesting result of this discussion is that 'the position of the decimal point is purely in the head of the programmer'. The processor just deals with bit patterns and does not care.

Thus programmers often consider that the sinusoid stored in the look-up table is a fractional signed number in the range ±0.9999. The right-most least-significant bits can be discarded as they are likely to contain numerical rounding error noise. The decision is often taken by working backwards from the final multiply-accumulate word length.

Other factors to consider:

- The multiplication of two signed numbers of width 'sign + B_1' and 'sign + B_2' results in a word of width 'sign + B_1 + B_2'. This saves one bit compared to the unsigned word length growth described earlier.
- The multiplication of two sinusoids $\cos(\omega_1 t)\cos(\omega_2 t) = \frac{\cos(\omega_1 t - \omega_2 t) + \cos(\omega_1 t + \omega_2 t)}{2}$ and the removal of one of the components using filtering results in a reduction of one bit (from the factor of two divisor).

Typical Worked Example – Working Backwards from 16-bit Output

Assume that the output of the multiplier-accumulator stage is a signed 16-bit variable.

Assume that the number of samples to be integrated is 256, the resulting word growth is 8-bits. Thus the maximum input word width is 16 – 8 = 8 bits (8.0 format) for a single frequency

component, or 16 – 8 + 1= 9 bits (9.0 format) for a two frequency components where one is filtered out (your case).

Assume that the output of the analogue-to-digital converter has been sign extended to a 16.0 format number and that the sinusoidal look-up table has a 0.16 format. The output of a 16x16 multiplier will be in a 16.16 format (only 10 bits will be of significance to the left of the decimal point). Thus to achieve the desired 9.0 format a sign-extended right-shift of 17 places is required. In reality, this would be achieved by taking the upper 16 bits of the multiplication (16 shifts equivalent) and sign-extended right-shift of 1 place before inserting into the circular buffer.

Looking at these parameters, it would be very difficult to justify using a 16x16 multiply instead of an 8x8 multiply. Similarly, an 8-bit sinusoidal look-up table might be adequate.

Typical Worked Example – Working Backwards from 32-bit Output

Assume that the output of the multiplier-accumulator stage is a signed 32-bit variable.

Assume that the number of samples to be integrated is 512, the resulting word growth is 9-bits. Thus the maximum input word width is 32 – 9 = 23 bits (235.0 format) for a single frequency component, or 32 – 9 + 1= 24 bits (24.0 format) for a two frequency components where one is filtered out (your case).

Assume that the output of the analogue-to-digital converter has been sign extended to a 16.0 format number and that the sinusoidal look-up table has a 0.16 format. The output of a 16x16 multiplier will be in a 16.16 format (only 10 bits will be of significance to the left of the decimal point). Thus to achieve the desired 24.0 format a sign-extended right-shift of 16 places is required. In reality, this would be achieved by taking the upper 16 bits of the multiplication (16 shifts equivalent) and inserting into the circular buffer (each cell of 16.0 format).

Looking at these parameters, discarding fractional post-multiplier bits might be considered as ‘noise inducing’. However, there appears more-than-adequate head-room for allowing word growth – at the expense of computation time.

This type of fixed-point format analysis takes place during the design of many signal processing applications, for example in mobile phones. The problems of what happens if a system overloads are severe and often add dramatically to the number of lines of code required and a corresponding reduction in throughput. The examples presented above show that even a 64 MHz clock speed processor is struggling to perform basic signal processing tasks.

Technical Note 2

Analogue-To-Digital Converter

The PIC 18F27K40 contains a single 10-bit analogue-to-digital converter. The details are covered in Section 31 of the data sheet. Examining the PIC18F27K40.h include file, we find interesting commands such as:

```
setup_adc_ports(sAN0 |sAN0 | VSS_FVR);
```

This sets up pins RA0 and RA1 to be analogue inputs and uses the Fixed Voltage Reference (FVR).

Similarly, we can set the data conversion clock to be derived from the master clock using a command such as:

```
setup_adc(ADC_LEGACY_MODE | ADC_CLOCK_DIV_64);
```

This would drive the ADC from a 1 MHz source if the master clock frequency was 64 MHz. The conversion time for one sample would be in excess of 11 µs.

The PIC 18F27K40 has an internal reference voltage that should be connected to the 'VREF+' pin of the analogue-to-digital module. First this the reference voltage should be switched on and then it should be selected to one of 1.024 V, 2.048 V or 4.096 V. This may be achieved by a command such as:

```
setup_vref(VREF_OFF | VREF_ADC_1v024);
```

The voltage from your sense coil is likely to be quite small, so setting the voltage reference to the smallest value should provide the greatest sensitivity.

The analogue-to-digital module will need to be repeatedly triggered from some source, or other. Previously you had configured Timer 2 to trigger a digital-to-analogue module at a 32 kHz rate, so routing this source to the ADC might require:

```
setup_timer_2(T2_DIV_BY_8|T2_CLK_INTERNAL,249,1);
set_adc_trigger(ADC_TRIGGER_TIMER2);
```

Within the 'main' function, the corresponding interrupt must be enabled, e.g.

```
enable_interrupts(INT_AD);
enable_interrupts(GLOBAL);
```

The corresponding interrupt routine must be preceded by the compiler directive:

```
#int_AD
```

The contents of the digital ADC buffer may then be read and processed within this interrupt routine.

If your code is only designed to process one analogue channel, then in the main code you will need to set the source of the analogue multiplexer using a command such as:

```
set_adc_channel(sAN0);
```

If you are processing two, or more, analogue channels, then at the end of the interrupt routine you need to toggle a suitable global flag and set the source of the analogue multiplexer prior to the next conversion. For example:

```
if (Global_AD_Source_Flag == 0)
{  // The conversion that triggered this interrupt was from sAN0
  set_adc_channel(sAN1);     // Next conversion will be sAN1
  Global_AD_Source_Flag = 1;
}
else
{  // The conversion that triggered this interrupt was from sAN1
  set_adc_channel(sAN0);     // Next conversion will be sAN0
  Global_AD_Source_Flag = 0;
}
```

Technical Note 3
Frequency Response

The software implementation of half of Figure TN1.1 could be that illustrated in Figure TN3.1.

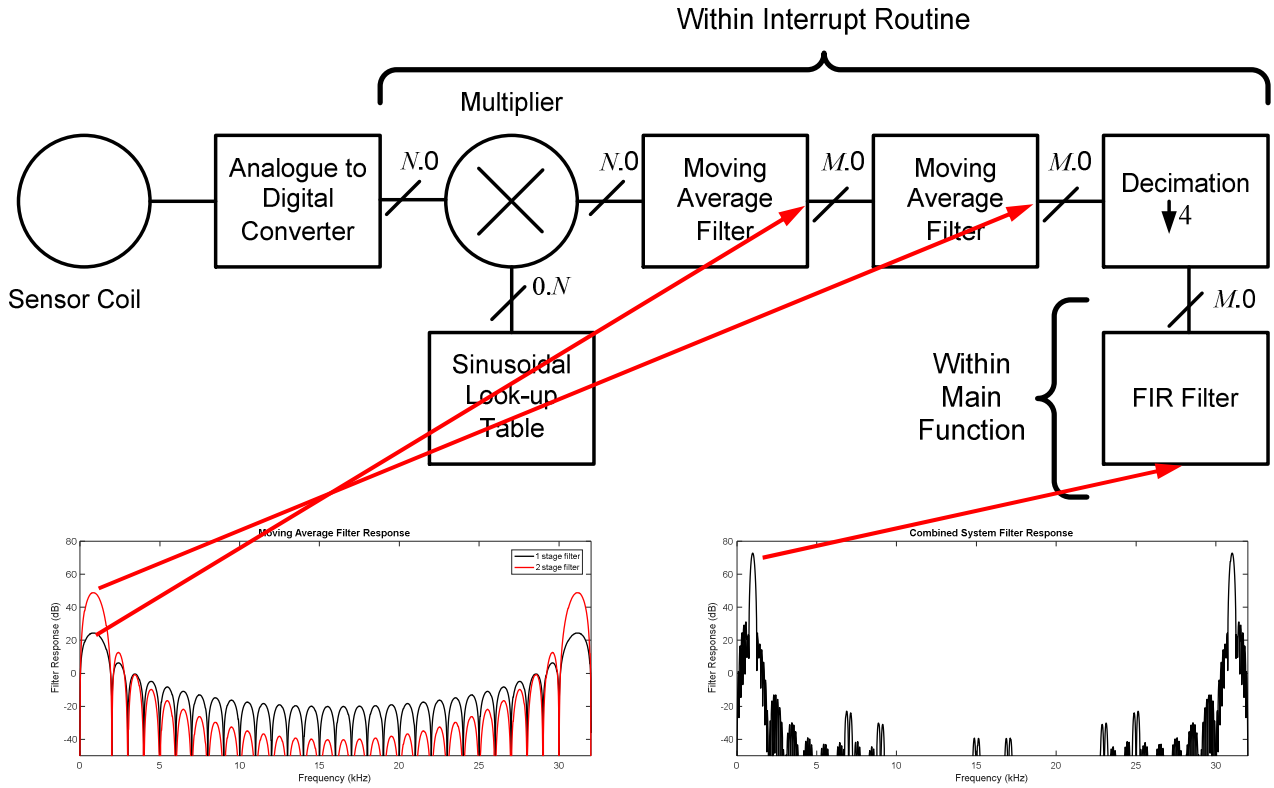


Figure TN3.1: Software implementation of half of an I/Q demodulator

It is assumed that a sensor coil is connected to an analogue-to-digital converters running in 8-bit, or 10-bit mode. The mean level is subtracted to result in a two's complement, *N*.0 format representation. This is multiplied by either a sine wave, or a cosine wave, using an *N*-by-*N* signed multiplier. The sine wave, or a cosine wave, would be stored using a fractional two's complement, 0.*N* format representation. With a 32 kHz sampling frequency and a demodulator tuned to 1 kHz, 32 samples would be required in the look-up table. It is assumed that the least significant bits of the multiplier product are discarded to leave a *N*.0 format representation.

Multiple stages of filtering follow the multiplier. One approach might be to use a moving average filter integrating over a multiple of an input signal cycle (e.g, 32, 64, 96, etc. samples). The black trace of the lower-left frequency response shows the output of a 32-point moving average filter. This is a traditional 'sinc' function filter where the first sidelobe is only suppressed by 13 dB but the deep nulls can be placed to coincide with interfering sources such as the power supply frequency.

To improve sidelobe suppression, a second stage of moving average filter is illustrated with the red trace of the lower-left frequency response, again using a 32-point moving average filter. The suppression of all sidelobes is doubled in the dB domain with the first sidelobe now being suppressed by 26 dB. In commercial sigma-delta converters, typically four stages of moving average filter are cascaded (but you probably will not have the processing power to achieve this).

Following two stages of moving average filter, the bandwidth of the signal has been significantly reduced. It is therefore possible to resample the signal at a much lower rate – this process is known

as decimation. For the figure illustrated here, a decimation rate of four has been applied (every three-out-of-four samples are discarded). The reduced sampling rate is now 8 kHz.

At this point, there is no longer a need to process all the computations within the original sampling rate of 31.25 µs. A finite impulse response (FIR) filter may now be used to provide greater suppression of the original sidelobes and define the final system bandwidth. For the case illustrated here, a 64-tap FIR filter was used with a Hamming window to provide a system band of 250 Hz and a deep null on the 750 Hz and 1500 Hz potential power supply frequency harmonics. The system frequency response is illustrated in the lower-right frequency response,

You will note that all the frequency responses cover the range 0 to 32 kHz (the sampling frequency). The demodulator is tuned to 1 kHz, so a peak appears at that frequency. A mirror image occurs at 32 kHz – 1 kHz = 31 kHz and multiple other peaks would occur at 33 kHz, 63 kHz, 65 kHz, 95 kHz, 97 kHz etc. These peaks can only be suppressed using a hardware anti-aliasing filter. Secondary peaks (sidelobes) in the range 0 to 16 kHz can be supressed by considered filter design and will allow interfering in-band signals to contaminate the desired measurement.

Although, much of the material covered in this Technical Note represents a ‘big ask’ for an EE2A laboratory experiment, the material is obviously the kind of material that will impress the Integrated Mechatronic Project assessors and would be expected in the final report of high-achieving groups (don’t forget to include your Matlab code).

Technical Note 4

Gotchas

Gotcha 1: You Didn't Read the C Standard?

The expectation is that many students will be able to demonstrate their signal processing offline in Matlab, but have significant difficulty in implementing the same algorithms in C on the PIC. The expected description of the results of your efforts will be 'it is generating rubbish'. Obviously, you have the capability to implement a bit-at-a-time of the signal flow diagram illustrated in Figure TN1.1 and send the results to a PC for comparison with the Matlab code. Consider the in-phase channel, we wish to implement $I(t) = \int s(t)\cos(\omega t) \quad dt$ in C. Initially, you might set up the analogue-to-digital converter to 8-bit resolution and use an 8-bit width look-up table for the sinusoid (probably in 0.8 format). You know that the result of multiplying two 8-bit numbers is 16-bits, so your code might look like:

```
int8 Signal=0b11111111, Cos_LUT=0b11111111;  // Initialised for test purposes
int16 Product;
Product = Signal * Cos_LUT;
```

In this case, the decimal equivalent of signal has been set to 255 and the decimal equivalent of cos table is 255 (or 0.9999). It would be reasonable to expect the result to be 65025. However, printing out the value of 'Product' reveals 1. **This is not a compiler error – it is your misconception error.** Why?

Now think about the word length of each of the variables – here I include the variable types as casts for indicative purposes.

```
(int16)Product = (int8)Signal * (int8)Cos_LUT;
```

The C standard, in common with virtually all programming languages, finds the largest word length to the right of the equate – in this case int8. It then extends all variable to that word length, if necessary. The variable on the left of the equate is assigned to this word width. So in the above case the variable 'Product' is actually considered to be int8, not int16. Thus a number greater than 255 will have its most significant bits truncated (hence the answer of 1).

The formal way of dealing with this is to increase the word length of the right-hand-side using casts. Thus the code becomes:

```
Product = (int16)Signal * (int16)Cos_LUT;
```

Adding the casts will provide the correct answer. However, a 16-by-16 multiplier is now called instead of an 8-by-8 multiplier and the operation could easily take forty-times longer to execute. The skilled engineer (**that is you**) will now take steps to override the normal action of the compiler and force it to act in the manner required – see gotcha 2.

Similar care must be taken with all operators. As a second example, you know that word growth occurs on addition. Thus an apparently sensible code fragment might be:

```
int8 A=0b11111111, B=0b11111111;  // Initialised for test purposes
int16 Sum;
Sum = A + B;
```

An expectation might be that 'Sum' yields 510, whereas the code actually produces 254. The correct approach is actually to code:

```
Sum = A + (int16)B;
```

'Sum' now yields 510 and only one cast is required (as both A and B are now extended).

The summation required in your signal processing is less likely to yield such interesting results as $I = \sum_{i=0}^{31} s(i)\cos(i)$ will be coded as:

```
signed int8 Signal, Cos_LUT[32]={…},Index;
signed int16 MAC_I;
MAC_I = 0;   // Reset integrator
for (Index=0;Index<32;Index++)
   MAC_I = MAC_I + Signal * (signed int16)Cos_LUT(i);
```

The right-hand-side of the summation equate has already been cast to the same width as the left-hand-side and in this case a `MAC_I` appears on both sides of the equate.

Gotcha 2: The multiplication is far slower than expected

By now you will be used to inserting a debug signal into the interrupt routines. Attempting to implement just one channel of Figure TN1.1 will reveal that an 8-by-8 signed multiplication is taking of the order 4.5 µs at a 64 MHz master clock rate. This is far too slow as at least four multiplications are required in well under 31.25 µs.

Searching the assembly code generated reveals that a software multiplier has been inserted. There is no sign of the 'MULWF' hardware multiplier assembly code statement. The adoption of the hardware multiplier would speed this operation up by a factor of fifteen times.

This does not present an embedded systems engineer with any real problems, they will just drop in a very few lines of assembly code. So as an example, we might need an 8-by-8 multiplier that acts on signed numbers and produces a 16-bit product to overwrite the defined behaviour of the C standard. This might look like:

```
signed int8 Signal,Cos_LUT;   // Working with signed numbers
signed int16 Product;

// Rename the multiplier registers to something meaningful
register _PRODH int8 Product_High_Byte;
register _PRODL int8 Product_Low_Byte;

#asm                    // Drop in a few lines of assembly code
MOVF Signal, W         ;// Move the input ADC value to the working register W
MULWF Cos_LUT          ;// Signal*Cos_LUT -> Product_High_Byte:Product_Low_Byte
BTFSC Cos_LUT, 7       ;// Test Sign Bit, skip over next statement if positive
  SUBWF Product_High_Byte, F  ;// Product_High_Byte = Product_High_Byte - Signal
MOVF Cos_LUT, W        ;// Use W as temporary storage of Cos_LUT
BTFSC Signal, 7        ;// Test Sign of ADC sample, skip next statement if +ve
  SUBWF Product_High_Byte, F ;// Product_High_Byte = Product_High_Byte - Cos_LUT
#endasm
Product = make16(Product_High_Byte,Product_Low_Byte);  // Combine 16-bit result
```

The assembly code can make use of simple variables defined in C (e.g. `signed int8 Signal;`). It cannot use more complex constructs such as elements from an indexed array. The new C construct introduced here is 'register' that is an alternative for '#BYTE'. Internal special function registers are referred to using an underscore e.g. '_PRODH'. You will find a text file (sfr.txt) listing these by searching for 'PRODH' within the compiler director structure.

A thorough description of efficient, signed, two's complement multiplication is available on Wikipedia.

EE2A Experiment 5

Wire-Following Sensor and Associated Signal Processing

Feedback Mark Sheet – Paste into the logbook

The following will be assessed during the laboratory session (PGTA to circle, date and sign)

Circuit construction and demonstration:	Yes	Partially	No
Demonstration of working sensor system (35%)	**W**	**WP**	**NW**

Inspection Mark For Log Book:	Could not be better	Good attempt	Room for improvement	Appalling
A 'flick-test' of the log book will be carried out. This will be based on the purpose of the log book – to convey useful information to other engineers and to allow others to carry on with the work.	**4**	**3**	**2**	**1**

At the end of the year, the log-book will be handed in and assessed using the following criteria:

Preparatory Work:
Evidence of adequate preparatory work undertaken outside laboratory (15%). Students to have decided what is 'adequate' as part of their work.
Source Code:
Existence and completeness of the program header (author, date, filename, target device, fuse settings, program function) (5%). Appropriateness and clarity of comments, labels and variable/constant definitions (5%). Efficient use of code (i.e. no redundancy) (5%). Program Elegance (5%). Technical content (10%). Code print-outs are assumed.
Reflective journal:
Care and neatness of preparatory work (written up outside lab) (5%). Sensible attempt at keeping a log-book (written up at the time) to convey engineering information to other professionals. (10%). If this student was a professional engineer who left his/her organization today, could another engineer pick up the pieces in six months time? Conclusions- sensible executive summary & record of the learning experiences gained by the student during this experiment. (5%).

Verbal feedback will be provided during the laboratory.

Timeliness (Spring Term)

Week in which experiment was demonstrated to supervisor (shaded blocks show overrun)

Week 3	Week 4	Week 5	Week 7	Week 8	Week 9	Week 10
Excellent Progress	Excellent Progress	Excellent Progress	Excellent Progress	Good Progress	Getting Concerned	Oops Serious Panic