EE2A

USB, Serial Interfacing, Analogue-to-Digital Conversion and Closed-loop Controllers

1.0 Aims

(i) To design, implement and test a USB and serial interface.

(ii) To implement an interrupt handler, circular receive buffer and command interpreter in C.

(iii) To implement a USB controlled PWM module capable of acting on human-entered text strings.

(iv) To implement an analogue-to-digital conversion sampled at a rate controlled by an internal timer.

(v) To implement a closed-loop control of the armature current of a DC permanent magnet motor where the demand is initiated from a USB text string.

Many microprocessor systems are required to interface with a standard PC. Possibly the simplest method of achieving this is to use a commercial USB interface module connected to the microprocessor via a serial interface, as shown in Figure 1. This implies that the software running on the microcontroller will require First-In, First-Out (FIFO) character buffers. The transmit buffer functions are inherently implemented within such C functions as '*fprint*'. However, the equivalent receive buffer functions are best implemented under interrupt control. Having received a stream of characters, a command interpreter is frequently used to parse the string prior to executing the desired actions. The microcontroller will convert the characters between software and a serial interface with a Universal Asynchronous Receiver Transmitter (USART). This serial interface will in-turn be converted into a USB interface using an FTDI UB232R interface module.

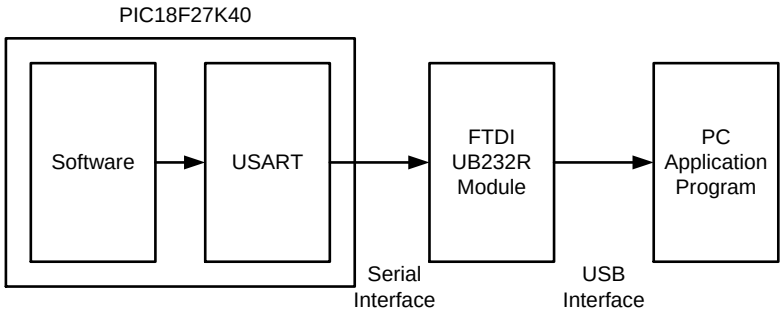


Figure 1: Information Flow Diagram

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 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 Control of LEDs via USB interface

A system is to be designed that following power-up sends a message to the PC such as '*Hello World ver. 1.1*'. The micro-controller should then wait for one of a number of 'Carriage Return' terminated messages such as '*switch on LED 2[CR]*', '*switch off LED 3[CR]*', '*flash LED 4[CR]*'and control one of four LEDs corresponding to the message. Should the message be decoded successfully, an '*OK*' should be sent to the PC, otherwise an '*Error*' message should be returned. It is suggested that you use SSH Tools as the Windows host application (Start > *Programs* > *SSH Tools* > *Putty* then choose "*Serial*" as the connection type followed by the COM port number that is associated with the FTDI driver).

3.1.2 PC Control of LED Brightness using a PWM Module

A system is to be designed that allow a human to control the brightness of an LED via a text string such as '*LED Brightness 47%[CR]*'. A number in the range 0 to 100% should be decoded (interested students may wish to decode numbers with a varying number of decimal places). Should the message be decoded successfully, an '*OK*' should be sent to the PC, otherwise an '*Error*' message should be returned.

3.1.3 Analogue-to-Digital Converter with USB Output

A system is to be designed that allow a snapshot of 1024 samples of an analogue input to be collected when initiated by a text string such as '*Collect Data[CR]*'. The collected data will then be transferred to the PC via the USB interface. Interested students may wish to perform all initiation and reception commands using Matlab and display the data.

3.1.4 PC interface to a DC Motor Closed-Loop Armature Current Controller using a PWM Module

A system is to be designed that implements a closed-loop control of the armature current of a DC permanent magnet motor where the demand is initiated from a USB text string such as '*Armature Current -1980mA[CR]*'. The current value should be in the range +/-2 Amps. An LM298 H-bridge will be supplied to aid you with this task. Demonstrate that stable proportional control can be achieved.

That's it – now do it!

4.2 Preparatory Work

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

4.2.1 Hardware

The PIC 18F27K40 is ideal for this laboratory experiment as it contains a hardware Universal Synchronous/Asynchronous Receiver/Transmitter (USART) suitable for efficient and high-speed implementations of serial interfaces. An asynchronous serial interface will be required to communicate to the USB interface. This USB interface will be implemented using an FTDI UB232R module, see Figure 1.

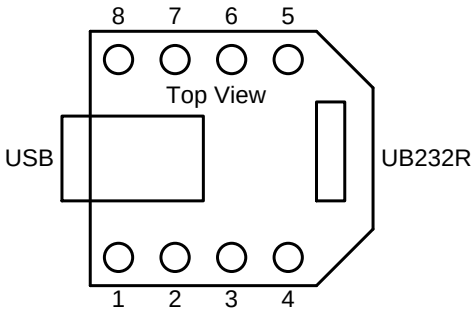


Figure 1: FTDI UB232R interface module

The pin connections are as follows:

Pin	Name	Description
1	GND	0V power pin. Connect to pins 8 & 19 of PIC 18F27K40
2	Vcc (out)	+5V power from USB interface. Only use if no other power supply available
3	CTS	Hardware handshake line (Clear To Send): connect to Request To Send
4	RTS	Hardware handshake line (Request To Send): connect to Clear To Send
5	CBUS1	General purpose hardware output line – not connected
6	CBUS2	General purpose hardware output line – not connected
7	RX	Receive data line: connect to any free pin on PIC 18F27K40 on Ports B or C
8	TX	Transmit data line: connect to any free pin on PIC 18F27K40 on Ports B or C

PW1. Construct a complete schematic circuit diagram of the PIC USB interface. This will comprise a PIC 18F27K40 and a FTDI UB232R module. This will be the circuit diagram that you will wire up within the laboratory. Leave as much of the circuitry used in earlier experiments in-place as possible. The USART can only be mapped to pins on Port B or Port C. Thus it is suggested that you disconnect two LEDs and rewire these to the RX and TX lines of the UB232R module.

4.2.2 Software

PW2. One of the advantages of using the PIC 18F27K40 is that the internal Resistor-Capacitor (RC) oscillator can be used, thus freeing up two pins for other purposes. Similarly, an internal reset circuit can be used, freeing up a third pin. The device then become really easy to use as all it needs is a 0V and +5V supply. Using the CCS wizard and/or the CCS manual, decide which processor fuses should be set in the '*#fuses*' compiler directive. I suggest that you run the processor at between 8 MHz and 32 MHz.

PW3. The PIC 18F27K40 contains a hardware USART. The compiler can only make use of this if the correct transmit and receive pins are defined in the '*#use RS232*' compiler directive. If you incorrectly define these pins then the compiler will insert a software emulation of a UART that consumes significant processor resources. Use the CCS C compiler manual in conjunction with the PIC data sheet in order to correctly define the compiler directive. The TX output of the processor should be connected to the RX line of the UB232R module. Similarly, the RX output of the processor should be connected to the TX line of the UB232R module.

PW4. The PIC 18F27K40 contains a clever hardware feature that allows most internal peripherals such as the USART to be mapped to many of the IO pins. This is described in 'Section 17.0

Peripheral Pin Select (PPS) Module' of the data sheet. You should also read the section on 'pin_select' of the compiler manual e.g.

```
#pin_select U1TX=PIN_C6
#pin_select U1RX=PIN_C7
```

PW5. Serial communication with a PIC requires the use of First-In, First-Out (FIFO) buffers. These must use a finite amount of pre-defined memory and thus the implication is that both input and output pointers are required. Find out about, or deduce from common sense, the need for circular buffer pointers. Show the code needed to allocate memory space and manipulate these pointers in order to implement a FIFO buffer. What flags, or error checking, is also required?

PW6. Asynchronous activities, such as receiving characters from a serial interface and placing them in a FIFO buffer, are best processed under interrupt control. The interrupt routines should be as simple as possible and execute in the shortest possible time, why? Write an interrupt routine to receive a character from the USART and place it in the FIFO buffer. A global scope flag should be set if a carriage return character is received.

PW7. How might a case-insensitive, command interpreter be implemented to recognise commands of varying length?

PW8. When a user presses the 'enter' key on a keyboard, nobody knows how this will be translated into a sequence of Carriage Return (CR) or Line Feed (LF) characters. For example, there could be four combinations 'CR, 'LF', 'CR-LF' or 'LF-CR'. It is a very challenging task to interpret any one of these combinations as a single event – try it!

PW9. Find out how to detect the occurrence of sub-strings within other strings and which hash-include files should be added to your program. You are attempting to implement a very simple 'command interpreter'. For example, white the code to spot '*switch on LED 2[CR]*' within the input data.

PW10. Find out how to extract a variable length number from within the input data e.g. '*LED Brightness 0%[CR]*', '*LED Brightness 99%[CR]*' and '*LED Brightness 15.25%[CR]*'.

PW11. Find out how to sample analogue data at a rate such as 32 kHz as determined by a hardware source such as Timer 2.

PW12. Hence, generate the code to implement a closed-loop armature current controller. It is suggested that proportional control only is considered within EE2A.

4.0 Additional Resources

4.1 Helpful Hints

Let's face it – this is yet another demanding laboratory experiment and is unlikely to work first time. Therefore, experienced engineers would break this experiment down into a number of stages.
Stage 1: Send a simple '*Hello World*' type transmit message to the PC. Two lines of C are required; '*#use RS232()*' and '*fprintf()/puts()*'. No interrupt routines should be incorporated at this stage. If this doesn't work, place this command inside a continuous loop and use an oscilloscope to examine the TX pin of the PIC and the RX line of the UB232R module.

Stage 2: Add an interrupt routine activated on the receipt of a serial character. This routine should do nothing more than flash an LED on-and-off. e.g. '*high-delay-low*'. This will provide the programmer with confidence that the processor is vectoring correctly to the interrupt routine.

Stage 3: Add code to the interrupt routine to implement a FIFO register using circular buffer pointers and to detect a carriage return. Within the main program continuously loop looking for a

'command detected flag' and echo the command to the serial interface using an '*fprintf/puts()*' statement.

Stage 4: Add code to the main program to compare a received command string with a number of pre-defined commands and undertake the required actions.

5.2 Code Issues

Many programmers would include sufficient comments at the start of the code to indicate pin/port connections, e.g.

```
/************************************************************************/
/*          EE2A2 Demonstration UB232R USB Interface Module             */
/*          Designed to run on a PIC 18F27K40 microcontroller           */
/*                Written by Phil Atkins                                */
/************************************************************************/

/***************   Input/Output Pin Definitions   *******************/
/* Port A Pin Definitions                                             */
/* Bit 0 :                                                            */
/* Bit 1 :                                                            */
/* Bit 2 :  LED3                                                      */
/* Bit 3 :  LED4                                                      */
/* Bit 4 :                                                            */
/* Bit 5 :                                                            */
/* Bit 6 :                                                            */
/* Bit 7 :                                                            */
/*                                                                    */
/* Port B Pin Definitions                                             */
/* Bit 0 :                                                            */
/* Bit 1 :                                                            */
/* Bit 2 :                                                            */
/* Bit 3 :                                                            */
/* Bit 4 :                                                            */
/* Bit 5 :                                                            */
/* Bit 6 : ICSPCLK, part of the In-circuit programming interface      */
/* Bit 7 : ICSPDAT, part of the In-circuit programming interface      */

/*                                                                    */
/* Port C Pin Definitions                                             */
/* Bit 0 : LED 1                                                      */
/* Bit 1 : LED 2                                                      */
/* Bit 2 : LED 3                                                      */
/* Bit 3 : LED 4                                                      */
/* Bit 5 :                                                            */
/* Bit 4 :                                                            */
/* Bit 6 : UB232R Module RX Pin                                       */
/* Bit 7 : UB232R Module TX Pin                                       */
/*                                                                    */
/* Port E Pin Definitions                                             */
/* Bit 3 : Master Clear Pin (MCLR), part of the In-circuit programming */
*/
/************************************************************************/

/************************************************************************/
/******************   SERIAL Command  Definitions   ******************/
/*    'L1 ON' - Switch on LED 1                                       */
/*    'L2 ON' - Switch on LED 2                                       */
/*    'L3 ON' - Switch on LED 3                                       */
/*    'L4 ON' - Switch on LED 4                                       */
/*    'L1 OFF' - Switch on LED 1                                      */
/*    'L2 OFF' - Switch on LED 2                                      */
/*    'L3 OFF' - Switch on LED 3                                      */
/*    'L4 OFF' - Switch on LED 4                                      */
/*    'L1 FLASH' - Switch on LED 1                                    */
/*    'L2 FLASH' - Switch on LED 2                                    */
/*    'L3 FLASH' - Switch on LED 3                                    */
/*    'L4 FLASH' - Switch on LED 4                                    */
/************************************************************************/
```

5.3 Useful Code ?

A command interpreter can use one of the string comparison functions such as '*strncmp*'. Such functions will require that leading spaces are stripped off before comparison takes place. The following is a routine that extracts characters from the FIFO buffer and strips off leading spaces.

```
/************************************************************************/
/*                  Obtain Command from Serial Interface                */
/************************************************************************/
void GetCom(char * command_string)
{
   int length;
   char c;
   length=0;
        /* Reset command received flag*/
```

```
        SerialCmdWaitFlg = FALSE;
        /* Ignore leading spaces */
        do
        {
              c=Serial_buffer[Serial_next_out];/* Get a character from the Serial buffer */
              Serial_next_out=(Serial_next_out+1) % SERIALBUFFSIZE;
        } while(c ==' ');
        /* A non-space character has been entered */
        command_string[length++]=c;
        /* Get rest of command string - until space or CR terminated or input string is too long*/
        while ((c != ' ')&&(c != 13)&&(length<20))
        {
              c=Serial_buffer[Serial_next_out];          /* Get a character from the Serial buffer */
              Serial_next_out=(Serial_next_out+1) % SERIALBUFFSIZE;
              command_string[length++]=c;
        }
        command_string[--length] = 0;          /* Null terminate string */
}
```

Are there any functional errors in this code ?

5.4 Code Jigsaw

There is no single correct way of implementing the code – just more elegant, or more maintainable ways. For those really struggling with this exercise, here are some lines of code (in a random order) that may, or may not, be useful.

```
puts("Phil Atkins USB-Serial System V1.0");
if ((((Serial_next_in+1) % SERIALBUFFSIZE) != Serial_next_out) && (Serial_buffer[Serial_next_in] != 0x00))
/* A bad command may have been received */
if (CmdDoneFlag == FALSE)  puts("Bad Command");   Serial_next_in = 0;   Serial_next_out = 0;
enable_interrupts(GLOBAL);            /* Global interrupt enable */
#include <18F27K40.h>   #include <stdio.h>   #include <string.h>
#int_RDA
void serial_isr() /* Serial Receive Data Available */
/* Just make sure that the Serial buffer is empty */
if(kbhit())   getc();
/* Look for a carriage return as this signifies the end of a command */
if (Serial_buffer[Serial_next_in] == 0x0D)
SerialCmdWaitFlg = TRUE;
/* Update buffer pointers unless buffer full or a null character was received */
Serial_next_in = (Serial_next_in+1) % SERIALBUFFSIZE;
char CmdStr[15];            /* Input character string from Serial interface */
char Cmd1[3] = {'L','1',0}; /* Switch on LED 1 */
enable_interrupts(INT_RDA);
Serial_next_in = 0;
Serial_buffer[Serial_next_in] = getc();        /* Get character and place in buffer */
CmdDoneFlag = FALSE;
#fuses NOWDT, NOPUT, NOPROTECT, NOBROWNOUT, NOMCLR, NOLVP, NOCPD
if (SerialCmdWaitFlg == TRUE)
/* Enter continuous loop looking for event driven activities */
if (STRICMP(CmdStr,Cmd1)==0)
#use delay(clock=32MHz)
/* Examine command string */
char      Serial_buffer[SERIALBUFFSIZE];
int       Serial_next_in = 0;
int       Serial_next_out = 0;
short int SerialCmdWaitFlg = FALSE;
short int CmdDoneFlag;
/* Test to see if an Serial command has been executed */
if (CmdDoneFlag==TRUE)
        puts("OK");         /* Send an OK CR LF to the Serial interface to signify completion */
        Serial_next_in = 0;   Serial_next_out = 0;   CmdDoneFlag = FALSE;
while( TRUE )
#use Serial(baud=9600,parity=N,bits=8,errors)
/* Serial Global variables */
GetCom(CmdStr);
#define SERIALBUFFSIZE 64
Serial_next_out = 0;
```

Warning: None of this code jigsaw is able to unravel the problem posed in PW 8.

To Do List

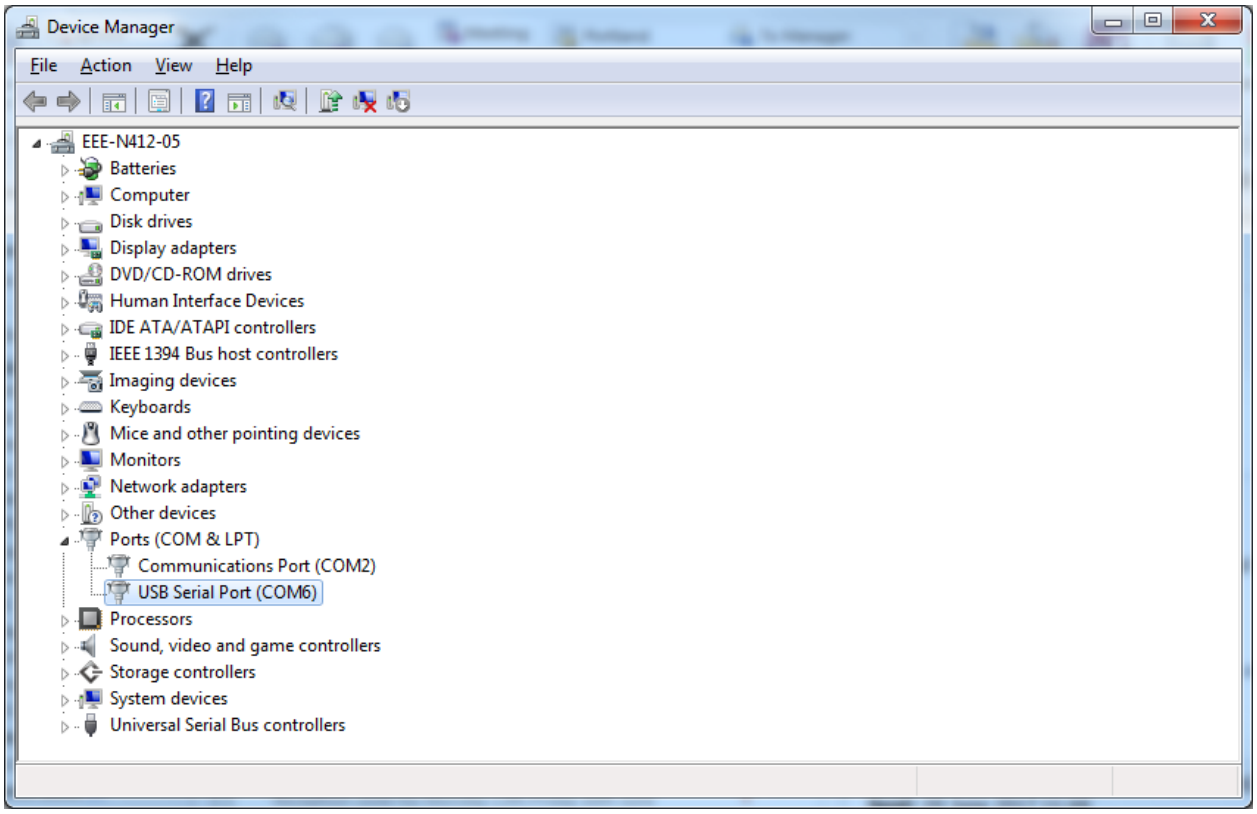
Section 5.4 section needs updating

I probably need a more structured what to look for on the serial interface

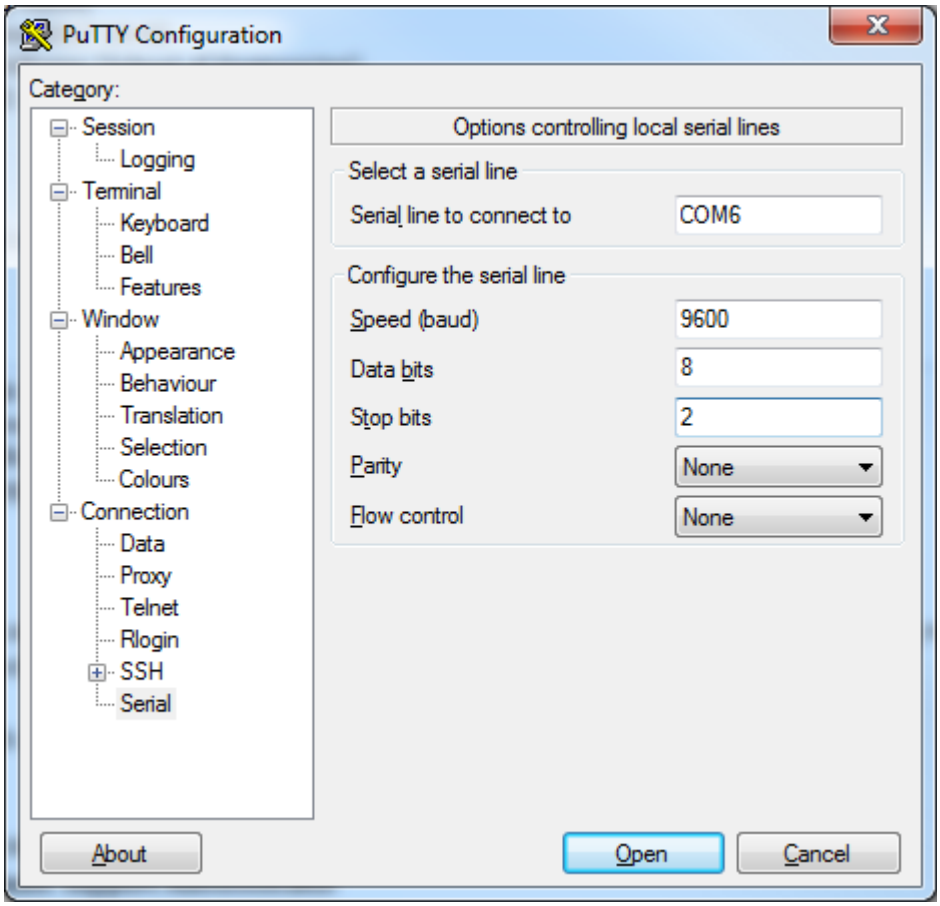
Appendix 1

Virtual Communication Ports

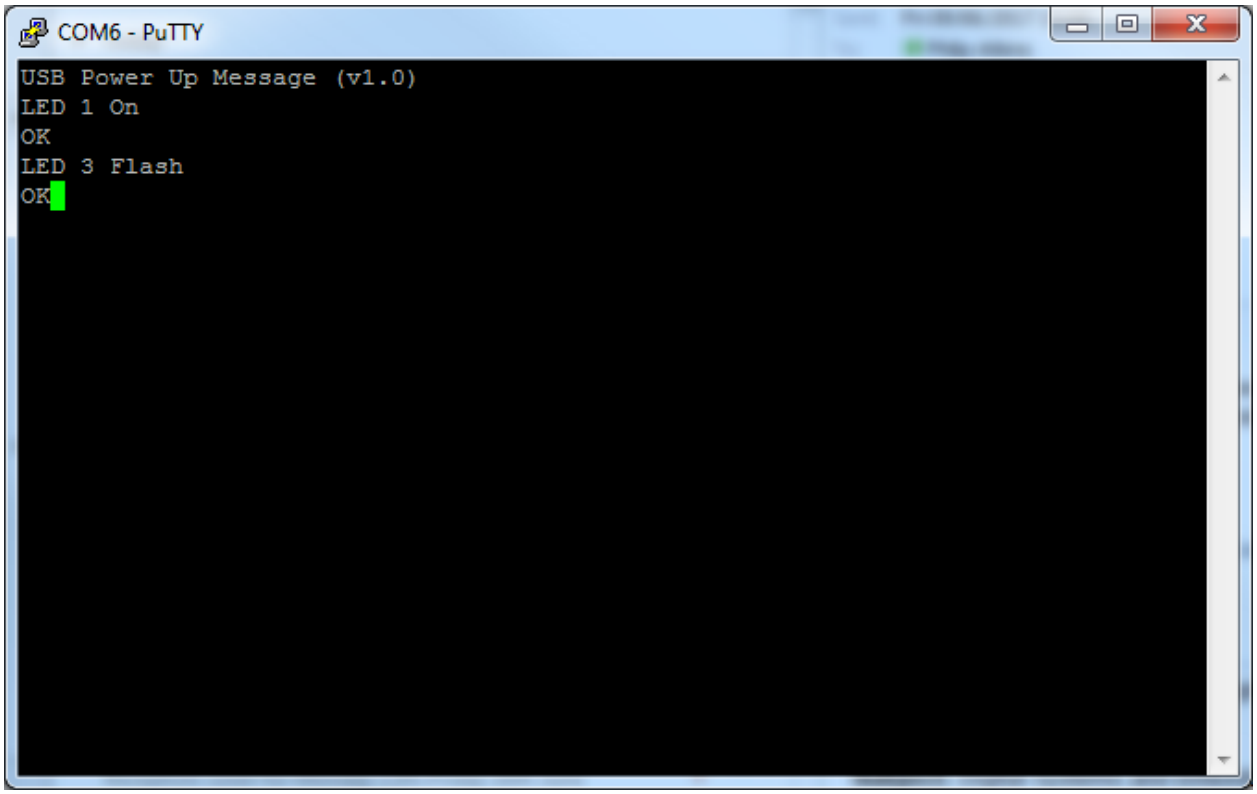
You will communicate from your PC to the microcontroller using a virtual communication port. Each port will be allocated a number by Windows, but the bad news is that this will change from machine-to-machine and possibly every time you connect the USB cable. First, connect the USB cable from the PC to the FTDI UB232R. Next press the Windows and X keys (on Windows 10). Now select the 'Device Manager' and click OK to any messages reminding you that you need Administrative Privileges. Expand the 'Ports' item by clicking on it and you should see something like 'USB Serial Port (COM6)' – now make a note of that number.



We now need a simple communication interface to this device – I suggest you run 'putty.exe'. First click on the 'Serial' radio button towards the upper-right. A text box appears titled 'Serial line' – update this with your COM Port number (e.g. COM6). Click on the 'Serial' Category (lowest left entry in the list) and ensure that the 'Serial line to connect' matches your COM Port number (e.g. COM6). I suggest you run the serial interface of your PIC at 9600 baud, 2 stop bits and 'None' for flow control.



Now press 'Open' and the main communication screen will open. Any characters you type on the keyboard should appear on the serial interface between the UB232R and the PIC. However, you will not see anything on the screen unless you have explicitly programmed your code to do so – this is why you included a power-up message. Most students build-in echoing of characters and a very simple editor.



Technical Note 1

Circular Buffers and FIFOs

In many interrupt-driven embedded controllers, there is a need to asynchronously communicate between routines using First-In, First-Out (FIFO) buffers, see Figure TN1.1. Characters are normally input to and output from the microcontroller under interrupt control. The use of interrupts frees up large amounts of time for the processor to perform other tasks.

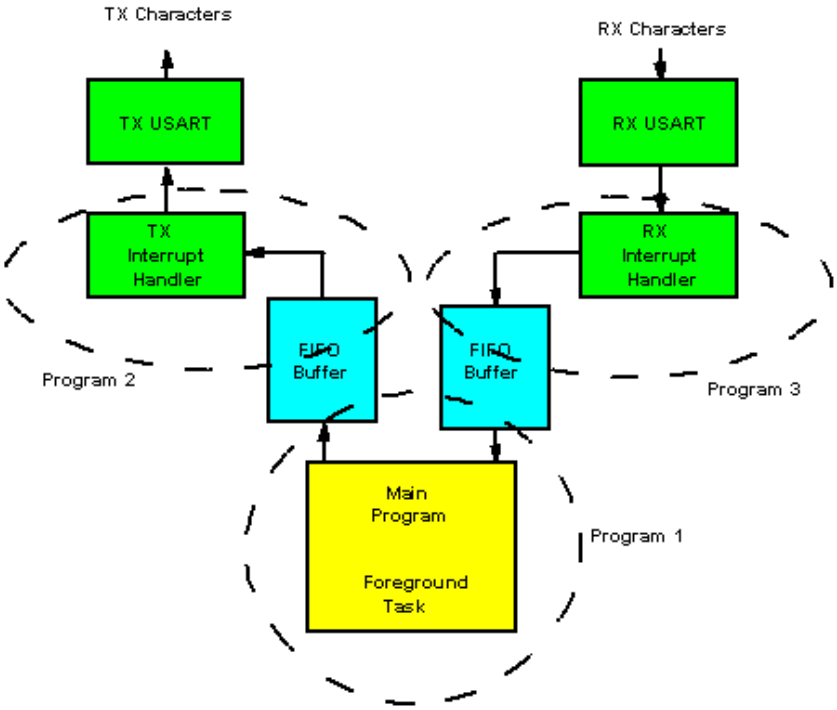


Figure TN1.1: Asynchronous communication between functions in an interrupt-driven environment

In Figure TN1.1, all three programs will be assumed to function independently. The data is transferred using First-In, First-Out buffers. Communication between modules will also require a variety of semaphores (flags) to indicate empty buffers, full buffers and error conditions.

We need to implement First-In, First-Out (FIFO) buffers of sufficient depth so as to not loose characters. A FIFO may be thought of as a pile of cards; new ones are added to the top and old ones are removed from the bottom. The obvious implementation step is to leave the data where it is in memory and to use pointers to access this data, as shown in Figure TN1.2. This will speed up the operation of a buffer significantly.

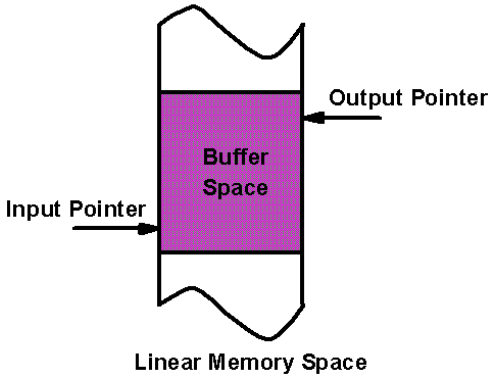


Figure TN1.2: Implementation of a FIFO buffer

When the pointers get to the end of the reserved area they need to wrap round to the start of the buffer. As soon as we have added this extra constraint we have implemented a circular buffer, as shown in Figure TN1.3. Such circular buffers are used in many systems. DSP devices implement indirect address pointers with no programming overhead, whereas you will have to add additional pointer arithmetic.

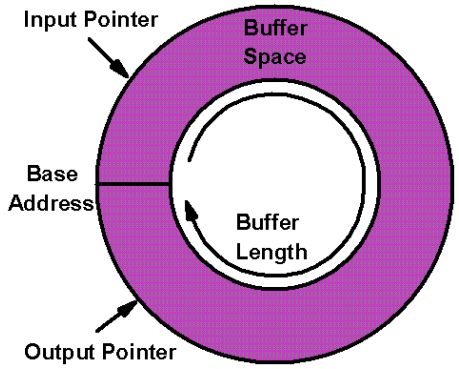


Figure TN1.3: Implementation of a FIFO circular buffer

The C code implementation for an addition to the FIFO would be:

```
/* Place received character in a buffer */
Serial_buffer[RS232_next_in] = getc();
Serial_next_in = (++Serial_next_in) % SERIAL_BUFFER_SIZE;
```

Normally, a circular buffer would continue over-writing characters when the buffer is full – an error flag would be raised to indicate that data has been lost.

A similar buffer pointer ‘Serial_next_out’ would be required to extract data from the buffer. Usually, null characters are returned and an ‘empty flag’ is set if the buffer is empty.

Technical Note 2

Analogue-To-Digital Converter

The PIC 18F27K40 contains a single 10-bit analogue-to-digital converter. Although the data sheet will make this look like it can sample multiple channel, this is achieved by using an analogue input multiplexor and system performance will seriously degrade when attempting to measure the voltage on multiple pins at any reasonable speed.

The schematic block diagram of the analogue-to-digital converter in shown in Figure 2.1. Twenty-four input pins (ANA0 through ANC7) may be switched through to the sampled input of the module. This signal will be compared against two voltage reference signals 'Vref+' and'Vref+'. The module may be clocked by the main system clock, or a unique clock source 'FRC'. Similarly, it may be triggered from a multitude of sources.

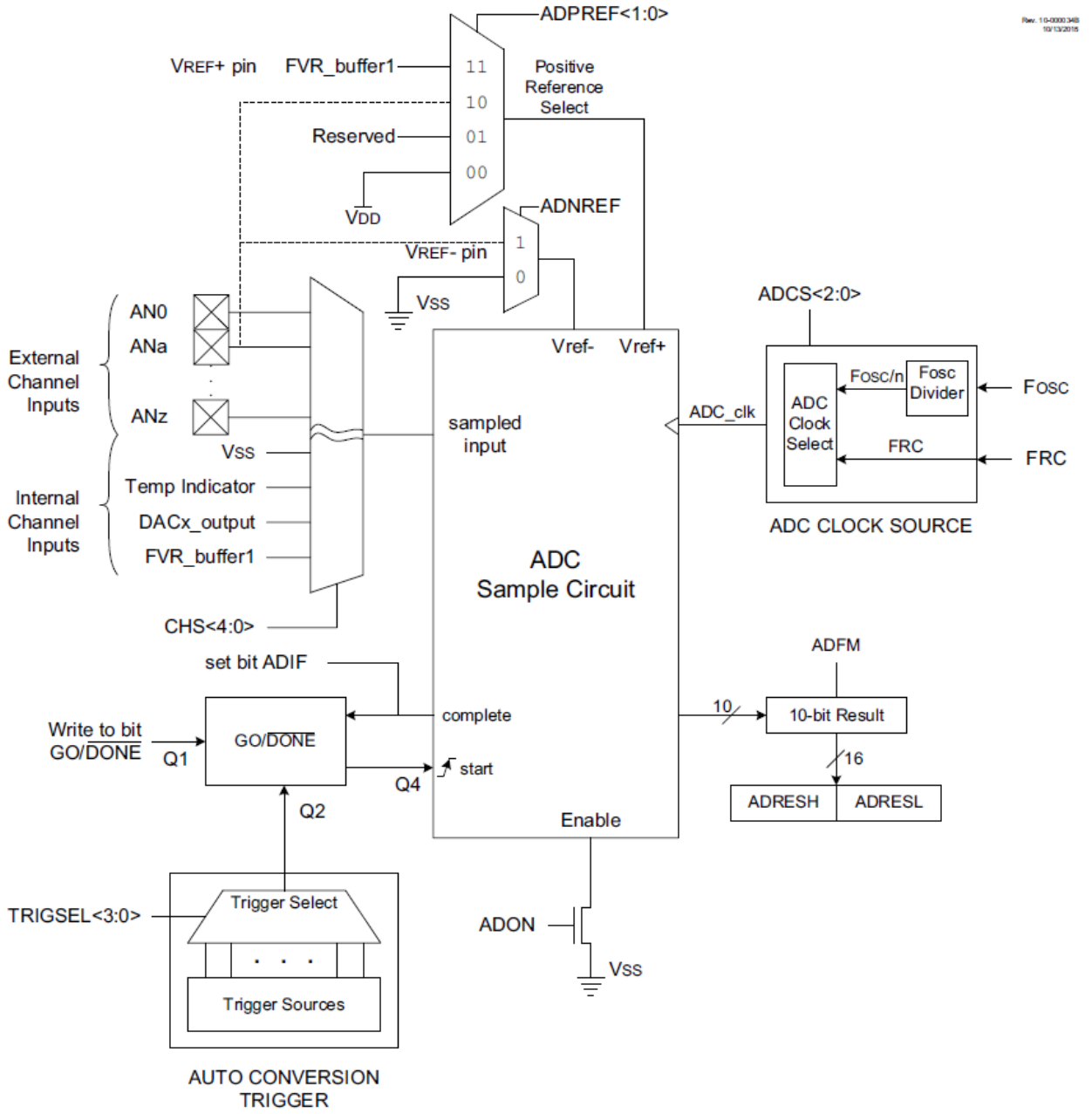


Figure 2.1: Schematic block diagram of analogue-to-digital module

Examining the PIC18F27K40.h include file, we find interesting commands such as:

```
setup_adc_ports(sAN7 | VSS_FVR);
```

This sets up only pin RA7 to be an analogue input 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_CLOCK_DIV_16);
```

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_2v048);
```

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:

```
set_adc_trigger(ADC_TRIGGER_TIMER2);
```

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

```
enable_interrupts(INT_AD);
```

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 stored to a buffer within this interrupt routine. The interrupt routine possibly only contains two lines of code.

Technical Note 3

DC Motor Control Systems

First, delete from your mind any idea of just connecting a voltage to a DC motor – this will not be good enough. For your Integrated Mechatronic Project you will be effectively nesting three control loops: position, velocity and current. For the purposes of this experiment, you will implement the current control loop only.

If you wished to implement a closed-loop, voltage-to-current converter using analogue components, the implementation might be as shown in Figure TN3.1. A high-power amplifier applies a voltage to the armature of a DC motor. The value of current flowing is measured using a low-value sense resistor. The potential difference appearing across this sense resistor is compared to the input demand signal and the difference amplified to form the closed-loop current controller.

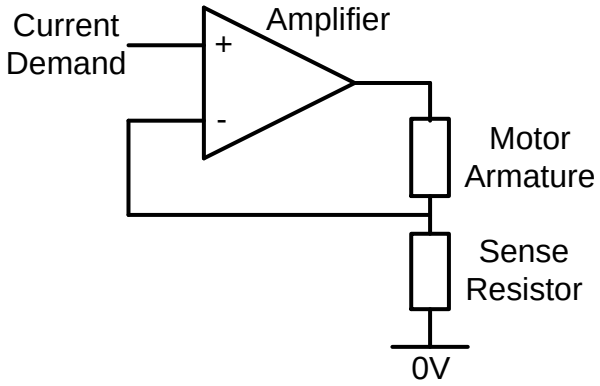


Figure TN3.1: Analogue Current Control Loop

Now this analogue closed-loop current controller takes the classical form as illustrated in Figure TN3.2. The plant will include gain, a pulse-width-modulator, an H-bridge driver and a DC motor armature. The feedback will contain the sense resistor and the analogue-to-digital converter. It is highly likely that this system will not be unconditionally stable.

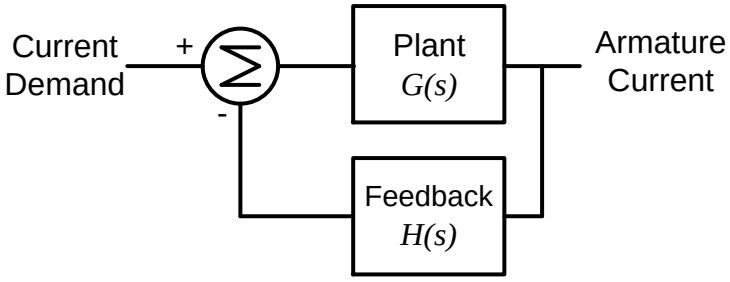


Figure TN3.2: Classical closed-loop feedback control

For the purposes of a simple introduction, we will consider the system as continuous-time is the *s*-domain and add in a transit-time delay as a result of the sampling rate of the microcontroller. Obviously, any rigorous analysis will be conducted in the *z*-domain.

Consider a simple model of a DC permanent magnet motor shown in Figure TN3.3. If we were to externally spin a un-powered motor, it would act as a generator to supply a potential difference, V_{EMF}, where

$$V_{EMF} = K_b\ \omega(s) \tag{1.1}$$

When acting as a motor, this EMF is subtracted from the applied armature voltage, $V_A(s)$.

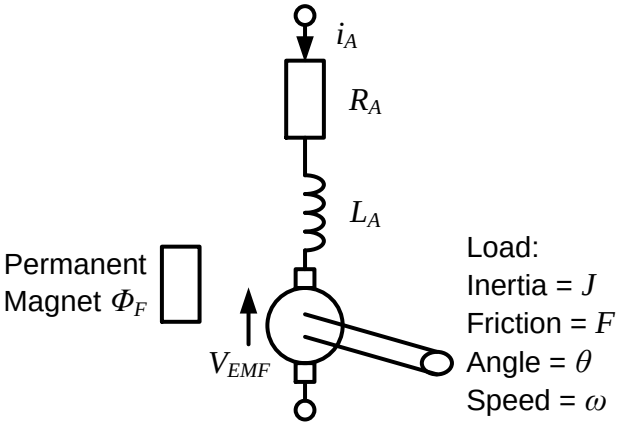


Figure TN3.3: Simple model of a DC permanent magnet motor

Thus the armature current, $i_A(s)$, is

$$i_a(s) = \frac{V_A(s) - V_{EMF}}{\left(R_A + L_A s\right)} \tag{1.2}$$

This equation tells you that the armature current changes with time and the speed that the motor is running at. The armature will consume very high currents when the motor is starting from stationary and relatively small currents when it is spinning under no-load conditions – hence you need a control-loop.

The torque, T, generated by the motor results in the acceleration of the vehicle. This torque is proportional to the magnetic flux originating from the permanent magnet used in the field of the motor, ϕ_F, and the armature magnetic field, ϕ_A.

$$T(s) = k\,\phi_A\,\phi_F = K_A\,i_A(s) \tag{1.3}$$

The load torque for a rotating inertia, *J*, and friction, *f*, is

$$T(s) = J\,s^2\,\theta(s) + f\,s\,\theta(s) \tag{1.4}$$

Therefore, the transfer function of the motor and load is

$$G(s) = \frac{\theta(s)}{V_A(s)} = \frac{K_A}{s\left(Js + f\right)\left(L_A s + R_A\right) + sK_A K_B} \tag{1.5}$$

This is third order system and will become unstable if the loop gain exceeds unity at the point at which the phase shift is 180 degrees. You can model this transfer function as a function of frequency in a very few of lines of Matlab. Thus an interesting problem for your integrated mechatronic project is how to determine all the parameters. In order of increasing difficulty:

- Measure the armature resistance, R_A, at say 1 A without the motor rotating (known as a locked-armature test). The value will vary with applied current and will typically be of the order of 2 – 8 Ohms.
- Measure the armature inductance, L_A, at say 1 A again without the motor rotating. The value will vary with applied current.
- Measure the inertia, *J*.
- Measure the friction, *f*.
- Measure the constants, K_A *and* K_B.

Any group that is able to document a sensible approach to all these measurements will surely impress the assessors.

Implementation Parameters

Your current control system will be implemented in software. At every stage we must analyse the effects of our component choices on the scaling factors introduced within the control system.

H-Bridge Driver

You have been supplied with an LM298, dual H-bridge driver capable of controlling two motors at up to 2 A per motor. The circuit diagram is shown in Figure TN3.4.

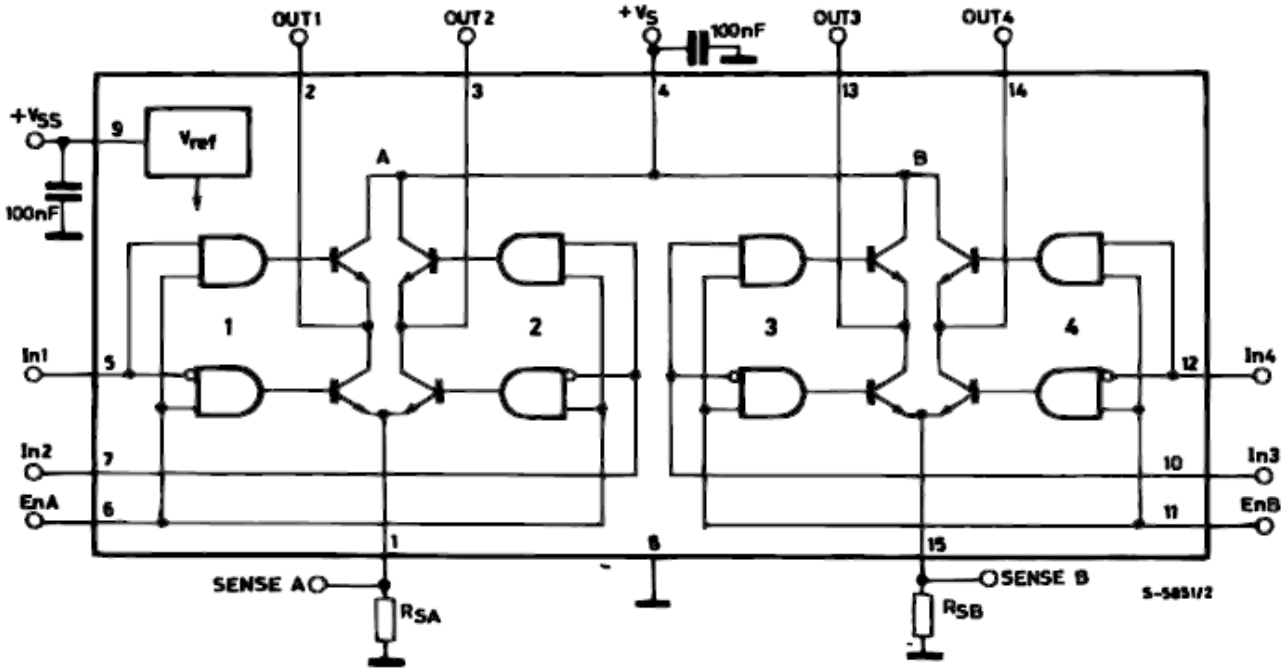


Figure TN3.4: LM298 Dual H-bridge driver

The first motor is connected between 'OUT 1' and 'OUT 2', whilst the second motor is connected between 'OUT 3' and 'OUT 4'. A total of eight external fly-back diodes will be required (already connected to your board, but limited to 1 A current flow). Usually, two PWM signals are applied to 'En A' and 'En B' to control the mean currents in each of the motors. The signals 'In 1', 'In 2', etc. are used to control the direction that the motor is running in. Two sense resistors, 'R_{SA}' and 'R_{SB}' are connected between the lower part of the H-bridge and ground to measure the modulus of the current flows in the motor. The sense resistors added to your board have value 0.5 Ohm. Thus the transfer function is $0.5V/A$.

This analogue signal will be connected to the analogue-to-digital converter of the PIC 18F27K40. This is a 10-bit device with a programmable reference voltage of 1.024 V, 2.048 V or 4.096 V. Now, the higher you set the reference voltage, the greater will be the armature current that the software can handle without overloading – but the worse the resolution. As an example, set the reference voltage to 2.048 V, the resulting transfer function in units of integers-per-Amp will be:

$$\frac{0.5\times 2^{10}}{2.048} \quad \text{integers per Amp} \tag{1.6}$$

In the software, the proportional control loop may be implemented by two lines of code, e.g.

```
Error_Integer = Gain*(Demand_Current - Measured_Current);
set_duty(Error_Integer);
```

The PWM driver has a maximum resolution of 10-bits, but this may be less if the ratio of clock period to cycle period is less than 1024. The transfer function relating the voltage applied to the armature to the armature current is:

$$\frac{(\text{Supply Voltage}) \times (\text{Error\_Integer})}{\leq 1024} \quad \text{Volts per Amp} \tag{1.7}$$

You will notice that the loop-gain is proportional to the supply voltage. As this is specified as varying over a range of nearly 4:1, you may find that your vehicle is stable at a low supply voltage, but unstable on Race Day when the voltage may be varied.

Finally, your software adds an unwanted delay that will make your system less stable. We know that the Fourier transform of a time-domain signal, x(t), is:

$$x(t) \Leftrightarrow X(\omega) \tag{1.8}$$

Now if we delay the signal by a time tau, often the sample time of a digital system, the result is:

$$x(t+\tau) \Leftrightarrow \exp(j\omega\tau)X(\omega) \tag{1.9}$$

The amplitude remains unchanged, but an additional frequency-dependent phase shift $\exp(j\omega\tau)$ is added to the signal. When $\omega = 1/\tau$, an extra 360 degrees of phase shift has been added.

The lessons are:

- Digital control systems are highly likely to be less stable than analogue control systems.
- Make the sampling frequency as high as possible, to minimise the transit-time delay. This will always be significantly higher that the Nyquist criteria.
- In time-honoured fashion, build-it, wind the gain up until it oscillates, then back the gain off until it is acceptably stable under all conditions. Finally, analyse it properly and pretend that you knew everything that you have just learnt by trial-and-error before you started the project.
- You must learn how to saturate the 'error' signal at the limits of the PWM control range. *Big Hint*: Might negative error signals be interpreted as very large positive signals?

Debugging Hints

This experiment is designed to provide at least one challenge for all students – so, of course, it is unlikely to work for you! Now your task is to start-over with the simplest code and circuit diagram possible. So what might be your plan? Perhaps:

- Reassure yourself that your processor is working. I always build in a routine that flashes any LEDs on the board, I will comment out these lines before shipping a product to the customer. For example:

```
void main()
{
   int Intro_Flash_Counter;
   IO_Port_Direction.LEDs = 0b00000;
   // Prove system is working by flasking LEDs five times
   for (Intro_Flash_Counter=0;Intro_Flash_Counter<5;Intro_Flash_Counter++)
   {
      IO_Port_Latch.LEDs = 0b1111111;  // Switch on LEDs
      delay_ms(500);                   // Waste time for half a second
      IO_Port_Latch.LEDs = 0b0000000;  // Switch off LEDs
      delay_ms(500);                   // Waste time for half a second
   }
}
```

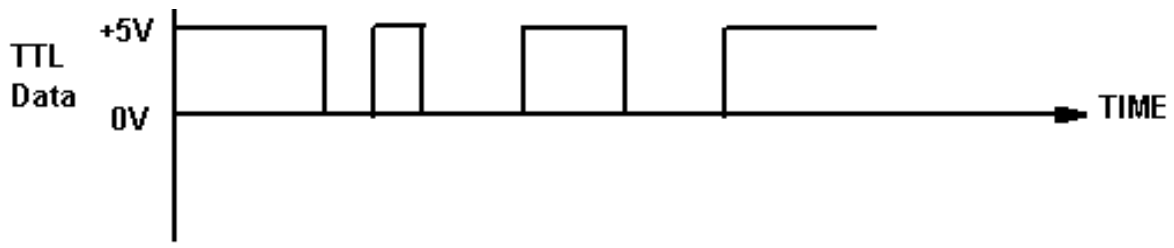
Now we know that we need to setup the serial interface somewhere near the top of the code. In this case, I have setup the first of the two UARTS to run at a data rate of 9600 baud. If a receiver error is encountered, the device will continue working instead of closing down awaiting an action to detect and correct the fault. Two additional compiler directives are used to route the input (RX) and output (TX) of the UART to the pins you have chosen.

```
#use rs232(uart1, baud=9600, ERRORS)
#pin_select U1TX = PIN_C6        // UART1 TX out pin
#pin_select U1RX = PIN_C7        // UART1 RX out pin
```

- If you can't see characters appearing on the PC, the next stage is to debug the hardware with your oscilloscope. For this, you need to repeatedly output a known character to the serial interface. After the light flashing code add:

```
   while(TRUE)
   {
      printf("A");
      delay_ms(10);
   }
```

This will output the character 'A', 100 times per second to the pin you have defined as a TX pin. Configure the oscilloscope to examine a 0 V to +5 V signal with a time-base of 200 µs per division. Probe the TX output of the PIC 18f27K40 and ensure that you have a 0 V / +5V signal of 8-bits at 102 µs per bit.



EE2A Experiment 4

Closed-Loop Current Controller and USB Interface

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 USB interface, PWM output and closed-loop armature current control (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 1	Week 2	Week 3	Week 4	Week 5	Week 7	Week 8
Excellent Progress	Excellent Progress	Excellent Progress	Excellent Progress	Good Progress	Getting Concerned	Oops Serious Panic