Lab 4: Using Analogue-to-Digital Converters (ADCs)

Introduction

The objective of this lab is to investigate how we can capture analogue information and turn it into a digital form that can be understood by a microcontroller. This is because the signals you are measuring in your project are inherently analogue in nature, so we have to convert them into binary values that can then be stored and processed, in order to work out the power/energy.

Read the entire lab manual before you start. This lab should take you approximately **5 hours**. There is a pre-lab, five parts and a few optional tasks. Before starting the lab, pull the lab repository from GitHub as it contains the Proteus model(s). Remember to commit and push your saved work (code and Proteus simulations) to the lab repository on GitHub after completing each task. **The final answers should be written in this document, with supporting calculations shown in your logbook.**

Pre-Lab: Digital I/O

At this point, you should be familiar with the ATmega328P/PB, Xplained Mini development board and how to use Atmel/Microchip Studio as well as how to simulate our compiled code using Proteus. Before the lab, we will refresh our understanding of digital input/output (I/O), as well as some elements of timing.

Q P.1: What is the purpose of a DDRn register?

Controls the Direction of each I/o

1) Open PreLab4_Proteus.pdsprj Proteus project from the Lab4_ADC folder in your local repository. Examine and understand the circuit diagram.
   a. Notice the virtual oscilloscope component – make sure you understand what this is reading.
   b. Make sure you can identify which pin on the microcontroller is connected to the LED, and which is connected to the Push Button.

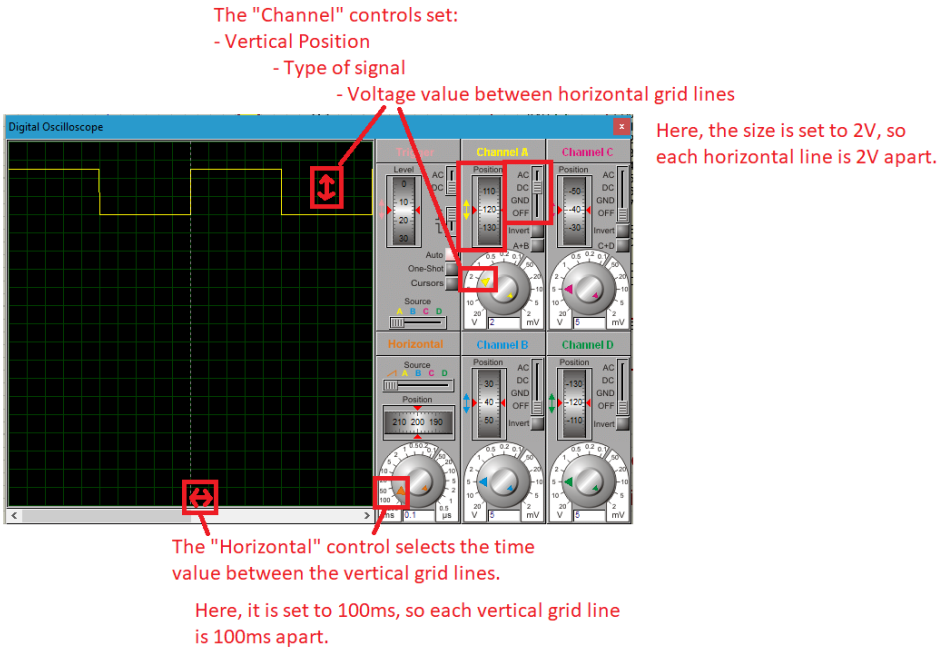
Q P.2: Which pin is the LED connected to? PB5

2) Create a new GCC C Executable Project in Atmel/Microchip Studio, selecting ATmega328P as the device family.
3) In your C program, use the appropriate DDRn registers to set PORTB to be outputs, and PORTC and PORTD to be inputs.
4) Now, extend your C program, using <util/delay.h> and _delay_ms(time) to toggle the LED connected to the ATmega328P so that it flashes with a 1Hz frequency and 50% duty-cycle continuously (i.e. turn-on the LED for half the period and turn-off the LED for half the period in the while loop). Use the datasheet to determine which pin you should use to control the LED. You may want to refer to your COMPSYS201 notes for some help.

As you did in Lab 2, compile the Atmel/Microchip Studio project, link the elf file with the ATmega328P processor in the PreLab4_Proteus.pdsprj Proteus project, and simulate your program in Proteus. Visually inspect that the LED seems to be

Version 1.0 – 2025
H. Pearce, A. Chen, T. Scott & D. Thrimawithana

flashing at a reasonable rate – Don't worry if the timing isn't looking fully correct, as **the simulation may not run in real time on less powerful computers**. However, the virtual oscilloscope can be used to confirm the timings are correct. Ensure you're reading Channel A by setting the slider associated with Channel A to DC. Ensure the horizontal grid size (the time axis) is set reasonably (e.g. 100ms/div is a good scale to use). Ensure the vertical grid size (the voltage axis) is set reasonably (e.g. 2V/div is a good scale to use). The trigger source should be set to Channel A and you may use a rising edge trigger. Now measure the period of the pulsing waveform by counting the number of grid lines. It should look something like:



You should be able to see from the above picture that the period passes through 10 vertical grid lines, and the amplitude passes through 2.5 vertical grid lines. Therefore, we can determine, based on the scaling knobs, that we have a period of 1s (10 * 100ms) and amplitude of 5V (2.5 * 2V).

5) Modify your C code to read the input of the push button and assign the inverse of that input to the LED (i.e. if button pressed LED should be on, if button is released the LED should be off).
6) Compile your code and then simulate it on Proteus to observe the input produced by the push button and the output generated by the ATmega328P on the oscilloscope. You'll need to enable Channel B (set Channel B slider to DC) to observe the input produced by the push button.

Q P.3: How much of a delay is there between the ATmega328P producing a logic high (5V) output to drive the LED from the time the push-button is pressed generating a logic low (0V) input? (Hint: Change the time scale). Why is this?

15µs. This time depends on the CPU clock. This delay is due to the programme execution in a continuous while(1) loop.

You may run your code on the Xplained Mini board to validate your solution experimentally. If required remember to recompile the code for an ATmega328PB.

Part 1: What is an ADC?

It can be easy to just think of the ADC as this magical device where you stick analogue signals in one end and digital data pops out at the other end. But quite a lot of smarts are going on inside! The biggest problem that we have to address is that analogue signals are *continuous*, but digital data is *discrete*. By converting from analogue to digital, we inherently have to go through a process of *quantisation*, and accept the associated *quantisation error* due to the loss of information. This is done by *sampling* the analogue signal.

Q 1.1: Based on the lecture slides, datasheet, COMPSYS201 notes, and other online resources, write a description for each of these terms in your own words:

a) Channel Selection

Choosing which input pin is currently connected to the ADC unit for next conversion.

b) Sample and Hold

Freezing the input voltage, so it doesn't change during the decision process before the ADC decide the digital value.

c) Successive Approximation

SAR ADC decides the result bit by bit over several ADC clock cycles. Each cycle compare the input voltage to internally generated voltage that is half of V_{ref} at start, if the input is higher, that bit set 1, otherwise set 0, and the search range halves next cycle. After N cycle, N-bit result obtained.

d) Reference Voltage

The voltage the ADC use as its comparison yardstick.

e) Sampling Rate

How many samples take in a time unit $t_{sample(max)} = \frac{1}{t_{acquisition(min)} + t_{conversion(min)}}$

Max sampling rate is limited by the minimum total sampling time

f) Resolution

The number of Bits (N) the ADC uses to represent input.

Q 1.2: How many input channels are connected to the ADC on the ATmega328P/PB?

8 channels ADC0 ~ ADC7

Q 1.3: How many channels can this ADC process simultaneously?

one

Q 1.4: How many clock cycles does a normal (single) ADC conversion take on the ATmega328P/PB? How long is this in real-world time if we have a 125kHz ADC clock?

13 cycles which equal to 104 us

Q 1.5: Which stage of the sampling process requires the most time? Why?

SAR decision stage because the logic solve result bit by bit, which take 10 cycles one of 13 cycles.

Q 1.6: Write a formula that gives the ADC count you expect to obtain when converting an analogue signal that has an amplitude of V_{analog}. Assume V_{ref} of your ATmega328P/PB is set to 5V and V_{analog}<5V.

$V_{step} = \frac{V_{ref}}{2^{10}} = \frac{5V}{1024}$, $ADC\_count = \frac{V_{analog}}{V_{step}} = \frac{V_{analog} \times 1024}{5V}$

Q 1.7: Based on the project specifications and the characteristics of the ATmega328P/PB ADC, what is the voltage resolution (i.e. how many volts does one ADC count correspond to)? 10kHz max

$V_{LSB} = \frac{V_{ref}}{2^{10}} = \frac{5}{1024} = 4.8828\ mV$

Q 1.8: What is the recommended operating frequency range for the ADC if we want to read 10-bit numbers?

50kHz ~ 200 kHz

Q 1.9: In this lab you will supply 125kHz to the ADC on the ATmega328P/PB. What is the prescaler value needed to achieve this ADC frequency? The system clock frequency is 2MHz.

$Prescaler = \frac{2\ MHz}{125\ kHz} = 16$

Part 2: Controlling the ADC

Just like most of the other peripherals on the microcontroller, we primarily interact with ADC through *registers*. Heading to the ADC section of the datasheet, we can see that there are three control registers (ADMUX, ADCSRA, ADCSRB) and two data registers (ADCL and ADCH). If you are interested, you can also look into PPR0 and DIDR0 for special circumstances. For the purposes of this lab, we will initially be reading from ADC2 using **polling with single conversions**, not interrupts. We will use AVCC as the reference in this example.

Q 2.1: What are our options for setting the reference voltage on this ADC, and which one is most fit-for-purpose for your project?

Options can be AREF, AVCC, Internal 1.1V reference. For this project, use AVCC so the ADC's full-scale matches 0-VCC analog scaling.

Q 2.2: For each of the control registers below, fill out the table with the short name for each bit, and the value we need to set it to during initialisation. Your options are 0 or 1 – if you don't care what the value is, set it to 0.

ADMUX

REFS1	REFS0	ADLAR			MUX2	MUX1	MUX0
0	1	0	0	0	0	1	0

ADCSRA

ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0
1	0	0	0	0	1	0	0

ADCSRB

	ACME				ADTS2	ADTS1	ADTS0
0	0	0	0	0	0	0	0

Q 2.3: Create a new Atmel/Microchip Studio project and using your answers thus far, write a function called adc_init(), which initialises the ADC component (select ATmega328P as the device family).

This is a good place to remind you about *modularity* – ideally, we should be putting all of the functions related to the use of ADCs in a separate .c file, with a corresponding .h file. Let's say your main code is in a file called **main.c**. Create a c source file called **adc.c**, and a corresponding header file called **adc.h**. In the header file, you should provide function declarations for all functions defined in the source file. Then, we should make sure that the **adc.h** header file is #include-d in **both** the main source file and the adc source file. The general rules for this are:

1) A .h file should #include as few other .h files as possible
2) A .h file should be protected against multiple #includes
3) A .c file should be able to #include any necessary .h files
4) You may also want a shared .h such as "common.h" for global #defines e.g. #define F_CPU 2000000
5) Avoid sharing global variables between .c files

For example:

main.c

```
#include <avr/io.h>
#include "adc.h"
#include "common.h"

int main(void){
	adc_init();
	while(1){};
}
```

adc.h

```
//Note the use of #ifndef, #define, #endif to protect against multiple inclusions
#ifndef _ADC_H
#define _ADC_H

#include <stdint.h>  //Necessary for definitions of uint8_t etc

void adc_init();
uint16_t adc_read(uint8_t chan);

#endif
```

adc.c

```
#include "common.h"
#include "adc.h"
#include <avr/io.h>  //Necessary for definitions of ADMUX etc

void adc_init() {
	ADMUX = ...   //Initialisation code goes here
}

uint16_t adc_read(uint8_t chan) {
	...           //Code written in Part 3 goes here
}
```

When you compile the program, the *linker* will automatically find everything and create the right references so that the functions can be used in the appropriate source files. However, you must have all files in the same project. Throughout your project, there are different microcontroller modules that are being used (e.g. USART, ADC, timers), and it is a good idea to separate the functions that are developed for each module. Note that there are marks allocated to code quality and here we will be looking at modularity and readability of your program.

Part 3: Reading from the ADC

When initialising the ADC, we have to assign a value in ADMUX to indicate which channel we are reading from. However, in our project, we have to be able to read from two different channels (representing the voltage and the current). Therefore, we will need to be able to change the channel that we are reading from.

Q 3.1: Write C code to set/change ADMUX to read from ADC1, without affecting the other bits in ADMUX that do not control channel selection:

```
ADMUX = (ADMUX & 0xF0) | 0xF0;
```

Q 3.2: Which bit do we set to start an ADC conversion (give name of the bit)?

ADSC

Q 3.3: Which bits can we check to know if the conversion is complete (give names of the bits)?

ADIF and ADSC (Auto-cleared to 0 when coversion complete)

Q 3.4: When we read the ADC result, we use ADCL and ADCH. Why are there two data registers for storing the ADC result?

Because ADC is 10-bit, but the data bus and registers are 8-bit.

Q 3.5: Using your answers thus far, modify your Atmel/Microchip Studio project to add a function called adc_read(channel), which completes the following steps:

- *Sets the channel of the ADC based on the argument*
- *Starts an ADC conversion*
- *Polls (i.e. blocking wait) for the conversion to finish*
- *Reads and returns the ADC result*

Ensure that you use a variable type with an appropriate width for storing the ADC result. Write the function in adc.c, with the relevant declaration in adc.h.

Part 4: Processing the Data

Of course, reading the ADC value on its own isn't very interesting because it only represents the number of steps between 0V and the reference voltage. Normally, in order to do something useful with this, we have to convert it back to the original voltage value.

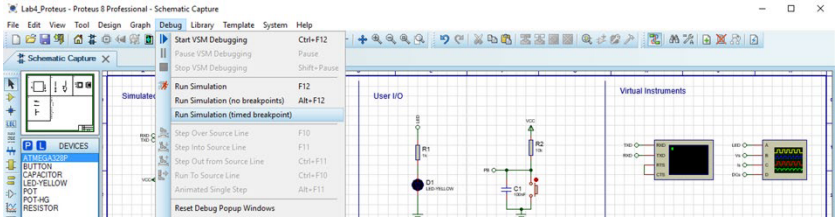
Q 4.1: Using your answers thus far, add a function called adc_convert_mv(value) to your program, which takes a raw ADC count and converts it back to the original voltage value in mV. Write out the pseudocode steps (like the ones shown in Q 3.5) here:

- Define reference voltage in millionsecond Set Vref_mv = 5000mV
- Define ADC resolution, ADC_max = 1024
- Scale raw ADC resolution, voltage_mv = (value × Vref_mv) / ADC_max

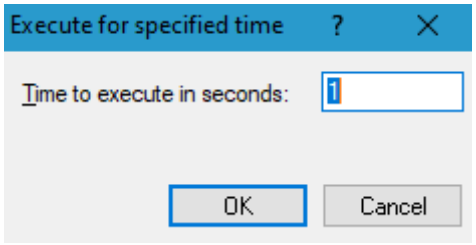
Version 1.0 – 2025
H. Pearce, A. Chen, T. Scott & D. Thrimawithana

- Return result, return voltage_mv

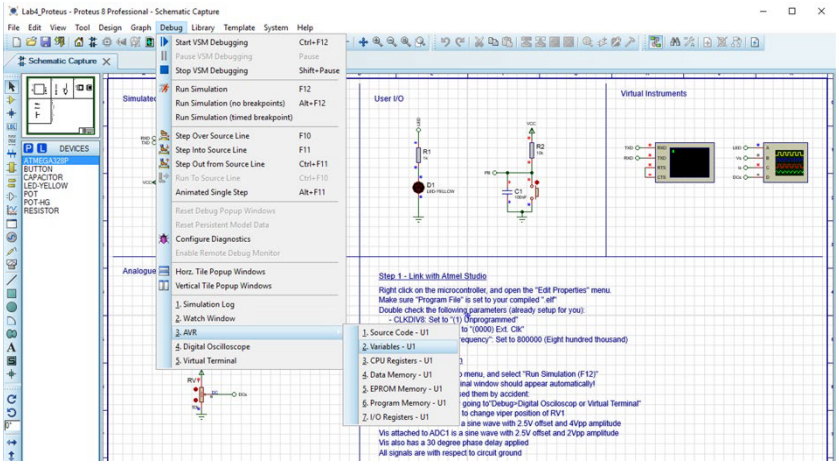
We are first going to use Proteus to test if our code works as intended. Modify the main function of the program you have written so far to read the value of a signal connected to ADC2 every 1ms (using _delay_ms(time)). Compile your code and link the elf file with the Lab4_Proteus.pdsprj Proteus project provided to you. In the Proteus project, a 5V DC source is connected to ADC2 through a variable resistor. After compiling your project and linking the ".elf" to the simulated ATmega328P microcontroller, in Proteus, in the top menu, select Debug > Run Simulation (timed breakpoint):



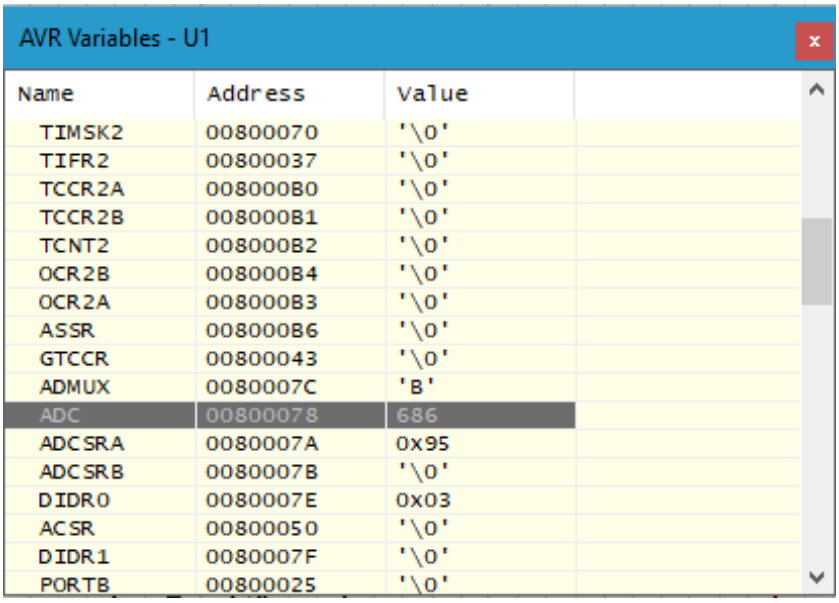
Instruct Proteus to run 1 second of simulation:



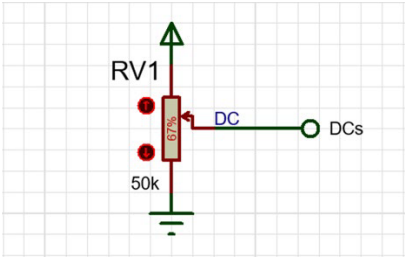
Press [OK] and wait for the simulation to finish executing 1 second of the code. Now, open Debug > AVR> Variables (if variables window does not appear):



Find the ADC register, and confirm that it captured data from your ADC2 port:



We captured the value 686 in this example. Let us check if the value makes sense. Find in the Proteus schematic the variable resistor connected to ADC2. It will indicate the percentage it is set at. As an example:



Here the variable resistor is set at 67%. Examining this circuit, we can see that the resistor goes from VCC (5V) to GND (0V). Therefore, the DC voltage at "DCs" will be 67% of 5V, i.e. 3.35V. Now, the ADC is also reading between 0 and 5V, meaning that it will also read 67%. The maximum value for the ADC in 10 bit mode is 1023, so the reading will be at 67% of 1023, i.e. 685.41. Our variable is at 686, indicating a small quantisation error has occurred, but the reading is otherwise correct.

Set the Proteus simulation to run normally. Now, drag the slider up and down on the variable resistor to change its resistance and thus the voltage at ADC2. Observe the change in the ADC register **(you need to pause the simulation for the variables window to update)**. Alternatively, for live updates, you can add the variable to the 'watch window' by right clicking on variable name and selecting 'Add to Watch Window'. Then open the 'watch window' from Debug>Watch Window.

Q 4.2: In the simulation when

RV1 is set to 100%, what does the ADC read? 1023

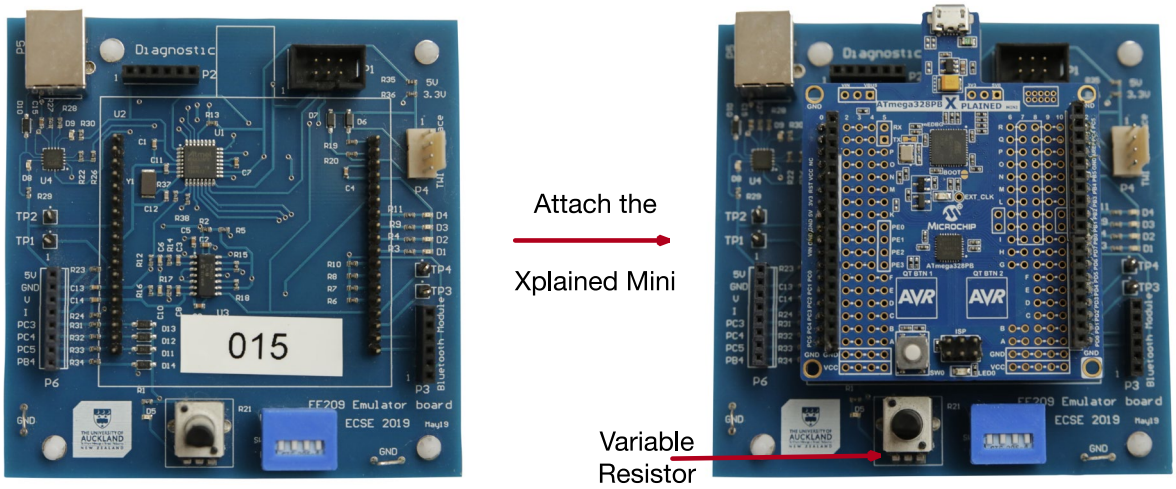
RV1 is set to 50%, what does the ADC read? 512

RV1 is set to 0%, what does the ADC read? 0

Let us now integrate our ADC code with the UART code you developed in the previous lab.

Q 4.3: Using the UART code that you have already developed (either as part of lab 2 or throughout the semester), write a function that will take the converted ADC2 voltage in mV, turn it into ASCII characters, and transmit it over UART to be displayed on the virtual terminal of the Proteus project. Compile the code and run the simulation in Proteus to verify your code developed so far.

Lets now test the code on actual hardware using the Xplained Mini development board. We have provided you an 'Emulator Board' that generates signals to test your C program without the need for the analogue hardware. This is quite common in industry as it takes time for analogue hardware to be developed, and an emulator gives the opportunity for the firmware team to start developing the code in parallel with hardware development. We will talk more about the emulator board later. For now, plug the Xplained Mini on to the 'Emulator Board':



The variable resistor on the 'Emulator Board' works the same as the variable resistor (RV1) you used in the Proteus simulation (i.e., one end connected to GND (0V) the other to VCC (5V), while the mid-point is connected to ADC2).

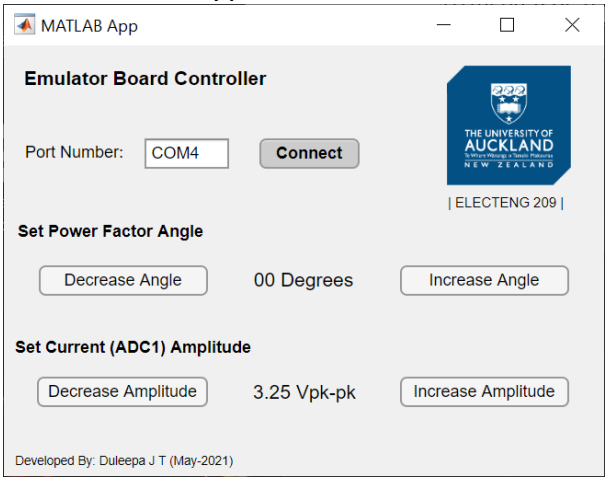
Q 4.4: Connect the Xplained Mini to the PC using a USB cable and load the program. If you have a Xplained Mini 328PB, remember to change the device to an ATmega328PB and re-compile your code. Observe the UART messages sent by the Xplained Mini on a terminal program such as PuTTY or RealTerm. Using a multi-meter to measure the voltage at the ADC2 pin (i.e., PC2 port pin) with respect to GND, validate the functionality of your code.

Modify your program using all the previously developed functions to read values from ADC0 and ADC1, alternating between these two channels after each read. Your program should read 40 samples from each ADC channel and store these values in two arrays (one for each ADC channel). Once you have read all 80 samples, you should transmit this data over UART to be displayed on a terminal program. The data printed on UART needs to be formatted correctly with extra ASCII characters so that these values can be copied and pasted on Excel as two

columns (e.g. 1st column with ADC0 readings separated by ", " and 2nd column with ADC1 readings). It may be easier to work with mV to avoid floats.

Q 4.5: Run your compiled code using the Lab4_Proteus.pdsprj which has two 500Hz sine wave inputs on ADC0 and ADC1 (remember to change the device back to an ATmega328P). The signal on ADC0 has an amplitude of 2V and an offset of 2.5V. The signal on ADC1, which is delayed with respect to the signal on ADC0 by a 30^0 phase-lag, has an amplitude of 1V and an offset of 2.5V. Observe the printed voltages on Proteus's virtual terminal. Plot these waveforms on Excel to verify the functionality of your code.

The 'Emulator Board' also generates two 500Hz sine wave signals at ADC0 and ADC1 pins of the ATmega328P/ATmega328PB on the Xplained Mini board. These signals, which are also available at test pins TP1 and TP2, have an offset of about 2.2V. The amplitude of the sine wave at ADC0 (or TP1) represents the signal you may be receiving from your voltage sensing circuitry. It has an peak-peak amplitude of 3.25V. The amplitude of the sine wave at ADC1 (or TP2) represents the signal you may be receiving from your current sensing circuitry. Its amplitude as well as phase (with respect to the signal at ADC0) can be controlled using the application given to you (to use this application the emulator board needs to be connected to the PC via the USB Type-B connector identified as P5):



Now re-compile your program to run on the Xplained Mini and use PuTTY/RealTerm to observe the printed voltages to make sure your code works on the actual hardware.

Matlab App broken

Q 4.6: Using an oscilloscope measure the two signals at ADC0 and ADC1 (i.e., PC0 and PC1 port pins) and clearly identify the amplitudes, offsets, and the phase-lag.

Amplitude : 1.97 V, offset : 2.12 V, phase-lag : 0

Part 5: Calculating Power

The signal connected to ADC0 in the Proteus project (as well as the 'Emulator Board') represents the signal derived from your conditioning circuitry that measures the voltage supplied by the AC source. Similarly, the signal connected to ADC1 represents the signal derived from your conditioning circuitry that measures the load current.

Let's assume that you designed your voltage divider and signal conditioning circuitry to step the voltage across the AC source down by 14.1 and add a 2.5V offset before emitting the signal supplied to ADC0. This means that the signal at ADC0 in Proteus, which has an amplitude of 2V and an offset of 2.5V, represents $20V_{rms}$ across the source (i.e. 2x14.1/1.414).
Similarly, the provided Proteus project also assumes that you designed the current sensor and its signal conditioning circuitry to step the load current down by 2 before adding a 2.5V offset to produce the signal supplied to ADC1. This means the signal at ADC1 that has an amplitude of 1V and an offset of 2.5V in the Proteus project provided represents $1.4A_{rms}$ load current (i.e. 1x2/1.414).
Note that in your project the gains and offsets provided by the signal conditioning circuitry will differ. This is just an arbitrary example. For your project, you will change these values as well as the magnitudes of these two signals and the phase of the signal representing the current as per your LTspice simulations. For now, assume the above. Given we now have the ability to read from these two ADC channels, you will now extend your program to calculate power represented by the signals fed to ADC0 and ADC1.
However, determining the power is not quite as simple as P = VI. One of the issues with reading from two channels is that we get the instantaneous current or voltage when we sample, and these two samples are not *synchronised* (i.e. they are not taken at the same point in time).

Q 5.1: Let us assume that both the voltage and current are sinusoidal signals with the same time period. If the ADC is supplied with a 125kHz clock, how much of a time-delay would there be between two consecutive ADC0 and ADC1 samples?

$T_{conversion} = \frac{1}{125\ kHz} \times 13 = 104\ \mu s$

In the lectures, a number of different methods of mitigating this problem have been presented.

Q 5.2: Use the data obtained in Q4.5 to evaluate which of these methods is most suitable for you in terms of accuracy and speed. You are expected to use Excel or MATLAB to do this evaluation.

Optional Task: Implementing Power Calculation

Let's implement the power calculation algorithm you experimented on Excel or MATLAB in C.

Q O.1: Modify your C program to implement your chosen method for calculating power, and print the results in the terminal. You may want to first test your code in Proteus before testing using the 'Emulator Board' and the Xplained Mini. Compare these results to your theoretical result obtained in Q5.2 – how much difference is there? Where does this difference come from?

Q O.2: Note that the design specifications limit the rate at which you can take ADC readings to a maximum of 10kHz (i.e., ADC conversion rate). This means we can't take as many ADC readings as we would like to. In the code you have developed so far, you are using a 125kHz ADC clock and operating it in the single conversion mode. At what rate are currently taking ADC readings? How can you verify that you are taking an ADC reading at this theoretical rate you calculated? How would you increase the rate at which you take ADC samples?

_____

_____

Q O.3: What could be an advantage of operating the microcontroller at a lower clock frequency and taking fewer ADC samples?

_____

Q O.4: In Q4.4 why did we first store the ADC values in two arrays and only send them over UART once all samples were collected? Can we not send the ADC values over UART as we read them from the ADC?

_____

To get signed off for the lab:

- Record all the workings in hardcopy or in digital format so we can give you marks for workings if final answers are not correct
- Modify the simulation models provided as per the lab tasks and save
- Commit and push your saved work to the lab repository on GitHub after completing each task and make sure it is up to date
- Complete this document summarising your final answers, commit and push to lab repository on GitHub
- Update logbook indicating yours and teams progress, meeting notes, etc.
- Go to your assigned interview session to check-in with a TA who will check your solutions and ask a few questions
- Report on your weekly progress