Lab Session 1

Equipment and System Familiarisation

Part I: Introduction

For those of you who are familiar with personal computer operating systems, and have some experience in programming in C, this first lab will be rather easy. During this first session, the idea is to become acquainted with the development environment (which you will use to write, compile, debug, download and run your programs), and also the AT89C51AC3 chip and development board.

Your development board is connected to the PC's USB port, which has been configured as a RS232 serial port. A number of peripherals have been constructed that can be connected to the development board, such as:

- an I/O (short for Input/Output) module consisting of 8 buffered LEDs, seven-segment-displays, and various connections,
- a digital to analogue converter (DAC),
- a pulse width modulation (PWM) and DAC motor driver,
- a dot-matrix liquid crystal display (LCD),
- a ten-digit keypad.

For this lab we will be using the I/O, Keypad and DAC modules. You will become familiar with all of the other peripherals as the labs progress.

Note: The computers you are using may be reconfigured between the writing of this manual and you completing the lab session. So follow the instructions of your demonstrator to access the software. The following two paragraphs may or may not be accurate for your initial log-on.

Log on, supplying your username and password if you have used the ECS network before (this year), if not enter "register" as both user name and password. Make sure the domain is set to `ECS.VUW.AC.NZ(<computer name>)`. If you are having difficulty logging on, ask your demonstrator for help.

Using My Computer/Windows Explorer, create an ECEN301 folder in your H: drive [default source dir = H:/Atmel Studio/7.0]. During the lab sessions you will save your work here in folders relevant to the lab being undertaken. The skeleton code for the current and future lab sessions is available in the course site under ECEN301 → lab files. Project templates can be loaded via Atmel Studio. Download the files into your newly created folder so that all skeleton code is now available on your H: drive.

Refer to the "Atmel studio quick start guide" for instructions on how to create and compile a new project.

The first project you will create is "portout" and we will use this project to demonstrate how to perform simple I/O. Create a project called "portout" and then locate the file "portout.c" and add it to your project as per the instructions in the quick start guide.

Now look at `Portout.c`. You should recognise it as being written in C, and that it is a very simple program designed to output a specific binary value out of port 1 of our micro controller. **P1_0** indicates bit zero of port 1, similarly **P1_7** indicates bit seven, ie: the 8th bit of the same port. We assign specific values to them by the Boolean expressions, TRUE and FALSE, or 1 and 0.

At this stage, it is useful to connect up the microcontroller to the PC. We need to do this in order to setup the flash tools. Connect the Microcontroller using a 9V DC power supply and use a USB cable and connect it to the PC. Both the power supply and the USB cable must be connected for the micro to communicate to the PC. Ensure that the UART switch on the box is turned to the 'on' position. There are two indicator LEDs, one beside the USB, and one beside the DC jack. **Check that both of them are on before continuing. If you are having problems, ask the demonstrator.**

Before compiling your project, you will also need to configure the flash tools. Refer to the quick start guide for instructions on how to do this.

Build/compile your project. If the compile was successful, your Build Results should tell you there were no errors. If you did get errors, congratulations, you've made your first mistake. Ask your demonstrator to help you fix it.

Note: when coding, you should check the compiler output very carefully. If the output of the compilation results in an error, this is a sign that something has gone wrong. It is useful to scroll up the build result window to see where things have gone wrong.

If you open a file explorer and look in the folder that you have created the project in, you will notice that a number of new files have been created. The end result is the .hex file, which contains the bit stream (binary file) that will be downloaded to the board.

The next step is to see your program running, (it is rather useless otherwise!). To do this, we need to download the compiled program to the microprocessor board. The steps for preparing the micro board for programming are:

- Make sure your board has power, and the USB cable is plugged in.
- Make sure the UART is switched on.
- Press and hold the Program button.
- While holding the Program button, press and release the Reset button.
- Release the Program button.

Use the Flash tool to download the program. After a few seconds, your program should be finished downloading. Press the RESET button to run the program. **You must press reset after each new upload to run your program.**

Connect an I/O module to the Micro module so you can see on the LEDs of the I/O module the value of port 1. Verify that the LEDs match the software (ask the demonstrator if this is not the case).

So we have now been into an edit environment, compiled a program, and "downloaded" it onto another microprocessor, from which our program was run. Pretty simple stuff so far isn't it? This may seem trivially easy, but it is vital that you get a good feel for the system before you attempt anything too complicated.

Open up MyComputer and look in your `portout` directory. Try opening up `portout.rel` by double clicking on it. The "Open With" dialogue box should appear. Make sure the "Always use this program to open these files" is **unchecked**, and select Notepad from the list. You should now be able to see the contents of the object (.rel) file. If you are prompted to save any of the files you are viewing, respond by clicking `No`. Do the same with `main.hex`.

Q1 Why can you not view the contents of an object or executable file? What do you think is the purpose of the .rel file?

Part I: Using ports to output data

You are now going to create your own programs to perform specific output tasks. Create a new Project called Count, and set its location to `H:\ECEN301\Count\.` (If it asks to save the current project, click Yes). Add the `count.c` file to your new project.

We do not need to set every single bit individually when we are writing to a port. This program sets the port value by a hexadecimal number.

`P1 = 0x00;` is equivalent to

```
P1_0 = FALSE;        P1_1 = FALSE;
P1_2 = FALSE;        P1_3 = FALSE;
P1_4 = FALSE;        P1_5 = FALSE;
P1_6 = FALSE;        P1_7 = FALSE;
```

but is much more efficient. Note that the '0x' before the number indicates the number is in hexadecimal form. This program initially sets all bits to zero, and then increments them. Have a look at the contents of this program and then build/compile it, ensure it is error free, download it and report on the results. **NB: You will need to configure the Flash Tools every time you create a new project**. Modify the program if necessary to provide a more satisfactory output. **Show this to your demonstrator.**

The files that are produced when compiling the project are named after the project, so for every new .hex file you wish to create you will need to create a new project. You can also remove c files from the project and replace them with other ones, if you are not worried about overwriting the old hex file.

Write a program to set 4 port bits hi, wait for a length of time, then make those 4 bits lo, and set the remaining 4 bits hi. Have this repeating indefinitely. To save retyping large segments of this program out again, you can create a second copy of the file `count.c` by using the `Save As` option from the `File` menu. When you are prompted for the file name, change it to read `changed.c` and hit `Enter` (or click on the SAVE button). Now remove the `count.c` file from the project, and add in `changed.c`.

Whenever you work on a new program get in the habit of saving it to a new C file. It's often useful having older programs to refer back to, and it makes it easier when it's time to attach your code for your lab report.

There are more ports available to us than just number 1. Identify which ports are available to use as outputs, and modify your program to use an alternative port. Only some of the bits in port 4 can be used as outputs, take careful note of these because you may find them useful in later experiments where all other ports are used. *Clearly write these results in your lab report*. Report on any hardware alterations that may be necessary.

Q2 List which pins/ports can and cannot be used. Which port will affect the UART, and what do you think the UART switch does?

Create another program that takes an input from port 2, reads it and outputs the same value to port 1. **Download this program and show it to your demonstrator.**

Hint: Because at this stage you are not expected to be totally familiar with C and the workings of the compiler environment, most of the programming will already have been done for you. In order to implement the input routine, you may find this procedure useful:

```
unsigned char ReadInput ()
{
  static unsigned char Input;
  Input = P2;
  return Input;
}
```

Appendix for Lab Session One, part 1:

THE MICRO DEVELOPMENT BOARD

The micro development board you will learn more about as you go on. At this time just be aware of the input/output ports which are available, (ports 0, 1, 2, 3 and 4). Connection to and from these ports may be made by connecting the ribbon to the associated connectors. Initially the board should be set up with the ribbon cable feeding into port 1. To program the microcontroller, first ensure the `UART` is switched on. While holding down the `PROGRAM` button, press and release the `RESET` button. Then use Keil to download the .hex file from the PC. Once downloaded, press the `RESET` button and program execution should begin.

THE INPUT/OUTPUT MODULE

Finally the I/O (Input/Output) module you have is effectively just a means of inputting data into the micro, and similarly displaying output from it. The switch that is located on this module, allows you to select whether the device is to be used for input or output. With the switch in the "Ext" (External input) position, we can see displayed on the LEDs the results of the information coming in through the connector ribbon. With the switch in the “Int” (Internal input) position, we can use the switches to provide an input signal into the ribbon cable. Note that the Digital Out connection should be used. Finally, power needs to be supplied to the I/O module through the plugs at the top of the module. Use +5V.

Handy C code operators

Operator	Effect	Example
>>	Shift Right	0x40 >> 3 = 0x08
<<	Shift Left	0x01 << 5 = 0x20
%	Modulo (remainder)	1234 % 100 = 34 1234 % 10 = 4

Part II: Keypad Scanning

The first part of the lab session was very much an introduction to the system and to the environment that you will be using and as such, most of the material was presented in a very basic form where you were led through every step. As the labs progress, you will be expected to undertake more of the actual program development by yourselves, and your write-ups will need to include such things as the problems that occur, how you isolate them, and ultimately how they were solved (if they can be solved at all).

At your bay you should have a numeric keypad with an LCD display. The keypad is organised in a matrix fashion, with 4 rows and 3 columns. Plug the keypad into port 2, and an I/O module to receive the output on port 1.

Create a new project and call it Keypad, setting the project location to `MY H:\ECEN301\keypad.` Add the library file `ECEN301libsdcc.lib`. These files are available from the course site or if you can't get them, the lab demonstrators will have a copy on a memory stick. The `ECEN301libsdcc.lib` is a previously compiled "library", that is, it is not a program by itself, but contains routines that can be employed by other programs. As you progress in the course you may be required to write source code that performs better than the routines in this library. As `ECEN301libsdcc.lib` is a library file, you are not able to examine its contents.

Notice the line: `#include “ECEN301libsdcc.h"`

This is a relative path pointing to the `ECEN301libsodcc.lib` header file, which tells the compiler what functions are available in the library file. It is relative to the location of the .c file, not necessarily your project. Look in the appendix or open the file up from My Documents to see what functions are available. Tip: If the positioning of the libraries and the .rel file is not working, consult your demonstrator. Always make sure that the object and header file are in the same place.

The function relevant to this lab is:

```
unsigned char sampleKeypad()
```

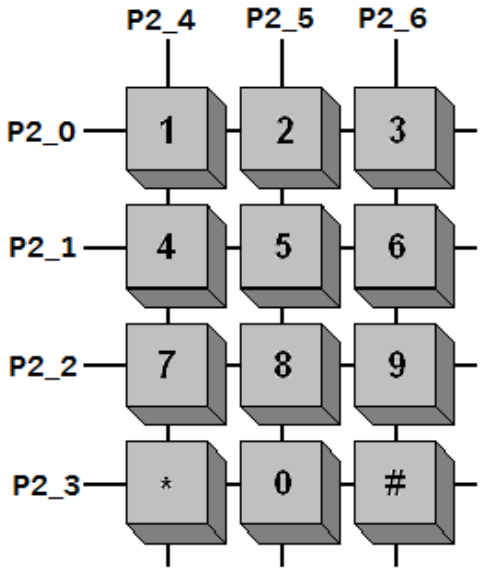
Now, get the project `keypad` going so that the I/O module displays some representation of the key being depressed on the keypad. Note that `main.c` will NOT work without some minor changes. Show this to your demonstrator.

Note that the 4 rows of the keypad (P2_0, P2_1, P2_2, P2_3) are configured as outputs and during the scan cycle these four lines will strobe, while at the same time the columns (P2_4, P2_5, P2_6) are reading the activity. When a key is pressed during the scan cycle, this is detected at the inputs.

Q1 What does this program display when there is no key depressed?

Q2 Describe the keypad’s behaviour when multiple keys are pressed at the same time.

Q3 What other issue needs to be addressed in the sampleKeypad routine?



Keypad Pin Connections

Part III: Digital to Analogue Conversion (DAC)

Supplied is a digital to analogue converter on a board with several other components. A schematic diagram is included in the appendix if you wish more information on how the circuit functions. Refer to these now, and at least satisfy yourself as to the form of the DAC's power supply, and the Analogue and LED outputs.

Create a new project called Sinefn (short for sine function) setting the project location to `MY H:\ECEN301\sinefn.` Add to it the `sinefn.c` file. This is a skeleton of a program, which you are required to complete, that generates a sine wave on the output of the DAC. Get this program working, display it on the Digital Oscilloscope, then calculate the frequency and the amplitude of the resulting sine wave. Ask the demonstrator if you are having problems obtaining a satisfactory display.

Hint: You will need to supply an argument to the SIN function in radians, and have the output changing from 0 to 255. This is NOT a programming course, so if you are having difficulties, ask the demonstrator. It is the results we are interested in, and there is no need to spend excessive amounts of time over minor programming concerns.

Try to increase the frequency of the output and hence determine the maximum frequency you can satisfactorily generate. Sine functions are generally generated internally by some power-series expansion formula (lots of multiplications and divisions) so they can take a long time to calculate each output value. Look up tables and interpolation is usually a faster method.

NOTE: When using the math.h file, only certain single letter variable names are allowed. Using variables other than the single letter ones may break your program. Because of this, the more comments you have in your program the easier it will be for demonstrators to read your code if you have problems.

```
e.g. int angle; -> int x;
     int data; -> int y;
     Data = (sinf(angle)*687) + 2;
     -> y = (sinf(x)*687)+2
```

Q5 Report on how you have set up the hardware, and any affects you notice on the DAC sine wave as you increase its frequency.

Part IV : DAC Motor Control

Okay, so you have had a look at the output of a DAC on an oscilloscope and have (hopefully) managed to get some LEDs to light up when they are supposed to. In the "real" world this is not of much use, so we shall now attempt to control something a bit more physically meaningful. To explore this, you will be provided with a 12 volt DC motor, and an output motor driver box. The schematic for this driver circuit is included later in this lab manual. Familiarise yourself with how this driver circuit operates, and then connect it to the DAC module and to the motor. If you are unsure of how to go about this, consult your demonstrator.

Once this has been set up, modify the previous project to produce a slowly changing DAC output. Then download the code and run the motor from the DAC output. What do you see happening? Make sure you understand how the driver circuitry works.

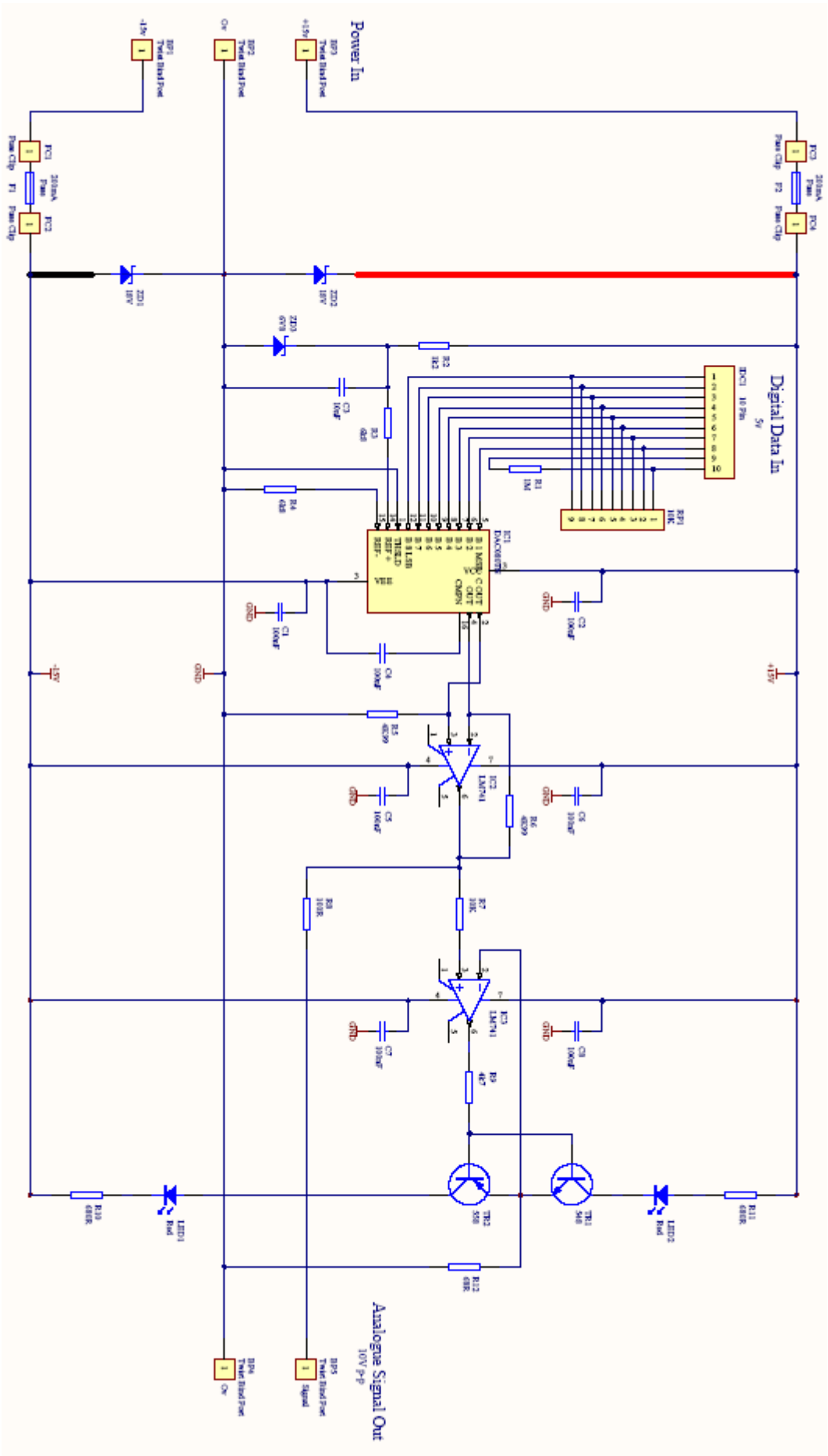
Q6 Using the oscilloscope, measure the DAC output voltage which goes to the motor driver. Comment on the resulting waveform in your write up, and also *briefly* explain how the motor driver module works.

Now write a program that permits you to control the direction and the speed of the motor by using the I/O module. Have speeds ranging from 0 to indicate stopped to 127 to indicate full speed. Use bit 7 to indicate direction, i.e. bit 7 on indicates forward direction, bit 7 off indicates reverse. Show this to your demonstrator. Record the following results:

a) The threshold (or minimum) voltage required to get the motor moving in the forward direction (from stopped), and the corresponding switch settings.

b) The threshold voltage required to get the motor moving in the backward direction, and the corresponding selection.

c) The maximum and minimum obtainable DAC output voltage. What can be said about the operation of the motor at these voltage levels?


Power In
Digital Data In
5v
Analogue Signal Out
10V P-P

Lab Session 2

LCD Displays and Analogue to Digital Conversion

In this laboratory session, you will learn how to use the ADC to make a measurement and to then use the LCD display to show the result.

Part I LCD Display

Each character on the display is represented by a 5 × 8 dot matrix grid. The display itself has two ports, data (8 lines) and control (3 lines). Plug the data lines into Port 0, and the control lines into Port 4.

Create a new project, add the file `lcd.c`, and also the object file `ECEN301libsodcc.lib` from the lib folder. The functions relevant to the LCD are:

```
void initLCD()                      Initialises dot matrix display
void writeCharLCD(char c)           Writes a character
void writeLineLCD(char * s)         Writes a string of characters
void setLCDPos(unsigned char p)     Sets the cursor position
void clearLCD()                     Clears the display
```

For the moment you can use these procedures to become familiar with the display, but at a later date you may have to write some LCD control procedures yourself. What you must do in this section is:

1) Write a program to display a standard character string on the display
2) Using the string library display a numeric value and a character string simultaneously.
3) Write a program to generate a counter, and display it on the LCD.

Part II Analogue to Digital Conversion

Create a new project, and add the file **adc.c**

This file contains one procedure that will produce a digital approximation of the analogue voltage coming in on bit **0** of the **ADC** port. Make sure you read the ADC section of the micro datasheet from page 105-112 to understand what is going on.

The ADC port of the micro shares bits with port 1, with a few differences. Pins 1–8 (ie, bits 0–7) should only be used as inputs to the micro. Pin 9 is connected to the micro ground (0 V), and Pin 10 is connected to AV_{ref}, the maximum allowable input voltage, which is approximately 3 V. (You can measure this if you want, just use the ADC port.)

Design a voltage divider circuit using the resistance substitution boxes and a discrete resistor to provide the ADC with an adjustable analogue voltage ranging from approximately 0 to AV_{ref} volts. Check with your demonstrator **BEFORE** wiring your circuit up. To test your ADC you can use the sampleADC() function of the ECEN301libsodcc library, showing the result on the LCD. Show this to your demonstrator and report fully on your observations of its operation. You can use the **Breadboard Modules** to conveniently add components to your system. Ask your demonstrator if you need help understanding them.

Now write your own `sampleADC()` function (you can choose to do this later). A pseudo-code example of how to do this is provided in the ADC section of the datasheet on page 108. **(Do not blindly copy it. There is a mistake in print.)** Special Function Registers such as ADCON, ADCF, ADDH and ADDL can be directly read/written. The code knows where to find them because they are listed in the at89c51ac3.h header file.

Q1 Explain what all of the bits in the ADCON and ADCF registers mean.

Part III Temperature measurement

With your equipment is a thermistor (assume for the moment that its resistance at 25 degrees Celsius is 4700 Ω), that you can use to provide the ADC with a temperature dependent input.

Design a circuit, and write the appropriate software, so that you can log the temperature of the room, sampling several times a minute, and display a sliding average over 5 samples. Calibrate it to an appropriate level of accuracy (it will probably be easiest to use ice-cold water), and use this as part of a temperature logger. **Show this to your demonstrator.**

***Warning*:** Do not assume that your thermistor will be linear in operation. (Do not necessarily assume that your thermistor will not be linear either.) Also do not assume that it will be the same as anyone else's. Take several measurements of the thermistor's resistance at known temperatures, for example, in ice-water, air temperature and warm water (measured with the mercury thermometer), and draw a graph of resistance versus temperature. Use readings at different temperatures to determine the response of your thermistor. Hint: Break the curve in to regions of straight lines.

It is useful to add some additional functionality to your system to make it nicer. Such useful features include:

- Displaying the Standard deviation of the readings in your sliding average.
- Displaying the minimum and maximum values of temperature encountered so far.

In your write-up, be sure to include your circuit, a copy of the program and evidence that you have completed the tasks. Remember, your demonstrator still needs to see all circuits and programs functioning as part of your lab assessment. Suggest possible improvements to your circuit (or program), if you do not have time to actually implement them.

Lab Session 3

PWM, LDRs, Interrupts & Timers

In this lab you will become familiar with:

- The PWM output
- External interrupts (using external events to trigger a particular task)
- The interval timers
- Internal interrupts (using the timers to trigger a particular task)

What is an interrupt?

Imagine a game of rugby: The game starts, you (hopefully) play by the rules until the game ends. However, at ANY stage in the game, the referee could blow the whistle and stop the normal course of play, and instead force the execution of a specific routine, for example a scrum, a penalty, or a lineout.

Let us refer to the referee as being an *interrupt handler*. In this instance, the referee monitors the game. In a similar manner a micro-controller has an interrupt handler monitoring it. At any stage in the game something could happen (for example knocking the ball forward, the ball going over the side line etc.) that will cause the referee to blow the whistle (the whistle signalling that the game is to be interrupted). In this example, there are several different forms (or sources) of interrupts. When an interrupt occurs *the interrupt routine* is begun. So if the interrupt source is the ball being knocked forward, the interrupt routine would be the formation of a scrum, if the interrupt source is the ball going over the side line, the interrupt routine would be the formation of a lineout. Obviously in a game of rugby there are many different interrupt sources, and correspondingly, many different interrupt routines. In a micro-controller, the equivalent of the whistle blowing is a bit called the *interrupt flag*. When this flag is set, the program that has been running stops and the microcontroller instead begins to execute the corresponding *interrupt routine* (a special type of software function). Once that interrupt routine has finished, the program continues running from where it left off (before it was interrupted) as if nothing had happened. Again this is similar to the rugby game, once the scrum or lineout is over, normal play recommences.

Priority Levels

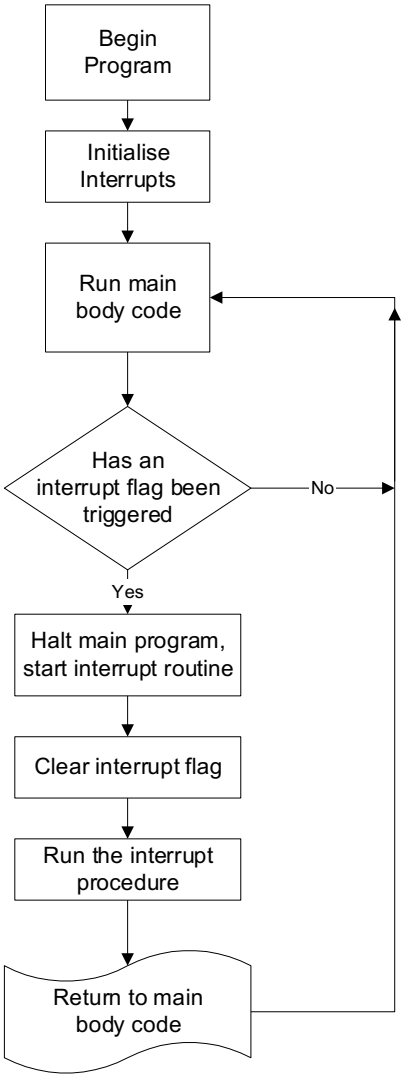
It is possible to have an interrupt occur while a previous interrupt is still being processed. In this case there is a priority structure that indicates if this new interrupt is to be processed now, or whether it must wait until the previous interrupt is complete. All interrupts have a priority level and an interrupt with a higher level can stop execution of an interrupt with a lower level - but obviously not the other way around. This can permit the user to enable multiple interrupts but still ensure that important interrupts get executed first. Going back to the rugby analogy, an example of this priority structure could be the referee stopping a fight that occurred during a lineout (that being the higher priority) before going back to complete that lineout, which must itself be completed before normal play could recommence.

Types of Interrupts

There are two main types of interrupts for the micro-controller: internal and external. An internal interrupt is communicated when an event happens in the internal circuitry of the micro-controller. This can include events such as a counter reaching a certain value (compare interrupts) or a byte being received through the serial port. External interrupts are activated by circuitry outside the micro-controller via an interrupt pin. Examples of externally triggered interrupts include (say) a switch being turned on and off, or a sensor being triggered. An external interrupt can be configured to trigger by an edge (change of state) or by a level (such as 0 or 5 Volts). When the interrupt has been activated, an interrupt flag is set inside the micro-controller.

What is an Interrupt Flag?

Every interrupt has associated with it an interrupt flag. The micro-controller continuously monitors these flags and when one is set, the associated interrupt handler is initiated. It is important to remember to manually clear the interrupt flag during the execution of your interrupt function. If you do not, as soon as the interrupt procedure exits, the micro-controller will check its flag and because it is still set, it will begin the interrupt procedure again.



Interrupt Types

External Interrupts 0 and 1:

These interrupts are set by an external signal entering the micro-controller. As mentioned, they can be set by either an edge (high to low signal or low to high) or by a level signal (low or high).

Internal Interrupts:

One example of an internal interrupt generating unit is the PCA (Programmable Counter Array) which can provide a number of different interrupts. It is essentially a 16 bit counter which counts up at a predefined rate for as long as it is enabled. When the count overflows, it will generate an overflow interrupt, and the count will automatically start again from 0 and carry on.

Other interrupts associated with the PCA are capture and compare interrupts. For compare interrupts, an interrupt is triggered when the PCA count is equal to a predefined value. For the capture interrupts, an interrupt is triggered by an external source and the current state of the PCA count is automatically stored.

The SDCC Compiler allows interrupt service routines to be coded in C, with some extended keywords.

```
void timer_isr (void) __interrupt (1)
{
...
}
```

Timer 0 interrupt service routine

The optional number following the *__interrupt* keyword is the interrupt number this routine will service. When present, the compiler will insert a call to this routine in the interrupt vector table for the interrupt number specified. If you have multiple sources files in your project, interrupt service routines can be present in any of them, but a prototype of the isr **MUST** be present or included in the file that contains the function main (main.c).

Interrupt #	Description	Vector Address
0	External 0	0x0003
1	Timer 0	0x000b
2	External 1	0x0013
3	Timer 1	0x001b
4	Serial	0x0023
5	Timer 2 (8052)	0x002b
...		...
n		0x0003 + 8*n

Table of interrupt numbers.

Part I: PWM Output

Read the section of the datasheet regarding the PWM outputs (under PCA from page 94). Note that in this section the datasheet incorrectly refers to Table 8. It should be Table 50.

We are going to configure module 0 as a PWM output. This involves the following:

1) Turn the PCA counter on (CCON register). You need to work out what to set the CR bit. This is bit addressable.
2) Set module 0 to be a PWM output (CCAPM0 register). You need to concentrate on the ECOM and PWM bits.
3) Set the PWM duty cycle by changing the CCAP0H register.

Q1 What is meant by 'bit addressable'?

Q2 Explain what the bits of the CCAPMn and CMOD registers do.

Now observe the output on an oscilloscope. Comment on the effect of changing CCAP0H.

Q3 What is the frequency, and how is it calculated? (Hint: The PWM is using the bottom 8 bits of the 16 bit PCA counter, and f_{pca} is equal to half f_{osc}). How could you go about changing the frequency?

Connect the PWM output to the DC motor driver box and the DC motor. Modify your program to be able to change the speed of the motor using an I/O module. **Show this to your demonstrator.**

Part II: LDRs and External Interrupts

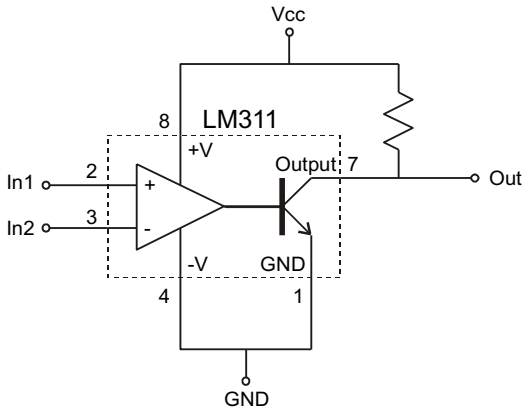
Now locate a Light Dependent Resistor (LDR). On your bread board, construct a simple circuit that will enable you to observe the variation in resistance of the device when you alter the amount of light falling on its surface. Report on the resultant resistance range.

We now want to be able to provide the micro with an interrupt signal, triggered by a reasonably substantial alteration in the amount of incident light falling on the LDR. To implement this, you will need to construct some form of voltage divider circuit that includes the LDR, so that the voltage swing from the output of this circuit is sufficient to generate an interrupt. Do this with the LM311 Voltage Comparator IC.

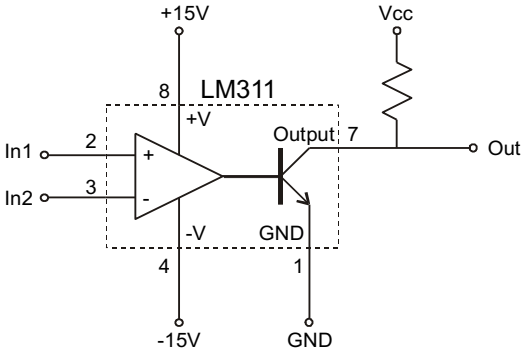
Voltage comparators do exactly what their name suggests. They compare two analogue input voltages and generate a digital output. The output is high if the +ve input is at a higher voltage than the -ve input, and a low output if the -ve input voltage is higher than the +ve input voltage.

The input side is very similar to an Op-amp. It has positive and negative input pins, and, positive and negative power pins. The output side consists of an open collector type digital output pin and a digital ground pin. Because the input voltages often do not relate to digital output voltage levels, the analogue power supply pins are independent from the digital ground pin.

If the analogue input voltages do not exceed 5 V or go below 0 V, the analogue power pins can be connected to the digital power supply. That is, +V to 5 V and –V to ground. Note however, that the digital GND pin **must** also be connected to ground. This is illustrated on the left figure below. If the analogue input voltages exceed 0 to 5 volts, a separate analogue power supply must be connected as per the figure on the right. Also note that because the output is open collector, an external pull-up resistor must be used.



LM311 set up for 5V operation



LM311 set up for ±15V operation

Hint #1: Use a variable resistor (as a variable voltage divider) to provide a reference level to one of the inputs and adjust it to suit the lighting conditions of the day. Ensure that you use a 5 V power supply to your voltage divider circuits and the voltage comparator. Verify that you do have a digital signal, possibly by using the LEDs of the I/O module.

You should now have an external event triggering an interrupt. Write a program to count the number of interrupts and display that number on the LCD screen. You might find pages 113 – 116 and page 70 of the microcontroller datasheet useful for this.

Hint #2: You will need to initialise (or setup) the interrupts. This consists of enabling external interrupt 0, disabling all other interrupts, and specifying whether the interrupt is to be edge or level triggered.

Hint #3: You will need to write an interrupt procedure for the external interrupt 0 (X0). In software an interrupt procedure takes the form of:

```
void my_isr_handler (void) __interrupt (n)
{
...
}
```

where the my_isr_handler is whatever you wish to call the function and “n” is the interrupt number. You can find out what the interrupt number “n” is, by using the table on page 20

of this document and the AT89C51AC3 datasheet. An interrupt handler performs some task when an interrupt is generated.

In the interrupt routine it is important to make sure the interrupt flag gets set to 0 at some point, otherwise the main code will jump straight back into the interrupt. In some cases the hardware will automatically **reset** this flag, so be sure to check the datasheet if this is the case.

Q4 Does switch bounce occur, and if so how do you deal with it? Does your software restrict the number of interrupts you can generate in real time (i.e. is the software working at a restrictively slow pace)?

WARNING:

Interrupt service routines open the door for some very interesting bugs:

- Common interrupt pitfall: *variable not declared volatile*

  If an interrupt service routine changes variables which are accessed by other functions these variables have to be declared volatile. See http://en.wikipedia.org/wiki/Volatile_variable.

- Common interrupt pitfall: *non-atomic access*

  If the access to these variables is not atomic (i.e. the processor needs more than one instruction for the access and could be interrupted while accessing the variable) the interrupt must be disabled during the access to avoid inconsistent data. Access to 16 or 32 bit variables is obviously not atomic on 8 bit CPUs and should be protected by disabling interrupts. You're not automatically on the safe side if you use 8 bit variables though. We need an example here: f.e. on the 8051 the harmless looking "flags |= 0x80;" is not atomic if flags resides in xdata. Setting "flags |= 0x40;" from within an interrupt routine might get lost if the interrupt occurs at the wrong time. "counter += 8;" is not atomic on the 8051 even if counter is located in data memory. Bugs like these are hard to reproduce and can cause a lot of trouble.

- Common interrupt pitfall: *stack overflow*

  The return address and the registers used in the interrupt service routine are saved on the stack so there must be sufficient stack space. If there isn't variables or registers (or even the return address itself) will be corrupted. This stack overflow is most likely to happen if the interrupt occurs during the "deepest" subroutine when the stack is already in use for f.e. many return addresses.

- Common interrupt pitfall: *use of non-reentrant functions*

  A special note here, integer multiplicative operators and floating-point operations might be implemented using external support routines, depending on the target architecture. If an interrupt service routine needs to do any of these operations on a target where functions are non-reentrant by default, then the support routines (as mentioned in a following section) will have to be recompiled using the --stack-auto option and the source file will need to be compiled using the --int-long-reent compiler option. Note, the type promotion required by ANSI C can cause 16 bit routines to be used without the programmer being !aware of it. Calling other functions from an interrupt service routine on a target where functions are non-reentrant by default is not recommended, avoid it if possible. Note that when some function is called from an interrupt service routine it should be preceded by a #pragma nooverlay if it is not reentrant. Furthermore non-reentrant functions should not be called from the main program while the interrupt service routine might be active. They also must not be called from low priority interrupt service routines while a high priority interrupt service routine might be active. You could use semaphores or make the function critical if all parameters are passed in registers. Refer to the sdcc user manual for more information.

Lab Session 4

Timers and Serial Communication

This lab continues with investigating various timer capabilities and finishes by introducing the serial communications UART library functions and how they can be used to communicate with the PC via the Putty application.

Part I: Capture Interrupts

The PCA modules can also be set to run in capture mode. In this mode the modules will be looking for an external interrupt coming in on their associated CEX pin, and as well as acting like an ordinary external interrupt, the hardware will automatically store the current count of the PCA at the time of the interrupt.

In this part we are going to use a PCA module in capture mode to calculate the speed of the DC motor. Set it up so that one of the CAP bits is set, and the ECCF interrupt is enabled. Use one of the optoswitches from the DC motor module to generate the interrupt signal, this will provide a signal which will give an indication of when a hole is passing the optoswitch. In the handler code, use variables to keep track of the current state of the PCA count, and compare that to the value of the previous interrupt, and hence calculate the time difference. This time difference can be used to calculate the motor's speed.

Q1 Given that you know the number of seconds in a minute; the number of counts occurring per second, the number of counts occurring per hole and the number of holes per revolution, find a formula for the RPM, the revolutions per minute.

Write software and set up the relevant hardware to show on the LCD the speed of the motor in revolutions per minute (RPM). Show this to your demonstrator. **NB: The old encoder wheel has 200 holes per revolution. We now use the Pololu 2823 motor. They have dual encoders that count 64 times per revolution of the motor shaft when both the positive and negative edges of the waveform are sampled on each encoder channel. This corresponds to 1920 counts per revolution.** The PCA counter can count both the positive

and the negative edge of the pulse, how would you achieve this? The PCA counter can also count from both encoders, how would you achieve this?

Part II Timer 0 and the calibrated delay

This task will use timer 0 to generate a calibrated delay. Note, do not blindly trust the pseudo code from the AT89C51AC3 book.

The on-board timers 0 and 1 are dedicated timers that function differently to the PCA timers. The timer 1 and timer 0 modules not only act as general purpose timers, but (as you have encountered previously) also provides timing functions for other parts of the microcontroller. This section of the lab will allow you to work with these timers by building code that will allow you to generate a calibrated delay.

Previously, you may have used an uncalibrated delay whereby you simply wasted clock cycles with an empty for loop with a configurable integer argument to change the length of the delay. This was suitable in times where you cared about a delay occurring and not about a particular length of said delay. (Even if you did, you could measure and quantify it by changing the integer and checking the length of the delay with a stopwatch. Then you would use this to infer int to delay length.)

A calibrated delay is a delay that uses a clock to count pulses of a known clock source to generate precise intervals. Using a dedicated clock module to generate delays comes with several major advantages:

- A calibrated delay function is non-blocking. Since the delay does not throw away clock cycles in the main code, (it uses its own dedicated timer hardware) your microprocessor program can perform other functions whilst the delay is occurring. Tasks such as pre-processing can occur in the dead-time, increasing overall efficiency.
- A calibrated delay can generate many different timer thresholds, and hence a single calibrated delay can time many functions, whereas a blocking delay can only time the main loop.

Start by writing an initTimer() function which sets up the timer. In this function, the steps you will need to perform are:

- Setup Timer 0 as an 8-bit reload timer.
- Set the timer select bit to count down the system clock.
- Set the requisite interrupt bits in the interrupt system to allow the timer 0 overflow to trigger an interrupt.
- Set the TH0 reload value to all zeros.
- Run the timer.

The timer 0 specification will generate an interrupt on overflow. Therefore, you must also enable the timer 0 interrupt flag in the interrupt service routine in the interrupt system. You will need to look at the TCON and TMOD registers to figure out how to do this.

Next, you must build a function startDelay(unsigned int msec) to set a global boolean delay flag to 1 to indicate that you wish to start delaying. Maintain another global variable denoting the number of overflow ticks that needs to be counted. You will need to look at the timer zero clock to figure out how fast the timer input clock is running and how many interrupts will be generated per second and the interrupt period. Use this to figure out how many interrupt service routine calls will be made to match the desired delay period.

Finally, build an ISR to trigger on the timer zero overflow interrupt. Inside the interrupt service routine generated by the timer 0 overflow, simply increment a global count variable. Check the number of overflow ticks and compare it with the current global count and determine if the desired time has been reached.

Use the preceding functions to write a program that will flash an LED at a rate of 1Hz (500ms on, 500ms off). **Show this to your demonstrator.**

Q2 Can you make the timer more accurate? To the millisecond? To the microsecond? What are the advantages/disadvantages of doing this?

Part III Serial Communication (optional)

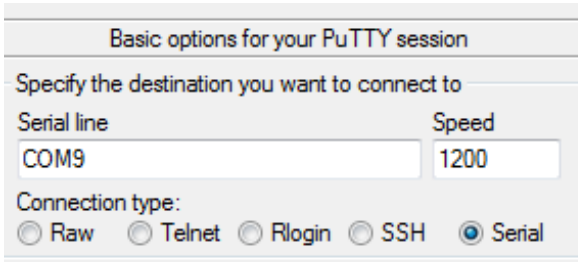
Using the ECEN301libsodcc library, write a program that will:

1) Initialise the serial port to a baud rate of 1200 Hz
2) Read incoming characters
3) Display the characters on the LCD display
4) Echo characters back to the PC.

After you have downloaded the program to the micro board, use Putty to test the program. To find Putty, all you need to do is use windows search, and type "Putty"

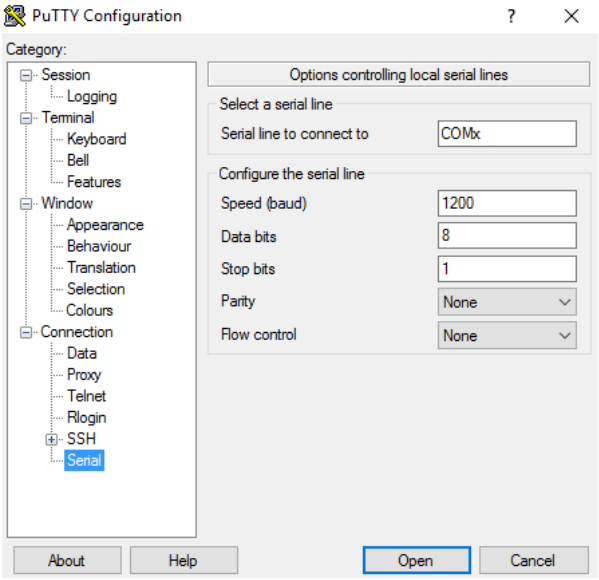
Side Note: There are many serial communication programs. Putty is one of them.

You will be met with the basic Putty interface. Under "Basic Options for you PuTTY Session," choose the radio button labelled: "Serial."



PuTTY main screen

Click on Connection > Serial in the sidebar and tell it to use the COM port that the micro is connected to. Set the bits per second (baud rate) to 1200, data bits 8, parity none, stop bits 1, and flow control none.



PuTTY Serial settings

Go back to the main menu by clicking on “Session.” At this point it may be prudent to save your connection settings. Whenever you close the Putty session, you will have to rerun this process. When you are ready to read data from the microcontroller, click “Open.”

If all goes well, the text you type into the Putty terminal will appear on the screen, as well as on the LCD display.