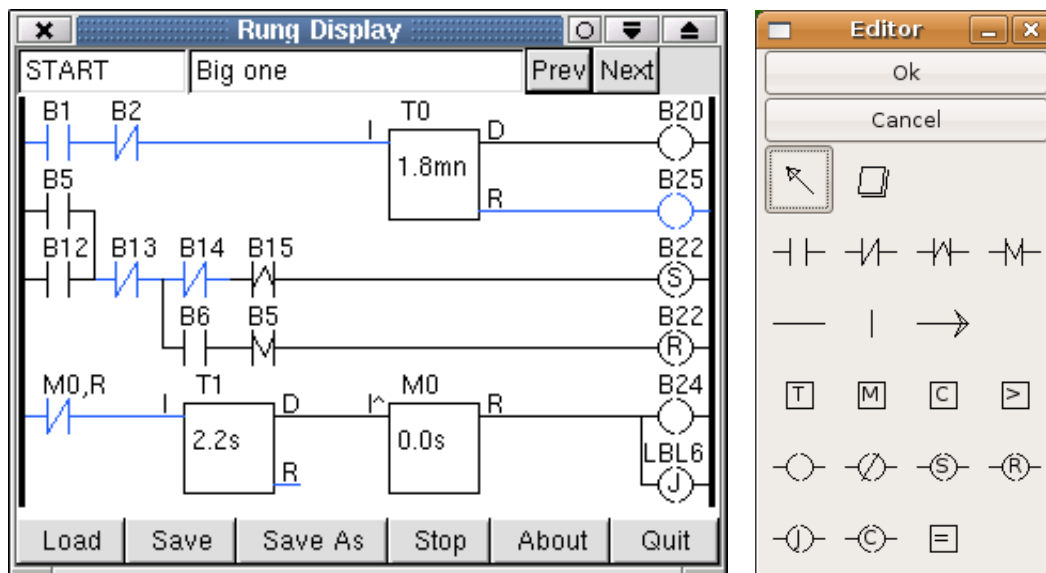
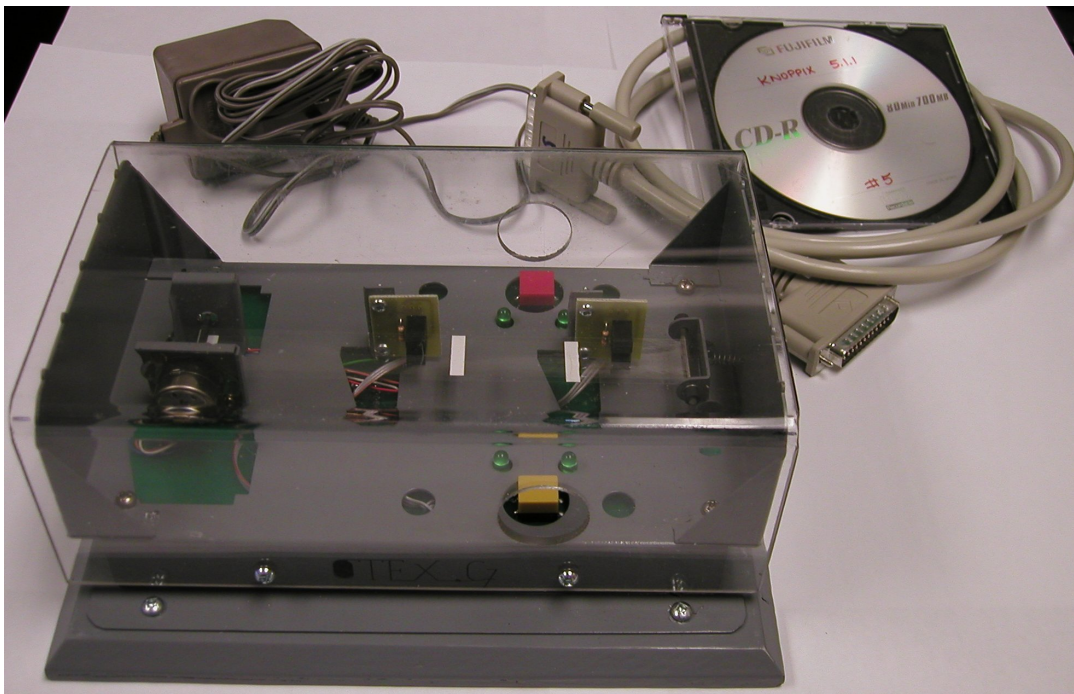


ENG-7680: SCADA Laboratory Experiments

1 Ladder Logic using ClassicLadder

Manual 1 <http://wiki.linuxcnc.org/cgi-bin/emcinfo.pl?ClassicLadder>

Manual 2 <http://mat.sourceforge.net/manual/logic/classicladder.html>



1.1 Getting Familiar with Timers

1.1.1 Experiment

Develop an algorithm that controls a dangerous process that requires the use of two palm buttons to be acted on at the same time to prevent any injury. To start the process, the two palm buttons must be pressed within 2.5s (a real process would require cca ten times shorter time frame) of each other, and must be held engaged for the entire process cycle. As soon as one of the buttons is disengaged, the process stops immediately.

Program this control algorithm and simulate it in Classic Ladder. Provide the ladder logic and simulation results.

1.1.2 Experiment

A motor is controlled by two momentary switches. The normally-open *START* switch starts the motor and the normally-closed *STOP* switch stops the motor. For an extra protection, the motor is started only if the *START* switch is thrown twice in two second time interval (a real process would require cca four times shorter time frame). Implement the control using the ladder diagram.

Program this control algorithm and simulate it in Classic Ladder. Provide the ladder logic and simulation results.

1.2 Sliding Door Control

Develop a ladder logic to operate a shop sliding door using the 'belt-conveyor' hardware. Here is the description:

1. To open the sliding door, a floor mat switch on either side must be acted on; if not engaged, the door will close after 5 seconds.
2. The marker on the belt will simulate the door travelling between two limit switches, the photo sensors.
3. After expiration of the time delay, door closing latches until the "CLOSED" position is reached.
4. If the floor mat switch is acted on again while the door closes, door immediately opens again.
5. LED's are indicating whether the sliding door is opening or closing.

NOTICE: The hardware units were custom built over several months by a number of volunteers including a generous donation from overseas; please look after the units carefully!

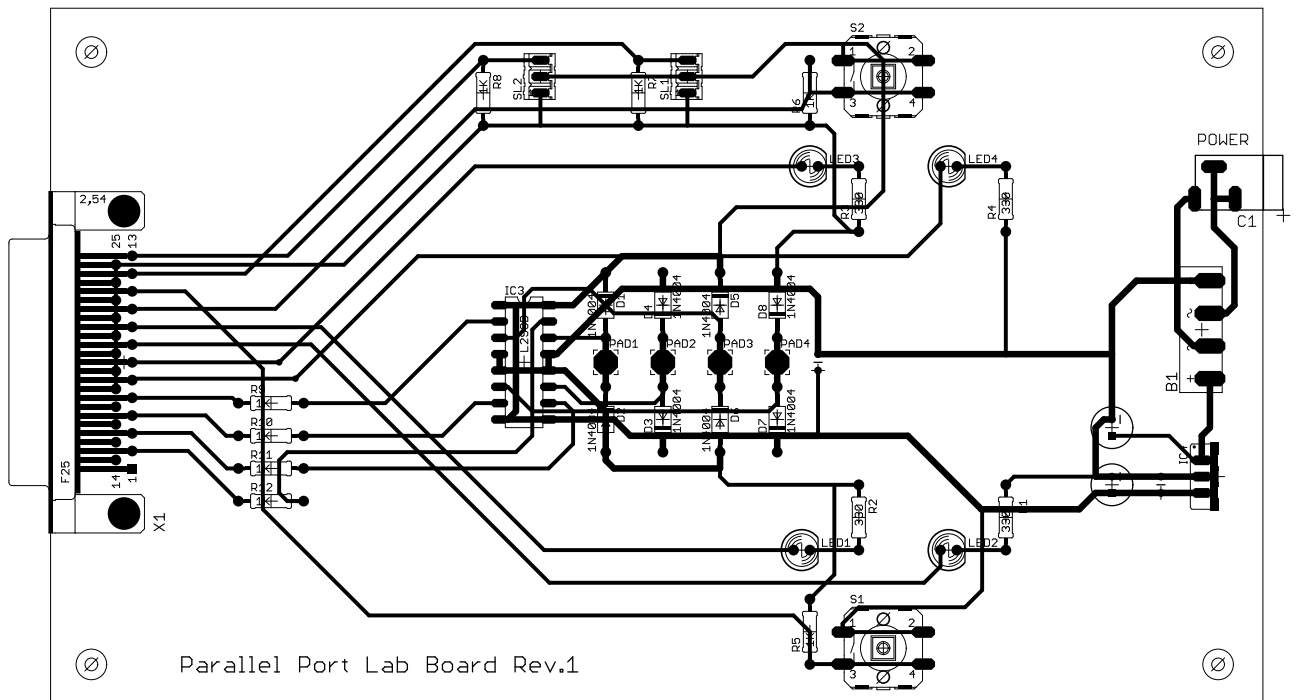
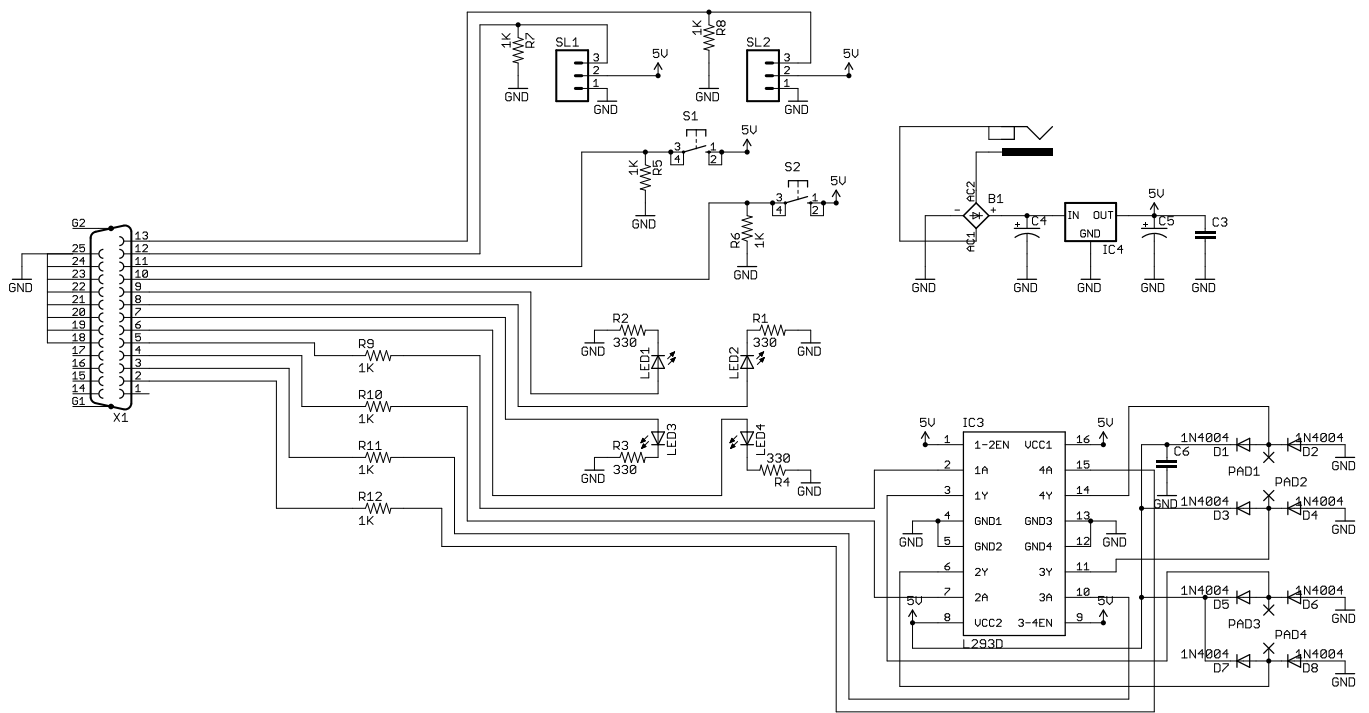
I/O Configuration:

Please refer to the circuit schematic, read document `/opt/classicladder/README.txt`, and run the example `/opt/classicladder/projects_examples/parallel_port_direct.clp` before configuring the I/O. The example tabulized below configures the direct parallel-port access such that the binary input `%I3` is read as the status of 3rd bit in a byte at address 379 (DB25 pin #15), and the binary output `%Q5` is changed by writing 5th bit of a byte at address 378 (DB25 pin #7). Note there are only 5 binary inputs in the parallel port interface.

DIRECT PORT ACCESS	INPUTS	OUTPUTS
1st I/O mapped	0 (<code>%I0</code>)	0 (<code>%Q0</code>)
Port Address	379	378
First Channel	0	0
Nbr of Channels	8	8

The table below shows the parallel port/cable pinout mapping:

	bit 0 (LSB)	bit 1	bit 2	bit 3	bit 4	bit 5	bit 6	bit 7 (MSB)
379 (Input)				#15	#13	#12	#10	#11
378 (Output)	#2	#3	#4	#5	#6	#7	#8	#9



2 Digital-To-Analog Converter

2.1 OBJECTIVES

In this lab, you will learn how to design, simulate and build a 3-bit D/A converter. In particular, you will get familiar with these aspects of instrumentation:

- Modeling and simulation of an interface circuit
- Design a PC to analog circuit interface
- Serial port programming in C using Linux
- Program D/A converter

2.2 BACKGROUND

To output an analog voltage from a PC or μ controller, the numerical value (integer) must be converted to an analog voltage by a D/A converter. Analog outputs are much simpler than analog inputs. This process is very fast, and does not experience the timing problems of sampling and conversion with analog inputs. However, analog outputs are still subject to quantization errors.

2.3 EQUIPMENT

Equipment
Variable dual DC power supply Prototyping breadboard with jump wires Multimeter Storage oscilloscope PC with serial port
Parts
Quad 741 Op Amp 3 diodes Serial port connector (female) with lead wires Resistors & Capacitors

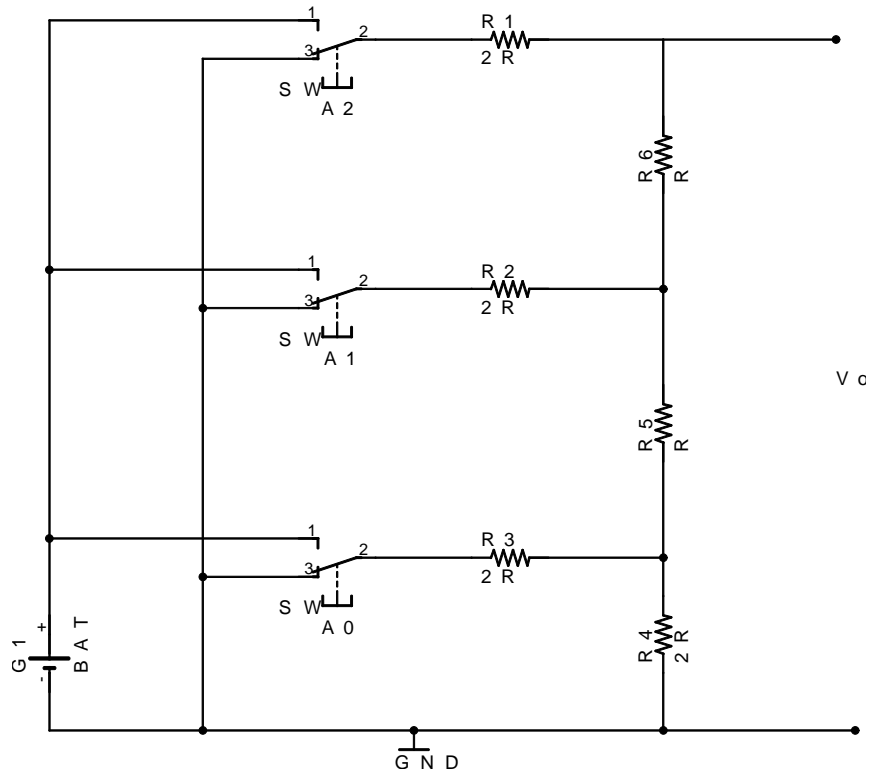
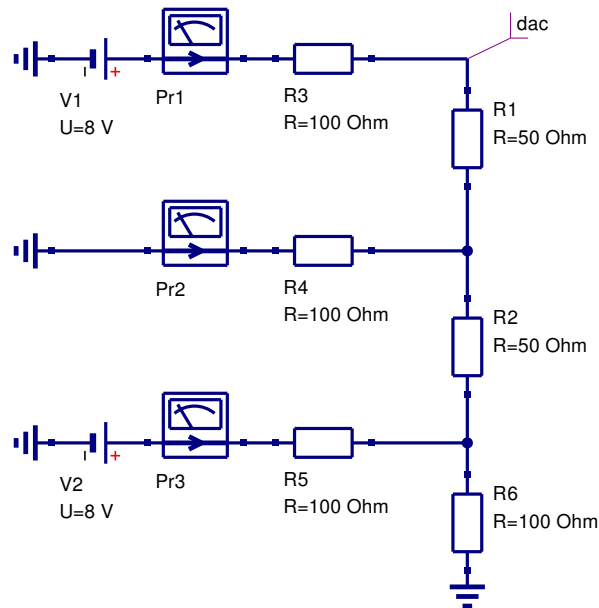


Figure 1: Ladder resistor network DAC

2.4 EXPERIMENTS

1. Figure 1 shows the concept of a 3 bit resistive ladder network D/A converter (DAC). This DAC design has a great advantage over the weighted resistor DAC in that only 2 values are used, R and $2R$ so the overall uncertainty/accuracy is much better.
2. Simulate the circuit using an open source circuit simulator QUCS available on the Live CD or at qucs.sourceforge.net shown in Figure :



dc simulation

DC1

number	dac.V	Pr1.I	Pr2.I	Pr3.I
1	5	0.03	-0.035	0.0425

3. Fill the conversion table below linking eight 3-bit inputs to 8 voltage outputs and 24 switch currents.

Binary Input	DAC Voltage	I_{A0}	I_{A1}	I_{A2}
000				
001				
010				
011				
100				
101				
110				
111				

4. At home, develop an analytical model for the DAC voltage output in terms of the R-2R resistances, reference voltages and the switch positions. Also develop an analytical model for 3 switch currents. List the analytical formulae and fill the table below using the formulae. Compare your calculated data with the simulated data above. The two tables should agree, if not, start again...

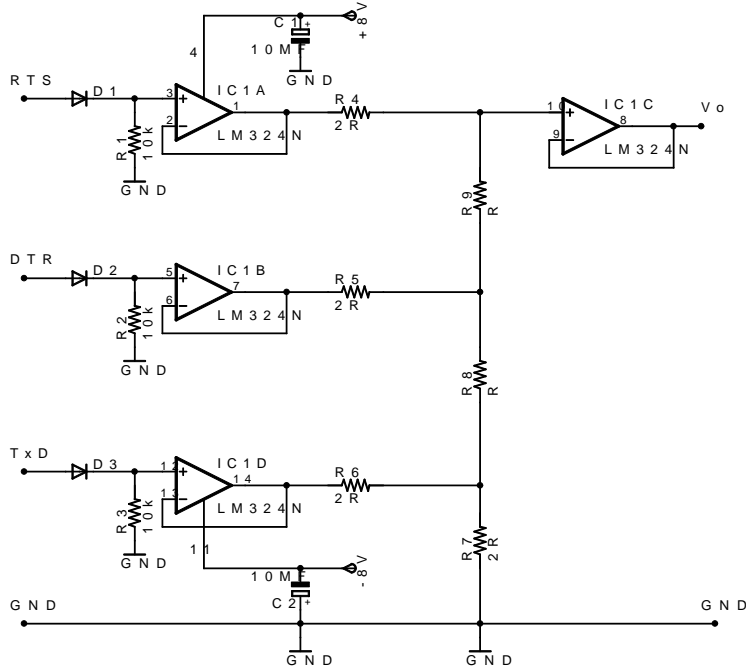
$$\begin{aligned}
u_{DAC} &= u_{DAC}(a_0, a_1, a_2) & U, R : constants \\
i_{pr_i} &= i_{pr_i}(a_0, a_1, a_2) & i = 0, 1, 2 \\
a_i &= \{0, 1\}
\end{aligned}$$

Binary Input	DAC Voltage	I_{A0}	I_{A1}	I_{A2}
000				
001				
010				
011				
100				
101				
110				
111				

- Design RS232 [-15V,+15V] to [8V,0V] interface as shown in Figure 2 that provides a low-impedance output in both reference levels [GND, +8VDC].
- From data sheets, determine the maximum allowable current at the OP Amp output. By using the simulation results, determine the values of R-2R ladder network that will protect the interface OP Amps from overloading. List the values.
- Assemble the resistive ladder network on the breadboard using the resistors determined above. Interconnect the buffer to the ladder network and add the voltage follower buffer at the DAC output; note this inverts the polarity!
- Verify the proper DAC function by switching the 3 inputs between [-15V,+15V] using jump wires; fill the table below:

Binary Input	DAC Voltage
000	
001	
010	
011	
100	
101	
110	
111	

- Connect DAC to PC serial port (TTY/COM). Follow the steps below to output a triangular (sawtooth) waveform. Record the waveform on the scope-screen and attach it to your report.



DB9 PIN	RS232	I/O	COLOR
1	DCD	in	brown
2	RXD	in	purple
3	TXD	out	red
4	DTR	out	orange
5	GND		black
6	DSR	in	grey
7	RTS	out	yellow
8	CTS	in	blue
9	RI	in	green

Figure 2: RS232 to R-2R DAC interface

2.5 Software Development

1. Boot Linux using Live Debian CD
2. Test individual pins:
 - (a) Open a root terminal window.
 - (b) `cp -R /var/MUN/DAC-ADC-lab-sources /home/user/Desktop/`
 - (c) `cd /home/user/Desktop/DAC-ADC-lab-sources`
 - (d) `ls -al`
 - (e) `gcc -o lp-tty-start.bin lp-tty-start.c`

- (f) `gcc -o port-write-then-read.bin port-write-then-read.c`
- (g) `ls -al`
- (h) `./lp-tty-start ./port-write-then-read 1020 0`
and check the RTS line output voltage
- (i) `./lp-tty-start ./port-write-then-read 1020 3`
and check the RTS line output voltage again
- (j) To exit from the running program, type `Ctrl + c`

3. Output Saw-Tooth voltage:

- (a) Open a text editor and save the C code listed below in `DAC-saw-COM1.c` file.
- (b) Compile: `gcc -o DAC-saw-COM1.bin DAC-saw-COM1.c.`
- (c) Run: `./DAC-saw-COM1.bin`
- (d) Record the resulting waveform on the scope screen and attach it to the report.

4. Shutdown the system by `shutdown -h now`

2.6 INTERFACING THE SERIAL PORT (RS232)

Serial Port	Base Address
COM 1	3F8
COM 2	2F8
COM 3	3E8
COM 4	2E8

Used Serial Port Registers		
Address	Read/Write	Register
BaseAddress+3	R/W	Line Control Register (LCR)
BaseAddress+4	R/W	Modem Control Register (MCR)

Bit 6 of LCR sets break enable. When active, the TxD line goes into "Space" state or logic '0' (positive voltage). Setting this bit to '0' disables the break, i.e. the line goes negative ('Mark' state or logic '1') in the idle state.

Bit 0 sets/resets DTR line; Bit 1 of MCR sets/resets RTS line.

```

#include <sys/ioctl.h>
#include <fcntl.h>
#include <sys/io.h>
#include <stdio.h>

int main(){
    int port, kbдин;
    char value, MSB, INB, LSB;
    int BASEPORT = 0x3F8; /* TYYS0 or COM1 */
    if (ioperm(BASEPORT, 8, 1)) {perror("ioperm"); return(1);} /* PORT OPENED */
    do
    {
for(value=0; value<=7; value++)
    {
        MSB = 0; INB= 0; LSB= 0;
        if ((value & 4) != 0) MSB = 1;
        if ((value & 2) != 0) INB = 1;
        if ((value & 1) != 0) LSB = 1;
        outb((MSB<<1) | (INB<<0), BASEPORT+4);
            // set RTS (MSB) {bit 1 of BASEPORT+4}
            // set DTR (INB) {bit 0 of BASEPORT+4}
        outb(LSB<<6, BASEPORT+3);
            // set TxD (LSB) {bit 6 of BASEPORT+4}
        printf(" RTS(7)  DTR(4)  TXD(3)  \n");
        printf(" MSB                LSB  \n");
        printf(" %d          %d          %d  \n",
                inb(BASEPORT+4)>>1 & 0x01,
                inb(BASEPORT+4)>>0 & 0x01,
                inb(BASEPORT+3)>>6 & 0x01);
        sleep(1);
    }

        printf("Enter '1' to run saw tooth again, enter '0' to exit. \n");
        scanf("%d",&kbдин);
    } while(kbдин);
    if (ioperm(BASEPORT, 8, 0)) {perror("ioperm"); return (0);} /* PORT CLOSED */
    return 0;}

```

3 ANALOG-TO-DIGITAL CONVERTER

3.1 OBJECTIVES

In this lab, you will learn how to combine a 3-bit D/A converter built in the previous lab with a comparator to make a simple successive approximation A/D converter (ADC). In addition, you will also build a four-channel ADC that does not require any external power supply. In particular, you will get familiar with these aspects of instrumentation:

- Build a sample & hold device
- Serial port programming in C using Linux
- Develop a mathematical model for ADC linearization

3.2 BACKGROUND

To input an analog voltage into a PC or μ controller, the continuous voltage value must be first sampled and then converted to a numerical value by an A/D converter. The process of sampling the data is not instantaneous, so each sample has a start and stop time. The time required to acquire the sample is called the sampling time t_s . A/D converters can only acquire a limited number of samples per second. The time between samples is called the sampling period T_s , and the inverse of the sampling period is the sampling frequency (also called sampling rate). The sampling time is often much smaller than the sampling period. The maximum V_{max} and minimum V_{min} readable voltages are a function of the control hardware such as 0V to 5V, 0V to 10V, -5V to 5V or -10V to 10V. The number of bits of the A/D converter is the number of bits in the result word. If the A/D converter is 8 bit then the result can read up to 256 different voltage levels. Most A/D converters have 12 bits, 16 bit converters are used for precision measurements.

Sample & Hold (S/H) circuitry takes a snapshot of the input signal and holds the value for the A/D converter to have a stable signal. A simple S/H circuit is shown in Figure 5. FET switch connects the capacitor to the buffered input once every sample period. The capacitor then holds the voltage value sampled until a new sample is acquired.

The sampling subsystem takes a period of time (aperture time) to capture a sample of the input signal. The holding subsystem holds the sampled voltage on the capacitor. This voltage slowly decreases over time despite of a high impedance

connection to a voltage follower. It is then necessary to perform the A/D conversion in a short period of time.

3.3 EQUIPMENT

Equipment
Function generator with GND isolation plug Variable DC power supply Prototyping breadboard with jump wires Multimeter Storage oscilloscope PC with serial port
Parts
Quad Op Amp LM324 2 Zener diodes 7.5V FET 2N5951 Serial port connector (female) with lead wires Resistors & Capacitors

3.4 EXPERIMENTS

3.4.1 Successive Approximation ADC

1. Use the previously developed 3-bit D/A converter and change the output buffer into a comparator to make 3-bit successive approximation A/D converter as shown in Figure 3. Connect the input voltage to the non-inverting terminal of the comparator, DAC output to the inverting terminal, and the comparator output to one of the RS232 inputs, e.g. CTS.
2. Compile the C program below listed below.
3. For 0.5V test voltage increments, record the input-output characteristic.

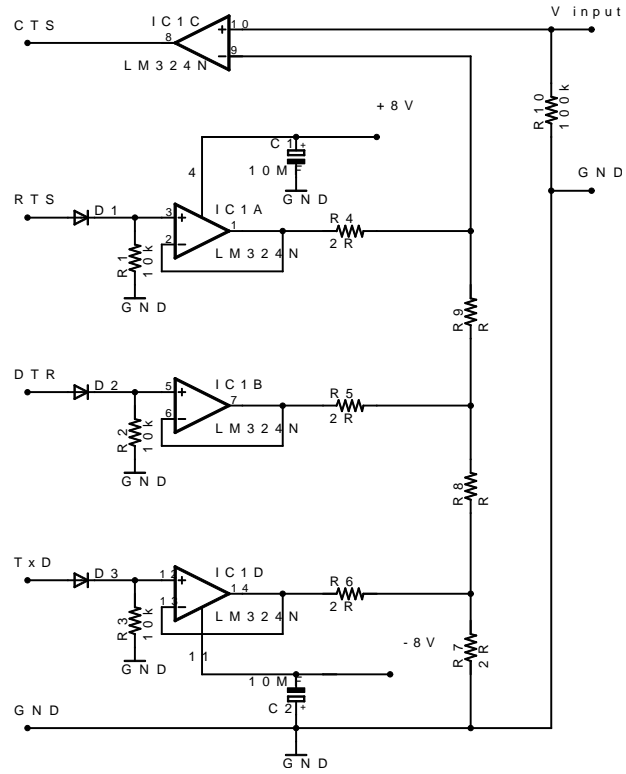


Figure 3: RS232 Successive Approximation ADC

ADC Input Voltage [V]	Measured Voltage [V]
0.0	
0.5	
1.0	
1.5	
2.0	
2.5	
3.0	
3.5	
4.0	
4.5	
5.0	
5.5	
6.0	
6.5	
7.0	
7.5	
8.0	

```

#include <sys/ioctl.h>
#include <fcntl.h>
#include <sys/io.h>
#include <stdio.h>

int main(){
    int port, exit;
    char value, MSB, INB, LSB;
    int BASEPORT = 0x3F8; /* TYYS0 or COM1 */
    if (ioperm(BASEPORT, 8, 1)) {perror("ioperm"); return(1);} /* PORT OPENED */
    value = 0;
    exit = 1;
    do
    {
        MSB = 0; INB= 0; LSB= 0;
        if ((value & 4) != 0) MSB = 1;
        if ((value & 2) != 0) INB = 1;
        if ((value & 1) != 0) LSB = 1;
        outb((MSB<<1) | INB, BASEPORT+4);
        // set RTS = MSB bit 1 of BASEPORT+4
        // set DTR = INB bit 0 of BASEPORT+4
        outb(LSB<<6, BASEPORT+3);
        // set TxD = LSB bit 6

        sleep(1);
        printf("VALUE= %d    CTS= %d \n", value, inb(BASEPORT+6)>>4 & 0x01);
        if((inb(BASEPORT+6)>>4 & 0x01)==0)
        {
            printf("Voltage is %d [V] \n", value-1);
            exit = 0;
        }
        value++;

        if(value==8) exit=0;
    } while(exit);

    if (ioperm(BASEPORT, 8, 0)) {perror("ioperm"); return (0);} /* PORT CLOSED */
    return 0;}

```

3.4.2 ADC using time interval

1. Measured analog voltages can also be compared to a variable voltage on a capacitor instead of DAC we used previously. The concept of charging a capacitor was covered in the class. Note RS232 outputs bipolar voltages that are conditioned by using two 7.5V Zener diodes to $\pm 8V$ max (Figure 4).

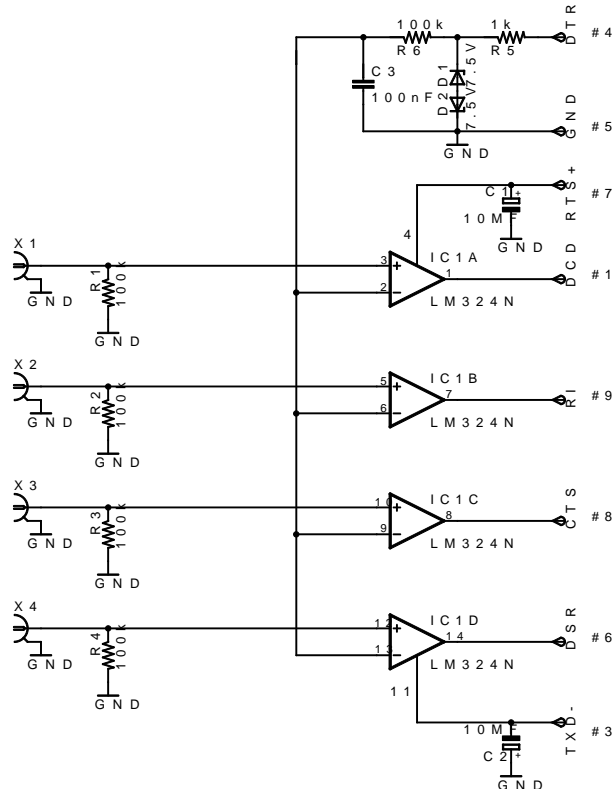


Figure 4: 4ch ADC using time interval

2. Modify C program below that measures the voltage using a time interval. You may need to terminate the timing loop in case the measured voltage is out of range. More references on the programming can be found at the end.
3. List your code and write the formula being used in your time-to-voltage conversion:
4. Test your system and fill the table below:


```

/* ***** */
/* SIMPLIFIED HW SETTING: */

/* DTR line (-12V/+12V) is used to charge a cap through a resistor (RC=93507us) */
/* GND line is connected to common ground */
/* CTS line is connected to comparator output */
/* OP AMP is powered from external power supply -15V/+15V */
/* A potentiometer is used to provide the input voltage between -12V/+12V */
/* Compile by gcc -lm -o ADCcapacitor.bin ADCcapacitor.c */
/* ***** */

#include <sys/ioctl.h>
#include <fcntl.h>
#include <sys/io.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>

int main(){
    int port;
    struct timeval tv;
    long time1, time2;
    double x;
    int BASEPORT = 0x3F8; /* TYYS0 or COM1 */
    if (ioperm(BASEPORT, 8, 1)) {perror("ioperm"); return(1);} /* PORT OPENED */
    outb(0, BASEPORT+4); // reset DTR (bit #0)
    sleep(1);
    outb(1, BASEPORT+4); // set DTR (bit #0)
    gettimeofday (&tv, NULL);
    time1=tv.tv_usec;
    do{
    }
    while(inb(BASEPORT+6)>>4 & 0x01); //read CTS
    gettimeofday (&tv, NULL);
    time2=tv.tv_usec;
    x=24.0*(1.0-exp(-(double)(time2-time1)/93507.0))-12.0;

    printf("time = %d [us], voltage = %5.2f [V] \n", (time2-time1), x);

    if (ioperm(BASEPORT, 8, 0)) {perror("ioperm"); return (0);} /* PORT CLOSED */
    return 0;}

```

ADC Input Voltage [V]	Measured Voltage [V]
-8.0	
-7.5	
-7.0	
-6.5	
-6.0	
-5.5	
-5.0	
-4.5	
-4.0	
-3.5	
-3.0	
-2.5	
-2.0	
-1.5	
-1.0	
-0.5	
0.0	
0.5	
1.0	
1.5	
2.0	
2.5	
3.0	
3.5	
4.0	
4.5	
5.0	
5.5	
6.0	
6.5	
7.0	
7.5	
8.0	

5. Implement S/H circuit shown below meeting the following specifications:

Input voltage V_{IN} range	0V – 8V
Max sampling frequency	10Hz
Min acquisition time	50ms

Determine the RC time constant and select the component values.

6. Using a function generator, determine
- (a) the sampling time constant by setting the FET switch ON and feeding a TTL square wave to S/H input, and

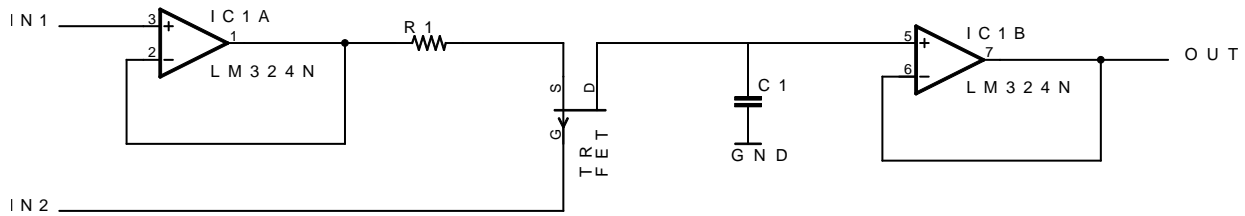


Figure 5: A simple S/H circuit

- (b) the holding time constant by feeding a TTL square wave to both S/H input as well as the FET's gate. This way the FET switch disconnects the holding capacitor from the input signal at the transition from high to low, thus allowing the capacitor to discharge through *bleeding*.

7. Conclude your work and findings.

3.5 SAMPLE CODE

3.5.1 MATH FUNCTIONS

Implementing $x = e^z$:

```
#include <math.h>
double x=exp(double z);
```

Compiling & Linking:

```
gcc -lm -o XXX XXX.c
```

3.5.2 TIME FUNCTIONS - excerpt from www.linux-mag.com/id/847

The `ftime()` function allows you to retrieve the time information in milliseconds instead of the somewhat coarsely grained seconds provided by the `time()` function. The prototype of the `ftime()` function as found in `<sys/timeb.h>` is `int ftime (struct timeb *tp);` Although declared with a return value, this function always returns 0. To obtain information from it, you pass it a pointer to a `struct timeb`. The struct is defined in `<sys/timeb.h>` as shown below along the sample code:

```

struct timeb
{
    time_t time;           // seconds since 01/01/1970  ~time()
    unsigned short int millitm; // milliseconds
    short int timezone;     // minutes west of Greenwich Mean Time
    short int dstflag;      // 1 if the system is using daylight savings time
};

/* sample code #1 */
#include <sys/timeb.h>
int main ()
{
    struct timeb the_time;
    ftime (&the_time);
    printf (Number of seconds: %dn, the_time.time);
    printf (Number of milliseconds: %dn, the_time.millitm);
    printf (Time zone: %dn, the_time.timezone);
    printf (Daylight savings time: %dn, the_time.dstflag);
}

```

The `gettimeofday()` function is very similar to the `ftime()` function except it provides even more precise time information. The prototype, as listed in `<sys/time.h>`, is as follows: `int gettimeofday (struct timeval *tv, struct timezone*tz);` The first struct (of type `struct timeval`) provides the time in seconds and microseconds. A sample program that prints out similar information to the previous example with `ftime()` is listed below:

```

/* sample code #2 */
#include <sys/time.h>
int main ()
{
    struct timeval the_time;
    struct timezone the_zone;
    gettimeofday (&the_time, &the_zone);
    printf (Number of seconds: %dn, the_time.tv_sec);
    printf (Number of microseconds: %dn, the_time.tv_usec);
    printf (Time zone: %dn, the_zone.tz_minuteswest);
    printf (Daylight savings time: %dn, the_zone.tz_dsttime);
}

```

4 RS485 LINE DRIVER

4.1 OBJECTIVES

In this lab, you will learn how to test a differential data bus driver/receiver for a multi-point communication. In particular, you will get familiar with these aspects of instrumentation:

- Differences between single-ended data transmission and differential data transmission
- Modeling and simulation of a dynamic circuit
- Design of an asymmetric delay for transient response

4.2 BACKGROUND

Line drivers and receivers are commonly used to exchange data between two or more points (nodes) on a network. Reliable data communications can be difficult in the presence of induced noise, ground level differences, impedance mismatches, failure to effectively bias for idle line conditions, and other hazards associated with installation of a network.

Standards have been developed to insure compatibility between units provided by different manufacturers, and to allow for reasonable success in transferring data over specified distances and/or data rates. The Electronics Industry Association (EIA) has produced standards for RS485, RS422, RS232, and RS423 that deal with data communications. Suggestions are often made to deal with practical problems that might be encountered in a typical network. EIA standards were previously marked with the prefix “RS” to indicate recommended standard; however, the standards are now generally indicated as “EIA” standards to identify the standards organization. While the standards bring uniformity to data communications, many areas are not specifically covered and remain as “gray areas” for the user to discover (usually during installation) on his own.

4.2.1 Single-Ended Data Transmission

Electronic data communications between elements will generally fall into two broad categories: single-ended and differential. RS232 (single-ended) was introduced in 1962 and has remained widely used through the industry. The specification allows for data transmission from one transmitter to one receiver at relatively slow data rates (up to 20K bits/second) and short distances (up to 50Ft. @ the maximum data rate).

Independent channels are established for two-way (full-duplex) communications. The RS232 signals are represented by voltage levels with respect to a system common (power / logic ground). The state MARK or 1 has the signal level negative with respect to common, and the state SPACE or 0 has the signal level positive with respect to common. RS232 has also numerous handshaking lines primarily used with modems. RS423 is another single ended specification with enhanced operation over RS232; however, it has not been widely used in the industry.

4.2.2 Differential Data Transmission

When communicating at high data rates, or over long distances in real world environments, single-ended methods are often inadequate. Differential data transmission (balanced differential signal) offers superior performance in most applications. Differential signals can help nullify the effects of ground shifts and induced noise signals that can appear as common mode voltages on a network.

RS422 (differential) was designed for greater distances and higher Baud rates than RS232. In its simplest form, a pair of converters from RS232 to RS422 (and back again) can be used to form an “RS232 extension cord.” Data rates of up to 100K bits / second and distances up to 4000 Ft. can be accommodated with RS422. RS422 is also specified for multi-drop (party-line) applications where only one driver is connected to, and transmits on, a “bus” of up to 10 receivers.

While a multi-drop “type” application has many desirable advantages, RS422 devices cannot be used to construct a truly multi-point network. A true multi-point network consists of multiple drivers and receivers connected on a single bus, where any node can transmit or receive data.

RS485 meets the requirements for a truly multi-point communications network, and the standard specifies up to 32 drivers and 32 receivers on a single (2-wire) bus. With the introduction of “automatic” repeaters and high-impedance drivers / receivers this “limitation” can be extended to hundreds (or even thousands) of nodes on a network. RS485 extends the common mode range for both drivers and receivers in the “tri-state” mode and with power off. Also, RS485 drivers are able to withstand “data collisions” (bus contention) problems and bus fault conditions.

4.3 EQUIPMENT

Equipment
Function generator
5V DC power supply
Prototyping breadboard with jump wires
Multimeter
Storage oscilloscope
Parts
SN75176 or equivalent
1N4448 fast diode or equivalent
74AC14 Hex Schmitt Trigger Inverter
Resistors, capacitors

4.4 EXPERIMENTS

Figure 6 shows the schematic of a point-to-point RS485 link shown as configured for left-to-right transmission only. Note the parasitic capacitances C1, C2, and C3. We will conduct experiments using only one 75176 chip as shown in Figure 7.

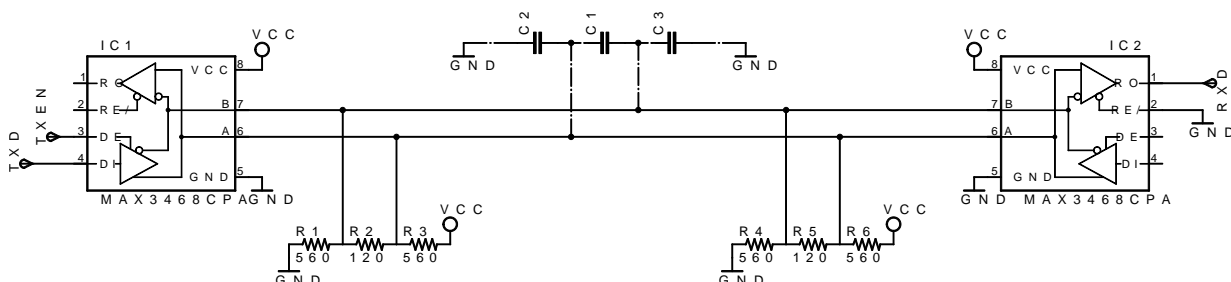


Figure 6: Two 75176 line drivers (one-way config)

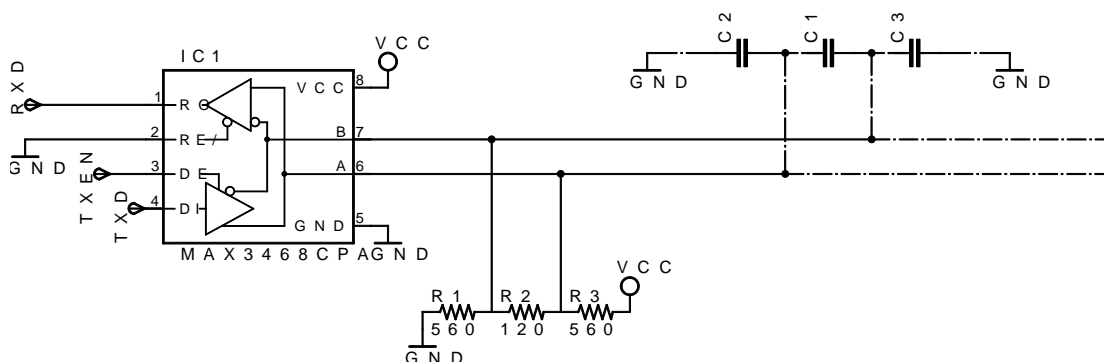


Figure 7: 75176 line driver test

1. Assemble the circuit in Figure 7 on the breadboard excluding the resistors R1 and R3 at the moment.
2. Connect a function generator to TxD and an oscilloscope to RxD.
3. Enable signal transmission by pulling high (+5V) the TxEN.
4. Verify the transmission of 10kHz TTL signal on the scope screen.
5. Now, feed the 10kHz TTL signal to TxD input and its inverted form to TxEN input (use one NOT gate from 74AC14. Dump the scope screen (ch1:TxD, ch2:RxD).
6. Add the resistors R1 and R3, and repeat the previous experiment.
7. Now, simulate the parasitic capacitances by adding capacitors C1, C2, and C3, all of value 10pF, and repeat the previous experiment.
8. Evaluate the maximum frequency a signal that can be transmitted assuming the sampling point being in the midpoint of each bit.

In principle, the signal transmitter (line driver) could be 'auto-enabled' by negated (inverted) TxD signal. This way, only '0' bits are put on the line while '1' bits (including the idle state) are driven by the pull-down and pull-up resistors R1 and R3. This would work as long as the parasitic capacitances of the line are negligible, which is often not the case. For this reason we will build an asymmetric delay line that will delay the transition edge from '0' to '1' on the TxEN input by a small amount of time to assist R1 and R3 to flip the polarity quickly.

1. Develop a simulation model for the circuit in Figure 8. If the logic part model is not available, simulate only the analog portion of the circuit (Figure 9). An open source circuit simulator is available at: qucs.sourceforge.net.
2. Feed a transition '0' to '1' and '1' to '0' to TxD and record the signal TEST and TxEN.
3. Now implement the TxEN circuit on the breadboard and measure its transient response. Discuss your results.

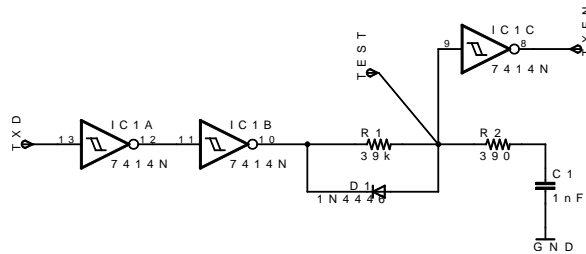


Figure 8: Auto-enable for 75176 line driver

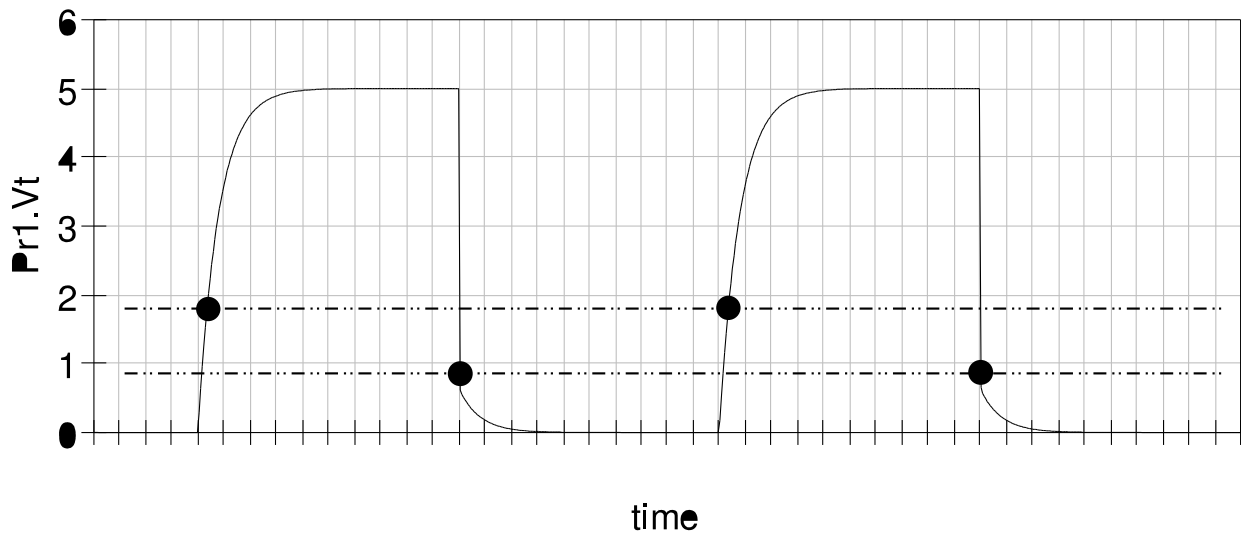
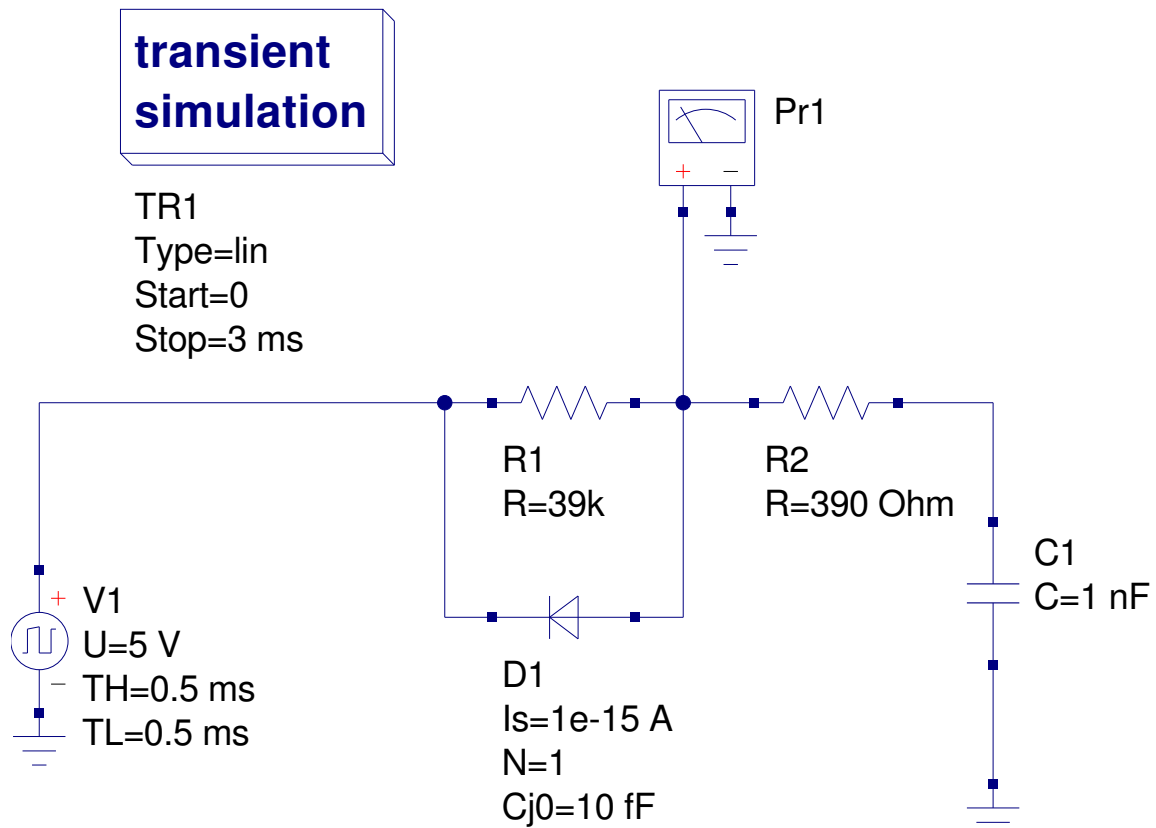


Figure 9:

5 RS485 BUS WITH MODBUS

5.1 OBJECTIVES

In this lab, you will build RS232↔RS485 converter for a multi-point communication bus. In addition you will be using a C Modbus library to create a custom application. In particular, you will get familiar with these aspects of instrumentation:

- Built a level shifter RS232 to TTL
- Solder the converter on a proto board
- Troubleshoot serial communication using a terminal software
- Implement Modbus protocol using C library

5.2 BACKGROUND

Modbus is a serial communication protocol used for transmitting information over serial lines between electronic devices. The device requesting the information is called the Modbus Master and the devices supplying information are Modbus Slaves. In a standard Modbus network, there is one Master and up to 247 Slaves, each with a unique Slave Address from 1 to 247. Versions of the Modbus protocol exist for serial lines (RTU and ASCII) and for Ethernet (Modbus TCP). Some functions are explained in detail below, for more information refer to *MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b*

Coil Numbers	Data Addresses	Type	Table Name
1-9999	0000 to 270E	Read-Write	Discrete Output Coils
10001-19999	0000 to 270E	Read-Only	Discrete Input Contacts
30001-39999	0000 to 270E	Read-Only	Analog Input Registers
40001-49999	0000 to 270E	Read-Write	Analog Output Holding Registers

Table 1: Modbus mapping

Function Code	Action	Table Name
01 (01 hex)	Read	Discrete Output Coils
05 (05 hex)	Write single	Discrete Output Coil
15 (0F hex)	Write multiple	Discrete Output Coils
02 (02 hex)	Read	Discrete Input Contacts
04 (04 hex)	Read	Analog Input Registers
03 (03 hex)	Read	Analog Output Holding Registers
06 (06 hex)	Write single	Analog Output Holding Register
16 (10 hex)	Write multiple	Analog Output Holding Registers

Table 2: Modbus functions

5.3 EQUIPMENT

Equipment
5V DC power supply Protoboard Multimeter Storage oscilloscope
Parts
SN75176 or equivalent MAX232/SP232A or equivalent 1N4448 fast diode or equivalent 74AC14 Hex Schmitt Trigger Inverter Resistors, capacitors

5.4 EXPERIMENTS

5.4.1

Build RS232 \iff TTL converter using MAX 232 chip, Figure 10.

5.4.2

Interconnect PC serial port with MAX232 and SN75176 including the auto-enabler. This will conclude the RS232 to RS485 converter hardware work.

5.4.3

Using *cutecom* terminal program, check your converter that always 'listens' to the bus (*RxEN* pulled down) using the loopback test:

TOP VIEW

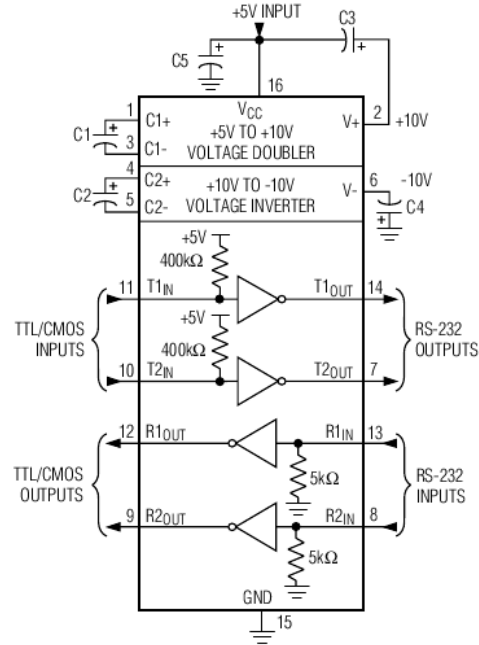
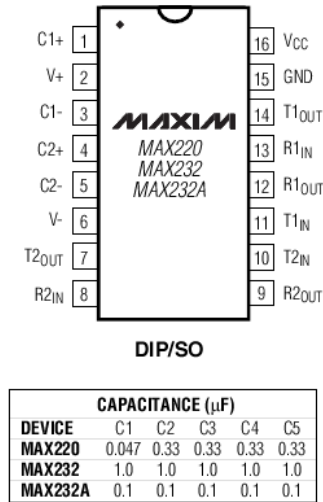


Figure 10: MAX232 converts RS-232 to TTL

1. Set the *device* to *ttyS0* or *ttyUSB0*¹
2. Set comm parameters to *9600/8-N-1*

In case you do not see what you sent out in the receive window , your converter is not working properly. Remove your converter and connect pin 2 to pin 3 on the DB9 RS232 connector (loopback test). Check the signal on the oscilloscope screen. Then reconnect the converter and check the signal on the bus. Connect line A to CH1 and line B to CH2, as there is no ground reference on RS485 bus! Then use a MATH function in the scope and perform (CH1-CH2) operation. Once you gain the proper function from your converter, proceed to the next part.

5.4.4

Download *libmodbus-0.0.4.tar.gz* (156.8 KB) from sourceforge.net/projects/libmodbus or fetch it from the LabNet server in `/autofs/pub/engr/courses/7680` directory using `ssh/sftp username@garfield.cs.mun.ca`. Untar the file and read the `README.txt` file.

Libmodbus is a dynamic library to use Modbus dialog protocol with GNU/Linux. LibModbus include master, slave and also serial port configuration functions. The

¹After plugging in the USB to RS232 converter, issue command `dmesg | tail` to see where your converter is mapped. Then write this map, i.e. `/dev/ttyUSB0`, into the CUTECOM device window since it is not included in the drop down options.

library is working only in RTU mode, so you must to configure every time 8 data bits.

Only register oriented functions listed below are implemented:

03 (0x03)	read n bytes
04 (0x04)	read n bytes
06 (0x06)	write 1 byte
07 (0x07)	read software status
08 (0x08)	line test
16 (0x10)	write n bytes

5.4.5

Develop software that responds to Master's queries using the Libmodbus library with these parameters: **9600** baud, **8** bit data, **1** stop bit, **NO** parity. Only the following two functions will be tested:

0x03	read holding registers
0x10	write holding registers

From each register, only the lower byte will be examined, i.e. the higher byte will be discarded upon reading. The slave database (memory map) will contain the following information:

A text message in ASCII that contains the **full name** and the **student number**, padded with zeros at the high end. This message will be located at even address registers starting at the offset **0xA0** as shown below for "John XXX":

:	:
0xA8	" "
0xA6	"n"
0xA4	"h"
0xA2	"o"
0xA0	"J"

Real-time updated **local time/-date** message in ASCII text format of "HH:MM:SS YYYY/MM/DD" will be also located at even address registers starting at the offset **0x40** as shown below for "12:31:56 2008/02/27":

:	:
0x48	"1"
0x46	"3"
0x44	":"
0x42	"2"
0x40	"1"

5.5 MODBUS references

5.5.1 Read Holding Registers 03 (0x03)

This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request specifies the starting register offset address and the number of registers.

The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits.

Function code	1 Byte	0x03
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D)

Table 3: Request 03

Function code	1 Byte	0x03
Byte count	1 Byte	2n
Register value	2n Bytes	data

Table 4: Response 03

5.5.2 Read Input Registers 04 (0x04) ~ Read Holding R's

5.5.3 Write Single Register 06 (0x06)

This function code is used to write a single holding register in a remote device. The Request specifies the address of the register to be written. The normal response is an echo of the request, returned after the register contents have been written.

Function code	1 Byte	0x06
Register Address	2 Bytes	0x0000 to 0xFFFF
Register Value	2 Bytes	0x0000 to 0xFFFF

Table 5: Request 06

Function code	1 Byte	0x06
Register Address	2 Bytes	0x0000 to 0xFFFF
Register Value	2 Bytes	0x0000 to 0xFFFF

Table 6: Response 06

5.5.4 Diagnostics (Serial Line only) 08 (0x08)

Function code 08 provides a test for checking the communication system between a client (Master) device and a server (Slave). The server echoes both the function code and sub-function code in a normal response.

5.5.5 Write Multiple registers 16 (0x10)

This function code is used to write a block of contiguous registers (1 to 123 registers) in a remote device. Data is packed as two bytes per register. The normal response returns the function code, starting address, and quantity of registers written.

Function code	1 Byte	0x10
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	0x0001 to 0x007B
Byte Count	1 Byte	2n
Data	2n Bytes	data

Table 7: Request 16