Introduction to Electrical and Computer Engineering Lab Manual

Professor Michael Lemmon
Department of Electrical Engineering
University of Notre Dame

Aknowledgements and Revisions

This lab manual was written for the Introduction to Electrical and Computer Engineering course by Professor Michael Lemmon. Much effort was put forth to create an interesting and fun introductory lab course that teaches electronic principles while exploring real applications.

Date	Editor	History
August 2001	M. Lemmon	Original publishing in pdf format
August 2002	M. Lemmon	Printed version published Power supply lab removed
August 2003	M. Lemmon	2nd printing Power supply lab reinserted as Lab 10
August 2004	M. Lemmon	3rd printing
August 2005	C. Manning	4th printing Lights and switches labs combined PWM labs combined Grading sheets moved to end of each lab

EE 224 - Introduction to Electrical Engineering
Lab Book Format

The grading of your labs will be based on successful demonstration of your working system to the teaching assitant (TA) as well as your lab book. You must have a bound lab book that has carbon inserts. The carbon copies of your lab report will be graded. The pages and carbons in your lab book are numbered. Your lab reports must only consist of sequentially numbered carbon pages. All results in the lab book are to be hand written. The only exception is for program listings. These program listings may be computer print outs. But copies of the programs must be in both your carbons and your lab book. The lab book is a record of your successes and failures in the lab, so it is crucial that everything that goes on in the lab is accurately recorded in your lab book. Reports that do not conform to these standards will automatically have their total grade reduced by 20 percent. No exceptions. Specific grading guidelines for each lab are given at the end of each lab. Lab books are an individual effort, the data gathering is a group effort. The lab counts for 30 percent of your total grade.

Each lab report must have 3 parts; an **pre-lab**, **in-lab**, and **post-lab** section. The pre-lab section discusses analysis work and preliminary software/hardware designs that were completed prior to starting the lab. The in-lab section discusses what actually happened in the lab and should include tables of experimental data that you may have taken to verify your system's functionality. The post-lab section of the lab report assesses the performance of your in-lab system against pre-lab predictions.

Each section must be written in English using traditional rules of usage and style that were discussed in your freshman composition seminar. Figures (schematic diagrams and breadboard layouts) and program listings must all be accompanied by written explanations of how the hardware/software works. Graphs, plots, and tables must be labeled and must be accompanied by written explanations of what the values are, how they were gathered, and what trends were observed.

Certain parts of the lab require the TA to check your breadboard or lab book prior to proceeding to the next task. Checking of the breadboard is done to prevent the student from building circuits that destroy the MicroStamp11. Checking of the lab book (usually pre-labs) is done to make sure that the student has completed their predictive work before going on to the in-lab tasks. Once a TA has completed the requested check, he/she will certify the check by signing and dating your lab book. This certification should also appear in the carbon copy of your lab report.

The labs build upon each other in a logical manner, which means that the labs must be completed sequentially. When you finish your lab, you must demonstrate it to the TA and turn in the final lab report. You will not be allowed to go on until the lab has been successfully demonstrated.

An example of a sample lab report , as well as the lab report presentation can be found on the lab website http://www.nd.edu/ eeuglabs/ee224/.

CHAPTER 1

Breadboarding Circuits

1. Objective

The purpose of this lab is to provide the student with hands-on experience in breadboarding simple resistive circuits.

2. Parts List

(1) **10 k-ohm trim pot.**
(2) **100 ohm resistor.**
(3) **light emitting diode (LED)**
(4) **breadboard and wire-kit**
(5) **digital multi-meter**

3. Background

In this lab the student will analyze, build, and test a simple *circuit* with a *resistor*, a variable resistor or *potentiometer*, a *light emitting diode* (LED), and an *independent source*. By *analysis*, we mean that the student will predict the voltage across and current through branches in a specified electrical circuit. By *building*, we mean that the student will use a *solderless breadboard* to construct the specified circuit. By *testing*, we mean that the student will use a *digital multi-meter* (DMM) to measure the voltage across and current through various branches of the specified circuit. This section discusses some of the background material required to complete the lab.

3.1. What is a circuit? A *circuit* (also called an *electrical network*) is a collection of electrical multi-terminal devices that are connected in a specified manner. For the most part, we'll be concerned with *two-terminal devices*. A two terminal device is an electrical device with two lines or leads coming out of it.

An example of an electrical network consisting of four two-terminal devices is shown in figure 1. This figure has two pictures. The lefthand picture is a graphical representation of the circuit called a *schematic diagram*. The places where device terminals are connected will be called *nodes*. Each node is labelled with a letter. In figure 1, there are three nodes with the labels a, b, and c. The lefthand picture is often abstracted into a graphical representation that emphasizes the important connections with the network. Such a *graph* representation for the circuit is shown in the righthand picture. In this picture, you'll see that the circuit element is drawn as an *arc* and all of the nodes are simply drawn as a point. The righthand representation is called a *graph*. In a graph, the circuit elements are always represented as *arcs* (also called *branches*) and the terminals are always represented as points or *nodes* of the graph.

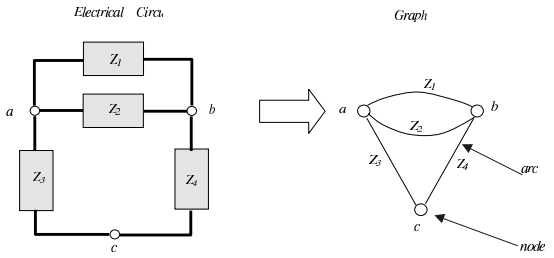

FIGURE 1. A simple electrical circuit and its graph

To characterize what a circuit does, we must be able to characterize the state of the circuit. The *state* of a circuit is determined by characterizing

- The *voltage* across each arc and
- the *current* through each arc.

Circuit analysis, therefore, is concerned with determining these two quantities for each branch of the circuit.

Electronic circuits are devices that do work by moving electrically charges about. The base unit of **charge** is the **coulomb**. One coulomb equals the charge of 6.24×10^{18} electrons. Charge at time t is denoted as $q(t)$. Moving charges generate an electric **current** that we denote as i or I. If $q(t)$ is the amount of charge at a point in a wire at time t, then the current passing through this point equals the first time derivative of q. In other words,

$$i(t) = \frac{dq(t)}{dt}$$

The basic unit of current is the **ampere** (denoted as A). One ampere equals one coulomb of charge passing through a point in one second. In other words, one ampere equals one coulomb per second.

Current is generated by an **electro-motive force** (EMF). Recall that a force that is applied for a specified distance generates **work**. So when we have an electro-motive force move charges over a specified distance (i.e. through a wire or device), then work is being done. This work is call **voltage**. In particular, a **volt** (abbreviated as v or V) is defined as the work done in applying a force of one newton on 1 coulomb of charge over a distance of one meter. Since one joule equals one newton-meter, this means that one volt equals one joule/coulomb.

3.2. What is a resistor? An ideal *resistor* is a two-terminal device in which the voltage across the terminals is proportional to the current flowing through the device. The constant of proportionality is denoted as R, the *resistance* of the device. This resistance is measured in units of *volts per ampere* or *ohms* (denoted by the Greek symbol Ω). In mathematical terms, this relationship is written as

$$v(t) = Ri(t) \tag{3.1}$$

where R is the *resistance*, $v(t)$ is the voltage across the resistor, and $i(t)$ is the current flowing through the resistor. Equation 3.1 is usually called *Ohm's Law*.

The symbol for a resistor is shown by the lefthand picture in figure 2. The righthand picture in figure 2 depicts the actual component. From this picture you will find that the resistor is a small cylindrical

component with two wire leads coming out of each end. Often the device will have colored bands around it. These bands are a color code specifying the value of the resistor in *ohms*.

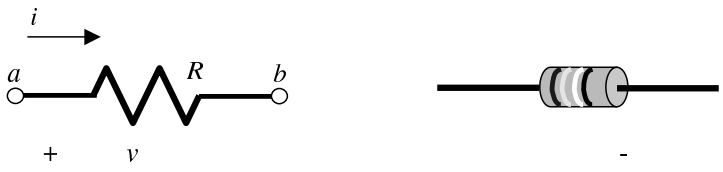

FIGURE 2. Resistor

Equation 3.1 is the equation for a *linear* resistor. The linearity of the device can be readily appreciated if we draw the current-voltage characteristic or I-V curve for the device. This curve plots the voltage $v(t)$ across the device as a function of the current $i(t)$ through the device. Figure 3 shows the I-V characteristic for a linear resistor. This characteristic is a straight line. The resistance is given by the slope of the line.

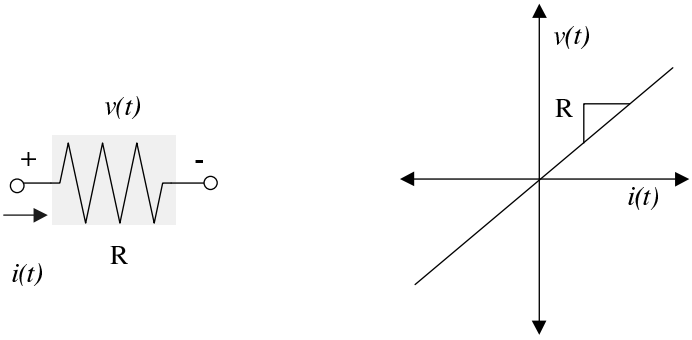

FIGURE 3. A linear resistor and its I-V characteristic

Two special types of resistors are the *short circuit* and *open circuit*. We define a *short circuit* as a two-terminal device device whose resistance is zero. An open circuit is a two-terminal device whose resistance is infinite.

A special type of a resistor is a *potentiometer*. We sometimes refer to them as pots. The potentiometer has three terminals. There are two terminals at either end of a resistor (a and c) and a third terminal connection (called the *wiper*) that taps into the middle of the resistor. The lefthand picture in figure 4 shows the symbol for a potentiometer, which is a resistor with the wiper lead tapping into the middle of the device. The righthand picture shows the physical device. This particular trim pot has a dial on the front that allows you to mechanically adjust the position of the wiper. The first and third leads on the bottom of the device correspond to the two ends of the resistor and the wiper lead is the lead in the middle.

You can use the potentiometer to construct a resistor whose resistance changes when you change the wiper position (by turning the dial on the front of the pot). This is simply done by connecting lead a to the circuit and connecting the wiper (lead b) to the circuit. The lefthand picture in figure 4 shows which two leads you

must connect in order to get a variable resistor. By changing the dial position you can change the resistance between leads a and b.

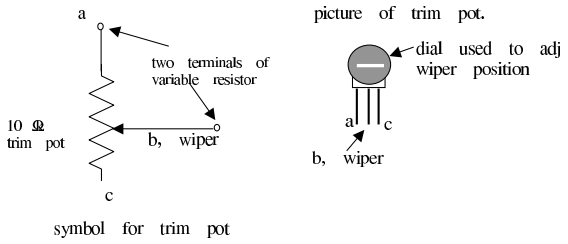

FIGURE 4. A variable resistor

3.3. What is a light-emitting diode? A *diode* is a two-terminal semiconductor device. It can be thought of as an electronic valve that only allows current to flow in one direction. The symbol for the diode is shown in the lefthand picture of figure 5. The symbol is shaped like an arrow that indicates the direction in which current may flow. The terminal marked with a positive sign is called the *anode* and the terminal marked with a negative sign is called the *cathode*. The righthand picture depicts the physical device. It looks similar to a resistor except that it has a single band on one end. In a forward biased diode, the current will flow from the end without a band to the end of the cylinder with the band.

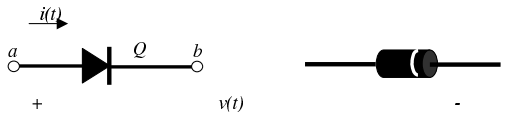

FIGURE 5. Diode

When the voltage v is positive and greater than a minimum threshold voltage V_t, then the diode is said to be *forward biased*. A forward biased diode will conduct current i, in the direction shown in the figure. If a diode is not forward biased, then we say it is *reverse biased*. A reverse biased diode will also conduct a current that has the opposite sense of that shown in figure 5. This reverse current, however, will be extremely small so that the forward biased diode is seen as conducting, whereas the reverse biased diode is seen as not conducting.

As with the resistor, the diode is completely characterized once we know the relationship between the voltage and current. The diode's IV characteristic satisfies the following equation

$$i = I_0 \left(e^{qv/kT} - 1 \right)$$

where q is the charge of an electron, k is Boltzmann's constant (1.381×10^{-23} J/K), and T is the material's temperature (Kelvin). The reference current I_0 is usually very small, on the order of 10^{-9} or 10^{-15} amperes. Plotting this function leads to the IV characteristic shown in the lefthand graph of figure 6. Note that this graph is actually the V-I curve since it shows how current varies as a function of voltage.

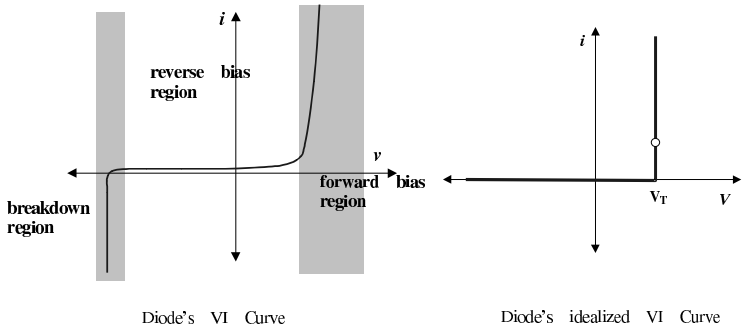

FIGURE 6. Diode's IV characteristic

The lefthand plot in figure 6 has three distinct operating regions. The *forward bias region* corresponds to those positive voltages that are above a specified threshold level. The threshold voltage, V_T, is a function of the physical properties of the semi-conductor material. Common values for this threshold voltage lie between 0.6 and 1.4 volts. For voltages that lie below this threshold, the diode essentially stops conducting. There is a small leakage current that is on the order of I_0. But as noted earlier this current is extremely small. If we further decrease the voltage, then we enter another region of operation known as the breakdown region.

We generally operate a diode in either its forward or reverse biased modes. In particular, we usually idealize this behavior so we can think of the diode as a valve that is open when v is greater than the threshold voltage V_T and is closed otherwise. These considerations lead to the simplified I-V characteristic that is shown in the righthand graph of figure 6. In this simplified plot, we see that the reverse bias region is idealized so that zero current is passed in this region if $v < V_t$. If the diode is forward biased, then the current is potentially unbounded, which means that the diode behaves like a short circuit. In other words, a forward biased diode behaves like a short circuit and a reverse biased diode acts like an open circuit.

An LED is a *light emitting diode*. The LED emits light when it is forward biased and it emits no light when it is reverse biased. The intensity of light is proportional to the square of the current flowing through the device. Figure 7 shows a picture of an LED. Note that LEDs have two leads. One lead is longer than the other. These leads are used to indicate which end of the diode is positive (anode) and which is negative (cathode). In many cases the longer lead is the anode, but you can easily test this by connecting the LED to a battery and seeing which orientation causes the LED to light up.

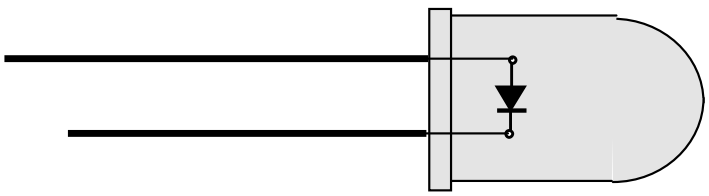

FIGURE 7. Light Emitting Diode

3.4. What is an independent source? Resistors are examples of so-called *passive* devices. We call them passive because they always dissipate energy. *Active* circuit elements actually generate energy. Examples of active circuit elements include *independent voltage sources* and *independent current sources*.

An independent voltage/current source is an idealized circuit component that fixes the voltage or current in a branch, respectively, to a specified value. Remember that the *state* of a circuit is given by the voltage across and current through each branch of the circuit. If a branch is a resistor, then we know that the current and voltage are related via Ohm's law. If that branch is an independent voltage source, then we know that the voltage across the branch has a fixed value, but the current is free. If the branch is an independent current source, then the voltage is free and the current through the branch is fixed.

Figure 8 shows the symbols for three independent sources. The lefthand symbol depicts an independent voltage source. The symbol is a circle with the voltage polarities marked on them and the voltage value V. The righthand symbol depicts an independent current source. The symbol is a circle with the current direction denoted by an arrow in the middle of the circle and the value or magnitude of the current i. The middle symbol is the symbol for a specific type of independent voltage source known as a *battery*. A battery is a physical realization of an independent voltage source. Physical realizations for independent current sources are often specially built transistor circuits (an important 3-terminal device that we'll introduce later).

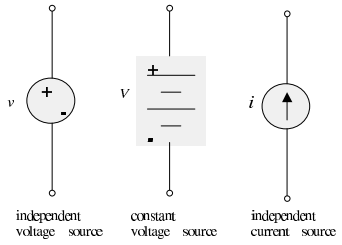

FIGURE 8. Independent Voltage and Current Sources

3.5. How is circuit analysis performed? As was mentioned earlier, a circuit is an interconnection of electrical devices. For the most part, we'll be concerned with interconnections of two-terminal devices such as resistors and diodes. An example of such a circuit is shown below in figure 9. The lefthand picture is the circuit diagram and the righthand picture is the circuit's graph.

Circuit analysis requires that we determine the voltage across and current through all branches of a circuit. For the circuit in figure 9, the independent voltage source makes it easy to specify the voltage across nodes a and b, but how do we analyze the rest of the circuit? To do this, we need to invoke two special physical laws that lie at the heart of all circuit analysis. In particular, we need to use the laws known as *Kirchoff's current law* or KCL and Kirchoff's voltage law *KVL*. These two laws are conservation principles that must always be obeyed by any passive circuit. We can use these laws to help determine the voltages and currents in the circuit's branches.

Kirchoff's Voltage Law (KVL) is stated with respect to a loop in a circuit's graph. It states that:

> the algebraic sum of the voltages around any loop equals zero.

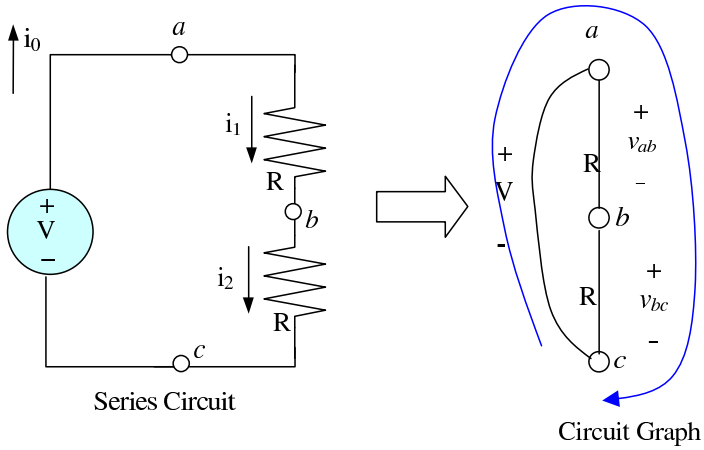

FIGURE 9. A Simple Circuit

A loop is a sequence of connected branches that begin and end at the same node. Figure 9 marks one of the loops in our circuit. This is the loop formed from branches

$$(a,b) \rightarrow (b,c) \rightarrow (c,a)$$

The voltages obtained by traversing this loop are

$$v_{ab}, v_{bc}, V$$

KVL says that the "algebraic" sum of these voltages must equal zero. By algebraic, we mean that the voltages are signed quantities. The voltage polarity or sign of each arc is determined by the direction in which we traverse the arc. If we start at node a and begin tracing out our loop in a clockwise direction, we see that the traverse of branch (a,b) goes from $+$ to $-$. This is considered as a negative change in potential (i.e. we're decreasing the potential). The same is true for the voltage over branch (b,c). Note, however, that in traversing branch (c,a) that we are going from a negative to positive polarity. The change in potential, therefore, is positive. On the basis of our preceding discussion, we can see that KVL will lead to the following equation:

$$V - v_{ab} - v_{bc} = 0$$

KVL is an energy conservation relation. It states, in essence, that the total work done in going around a loop will be zero.

The other important circuit relation is Kirchoff's current law. *Kirchoff's Current Law (KCL)* is stated as follows:

> The algebraic sum of current at any node is zero.

To explain what this statement means, let's consider the circuit shown in figure 10. This figure shows an independent source of V volts connected to a resistive network . The single node a of this circuit is shown in the righthand drawing of figure 10. At this node, we see three currents. Two of these currents i_1 and i_2 are leaving node a and the third current i_0 is entering node a. Currents that are entering a node are assumed

to have a positive sign, whereas currents leaving a node have a negative sign. By Kirchoff's current law, the algebraic sum (which takes into account the sign of the currents) must be zero. This means, therefore, that

$$i_0 - i_1 - i_2 = 0$$

Note in this equation that the sign preceding current i_1 and i_2 is negative. This is because those currents are leaving node a and therefore have a negative sense.

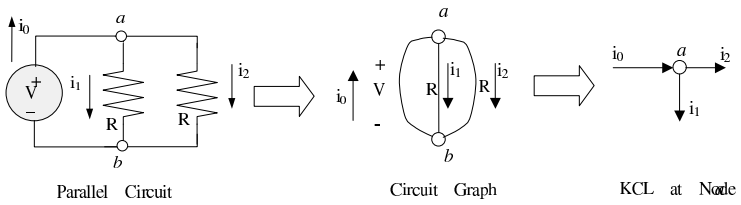

FIGURE 10. KCL at node b

Kirchoff's current law is simply a statement that charge cannot accumulate at the nodes of a circuit. This actually makes quite a bit of sense if you realize that the nodes are perfect conductors and therefore provide no place for charges to rest. This principle is identical to concepts found in fluid dynamics. Namely that if you look at the fluid flowing into one end of a pipe, you expect the same amount of fluid to flow out the other end. If this did not occur, then fluid would accumulate in the pipe and eventually cause the pipe to burst. KCL is nothing more than an electrical equivalent of this intuitive physical idea from fluid mechanics.

The key issue is to see how we can use KVL, KCL and Ohm's law to determine all of the branch voltages and currents in a specified circuit. We use the circuit in figure 9 to illustrate this process. What we will now do is determine all of the currents and voltages in this circuit.

We begin by using KCL at node a, b, and c. Remember that KCL states that the sum of the currents entering a node must equal the sum of the currents exiting a node. Applying KCL to nodes a, b, and c will therefore result in three different equations

$$\begin{aligned} i_0 &= i_1 \\ i_1 &= i_2 \\ i_2 &= i_0 \end{aligned}$$

These three equations, of course, simply say that the current going through all branches in the circuit must be equal. In other words, KCL allows us to conclude that $i_0 = i_1 = i_2$.

We now look at the voltages over each of the branches in this circuit. Because arc (c, a) is an independent voltage source, we know that $v_{ac} = V$ volts. The other arcs, however, are resistors and this means that they must satisfy Ohm's law. Applying Ohm's law to these branches allows us to conclude that

$$\begin{aligned} v_{ab} &= i_1 R \\ v_{bc} &= i_2 R \end{aligned}$$

Note that we've already deduced that $i_1 = i_2 = i_0$, so that because the resistors have identical values of R ohms, we can also conclude that $v_{ab} = v_{bc} = i_0 R$.

Finally, we use KVL (as before) to write down a single loop equation relating all of the voltages,

$$V - v_{ab} - v_{bc} = 0$$

Using our previous results, this equation can be rewritten as

$$V - i_0 R - i_0 R = 0$$

which is an algebraic equation in a single unknown quantity i_0. We can now solve for i_0 to deduce that

$$i_0 = \frac{V}{2R}$$

So we've determined the current leaving the voltage source i_0 as a function of V (independent voltage source) and R (the resistance).

On the way, however, we determined that all of the other currents and voltages in the circuit can be written as functions of this current i_0. Recall, that we deduced that $i_0 = i_1 = i_2 = V/2R$, so that we now know all of the currents in the circuit. Once the currents are known, we can use Ohm's Law to readily deduce that $v_{ab} = v_{bc} = V/2$. In other words, this circuit evenly divides the voltage supplied by the independent voltage source between the two resistors in the circuit.

3.6. What is a Solderless Breadboard? The μStamp11 module is built on a printed circuit board (PCB). A PCB is a non-conducting board upon which there are conducting strips. The components of your circuit are then connected to these conducting strips. The connections can be made using solder or wire-wrap. The problem is that these two types of connections are rather permanent. If you make a mistake in your initial circuit, it is difficult to "undo" what you've done. As a result, these methods are inconvenient for prototyping circuits.

To build prototype circuits, we'll use a special device known as a *solderless breadboard*. We often refer to such breadboards as *proto-boards*.

Figure 11 is a top down view of a standard proto-board. The protoboard consists of a set of holes that are just the right size for accepting the leads of electrical devices. (such as a resistor lead). The holes in the proto-board are electrically connected in a systematic manner so you can easily build electrical circuits by simply inserting the leads of your circuit components into the protoboard's holes.

The proto-board's holes are electrically connected in a systematic manner. A long row of holes on the top (bottom) of the board are electrically connected. These rows are usually connected to the power supply and ground and we refer to them as *power buses*. In the middle of the board, you'll find two columns of holes stacked on top of each other. These columns are also electrically connected. We usually insert components into these holes. In figure 11, we've circled the electrically connected groups of holes on the proto-board.

The nice thing about a proto-board is that you can easily build circuits by inserting one end of a device's lead into one hole and then inserting another component's lead into one of the electrically connected holes. This means that it is easy and fast to build circuits.

It is important, however, that one is NEAT in building prototype circuits. Being neat means that wires of appropriate lengths are used and that wires and components lie flat against the proto-board (if possible). It is highly recommended that your wires and components run in vertical and horizontal directions. Neat breadboards are important for more than aesthetic reasons. Careful wiring makes it easier to debug your circuits when they don't work. It also prevents accidental shorting of components and excessive parasitic effects that can greatly degrade the performance of your circuit.

3.7. What is a Multimeter? This lab will ask you to measure various voltages and currents in a circuit. You will need to use a special measurement device known as a *multimeter* to accomplish this.

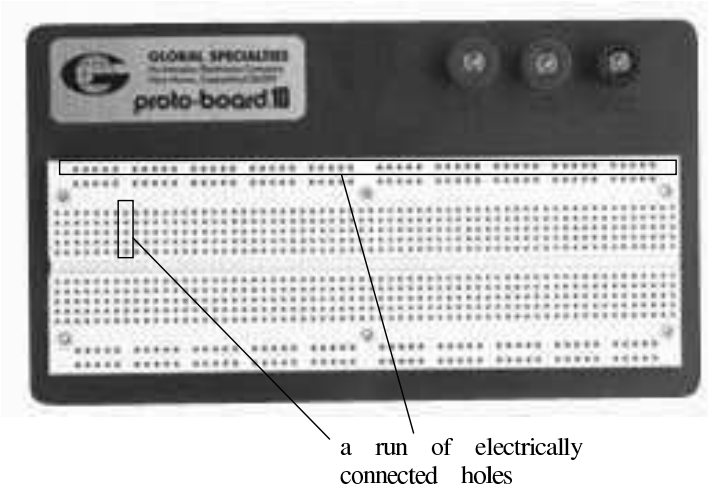

FIGURE 11. Solderless Breadboard

A multimeter is a device that can be used to measure multiple quantities (hence the name *multi*-meter) such as current, voltage, and resistance. Figure 12 shows the face-plate of an inexpensive hand-held digital multi-meter (DMM).

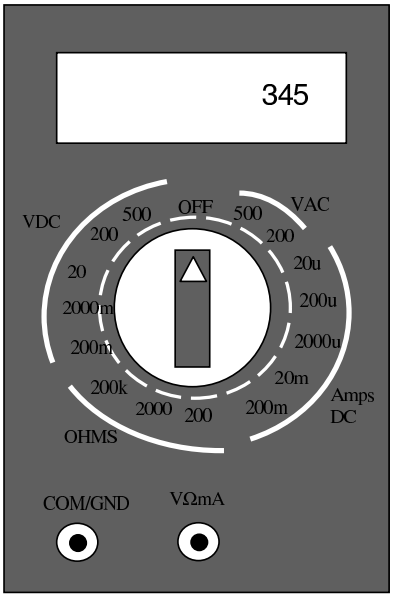

FIGURE 12. Digital Multi-meter

In the middle of the DMM's faceplate you will find a large rotary rotary switch, an LCD (liquid crystal) display, and two (sometimes 3) jacks for probes. The rotary switch is used to switch the DMM into its mode. It has one of 4 operating modes. It can either be used to measure constant voltages (VDC), sinusoidal voltages (VAC), constant currents, or resistances (ohms). Note that for each of these operating modes, there are 3-4 additional submodes that determine the largest value that can be displayed on the DMM's LCD. For example, if you were to set the switch to VDC 200m, then the largest voltage to be displayed on the LCD will be 200 milli-volts.

The actual measurements are made with two probes. One probe is black and the other is red. The black probe is usually connected to the COM/GND jack on the front of the DMM. The red probe is usually connected to the other jack ($V\Omega mA$). The black probe is usually taken to be the reference or ground node.

You can use the DMM to measure DC currents, DC voltages and resistance. To measure a DC voltage between two points on a circuit, you need to connect the black probe to the bottom end of the branch and the red node to the top end of the branch. Once the DMM switch is placed to one of the VDC positions, the voltage over that branch will be displayed on the DMM's display. Figure 13 shows how to make the connections. The lefthand picture shows the circuit schematic and the righthand picture shows where to connect the DMM to measure the voltage across the second resistor.

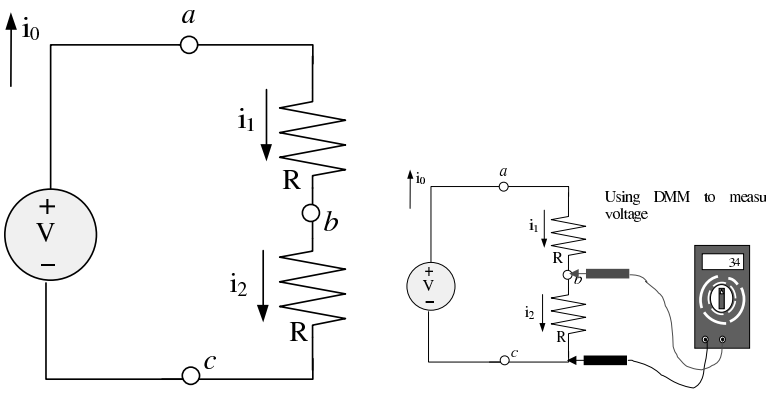

FIGURE 13. Using a Digital Multi-meter

To use a DMM to measure current, we need to make some changes to the original circuit. Recall that current is the rate at which electrons flow past a point in a circuit. To measure this flow, we need to insert the DMM into the flow. To do this we need to break the circuit. Figure 14 shows how to do this. The lefthand drawing in the figure is the circuit and we want to measure the current flowing through the second resistor R_2. The righthand drawing shows that we actually have to disconnect one terminal of the resistor. The red lead of the DMM will be connected to one of the free ends and the black lead will be connected to the other free end. These connections are shown in the righthand picture of figure 14. Once the DMM rotary switch is set to the current measuring position, the display on the DMM will show the actual current flowing through the second resistor.

You may also use the DMM to measure a resistor's resistance. To do this, first remove the resistor from the circuit and set the DMM's rotary switch to one of the resistor positions. Connect one probe to one end of the resistor and the other probe to the other end of the resistor. The measured resistance should appear in the DMM's display. Note that it is important that the resistor actually be removed from the circuit. If you

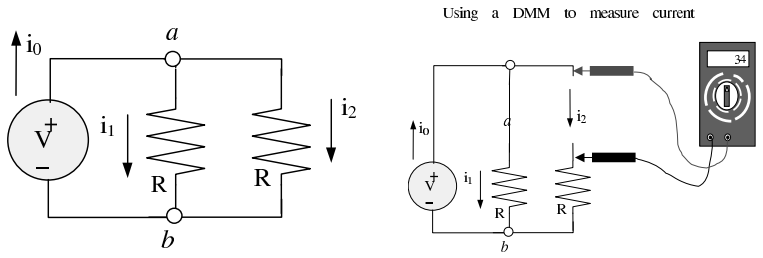

FIGURE 14. Using a Digital Multi-meter

attempt to measure the resistance of a resistor while it is still in the circuit, then you will not get the correct answer. You will be measuring the so-called equivalent parallel resistance of your resistor with the rest of the circuit.

In this particular laboratory we will be using a more elaborate DMM, the Fluke 45 located on your bench. See the Multimeter supplement at the very end of this lab for more detailed information.

4. Tasks

4.1. Pre-lab Tasks: Consider the circuit shown in figure 15. This circuit consists of a 5 volt independent voltage source driving a resistive circuit with a single LED. One of the resistors is a 100 ohm resistor. The other resistor is a variable resistor whose value can be changed between 0 and 10 kilo ohms. Since the second resistor's value is variable, the second resistor's value is denoted by the variable R.

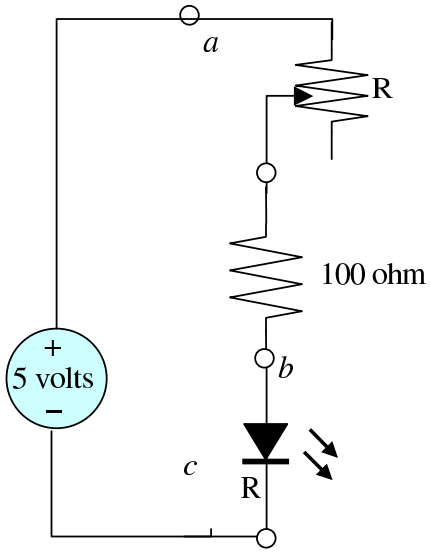

FIGURE 15. Circuit for Lab 1

Before coming to the lab you should do the following:

(1) Draw a *labelled* schematic diagram of the circuit with an explanation of how the circuit works.
(2) Draw a picture showing how you plan to breadboard the circuit.
(3) Derive an expression for the current going through the diode as a function of the variable resistance, R.
(4) Plot the current through the LED as a function of the variable resistance R. Do this using at least 10 different resistance values.

Ask the TA to check your completed pre-lab analysis. If your answers are correct, then you can proceed to the In-lab task.

4.2. In-Lab Tasks: The In-lab task asks you to breadboard the circuit shown in figure 15 and then to use a DMM to make current and resistance measurements on the circuit. Remember that to use the amp meter you will need to actually break the circuit and place the amp meter in series with the resistors. Your lab book's description of the IN-LAB task must include the following:

(1) A description of what happened during the IN-LAB task. This is a description of what you did. If things did not work out and you had to make a change, go ahead and write this down as well. This description is a record that you can use later to reminder yourself of what actually occurred in the lab.
(2) Construct the circuit on the breadboard shown in figure 15. You should use the following steps
    (a) Hook up the power buses to the breadboard terminals to the power rails.
    (b) breadboard the resistive network
    (c) connect power supply to the breadboard.

    Your wiring must be neat with components and wires lying flush against the breadboard and arranged in a rectangular manner. You will be asked to rebuild the circuit if these guidelines are not followed.
(3) Use the DMM to measure the resistance of the variable resistor. You can change this resistance by turning the knob on the component. Using the same resistance values used in pre-lab, measure the current going through the diode and observe the brightness of the diode. Make qualitative observations about the LED's brightness as a function of current. Note that you will need to remove the resistor from the circuit to accurately measure the potentiometer's resistance.
(4) Use the DMM to measure the voltage drop across the LED. Is this what was expected? Why or why not?

4.3. Post-lab Exercises: The POST-LAB portion of your lab book should contain the following.

(1) A graph plotting the expected and measured current through the diode as a function of the variable resistance, R.
(2) Assessment of how well your in-lab measurements agree with your pre-lab predictions. If there is a big disagreement, figure out why and re-do the experiment to see if you can get better agreement between predicted and measured current/resistance relationship.

5. What you should have learned

The purpose of this lab was to provide you with experience in breadboarding simple electrical circuits. You were asked to build simple resistive circuits driving a light-emitting diode. In completing this lab you should have learned the following:

(1) How to breadboard circuits.
(2) How to use a DMM to measure resistance, voltage, and current.
(3) How to use Ohm's law, KVL, and KCL to predict the voltage and currents in a branch of a resistive network.

6. Multimeter supplement

EE224 - Lab 1 Multimeter supplement

This supplement is intended to assist in your familiarity of the equipment used in the lab. Thus, it is an addition to, and not a replacement for pages 12-14 of lab 1. The method for taking different types of measurements are the same as outlined in your lab. You will find this true of any multimeter you use. What changes are the buttons and knobs, and input terminals. Figures 2-5 to 2-8 give some assistance in taking basic measurements, while Figure 3-4 addresses which input terminals to use.

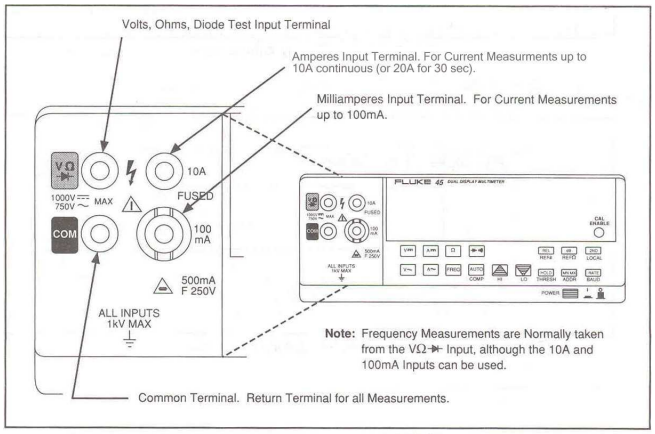

Figure 3-4. Input Terminals

Instead of a rotary knob with different range selections, the Fluke 45 has function buttons to select the type of measurement. Instead of selecting the range, for most of your measurements in this course, you will be using the AUTO range function of the Fluke 45 Digital Multimeter. AUTO is the default range. Figure 3-5 is a diagram of the function buttons.

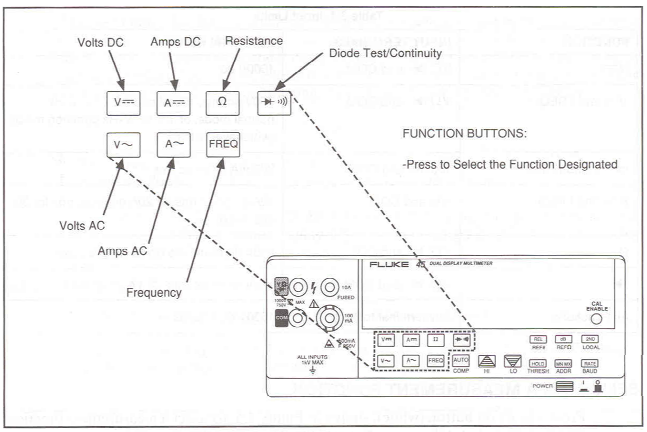

Figure 3-5. Function Selection Buttons

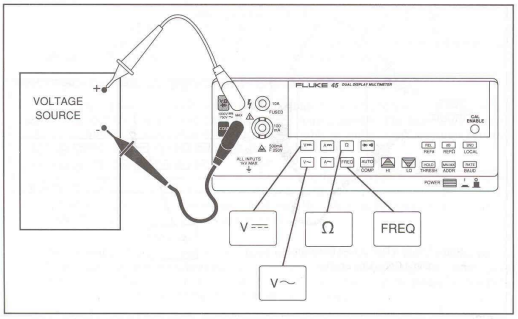

Figure 2-5. Measuring Voltage, Resistance, or Frequency

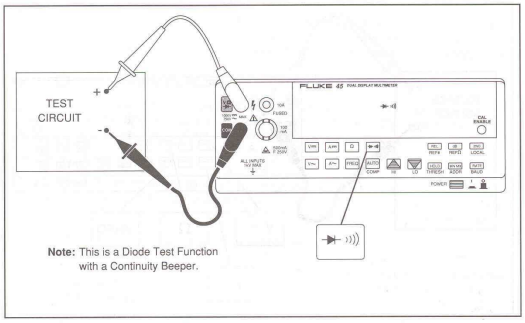

Figure 2-7. Continuity Testing

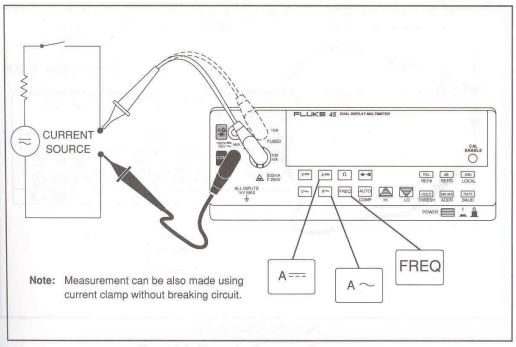

Figure 2-6. Measuring Current or Frequency

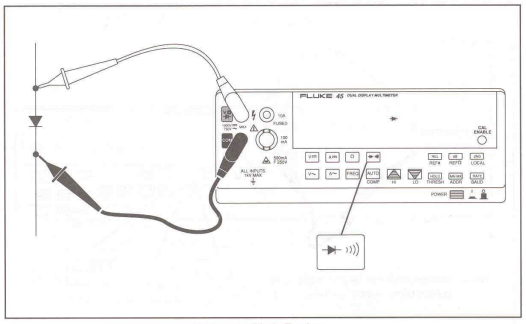

Figure 2-8. Diode Testing

Your TA has a complete operators manual if you would like more information.

7. Grading sheet

LAB 1 - Breadboarding Circuits

10 Pts	**PRELAB TASKS**	
2	Explanation of Circuits Operation	
1	Labelled Circuit Schematic.	
1	Picture of Breadboard Layout.	
3	Analysis determining current through resistive network as a function of the variable resistance, R.	
2	Graph plotting expected current through resistive network as a function of the variable resistance, R.	
1	TAs verification that the pre-lab tasks were completed correctly.	

8 Pts	**IN-LAB TASKS**	
3	Description of what happened during the IN-Lab Task.	
3	10 current/resistance measurements.	
1	Brightness observations as a function of current.	
1	Voltage drop across LED measurement.	

5 Pts	**POST-LAB TASKS**	
3	Graph plotting expected and measured current through resistor as a function of variable resistance, R	
2	Assessment of the agreement between pre-lab predictions and in-lab measurements.	

2 Pts	**DEMONSTRATION ..... STOP if not checked off**	
2	Successful demonstration of completed system.	

GO ON to next lab if completed.

Name _____ .

Lab Group Number _____ GRADE _____ out of 25

TA signature _____ Date _____ .

CHAPTER 2

MicroStamp11 Familiarization

1. Objective

The purpose of this lab is to provide the student with hands-on experience in programming the MicroStamp11.

2. Parts List

(1) **MicroStamp11 with docking module**
(2) **serial cable**
(3) *wiring kit and solderless breadboard*
(4) **Computer with compiler and loader**

3. Background

In this lab the student will write and compile a C-language program that will then be downloaded and executed on a *micro-controller* known as the *MicroStamp11*. To complete the lab, the student will need to provide *power* to the MicroStamp11 and will need to *communicate* with the device over a personal computer's (PC) serial port. The student will write and compile a C-language *program* on a PC. The program will use *kernel functions* to facilitate communication with the PC. The student will then *download* this program into the MicroStamp11. After the program is stored in the MicroStamp11's memory, the student will need to *start* executing the program and demonstrate that the program works as expected.

3.1. What is a micro-controller? The MicroStamp11 is a *micro-controller* module that is built around the Motorola 68HC11 micro-controller. A micro-controller is a special type of micro-computer (a computer on a single integrated circuit or IC) that has been specially designed to interact with the outside world.

To see how a micro-controller differs from a regular micro-computer, let's first look at the architecture of a regular computer. All computer systems are characterized by similar subsystems. The lefthand drawing in figure 1 is a block diagram for a generic computer system. The block diagram shows that the computer consists of a *central processing unit* (CPU), *clock*, *memory*, and *peripheral* or *input/output* (I/O) devices. All of these subsystems communicate over a communication subsystem called the *CPU bus*. The bus is, essentially, a pair of wires that interconnect all of the subsystems. In general, only one pair of devices can talk to each other at a time, so that communication over the bus must be coordinated to prevent message collisions.

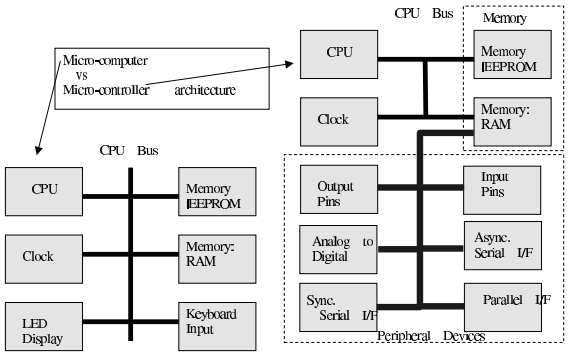

FIGURE 1. Block diagrams for micro-computer and micro-controller

The central processing unit (CPU) executes instructions contained in *memory* in synchrony with a hardware *clock*. The instructions contained in memory originate in a computer program that has been loaded into the computer's memory. Occasionally, the computer requires inputs from the outside world or must communicate its results to a user. This is done by reading or writing to a *peripheral* or I/O device. Common input devices are keyboards, sensor such as digital thermometers or potentiometers. Output devices include video displays, liquid crystal displays (LCD), light emitting diodes (LED), and servo motors.

Computational memory is arranged into single indivisible units called *bits*. A bit is a single digit that has a value of either zero or one. Bits are grouped together. A group of eight bits is called a *byte*. A group of 16 or 32 bits is called a *word* (depending upon the actual micro-computer being used).

There are two different types of *memory*. These two types are called *read-only memory* (ROM) or *random access memory* (RAM). In general, ROM is used to store permanent programs and data. RAM is used as a scratchpad to store variables generated by an executing program. The MicroStamp11 has 256 bytes of RAM and 32 kilo-bytes of ROM. In contrast, a personal computer (PC) may have several mega-bytes of RAM and several Giga-bytes of ROM held in the PC's hard drive.

The ROM used in the MicroStamp11 is a special type of memory called *electrically erasable programmable read-only memory* or EEPROM. This memory can be electrically erased and rewritten through a special procedure. EEPROM is non-volatile, which means that the stored data remains in memory even if power is removed the device. Since we use ROM to store the MicroStamp11's program, this means that the next time you power up the MicroStamp11, the previously stored program will be sitting there waiting to execute. In contrast, the RAM in the MicroStamp11 is volatile which means that the stored data is lost when power is removed from it.

A micro-controller such as the MicroStamp11 is a micro-computer that has been specially designed to speed up its access to its I/O devices. Most micro-computers access their I/O devices over the CPU bus. Since the CPU bus is also used by other computer subsystems, a micro-computer is somewhat limited in its ability to respond quickly to I/O events. In order to speed up the response to external physical events, a micro-controller modifies the CPU bus so that peripheral devices are directly mapped into the computer's RAM. In other words, a micro-controller's peripheral ports bypass the CPU bus by mapping the I/O port registers into the system's RAM address space. As a result of the memory-mapped I/O, information from the outside world reaches the computer's memory as soon as those events are sensed by the peripheral device. The

righthand drawing in figure 1 shows the architecture of the micro-controller. Note that this system has a much richer set of I/O devices than the standard micro-computer (shown in the lefthand drawing). Also note that the direct memory access (DMA) subsystem essentially provides a secondary bus that can work concurrently with the CPU bus.

3.2. What is the MicroStamp11? The MicroStamp11 is a micro-controller module. Physically, the MicroStamp11 module is a small 1 by 2 inch printed circuit board (PCB) that is built around the Motorola 68HC11 micro-controller integrated circuit (IC). In addition to containing the micro-controller chip, the module holds a modest amount of circuitry devoted to power and memory management. The lefthand picture in figure 2 depicts a MicroStamp11 that has been plugged into a solderless breadboard. In the center of the module's front side is a large square IC. That IC is the 68HC11 micro-processor, the heart of the module. In addition to this large IC, however, you'll also see a number of smaller IC's.

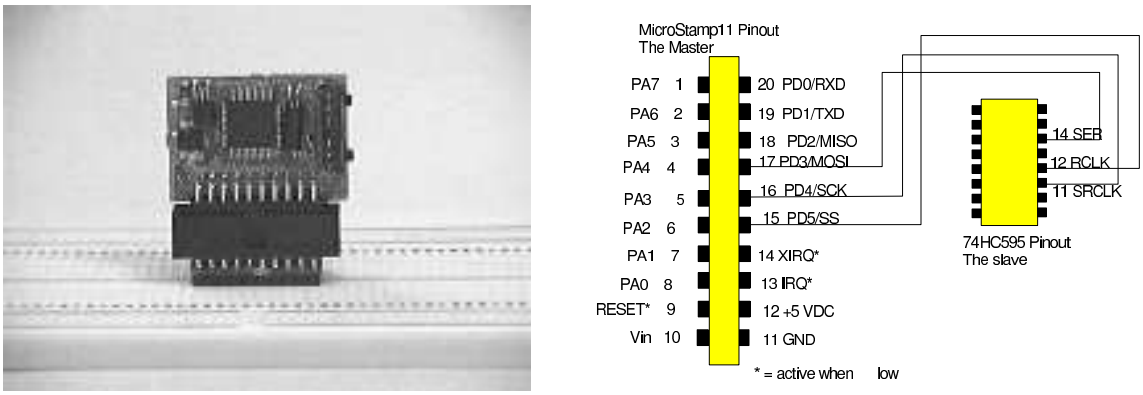

FIGURE 2. MicroStamp11 Module and its pin out

In order to function the IC needs a power source. The power source for most IC's is a direct-current (DC) voltage of either 5 or 3.3 volts. The MicroStamp11 requires a 5 volt voltage source. If we were to apply a voltage in excess of this specified voltage level, then the IC would probably be destroyed. To prevent this from happening, the MicroStamp11 module has a *voltage regulator* that is capable of stepping down DC voltages to the required 5 volt level. The voltage regulator is one of the smaller components on the upper lefthand side of the MicroStamp11 module.

Micro-controllers are *digital synchronous* devices. By *synchronous* we mean that all instructions are executed in synchrony with a hardware clock. The speed of the micro-controller's hardware clock is set by an external *crystal*. The crystal is the silver cylinder on the righthand side of the micro-processor IC.

Finally, this particular micro-processor has a very limited amount of internal memory. For most of the programs you'll be writing, you'll need more memory than the chip possesses. Additional memory will be found on the back side of the module. The large IC on the back of the module is a 32 kilo-byte memory module. The other chips on the backside of the module help interface this memory chip to the 68HC11.

Finally, you'll find two sliding switches on the module. These switches are used to control the operational *mode* of the device. For our purposes there are two operational modes of interest: the *boot* mode and *run* mode. MicroStamp11 programs are written and compiled on a personal computer (PC) and then downloaded into the MicroStamp11 module over the PC's serial port using a special *loader* program. The downloaded

program will only be stored in EEPROM, however, if the module is in *boot mode*. Once a program has been stored in memory, the application will begin running as soon as you switch the module into *run*-mode. You manually switch between boot and run mode using the switches shown in figure 2.

The MicroStamp11 module accesses the outside world through external pins. The righthand drawing in figure 2 shows the pin out for the module. This is a top down view of the device's pin out. Each pin on the device is labelled with one or two names. The precise function of these pins will be explained as needed in the labs.

In these labs, we've installed the MicroStamp11 into a *docking module*. The docking module is another PCB that has a special socket that accepts the MicroStamp11. The docking module also has a *serial interface* built onto it that can be used to connect the MicroStamp11 to a PC. It is through this serial interface that programs are downloaded into the module. In addition to the serial interface, the docking module has a ribbon cable that connects to the 20 pins coming out of the module. The ribbon cable terminates in a connector that can be plugged into a solderless breadboard, thereby allowing you to interface breadboarded circuits to the MicroStamp11 module. The docking module has a couple of light emitting diodes that provide visual indicators for "power" and module activity. Finally, the module has a "reset" button that allows you to restart a program that has been stored in the MicroStamp11.

3.3. How is the MicroStamp11 powered? Since the MicroStamp11 is an electronic device, it must be supplied with power before it can be used. The power is delivered by connected a 5 volt independent voltage source over two pins on the device. This independent source can either be a battery or a power supply on your lab bench. The positive terminal of the independent source is connected to pin 10 (Vin) and the negative terminal of the independent source is connected to pin 11 (GND).

In general, we don't connect these pins directly to the battery or power supply terminals. The independent voltage source is used to power all of the IC's in a circuit and what you will need to do is connect the voltage source to the breadboard in such a way that the breadboard distributes the 5-volt and GND lines in a systematic way over the board. In this way, various parts of your circuit can easily access the ground and positive terminals of the power source.

The breadboard is built in a way that facilitates the distribution of power throughout the circuit. At the very top of the breadboard you'll find a number of terminals that are not electrically connected to the breadboard. The positive terminal of your power supply is plugged into the red terminal through a red patch cable. The negative terminal of your power supply is plugged into the black terminal through a black patch cable. At the top of the breadboard you'll see two horizontal lines of holes. We call these lines the *power rails* of the breadboard. The red (positive) terminal can be connected to one of these lines of holes using a small wire. This action turns one of the power rails into a positive 5-volt power bus for the circuit. The black (ground) terminal is connected to the other line of holes using another wire, thereby creating a ground power bus for the circuit.

In addition to the power rails running along the top of the board, you'll see four double columns of holes running vertically. These double columns will also become 5 volt and ground power buses by connecting them to the top two power buses. These connections allow you to distribute the 5 volt and ground connection over the entire breadboard so that circuit components can directly access the 5 volt and ground points of your circuit without stringing long wires across the breadboard. In particular, to power the MicroStamp11 module you will need to connect your 5 volt power bus to Pin 10 and your ground power bus to Pin 11. Figure 3 shows the breadboard with the power rail connections. This figure also shows how you might connect the MicroStamp11 module to the power rails as well as some other connections, thereby illustrating the use of

these power lines. In this lab you only need to worry about the power connections to the MicroStamp11. In later labs, you will build other circuits that connect to the MicroStamp11.

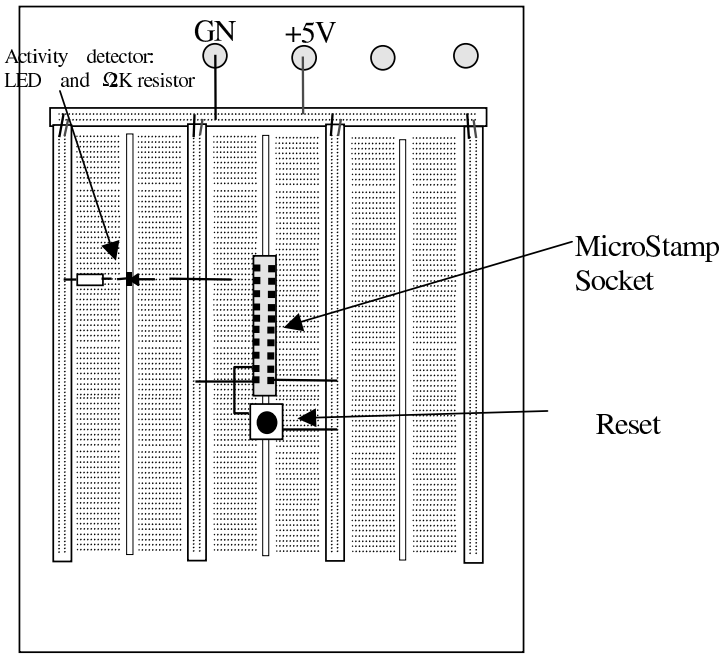

FIGURE 3. Getting Power to the MicroStamp11

If the power connections are made correctly to the MicroStamp11, then the *power* LED on the docking module will be lit. Note that it is extremely important that the voltages be applied to the correct pins. Modules such as the MicroStamp11 are easily destroyed if the wrong voltages are applied to the wrong pins. As these modules are somewhat expensive ($100 dollars apiece), it is crucial that you make these connections with some care.

3.4. How does one communicate with the MicroStamp11? You will need to load a program into the μStamp11's EEPROM before it can do anything. To load such a program, raises an obvious question; how do you communicate with the μStamp11? Unlike your personal computer, the μStamp11 does not have a keyboard or terminal associated with it. The μStamp11 is an embedded system that is designed to talk to other electronic circuits. It wasn't designed to talk directly to a human user. If you wish to talk to the μStamp11, it must therefore be done through an intermediary. That intermediary is your personal computer.

Your personal computer communicates with the μStamp11 through its serial port. The human user communicates with the serial port through a *terminal program* such as `HYPERTERM`. You can identify the serial port on the back of your PC by its distinctive 9-pin D-shaped connector (a so-called DB9 connector). There should be a null modem cable connected to this connector. The other end of that cable will be connected to the μStamp11's docking module. The connection between the μStamp11, the breadboard, and the personal computer is shown in figure 4. In your lab kit there should be a 20-pin socket that you can plug into the breadboard. The ribbon cable of the docking module plugs into this socket. The ribbon cable maps the pins

of the MicroStamp11 directly to the pins on the 20-pin socket, so that the top-down pin out shown in figure 2 corresponds to the top down view of the socket.

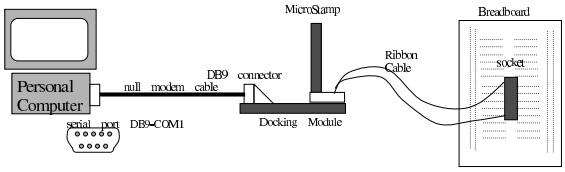

FIGURE 4. Communication connections for μStamp11

3.5. What are kernel functions? An *operating system* (OS) is a set of functions or programs that coordinate a user program's access to the computer's resources (i.e. memory and CPU). Large operating systems such as UNIX and Windows are probably familiar to most students. The MicroStamp11 is also a computer but it is so simple that no OS is hardwired into it. In particular, every program you write for the MicroStamp11 will need to include certain functions that can be thought of as forming a primitive OS for the device. These functions are called the MicroStamp11's *kernel functions*.

We've written a set of kernel functions that can be used by your program. These functions are contained in `kernel.c`, a file that can be downloaded off the course's website. A partial list of the functions that you might need for this lab are provided below. A more complete list will appear in some of the later labs.

For the program you'll be asked to write in this lab you will need functions that allow the MicroStamp11 to communicate with the PC over a serial link. These communication kernel functions are:

- `void init(void);`
  Description: This function initializes the global variables within the MicroStamp11 kernel functions. It must be the first function called by any MicroStamp11 program using the kernel.
  Usage: `init();`
- `void OutChar(char data)`
  Description: This function writes a single byte `data` to the MicroStamp11's asynchronous serial interface (SCI). This function is used to write single characters to a terminal program running on a PC connected to the MicroStamp11 over the SCI port. Some special character macros are `CR` (carriage return), `LF` (line feed), `SP` (space), `BS` (backspace), `DEL` (delete), `ESC` (escape). These special characters can be used for advanced control of the terminal's output.
  Usage: The following statement
  ```
  OutChar('n');
  ```
  outputs the ASCII string for the letter 'n' to the serial port.
- `void OutString(char *pt)`
  Description: This function writes a character string defined by the character pointer `pt` to the MicroStamp11's asynchronous serial interface (SCI). This function is used to write a string of characters to a terminal program running on a PC connected to the MicroStamp11 over the SCI port.
  Usage: The following statement
  ```
  OutString("Hello World");
  ```
  writes the string `Hello World` to the terminal window.

- `void OutUDec(unsigned short number)`
  Description: This function translates an unsigned short integer `number` into an ASCII string and then sends that string to the MicroStamp11's asynchronous serial interface (SCI).
  Usage: The following statements

```
i = 5;
OutUDec(i);
```

  would write the integer 5 in the terminal window.
- `char InChar(void)`
  Description: This function waits for the asynchronous serial port to receive a single character from the terminal program and returns the received character byte.
  Usage: The following statement

```
char i;
i=InChar();
```

  reads a single character from the terminal window and assigns that character to the variable `i`.
- `void InString(char *string, unsigned int max)`
  Description: This function waits for the asynchronous serial port to receive data from the terminal program. The received string is then stored in the character string with pointer `string` having a maximum of `max` entries.
  Usage: The following statements

```
char my_string[10];
InString(my_string,10);
```

  declare a character array `my_string` consisting of 10 characters, waits for the SCI port to receive data, and then fills the declared string `my_string` with the received characters.
- `unsigned short InUDec(void)`
  Description This function waits for the asynchronous serial port to receive ASCII bytes representing unsigned integers from the terminal program. The received bytes are then translated from their ASCII format to an unsigned integer and the resulting number is returned by the function.
  Usage: The following statements

```
unsigned short i;
i=InUDec();
```

  wait for the SCI system to receive data, transforms the received bytes into a short unsigned integer and then stores the received number in the variable `i`.

3.6. How does one program the MicroStamp11? Programs for the MicroStamp11 are written and compiled on a personal computer (PC) and then downloaded into the MicroStamp11 through the PC's serial port. The programs are C-language programs and we're assuming the student is already familiar (from EG111/112) with either C or C++ programming. If you need additional information, a primer on C-language programming for the MicroStamp11 will be found on the project's website.

The lab's PC's have an integrated development environment (IDE) called ICC11 that can be used to write and compile programs for the MicroStamp11. You use the IDE's editor to create C-language source files with the extension *.c. The IDE's compiler/linker then builds an executable file with the extension *.s19. The lab

PC's have a batch program, `pms91.bat`, that is used to download the executable file into to MicroStamp11. You may also use the program MicroLoad to download the *.s19 file into the MicroStamp11.

As an example let's consider creating a simple C-language program. Before starting, you should make sure that the home directory `c:/EE224` is empty. This means that there should be no files in that directory. If any files are in the directory, have the TA delete them. You should have downloaded the files `kernel.c` and `vector.c` from the lab's website. Copy these files into the directory and now you're ready to start.

You begin by opening the ICC11 icon on the PC's desktop and then create a new file `hello.c`. An editor window should open up and you can type in your program. For starters, let's try the following program,

```
#include "kernel.c"

void main(void){

 init();
 while(1){
   OutString("Hello World");
   OutChar(CR);OutChar(LF);
   pause(100);
 }
}
#include"vector.c"
```

The preceding program is more involved than the usual "Hello World" program you may have written for a UNIX operating system (OS) and it is significantly less complex than the program you may have written for the Windows OS.

The preceding program has two include files. The included file `kernel.c` implements a set of *kernel functions* that form something like a primitive OS for the MicroStamp11. The included file `vector.c` defines the absolute address of your program's starting point. The only part of the program you really need to worry about lies within the scope of the `main` function.

The function `main` begins by initializing the kernel functions through the function `init()`. This function must always be the first thing executed by your program.

After initializing the kernel, you will find a `while(1)` control statement. Since `1` is a logical TRUE, the `while` loop's pre-condition is always satisfied. This command, therefore, sets up an infinite loop that repeatedly executes the instructions within the curly brackets. The use of `while(1)` constructions is unusual in traditional programming, but it is often found in embedded programming. Most traditional programs execute a specific list of statements and then terminate after some specified condition has occurred. Embedded programs, on the other hand, are usually intended to execute forever. These programs are often used to monitor a sensor or control some other physical device. In this situation, one doesn't want the program to ever stop. One way to achieve this goal is to use the `while(1)` statement to set up a non-terminating loop. A number of your programs throughout these labs will require this construction.

Within the `while` loop are kernel functions `OutString()`, `pause`, and `OutChar()`. The function `OutString` sends a character string down the serial line to your computer. The function `OutChar` sends a single character

byte to the serial port. You use these functions to write to a terminal program running on your PC. The `pause()` function forces the program to wait for a specified number of clock ticks.

Note that the arguments for the function `OutChar()` are `CR` and `LF`. These two arguments are macros defined in `kernel.c`. These macros associate the logical names `CR` and `LF` with the ASCII character byte generating a "carriage return" and "line feed". So the end effect of the instructions

```
OutChar(CR); OutChar(LF);
```

is to start a new line. In reading the listing, it should be apparent that the first thing the program does is write "Hello World" to the `Hyperterm` window, generate a new line, and then wait for 100 clock ticks. Because these statements are embedded within an infinite `while` loop, the program will repeatedly execute these instructions over and over again. So the output delivered to the PC's screen is an unending column of "Hello World"'s that are generated at a rate specified in the `pause` command.

To compile your program, pull down the Projects menu and select new to create a new project file. Make sure you save it in the `c:\ee224` directory. Once this is done, your project should appear in the far right hand window. Looking at the PROJECT window, you'll find that there are no files in your project. Select the Projects menu and select add file(s) to add the file `hello.c`, that you just created. (DO NOT add the file `kernel.c` or `vector.c` to the project. These files are already includes from `hello.c`) you can then compile your entire project by hitting the build button in the tool bar (looks like a wall of bricks). The status of your build is displayed in the STATUS window (across the bottom). Building your program in this way creates the file `hello.s19` . This file is a packed binary file that you will later download to the MicroStamp11.

3.7. How does one download a program to the MicroStamp11? Before downloading your program you'll need to set the MicroStamp11 in *boot* mode. This is done by setting switches SW2 and SW1 in BOOT position and then pushing the reset button on the docking module. Figure 5 shows these switch positions. You place the device in boot mode by pushing the two switches towards each other. The device is placed in run mode by pushing the two switches away from each other.

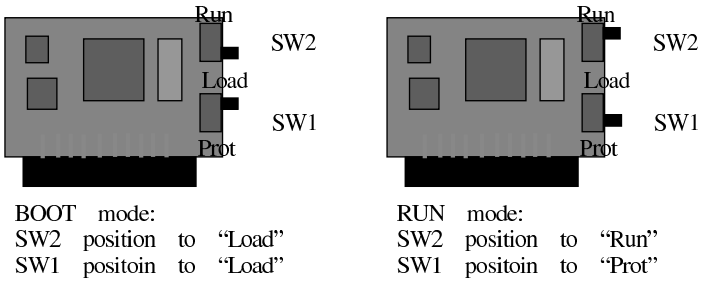

FIGURE 5. Switch Position for MicroStamp11

Once in boot mode, the MicroStamp11 is ready for program download. You can choose one of two methods to download your program. Either download may be used to program the MicroStamp. It is the student's choice.

To use the batch file : You begin downloading your program by executing the `pms91.bat` batch file from an MSDOS command window. So first open a command window and make sure you are in the `EE224` directory. Then type the command

```
pms91 hello
```

The batch file will return asking you to make sure your MicroStamp11 module is in boot mode. This program first downloads the bootloader program `msload9.bin`. The bootloader then directs the installation of `hello.s19` into the MicroStamp11's EEPROM.

Note that the `s19` file created by the compiler has the same name as the PROJECT, not the source file that you added to the project. So if you called your project something other than `hello`, then you `s19` file will be under this other name. The loader `pms91.bat` uses the name of your `s19` file. So if you name your project other than `hello`, you will need to modify the `pms91` command line accordingly.

To use the MicroLoad program : Start the program from the desktop icon. The first time you run the MicroLoad program it is best to verify the settings. To do this click on the options menu. The target should be set to MicroStamp11, Turbo should be selected, and the port is COM1. Click OK. Click the large Load button. Select your file and click open. Now just follow the instructions on the screen.

Once the file has been downloaded you must place the MicroStamp11 back in RUN mode. This is done by sliding SW2 to RUN and SW1 to PROT position. After doing this you open the Hyperterm window. Your Hyperterm program should be configured to connect to serial port COM1 at 38000 baud with 8 data bits and 1 stop bit. You then hit the RESET button and your program should start executing. The Hyperterm window should first type out `Hello World` followed by a new line.

3.8. How is the MicroStamp11 started? Once you have power connected to the μStamp11, how do you start it running? In other words, let's assume that there is a program already loaded into the module's EEPROM. How do we start the μStamp11 executing this program?

What we want is something like the CTRL-ALT-DEL key sequence that a computer running Microsoft Windows might use to reboot the system. If you take a look at the pin-out for the μStamp11 (see figure 2), you should notice that pin 9 has the label RESET. This pin is associated with a hardware interrupt that automatically causes the micro-controller's program counter to jump to the "start" location in EEPROM as soon as the pin is set to zero volts. After the computer jumps to this "start" position it begins executing whatever program was previously stored in EEPROM.

The reset pin on the docking module connects pin 9 (RESET) to ground when it is pushed. So you can start the μStamp11 by simply pressing this reset button.

4. Tasks

4.1. Pre-lab Tasks:

(1) Design a breadboard layout that gets power to the MicroStamp11. Draw a picture of your breadboard layout. This picture should include the connections for the power rails as well as the connections for the MicroStamp11. Do not include any of the button or diode connections shown in figure 3.

(2) Explain (in your own words), the procedure you intend to follow in programming the MicroStamp11.
(3) Write a C-language calculator program for the MicroStamp11. Your program should read two unsigned integers from the Hyperterminal, adds these integers together, and then write the answer to the Hyperterminal. After doing this the program should wait until a new addition problem is entered from the Hyperterminal. Your pre-lab writeup should have a listing of this program as well as an explanation of how the program works.

4.2. In-lab Tasks:

(1) Wire the breadboard to get power to the MicroStamp11. After completing your wiring, have the teach assistant (TA) check the correctness of your circuit. After the TA has verified your breadboard's correctness, you'll be given a MicroStamp11/docking module to connect to your breadboard.
(2) Write, compile, and download your program to the MicroStamp11. Be sure to include the final program listing in your lab book as well as a description of the procedure you used to test the correctness of your program.
(3) Write a description of what happened during your lab.

4.3. Post-Lab Tasks:

(1) Demonstrate the functionality of your program to the TA and have the TA double check the completeness of your lab book. Your lab-book should contain your answers to the pre-lab tasks as well as a listing of the final program that you obtained during the In-lab task.
(2) Your post-lab portion of the lab book should contain the final program listing you ended up with.
(3) You will need to explain the differences between your pre-lab and postlab programs. You will need to explain why these changes were made.

5. What you should have learned

At the end of this lab, the student should be able to download a program to the μStamp11 and then start the program.

6. Grading sheet

LAB 2 - MicroStamp11 Familiarization

8 Pts	**PRELAB TASKS**
1	Picture of Breadboard Layout (power rails and MicroStamp11)
3	Explanation of how to program the MicroStamp11
1	Program Listing
2	Explanation of how program works.
1	TAs verification that the pre-lab tasks were completed correctly.

5 Pts	**IN-LAB TASKS**
3	Description of what happened during the IN-Lab Task.
1	Group number included on report
1	name and date on each page

3 Pts	**POST-LAB TASKS**
1	Final Program Listing
2	Explanation of differences between pre-lab program and final program

2 Pts	**DEMONSTRATION ..... STOP if not checked off**
2	Successful demonstration of completed system.

GO ON to next lab if completed.

Name _____ .

Lab Group Number _____ GRADE _____ out of 18

TA signature _____ Date _____ .

CHAPTER 3

Lights and Switches

1. Objective

The purpose of this lab is to let the student design and build circuits that interface the MicroStamp11 to light emitting diodes (LEDs) and switches (buttons). The student will also learn how to *debounce* a switch input.

2. Parts List

Italicized parts were used in the previous lab.

(1) *wire kit and breadboard*
(2) *MicroStamp11, docking module, and serial cable*
(3) *Laptop with ICC11 compiler and PMS91 loader*
(4) **one 7-segment LED display (LSD3221-11)**
(5) seven 2.2 k-ohm resistors
(6) one 10 k-ohm resistor
(7) one button switch

3. Background

In this lab the student will build circuits that interface the MicroStamp11's *input/output ports* to *light emitting diodes* and *buttons*. In this particular lab, The student will then use some new *kernel functions* to write a program that displays the number of times (modulo 10) a button was pushed.

3.1. What is an input/output port? A micro-controller such as the MicroStamp11 has a number of pins as shown in the pinout of figure 1. The MicroStamp11 communicates with the outside world by changing the logical state of these pins or by reading the logical state of the pins. The *logical state* of a pin is said to be *high* if the voltage of the pin relative to ground is 5 volts. The logical state is low if the voltage on the pin is zero (relative to ground).

The majority of the pins on the MicroStamp11 are arranged into two *ports* that have the logical names `PORTA` and `PORTD`. PORTA has 8 pins associated with it (pins 1 through 8 on figure 1). PORTD has 6 pins associated with (pins 15-20). The other 6 pins on the MicroStamp11 are either used for power (pins 10-12) or they are special input pins that can interrupt the execution of a program (pins 9, 13, and 14). Our current interest is with the I/O pins associated with PORTA/PORTD (pins 1-8 and 15-20).

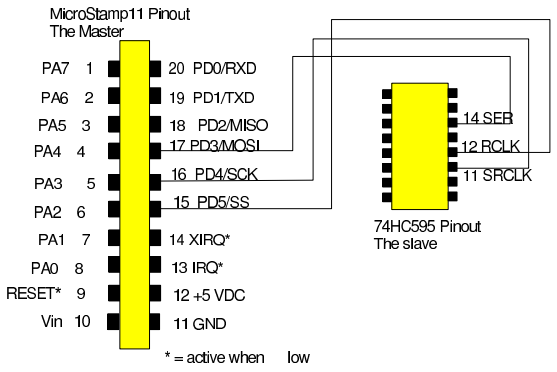

FIGURE 1. MicroStamp11's pinout

The pins on PORTA/PORTD have two distinct types of states. The *logical state*, as mentioned above, refers to the voltage level on the pin (5 volts or zero volts). In addition to this, however, each pin has a *direction state*. In other words, the MicroStamp11 either reads from or writes to a pin; it cannot do both at the same time. This means that each pin on PORTA/PORTD has a directional state that is either IN or OUT. When a pin has the OUT directional state it behaves like an independent voltage source of 0/5 volts. When a pin has the IN directional state it behaves as a high resistance load on the circuit it is connected to.

The MicroStamp11 can control the direction state and logical state of the I/O pins by writing to specific memory locations in its RAM's address space. Remember that Micro-controllers like the MicroStamp11 directly map their I/O pins to hardware registers that are in turn mapped to specific locations in the device's address space. The file `kernel.c` defines the logical names for these hardware registers controlling the port's logical state. These logical names are `PORTA` and `PORTD`. They are 8-bit variables in which each bit is associated with one of the pins on the I/O port.

As mentioned above, these ports can serve as either *inputs* or *outputs*. At a given time a pin can only act as input or output, not both. The directional state of an I/O pin is determined by setting appropriate bits in a *direction register*. Direction registers are hardware registers that can be written to or read from by a program because they are mapped directly into the device's address space. The logical name for PORTD's direction register is `DDRD`. If the ith bit in `DDRD` is set high, then the ith pin on PORTD has its directional state set to output. If the ith bit in `DDRD` is low (0), then the ith pin on PORTD is treated as an input pin.

The pins on PORTA are somewhat special in that not all of the pins' are bi-directional. In fact, only two of the pins (PA3-pin5 and PA7-pin1) can have their direction states changed. This is accomplished by setting the appropriate bits in a hardware control register with the logical name `PACTL` (PortA's control register). The logical names `DDRA7` and `DDRA3` refer to bytes in which only the 7th and 3rd bit are set to one. Through the use of bitwise operators, we can use these logical names to set, clear, or toggle the specified bits in `PACTL`, thereby controlling the directional state of these pins. The other pins in PORTA have their directional states fixed because they are associated with specific input or output interrupt functions. In particular, the directional state of pins 2-4 (PA4-PA6) is always OUTPUT, whereas the directional state of pins 6-8 (PA0-PA2) is always IN.

If an I/O pin's directional state is OUTPUT, then we can change its logical state by simply writing a 1 or 0 to the appropriate bit in the port's register. So let's assume that we wish to set pin PD5 (To "set a pin"

means to make its logical state HIGH. To "clear a pin" means to set its logical state LOW). The following C-language code segment uses bitwise OR operators to accomplish this

```
DDRD |= bit(5);
PORTD |= bit(5);
```

The macro `bit(i)` produces a byte that only has its ith bit set to one. This macro is defined in the kernel (`kernel.c`). The first statement sets the 5th bit in the DDRD register to one, thereby setting the pin's directional state to output. The second statement sets the logical state of the 5th bit to high. If we wish to clear this pin, then we must use a bitwise logical operator on the NOT (complement) of `bit(5)`. This is done in the following code segment

```
DDRD |= bit(5);
PORTD &= ~bit(5);
```

As before the first statement sets PD5's directional state to OUT. The second statement is equivalent to the statement `PORTD = PORTD & (~bit(5))` which simply switches the 5th bit in PORTD to zero.

Setting the logical state for pins in PORTA is similar. Recall that only pins PA3-PA7 can take an OUT directional state. The following code segment sets pin PA7, clears PA5, and toggles PA3.

```
PACTL |= DDRA7;
PORTA |= bit(7);
PORTA &= ~bit(5);
PACTL |= DDRA3;
PORTA ^= bit(3);
```

We used `DDRA7` and `DDRA3` to set the appropriate bits in PORTA's control register so that pins PA7 and PA3 are output pins. Note that we did not need to do this for PA5 since it is always an output pin.

To read the logical state of an I/O pin, the pin's directional state must be set to IN. This is accomplished by setting the appropriate bit in the PORT's control/direction register to zero. Once this is done, we can simply read the bit by testing it to see if that bit is one or zero. The following code segment does this for pin PD5 on PORTD.

```
DDRD &= ~bit(5);
  if((PORTD & bit(5))==0){
    OutString("PD5 = LOW");
  }else{
    OutString("PD5 = HIGH");
  }
```

As before the first statement clears the 5th bit in PORTD's direction register thereby making sure PD5 is an input pin. The logical test computes the logical AND of PORTD and `bit(5)`. This logical AND returns a 0 only if the fifth bit in PORTD is zero. If this occurs, our program writes out that PD5 is LOW. If the

logical AND does not return 0, then we know the 5th bit must have been set and so the program writes out that PD5 is HIGH. A similar type of test can be used to test the status of pin PA2.

```
if((PORTA & bit(2))==0){
  OutString("PA2 = LOW");
}else{
  OutString("PA2 = HIGH");
}
```

Notice that we didn't need to set any bits in PORTA's control register because PA2 is always an input pin. If we had attempted to read pin PA3 or PA7, of course, then we would need to set the appropriate pins (`DDRA3` or `DDRA7`) in control register `PACTL`.

3.2. What is a 7-segment LED?. An LED is a *light emitting diode*. A diode is a two-terminal semi-conductor device that behaves something like an electronic valve. When the diode is forward biased, then the diode can conduct a substantial current and the LED emits lights. The intensity of this light is proportional to the current flowing through the diode. A forward biased diode acts, essentially, like a short circuit. When the diode is reverse biased then only a small leakage current can flow and the diode is dark. The reverse biased diode, therefore, behaves like an open circuit. We use LED's to provide a visual indicator of the Micro-controller's state. On the docking module, for example, you will find two LEDs. One of the LEDs is lit whenever there is power applied to the module. The other LED is connected to one of the I/O ports of the MicroStamp11 and can be used to monitor the activity level of the device.

This project asks you to use the MicroStamp11 to drive a special integrated circuit that contains seven LEDs. The LEDs are arranged in a way that allows you to display numbers and letters. The typical arrangement of LEDs is shown in figure 2. By turning on the appropriate segments, we can display numbers between 0 and 9.

Figure 2 shows the pin assignments for the seven segment LED (LSD3221-11). You'll need to connect these LED's in series with a 2.2 k-ohm resistor in order to limit the current load on the μStamp11 to a safe level. One possible connection that uses pins 1-3 and 15-18 on the μStamp11 is shown in figure 2. This connection uses 4 of the 6 available I/O pins on PORTD and 3 of the 4 output pins on PORTA. This configuration keeps pins 19 and 20 (PORTA) free so you can use the serial interface to your personal computer.

In reviewing figure 2, you should notice that each of the LED's in the package is connected through a 2 kilo-ohm resistor to the output-pin of the MicroStamp11. This resistor is used to limit the current that flows through the diode. Remember that a forward biased LED acts as a short circuit. If we had connected the diode without the resistor, then setting one the output pins low, would have forward biased the diode. But because the diode acts as a short circuit, the current flowing through the diode and hence the MicroStamp11 would be extremely large. It would in fact be large enough to damage the Micro-controller. The resistors shown in figure 2 are in series with the diode, so that when 5 volts is dropped across the diode/resistor series combination, the current flowing through the diode will be limited by the resistor to a finite value that will not damage the MicroStamp11. It is for this reason that each of the LEDs in figure 2 has a current limiting resistor attached to it.

It is extremely important that you keep this in mind when interfacing devices to the output pins of the MicroStamp11. In all cases, you must make sure that the current drawn out of the MicroStamp11 is consistent with its internal ratings. The internal circuitry within the MicroStamp11 can only source around

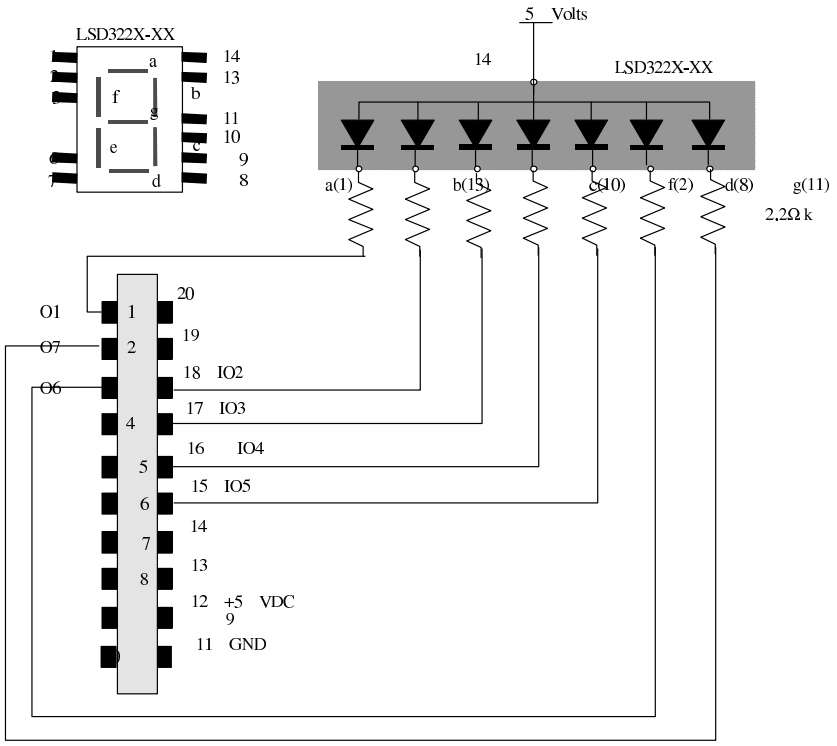

FIGURE 2. Seven segment display

10 mA at most. Anything greater than this will eventually destroy the device. Figure 3 emphasizes the importance of using current limiting resistors by showing the "right" and "wrong" way of connecting an LED to the MicroStamp11.

3.3. How does one connect a button to an I/O port? The preceding section showed how you might connect the μStamp11 to a seven segment LED display. We now examine the question of reading a logical level off of an input pin.

To read the logical state of a pin, you must first make sure that the pin's direction state is set to input. After that you must provide a valid logical voltage level of zero or 5 volts to the input pin. You can then look at the appropriate bit in the port variable (PORTA, PORTD) to have the program read the logical state of the pin.

A valid logical voltage level can be applied to the pin by using a *button*. The schematic symbol for a button is shown in figure 4. The righthand drawing is a picture of this particular button. This particular switch has a single button. When pushed, the switch closes the connection between terminals a and b. When released, the connection between terminals a and b is an open circuit. Sometimes, we can have buttons that close the connection on multiple pairs of terminals. Some of the buttons in the lab may have these multiple terminal

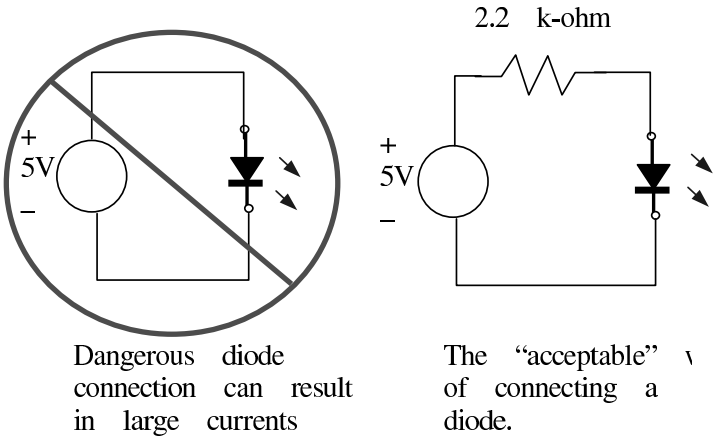

FIGURE 3. The right and wrong way of connecting an LED to the MicroStamp11

pairs. It is recommended that you use the DMM to check which pairs of terminals are electrically connected when the button is pushed.

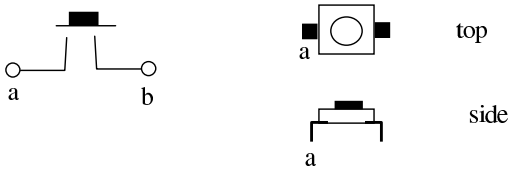

FIGURE 4. Schematic Symbol for button and a picture

Remember that for PORTD, you can set the direction state of the pin by setting the appropriate bits in the `DDRD` register. Only two of the pins on PORTA are bi-directional and their direction states are set by the bits `DDRA7` or `DDRA3` in the hardware register `PACTL`.

So how do we supply a valid logical voltage level to the input pin? One might suppose that the circuit shown on the left-hand side of figure 5 would work. But this circuit isn't a good design.

The reason why this particular circuit won't work well can be explained as follows. First, let's assume that the switch is closed. At this point the input pin will have a specified amount of charge sitting 5 volts away from ground. To keep that charge from draining way, the input pin is, essentially, an open circuit. So if we open the switch, there is still all of this charge sitting on the input pin. We need to give that charge somewhere to go and this is done by connecting a resistor to either the +5 volt supply or to ground. This modified and much better circuit is shown in the right-hand drawing of figure 5.

In this case, when the switch is closed, then we will have a small current going through the 10 k-ohm resistor and the potential between the input pin and GND is 5 volts. When the switch is opened, all of the charge

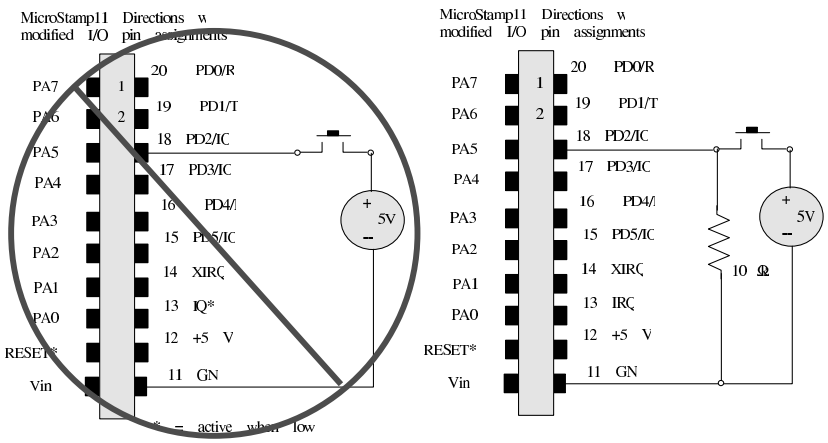

FIGURE 5. Use of pull-down resistor

left on the input pin will drain away to ground through this resistor and the potential at the input pin will be *pulled-down* to ground.

Note that the input pin circuitry is very sensitive to the voltage level applied over the device. Applied voltages in excess of 5 volts will probably destroy the MicroStamp11 in a puff of smoke.

3.4. What are the kernel functions? This particular lab introduces a number of additional kernel functions that you may need to complete the lab. Remember that the kernel functions in `kernel.c` act as a primitive operating system for the MicroStamp11. The kernel functions introduced in this lab either facilitate the setting/clearing of individual I/O pins on PORTA and PORTD, or they provide a way of reading the state of push buttons attached to the MicroStamp11.

A number of the kernel functions read/write to I/O pins of the MicroStamp11. Remember, however, that not all of the pins are bi-directional, so we must be careful in specifying which pins we're writing to and which port they are on. In order to simplify the access of I/O pins, a special set of kernel functions were written that assume certain "addresses" for the pins. In particular we provide 8 I/O pin addresses. The addresses are numbers between 0 and 7. Each address corresponds to a specific hardware pin on a port. The following two tables list these addresses for both input and output directions. Note that I/O pins don't always map to the same hardware pin since some of the PORTA pins have fixed directional states.

INPUT pins			OUTPUT pins		
address	PORT	hardware	address	PORT	hardware
I0	PA3	5	O0	PA3	5
I1	PA7	1	O1	PA7	1
I2	PD2	18	O2	PD2	18
I3	PD3	17	O3	PD3	17
I4	PD4	16	O4	PD4	16
I5	PD5	15	O5	PD5	15
I6	PA0	8	O6	PA5	3
I7	PA1	7	O7	PA6	2

The additional kernel functions introduced in this lab are itemized below. We first show the function's prototype, then we describe its function, and then we provide an example of its usage.

- `void set_pin(int i)`

  Description: This function is used to set pin-address `i` to high. If the pin is not already an output pin, its direction state is set to output.

  Usage: The following instructions set pin O6 to high. Referring back to the preceding table this means that hardware pin 3 will be at 5 volts.

```
i=6;
set_pin(i);
```

- `void clear_pin(int i)`

  Description: This function is used to set pin-address `i` to low (0). If the pin is not already an output pin, its direction state is set to output.

  Usage: The following instructions set pin O6 to low. Referring back to the preceding table, this means that hardware pin 3 will be set to zero volts.

```
i=6;
clear_pin(i);
```

- `void toggle_pin(int i);`

  Description: This function toggles the current state of pin-address `i`.

  Usage: The following instructions toggle pin-address O6 between its high (1) and low (0) state for 100 times.

```
i=6;
for(j=1;j<100:j++) toggle_pin(i);
```

- `int read_pin(int i)`

  Description: This function returns the logical value of pin-address i. If the pin is not already an input pin then its direction is set to input.

  Usage: The following instruction reads the logical value of pin-address I6. From the preceding table, this corresponds to hardware pin 8.

```
int i=6;
int value;
value = read_pin(i);
```

- `void wait_pin(int i)`

  Description: This instruction will wait until the state of pin-address `i` toggles from state 0 to 1. This function may never return if the state of pin `i` never changes.

  Usage: The following code waits until pin I3 becomes high.

```
int i=3;
wait_pin(i);
```

4. Tasks

4.1. Pre-lab Tasks:

(1) Design a button circuit that changes the state of input pin PA0 on the MicroStamp11. Draw a labelled schematic diagram for your circuit and draw a picture showing how you plan to layout the circuit component on your breadboard. Explain how button circuit works.
(2) Design a C-language program that waits for the input on pin `PA0` to transition from low to high, increments an unsigned integer variable upon detecting the transition, and then writes out the value of the variable to the PC's terminal program after each low-to-high transition. Include program listing and explanation of how the program works.
(3) Design the 7-segment LED circuit so that a segment in the display can be turned on or off by using a single call to either the `clear_pin()` or `set_pin()` kernel functions. Draw a labelled schematic for your circuit and draw a picture showing the layout of circuit components on your breadboard. You should arrange your breadboard layout so the LED display is in the upper righthand corner of the breadboard. In your lab book include the schematic diagram and breadboard layout. Also provide an explanation of of how the LED circuit works.
(4) Write a C-language *function* that accepts an integer between 0 and 9 as input and then uses the `clear_pin` and `set_pin` kernel functions to display that digit on your LED display. A partial example of such a function is provided below.

```
void display_digit(int data){
  int i;

  for(i=1;i<8;i++) set_pin(i);
  switch (data){
    case 0:
      clear_pin(1);clear_pin(2);clear_pin(3);
      clear_pin(4);clear_pin(5);clear_pin(6);
      break;
    case 1:
      clear_pin(2);clear_pin(3);
      break;
  }
}
```

This function accepts the integer `data` and sets the appropriate pins on the 7 segment LED display to show either a `0` or `1`. Include a listing of your final function in your lab book and discuss how the program works.

4.2. In-lab Tasks:

(1) Build and verify the button circuit.
- Breadboard the circuit you designed in the pre-lab.
- Have the TA check your breadboard before continuing.

- Program the MicroStamp11 with the first program you wrote in the Pre-lab and verify that the button circuit works correctly.

(2) Build and verify the display circuit.
- Build the 7-segment LED circuit you designed in the pre-lab.
- Have the TA check your breadboard before continuing.
- Use the display function you wrote in the pre-lab to modify your first program so that the system displays the output variable (modulo 10) on the LED display.

(3) Experimentally verify the correctness of your system. Plot the number of button pushes against the number that your device actually recorded.

(4) Most likely your system was not very accurate in counting actual button pushes. Your system will need to *debounce* the switch in order to keep an accurate count of the button pushes. Read the text below and modify your program to *debounce* the switch. Then experimentally verify the correctness of your system. Plot the number of button pushes against the number that your device actually recorded.

What is button debouncing? The first part of the lab discussed the electrical connection to an input pin, but there is still a mechanical problem that needs to be dealt with. When two contacts of the switch hit together, they tend to bounce off of each other a few times before settling down. Most mechanical switches have this bounce problem. You have probably never noticed this bouncing because it only lasts about 1/100 of a second. The problem, however, is that the MicroStamp11 is fast enough to see each of these bounces as a separate hit. So if you are trying to create a system that counts the number of times a button is pressed, you might count individual presses as multiple hits.

The solution to this problem is called *debouncing*. Debouncing can be done in many ways. One may, for instance, use special input circuitry on the switch. It is, however, easier for us to do the debouncing in software. In particular, we simply deactivate the switch for a specified length of time after the first contact is made. This approach to software debouncing is what you can use in the lab. In particular, we've provided a couple of special functions in the lab's `kernel.c` source file that you can use to debounce a switch and to count the number of times a button was pushed. The function that you'll use to test the state of an input pin is `button()`. This function, essentially, does nothing more than wait for the input pin to go high and then it "waits" for 10 milli-seconds before continuing. The implicit assumption, of course, is that any switch bouncing will be over once this 10 milli-second interval is over.

What are the new kernel functions? This particular lab introduces additional kernel functions that you may use to complete the lab. Remember that the kernel functions in `kernel.c` act as a primitive operating system for the MicroStamp11. The kernel functions introduced in this lab control the timing of events. The additional kernel functions introduced in this lab are itemized below. We first show the function's prototype, then we describe its function, and then we provide an example of its usage.

- `void pause(int duration);`
  Description: This function causes program execution to wait for `duration` clock ticks. The actual length of a clock tick can be controlled. In this version of `kernel.c`, we assume a clock tick takes about 2000 clock cycles where each clock cycle is 500 nanoseconds. So this pause is 1 m-sec.
  Usage: The following statement forces program execution to wait for 10 m-sec.
  ```
  pause(10);
  ```
  A more complete example is shown in the following program. This program toggles the logical state of an output pin once every second,

```
void main(void){
 init();
 while(1){
   pause(1000);
   toggle_pin(3);
 }
}
```

- `void button(short pin)`
  Description: This function causes program execution to pause until a button attached to pin-address `i` sets the pin state high. This function automatically performs button debouncing by causing program execution to wait 10 m-sec before returning from the function.
  Usage: The following code waits until a button attached to pin I3 is pressed (sets pin I3 high).
  ```
  button(3);
  ```
- `int count(short pin, short duration)`
  Description: This function waits until the state of pin-address `i` toggles from 0 to 1. The function then counts the number of times the button is pushed (changes state from high to low to high) in `duration` clock ticks. Debouncing is automatically performed on the button pushes and the button count number is returned by the function call when the time `duration` has expired.
  Usage: The following code counts the number of times a button attached to pin I3 is pushed in 1 sec (assuming a 1 m-sec clock tick).
  ```
  value = count (3, 1000);
  ```

4.3. Post-Lab Tasks:

(1) Provide a listing of the final project program along with an explanation of how the program works. Be sure to discuss the differences between your pre-lab program listing and your final program listing.
(2) Assess how well your system worked by discussing the experimental results obtained during In-lab task.
(3) Demonstrate the functionality of your completed system to the TA. The TA will review your pre-lab tasks and the empirical data you gathered to verify the correctness of your design. You may be asked to redo parts of the lab if they are not correct or complete. You may proceed to the next lab after the TA has signed off on this one.

5. What you should have learned

After completing this lab you should know

- how to drive a 7-segment display,
- how to connect a push-button to the μStamp11
- how to write a C-program that reads to and writes from the I/O pins of the μStamp11.
- how to use software to *debounce* a switch

6. Grading sheet

LAB 3 - Lights and Switches

10 Pts	**PRELAB TASKS**
1	Button Circuit schematic diagram and breadboard layout
2	Explanation of how the button circuit works.
2	Button program listing and explanation of how it works.
1	LED circuit schematic diagram and breadboard layout.
2	Explanation of how the LED circuit works
2	Program listing of function that displays a number on the LED display and explanation of how the function works.

8 Pts	**IN-LAB TASKS**
1	Breadboard of button circuit checked by TA.
1	Breadboard of LED display checked by TA.
3	Description of what happened in the lab.
3	Experimental data (before and after debouncing)

5 Pts	**POST-LAB TASKS**
1	Final Program Listing
2	Explanation of differences between pre-lab program and final program
2	Assessment of final program performance.

2 Pts	**DEMONSTRATION ..... STOP if not checked off**
2	Successful demonstration of completed system.

GO ON to next lab if completed.

Name _____ .

Lab Group Number _____ GRADE _____ out of 25

TA signature _____ Date _____ .

CHAPTER 4

Digital to Analog Conversion

1. Objective

The MicroStamp11 is a *digital* device. This means it only works with terms like on/off, true/false, 0/1, etc. The physical world, however, is *analog* and its variables takes values in a continuum. So how do we get a digital device to interact with an analog world?

The purpose of this lab is to build a simple device that converts a 3-bit digital number into an analog voltage between 0 and 5 volts. This type of device is called a digital-to-analog converter or DAC.

2. Parts List

The italicized parts were used in previous labs.

(1) *wire kit and breadboard*
(2) *MicroStamp11, docking module, and serial cable*
(3) *Laptop with ICC11 compiler and PMS91 loader*
(4) *one 7-segment LED display (LSD3221-11)*
(5) *seven 2.2 k-ohm resistors*
(6) **two 10 k-ohm resistor**
(7) two buttons
(8) two R ohm resistors
(9) five $2R$ ohm resistors

3. Background

A rough block diagram of the system you will build in this lab is shown in figure 1. In this figure, the MicroStamp11's output pins 16-18 take output values that are consistent with a desired voltage level. The figure, for example, shows that the three output pins take values of Pin16=1, Pin17=1, and Pin18=0. We let the value of Pin 18 denote the units place in a binary number, Pin 17 denote the 2's place and Pin 16 denote the 4's place. With this assumption, we see that the binary number `110` encoded on these three pins has the associated decimal value of

$$(1 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) = 4 + 2 + 0 = 6$$

This particular binary number is then transformed by the DAC subsystem into a 6 volt voltage. In other words, the DAC subsystem converts the binary digital number encoded on the output pins into the associated analog value.

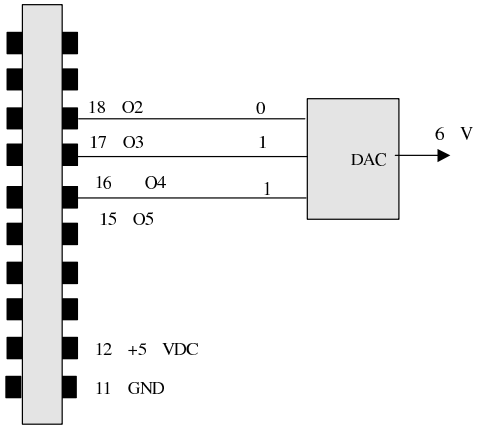

FIGURE 1. The DAC's block diagram

How does the DAC work? There are may ways of building such circuits, this lab looks at a specific approach that adjusts the DAC's output voltage through a *voltage divider*.

If we wish to generate a voltage that lies somewhere between the low ground voltage level (zero) and the "high" voltage level (5), we need to find a way of "dividing" the voltage. A voltage divider is a circuit that does this. Figure 2 shows a simple voltage divider circuit. The voltage V_2 over the second resistor, R_2 will be some fraction of the total voltage over the total voltage generated by the source. Our problem is to determine how the voltage V_2 varies as a function of the two resistances R_1 and R_2. From the course lectures, you should readily recognize that the voltage V_2 will be

$$V_2 = 5\frac{R_2}{R_1 + R_2}$$

Since $R_1 + R_2$ is always greater than R_2, we know that V_2 is somewhere between the high level of 5 volts (V_{pin18}) and zero volts. So we've generated an "analog" signal (a signal that takes a value between two digital values of high and low) using a simple resistive network.

Our challenge, of course, is to generalize this idea so we can transform a digital signal on pins 16, 17, and 18 into an analog voltage that takes one of 8 values between 0 and 5 volts. In other words, we wish to build a resistive network that acts as a 3-bit DAC. What is interesting here is that we can think of each of the three pins in our 3-bit binary number as being an independent voltage source. In other words, a preliminary circuit diagram for our DAC might be something as shown in figure 3. This is a large resistive circuit with three independent sources, whose output is the analog voltage we're looking for. The problem, of course, is what should this large resistive circuit look like. In this lab, this resistive circuit will be an $R2R$ *ladder network*. A schematic drawing of the R2R ladder network is shown in the righthand side of figure 3. This network requires two different values of R and $2R$ ohms. You can choose R to be 2.2 kilo ohms. The independent voltage sources shown in the figure 3 are the MicroStamp11's output pins.

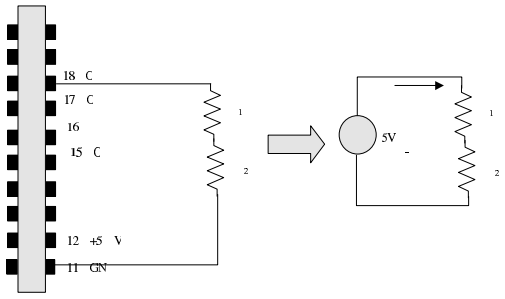

FIGURE 2. Voltage divider: a preliminary DAC design

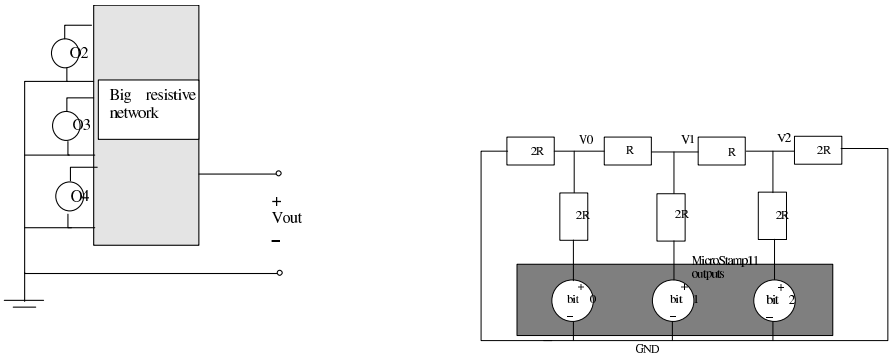

FIGURE 3. Our DAC circuit will have multiple independent sources driving a resistive network. Our 3-bit DAC will use an R2R ladder network to transform these digital voltage levels into an analog voltage.

4. Tasks

4.1. Pre-lab Tasks:

(1) Design a 2 button-circuit that connects to pins PA0 and PA1, respectively. Draw a labelled schematic of this circuit and draw a picture showing the proposed layout of your breadboard. You should be able to reuse the single button you already assembled in the previous lab.
(2) Write a program that increments an unsigned integer variable when the PA0 button is pressed and decrements an unsigned integer variable when the PA1 button is pressed. Your program must display the integer variable (modulo 8) on the 7-segment LED display you designed in the previous lab. Once again you should be able to reuse parts of your previous work.
(3) Design the R2R ladder network that is driven by the 3 output pin PA3, PD0, and PD1. Draw a labelled schematic of your circuit and draw a picture showing how you plan to breadboard this

circuit. It is recommended that you place the ladder network to the right of the MicroStamp11 socket.
(4) Determine the expected output voltages, V_0, V_1, V_2 (refer to the R2R ladder network in figure 3) for each of the 8 possible input voltage combinations that can drive the R2R ladder network. Be sure to show all of your analytical work.
(5) Plot the predicted voltages for V_0, V_1, and V_2 as a function of the requested voltage level on the same graph. From this graph determine which point to use as your output.

4.2. In-lab Tasks:

(1) Construct the two button-circuit you designed in the pre-lab part of this project. Have the TA verify your wiring.
(2) Run the program you designed in the pre-lab and double check to make sure that the system actually outputs the desired voltage level on the 7-segment LED display.
(3) Build the R2R ladder network you designed in the pre-lab. Have the TA double check your circuit before attempting to drive the circuit from the MicroStamp11 (we want to make sure you don't short the micro-controller module).
(4) Modify your original program so the requested voltage value (modulo 8) selected by your button circuit is output to the output pins driving the R2R ladder network. For each of the 8 possible requested voltages, measure the actual voltage level generated by your device. You will be using ports PD0 and PD1 to drive your DAC. These pins are used by the microstamp to communicate to the RS232 interface that talks to the computer. You will need to use the kernal function `disable_sci()` to turn off this default communication in order to use the pins for general purpose outputs. Once you call this function, you will not be able to use any of the functions that communicate with the terminal program.
(5) Provide a description of what happened in the lab.

4.3. Post-Lab Tasks: Your post-lab section of the lab book must include

(1) a final program listing and an explanation of how the program works.
(2) A graph of the measured voltage versus possible inputs states
(3) A graph of the predicted output voltage levels against what you actually measured in the lab.
(4) A written assessment of how well your DAC works.
(5) Demonstrate the functionality of your final system to the TA.

5. What you should have learned

After completing this lab you should

- know how to build a 3-bit digital to analog converter,
- know how to analyze an R2R ladder network,
- and have acquired more experience in programming the MicroStamp11 and breadboarding circuits.

6. Grading sheet

LAB 4 - Digital to Analog Conversion

12 Pts	**PRE-LAB TASKS**
1	2 Button Circuit schematic diagram and breadboard layout.
3	First program listing and explanation of how it works.
1	R2R ladder networks schematic diagram and breadboard layout.
4	R2R network analysis that predicts R2R ladder voltages (V0,V1, and V2) as a function of the 8 possible inputs.
3	Graph of predicted voltages versus possible inputs.

8 Pts	**IN-LAB TASKS**
1	Breadboard of 2-button circuit checked by TA.
3	Explanation of what happened in the lab.
1	Breadboard of R2R ladder network check by TA
3	Experimental data measuring the DAC voltage as a function of the 8 possible inputs.

3 Pts	**POST-LAB TASKS**
2	Final Program Listing and an explanation of how this program works.
3	Graph of measured voltages versus possible inputs
3	Comparison of the predicted and measured voltages. Assessment of how well the DAC works.

2 Pts	**DEMONSTRATION ..... STOP if not checked off**
2	Successful demonstration of completed system.

GO ON to next lab if completed.

Name _____ .

Lab Group Number _____ GRADE _____ out of 25

TA signature _____ Date _____ .

CHAPTER 5

Analog-to-Digital Conversion - Part 1 - Hardware

1. Objective

The previous lab built a 3-bit digital-to-analog converter (DAC). This lab begins looking at what is required to do the inverse operation, *analog-to-digital conversion* or ADC. As it turns out the heart of the ADC will be the 3-bit DAC you built in the previous lab.

This particular project has been split into two parts. The first part (lab 5) focuses on the additional hardware you will need to add to your system. The second part (lab 6) focuses on the software you will need to make your ADC function correctly.

This lab asks you to design, build, and test a circuit that compares the analog voltage, V_r, generated by your DAC against a reference voltage, V and to return a high logical voltage of 5 volts if $V > V_r$ and return zero volts otherwise.

2. Parts List

(1) *wire kit and breadboard*
(2) *MicroStamp11, docking module, and serial cable*
(3) *Laptop with ICC11 compiler and PMS91 loader*
(4) *one 7-segment LED display (LSD3221-11)*
(5) *seven 2.2 k-ohm resistors*
(6) *two 10 k-ohm resistor*
(7) *two buttons*
(8) *two R ohm resistors*
(9) *five 2R ohm resistors*
(10) **one LM660 quad op-amp IC**
(11) two resistors (R) for op-amp buffer
(12) two 1n4007 diodes
(13) one 10 k-ohm trim potentiometer
(14) two additional resistors for diode clamp circuit

3. Background

The heart of the ADC you will build in the next two labs is the DAC you built and designed in the previous lab. The DAC is used to generate a test voltage, V_r, that is compared against the voltage, V, you wish to convert into a 3 bit digit. The *reference voltage* V will be generated by a potentiometer. In this lab you will use diodes and *operational amplifiers* to build circuits that *compare* the two voltages, *buffer* your R2R

ladder network from this new circuitry, and then *clamp* the output voltages generated by your circuit to lie within the 0-5 volt range that can be read by the MicroStamp11.

3.1. What is the reference voltage? The reference voltage is a voltage that you want your ADC to convert into a binary number. You can generate the reference voltage using a potentiometer. In particular, you would tie the top end terminal of the potentiometer to a high voltage level and the bottom end-terminal to ground. The reference voltage is then the voltage between the wiper and ground as shown in figure 1

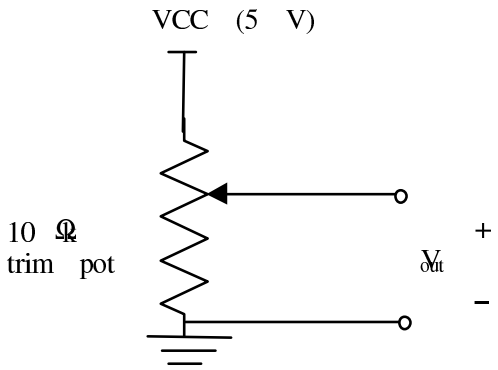

FIGURE 1. Reference voltage source is supplied by a potentiometer

3.2. What is an operational amplifier? A voltage *amplifier* is a special circuit that accepts an input voltage, V_{in} and outputs a voltage, $V_{\text{out}} = AV_{\text{in}}$ that is proportional to the input voltage. The proportionality factor A is called the *gain* of the amplifier. If $A > 1$, then the amplifier actually does *amplify* the input voltage. If $A < 1$, then the amplifier *attenuates* the input voltage.

An *operational amplifier* or op-amp is a special integrated circuit that accepts two input voltages, V^+ and V^-. The op-amp's output is a single voltage (relative to ground), such that

$$V_{\text{out}} = A(V^+ - V^-)$$

and such that A is a very very large number. In other words, an operational amplifier is an integrated circuit that behaves like a high-gain difference amplifier. It amplifies the difference between two input voltages.

The symbol for an operational amplifier is a triangle that has two inputs and a single output. This symbol is shown below in figure 2. The input with a positive sign is called the non-inverting terminal and the input with the negative sign is called the inverting terminal. In addition to the two inputs and single output, the op-amp must have two *supply voltages*. These are shown be the two extra lines coming out of the top and bottom of the triangle in figure 2. The output voltages generated by the op-amp will be confined to lie within these two supply voltages. To function properly the top supply voltage should be at least 7-9 volts and the bottom supply can be anything less than or equal to 0 volts.

As mentioned above the op-amp is an IC that acts as a high-gain difference amplifier. The gain is, in fact, very large, somewhere on the order of $10^5 - 10^7$. In addition to this the op-amp circuitry is designed so that

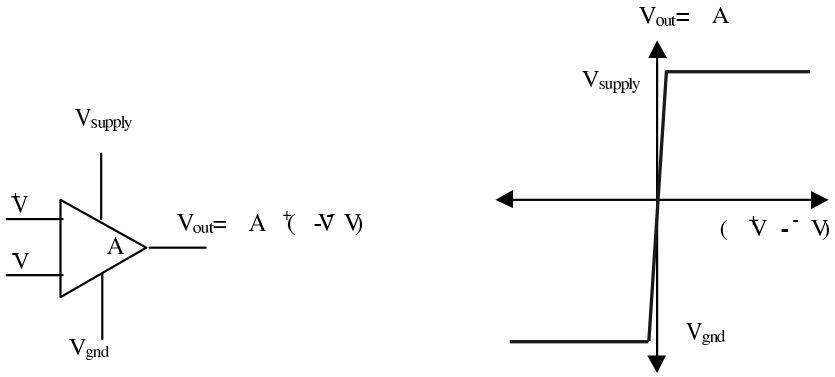

FIGURE 2. Opamp symbol

the device has a very high input resistance and very low output resistance. This means that we can model the op-amp using a dependent voltage controlled voltage source. A *dependent* source is a voltage/current source whose value is a function of some other voltage/current in the circuit. Your textbook should discuss these idealized circuit elements in more detail.

Dependent voltage sources are a very good approximation for the op-amp's behavior. In other words, the op-amp is a circuit that has been *engineered* to be well approximated by an idealized circuit element . This means that we can use op-amp models in a reliable manner to predict the behavior of op-amp circuits with high confidence that our analytical predictions will be duplicated by the physical device. This simple fact makes the op-amp one of the most useful building blocks in analog circuit design.

To operate properly, the op-amp must be supplied a voltage that is larger than the range of differential input voltages. These other voltages are called *supply voltages* and they are denoted as V_{supply} and V_{gnd} in figure 2. In practice there are two types of op-amps. Double side op-amps have supply voltage of $\pm V_s$ volts (where V_s is some positive voltage between 9 and 15 volts). This means that the output of the op-amp can swing between these positive and negative supply voltages. A single sided op-amp has a supply voltage of $+V_s$ volts and ground. This means that the output can only swing from 0 to V_s volts. In our labs we'll be using a single sided op-amp known as the LM660.

The op-amp you've been supplied with in your kit is a standard single-sided quad op-amp (LM660). By single sided, we mean that the supply voltages are $+V_s$ volts and ground (rather than $\pm V_s$ volts). By quad, we mean that there are 4 op-amps on a single chip. The pin-out for the LM660 is shown below in figure 3.

3.3. What is a buffering circuit? Op-amps have a variety of uses. One use is as a so-called *buffer*. A buffer is something that *isolates* or *separates* one circuit from another. In order to explain this more precisely, let's take a closer look at our 3-bit DAC.

The 3-bit DAC constructed in the previous lab produced a digitally controlled voltage, but it turns out that we can't really use this voltage as a *source* to drive other circuits. The problem is that if we were to attach another circuit to our DAC, then we would be changing the $R2R$ ladder network and hence would change the

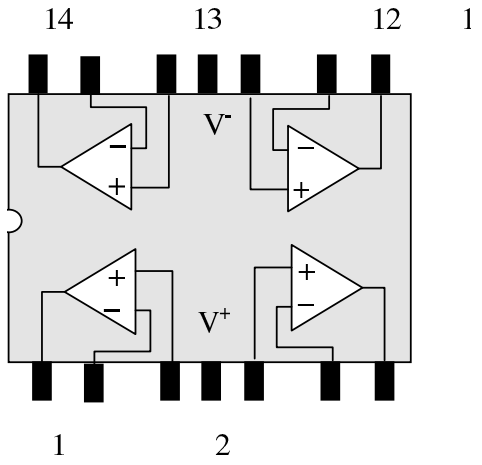

FIGURE 3. Pin out for LMC660 Quad Op-amp

voltage produced by that network. We refer to this phenomenon as *loading*. The problem with our circuit is that it produces a voltage that is not *insensitive* to the load on the circuit.

We now use our preceding discussion about Thevenin circuits to study the *loading problem*. Our preceding discussion asserted that a simpler circuit known as the Thevenin equivalent can always produce the output voltage of any resistive network with independent sources. Figure 4 shows the original DAC network (assuming only one of the output pins is high) and its associated Thevenin equivalent.

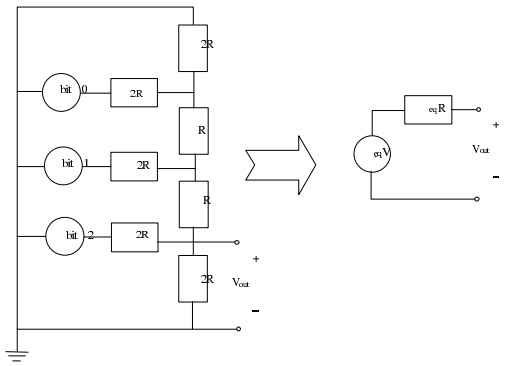

FIGURE 4. Thevenin equivalent of the DAC network

Assuming that the Thevenin equivalent voltage V_{eq} and resistance R_{eq} are known, then we can go ahead an determine the effect that a load resistance has on the circuit's output voltage by a simple application of the

voltage divider law. If we place a load with resistance R_{load} between the DAC's output node and ground, then the loaded Thevenin equivalent circuit would be as shown in figure 5 and the resulting output voltage would be

$$V_{\text{out}} = \frac{R_{\text{load}}}{R_{\text{eq}} + R_{\text{load}}} V_{\text{eq}} \tag{3.1}$$

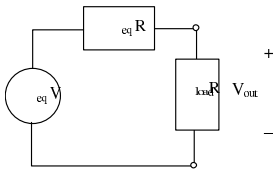

FIGURE 5. Loaded Thevenin equivalent circuit

Remember that V_{eq} is the open circuit voltage generated by the circuit and this is precisely the voltage that we wanted our DAC to generate. Since the resistances R_{eq} and R_{load} are positive, this means that the ratio. $R_{\text{load}}/(R_{\text{load}} + R_{\text{eq}})$ must be less than one. In other words, the output voltage of the loaded DAC will always be less than what we want it to be.

As a numerical example, let's assume that R_{eq} is 1 k-ohm and let's assume that R_{load} equals 8 ohms. This rather low load resistance is common for some devices such as audio speakers. The ratio is now readily seen to be

$$V_{\text{out}} = \frac{R_{\text{load}}}{R_{\text{eq}} + R_{\text{load}}} V_{\text{eq}} = \frac{8}{1000 + 8} V_{\text{eq}} \approx 0.008 V_{\text{eq}}$$

In other words, the output voltage is dramatically less than what we wanted our DAC to produce.

The bottom line in our preceding discussion is that connecting a load to a circuit always effects the output voltage that the circuit will generate. We can minimize the sensitivity of the output voltage to the load resistance by designing the circuit so its Thevenin equivalent resistance, R_{eq}, is large. From equation 3.1, we see that the ratio $R_{\text{load}}/(R_{\text{load}} + R\text{eq})$ can be made arbitrarily small by selecting R_{eq} arbitrarily large.

In order for our DAC to be useful, we'll need to find a way of redesigning the DAC, so that its Thevenin equivalent output resistance is very large. If this is done, then the output voltage generated by the DAC will be insensitive to variations in the load resistance. We can accomplish this feat by simply augmenting our existing ladder network with a *buffering amplifier*.

A buffer is a unity-gain amplifier that has an extremely high input resistance and an extremely low output resistance. This means that the buffer can be modelled as a voltage controlled voltage source that has a gain of one. We connect the buffer to our DAC as shown in figure 6. Note that we've represented the DAC by its Thevenin equivalent circuit. Since the buffer has an infinite input resistance, there is no loading effect so that $V_{\text{in}} = V_{\text{eq}}$. Moreover, we know that the output voltage produced by the buffer must be equal to V_{in} since it has a gain of 1. In other words the voltage produced by the buffer is precisely the voltage generated by the DAC. The output voltage from the buffer is insensitive to the load resistance because the idealized buffer has an output resistance that is essentially zero. By placing a unity gain buffer between the DAC and the load, we have, therefore, solved our loading problem.

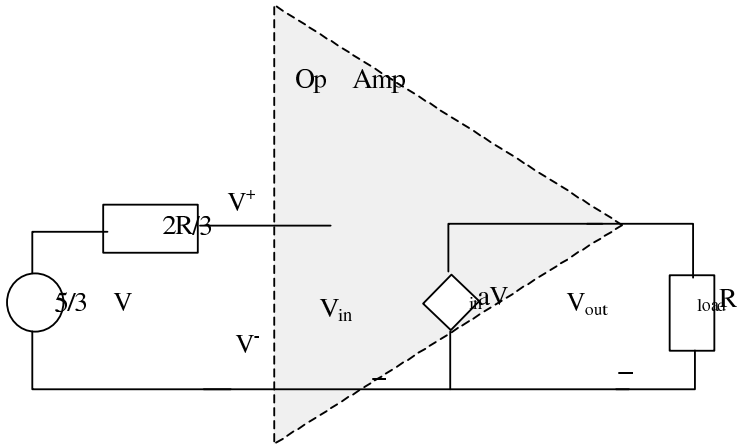

FIGURE 6. Circuit diagram for DAC buffered by unity gain amplifier

Unity gain buffers are idealized circuit elements. While it is possible to buy integrated circuits that serve as these idealized buffers, it is easy to build your own buffer from an operational amplifier. Recall that the op-amp has a large gain, near infinite input resistance and near zero output resistance. In order to turn it into a unity gain buffer, all we need to do is find a way of reducing the overall gain of the op-amp to unity. This can be done using the non-inverting op-amp circuit shown in figure 7. You will be asked to analyze this circuit as part of the pre-lab.

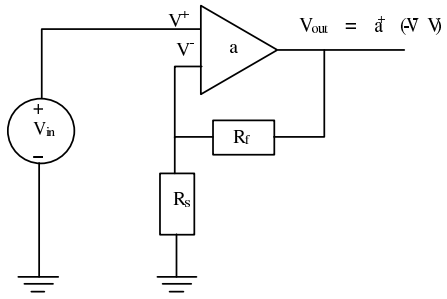

FIGURE 7. Non-inverting op-amp connection

3.4. What is an op-amp feedback circuit? As mentioned elsewhere, an operational amplifier is a differential voltage amplifier circuit that has very large voltage gains ($10^5 - 10^7$), near infinite input resistance, and near zero output resistance. This means that we can model the op-amp as a dependent voltage source controlled by a voltage. The circuit model for the operational amplifier is shown in figure 8.

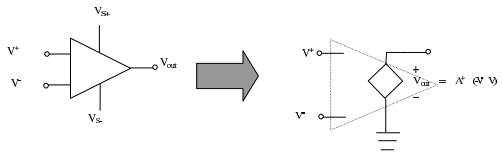

FIGURE 8. Op-Amp Equivalent Circuit

The idealized model shown in figure 8 is a good approximation for the "real-life" op-amp's behavior. In other words, the op-amp is a special circuit that has been specially engineered to behave like its idealized circuit model. We can therefore use op-amp models in a reliable manner to predict the behavior of op-amp circuits, with high confidence that our analytical predictions will hold true in real-life. This means the op-amp is a useful building block in analog circuit design.

The fact that the operational amplifier has an extremely large voltage gain is very useful when we connect the op-amp in a **feedback circuit**. In particular let's consider the **inverting feedback connection** shown in figure 9. This circuit shows the positive terminal of an independent source connected to the inverting terminal of the op-amp through a resistor R_1. A portion of the output voltage, V_{out} is applied to the inverting input terminal through the voltage divider formed by resistors R_1 and R_2. In other words, a portion of V_{out} is **fed back** into the op-amp's input, hence the name "feedback circuit".

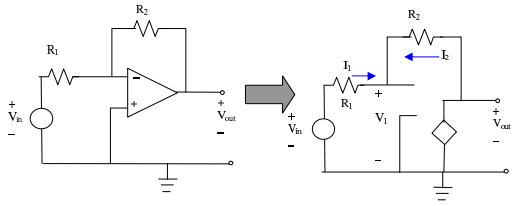

FIGURE 9. Inverting Op-amp Feedback Circuit

To analyze this circuit, we want to derive the relationship between the input voltage V_{in} and the output voltage V_{out}. In particular, we note that

$$V_{\text{out}} = -AV_1$$

where V_1 is the voltage from the negative terminal to ground. Because A is large, on the order of 10^6, we know that if V_{out} is around 1 volt, that V_1 must be 1 micro-volt. This is a tremendously small voltage and it means that the negative terminal of the op-amp is very close to being zero volts (ground). So we may assume that V_1 is equal to zero volts. We sometimes refer to this as the **virtual ground** assumption.

Under the virtual ground assumption, we know that the current I_1 through resistor R_1 must be $I_1 = \frac{V_{\text{in}}}{R_1}$. We also know that the current, I_2, through the feedback resistor, R_2, must be $I_2 = \frac{V_{\text{out}}}{R_2}$. Moreover, we know that the input resistance of the op-amp is extremely large so that the current going into the negative terminal is also nearly zero. By Kirchoff's law, we can therefore conclude that $I_1 = -I_2$. Because the current

going through both resistors is nearly the same, we can immediately see that

$$\frac{V_{\text{out}}}{R_2} = -\frac{V_{\text{in}}}{R_1}$$

which simplies to

$$V_{\text{out}} = -\frac{R_2}{R_1} V_{\text{in}}$$

In other words, the output voltage is proporitional to the input voltage with the proportionality constant (also called the voltage gain) of $-R_2/R_1$.

Note that the above analysis is approximate in the sense that we used the high gain of the op-amp to assume that the negative terminal was at ground (the virtual-ground assumption) and we used the high-input resistance of the op-amp to show that the current through R_1 and R_2 were equal to each other. It is because of these approximations that we find that the voltage gain of the inverting feedback connection is insensitive of the op-amp's gain.

Finally note that the preceding analysis can also be used with slight modifications to derive the voltage gain for the non-inverting op-amp feedback circuit shown in figure 7. As part of the pre-lab, you will need to do this analysis.

3.5. What is a comparator? A comparator is a circuit that accepts two voltages, V_1 and V_2 and outputs zero volts if $V_1 > V_2$ or outputs a positive voltage level if $V_2 > V_1$. Comparators can be built from operational amplifiers.

Remember that the gain of the op-amp is extremely large, somewhere on the order of 10^6. So if the difference between the two input voltages is around 1 volt, would we expect an output voltage of one million volts? Obviously this can't happen. The large gain of the op-amp is only valid over a small range of input voltages. If the output voltage becomes larger than the supply voltages for the op-amp, then the output will saturate or *clip* at that level. This means that uncompensated op-amps output voltage as a function of its input voltage will appear as shown in figure 2.

The implication inherent is that an uncompensated op-amp can be used to compare two voltages. The two inputs to the circuit are analog voltages. But if the input voltage difference is only a few millivolts, then the output will be one of two voltages, pegged at one of the two power supply voltages. In other words, the output will be binary in nature and we can use these binary voltages as a way of testing whether or not one voltage is greater than another.

3.6. What is a clamp circuit? We will later use the output of the comparator circuit as an input to the MicroStamp11. There is, however, a big problem with this approach. The problem is that the voltage levels generated by the comparator circuit are too large. Recall that for the op-amp to work well, the supply voltage must be around 9 volts. This means that the output voltage of your comparator will be either 0 or 9 volts. If we were to apply the 9 volt output to an input pin of the MicroStamp11 we would immediately destroy the MicroStamp11. The MicroStamp11 is only designed to accept voltages that are either zero or 5 volts. Any applied voltages outside of this range will destroy the delicate circuitry within the device. So if we are to use the output of the comparator, we will need some way of reducing the 0-9 volt range produced by the comparator to a 0-5 volt range. This can be done using a *clamp circuit*.

A *clamp circuit* is a special type of circuit that is used to limit or *clamp* the output voltage to a specified range. The clamping action is accomplished through the use of diodes. Remember that a diode is like an

electronic valve. When it is forward biased, it acts like a short circuit and when it is reverse biased it acts like an open circuit. Figure 10 depicts the schematic for a clamp circuit. When V_{in} is greater than 5 volts, then the bottom diode is reverse biased and the top diode is forward biased. As a result, the circuit can be replaced by the equivalent circuit shown in the top righthand drawing. In this case, we see that the input source is connected directly to the 5 volt supply and it is disconnected from the resistor R_2. This means, therefore that V_{out} will equal zero. The resistor R_1 is put in here to limit the current drawn out of voltage source V_{in}. If V_{in} is less than 5 volts, then the bottom diode is forward biased and the top diode is reverse biased. The equivalent resistive network is shown in the bottom righthand schematic diagram. We now see that the source is connected to ground through resistors R_1 and R_2. Once again we choose R_2 to be larger than R_1 so that most of V_{in} drops over the second resistor.

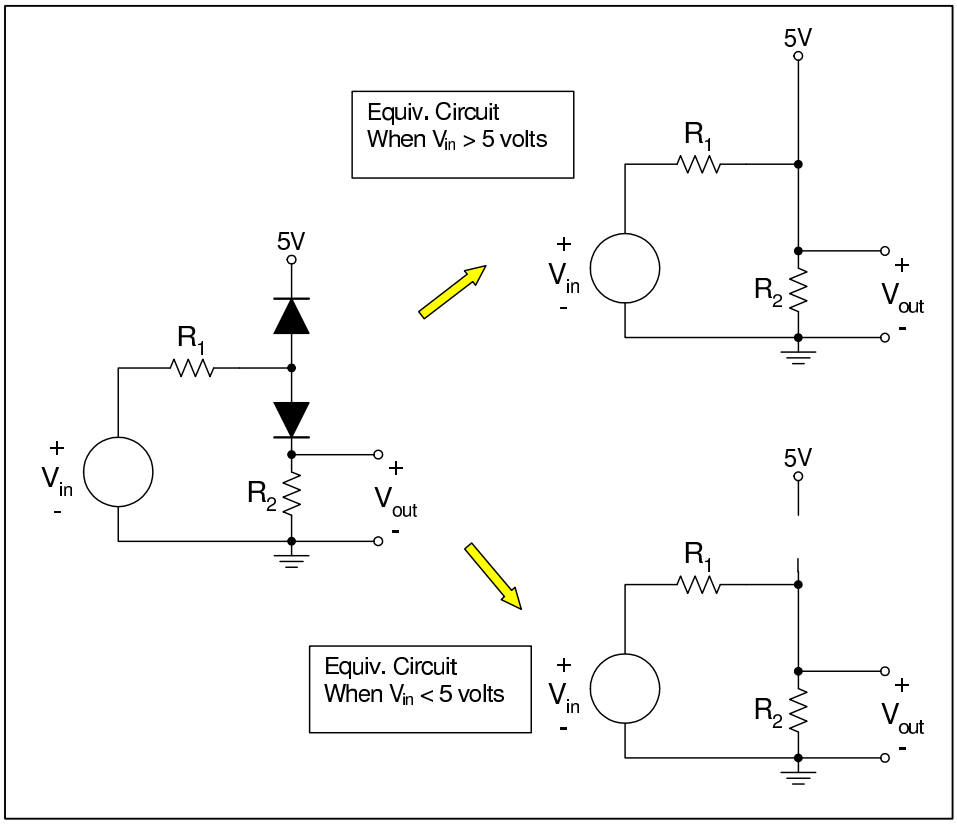

FIGURE 10. Clamp Circuit

The choice of R_1 and R_2 in the above circuit is dictated by two concerns. First we see that R_1 is essentially a current limiting resistor that prevent the source from being unduly stressed if $V_{\text{in}} > 5$. A logical value for this would be around 1 k-ohm. R_2 is then chosen to be much larger than R_1 in order to drop most of the input voltage across this second resistor. Choosing R_2 an order of magnitude greater than R_1 is a realistic choice.

4. Tasks

In this lab you will design, build, and test a circuit that

- buffers the DAC network from the rest of the circuit and amplifies the DAC output by a factor of two,
- compares the amplified DAC output coming out of the buffer against a reference voltage to produce a binary voltage between zero or 9 volts,
- and clamps the output of the comparator to 0 or 5 volts.

A block diagram of the complete circuit that shows the interconnection of the ladder network, buffer, reference voltage source, comparator, and clamp will be found in figure 11. This figure outlines that portion of the block diagram that was constructed in previous labs as well as the new part you are to construct in this lab. Please note that the connection between the clamp circuit's output and the MicroStamp11 is left open. We don't want you to close this connection until you are certain that the clamp voltages lie within the proper range. You will close this connection in the next lab.

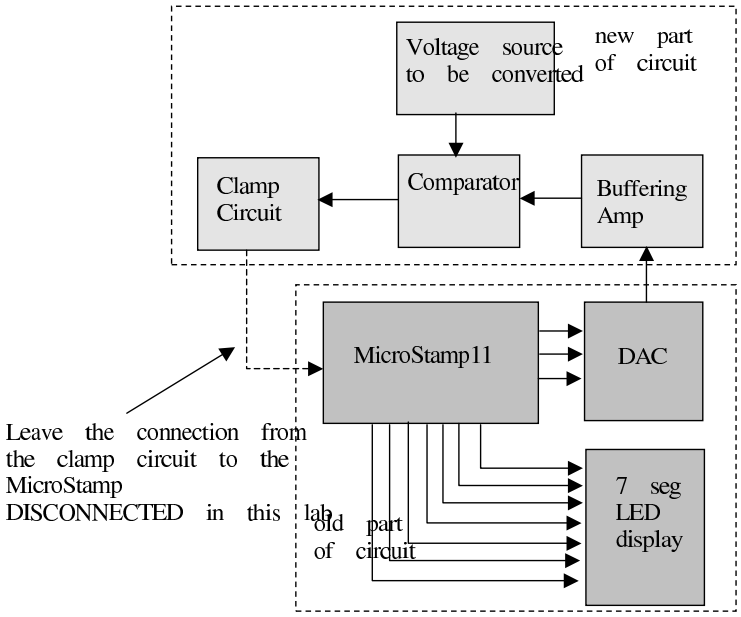

FIGURE 11. Block diagram of completed system

4.1. Pre-lab Tasks:

(1) Design a non-inverting op-amp circuit that has a gain of two. Show all of your analysis work, draw a schematic diagram of the circuit and explain how the circuit works.
(2) Design a comparator circuit that compares a reference voltage generated by a 10 kilo-ohm potentiometer against the input voltage generated by your buffer circuit. Your circuit should generate

a voltage that is either 0 or 9 volts. Draw a schematic diagram of your circuit and explain how it works.
(3) Design a clamp circuit that clamps the comparator's voltage to either zero or five volts. Choose the resistors so that the maximum current drawn from the source is around 1 mA. Show all analysis work and draw a schematic diagram of your circuit.
(4) Draw a picture showing how you plan to breadboard the three circuits and connect them to the DAC circuit.

4.2. In-lab Tasks:

(1) Build the buffer circuit and connect it to the DAC you built earlier. Test your circuit by measuring the voltage level generated by the buffer and the voltage level generated by the DAC for each of 8 possible output voltages.
(2) Build the comparator circuit and connect it to the buffer. Test your circuit by measuring the comparator's output voltage as a function of at least 10 different reference voltage levels between 0 and 9 volts. Repeat this test for each of the 8 possible voltages that can be generated by your DAC.
(3) Build the clamp circuit and connect it to the comparator. Test your circuit by measuring the clamp's output as a function of at least 10 different reference voltage levels. Repeat this test for each of the 8 possible voltages that can be generated by your DAC.
(4) Measure the *threshold* voltage where the comparator/clamp circuit transitions from zero to 5 volts as a function of the reference voltage for each of the 8 possible DAC voltages. Make a table of your data.
(5) Describe what happened in the lab.
(6) Demonstrate your circuit to the TA. The TA will double check the correctness of your results and completeness of your lab book, sign off on the book and let you move on to the next lab. You may be asked to redo some of the tasks if they are not correct or complete.

4.3. Post-Lab Tasks:

(1) Plot the buffer output voltage and DAC voltage level as a function of the requested DAC voltages. Assess how well your circuit works.
(2) Plot the experimental data for the DAC/Buffer/Comparator circuit. Assess circuit's performance.
(3) Plot the experimental data for the DAC/buffer/Comparator/clamp circuit. Assess circuit's performance.

5. What you should have learned

At the end of this lab you should know

- how to design a non-inverting buffer circuit,
- how to design a clamped comparator circuit that generates a binary voltage between 0 and 5 volts,
- how to connect up an operational amplifier IC,
- how to use a potentiometer to generate a reference voltage.

6. Grading sheet

LAB 5 - Analog to Digital Conversion - Hardware

10 Pts	**PRE-LAB TASKS**
3	Design and analysis of non-inverting op-amp buffer and schematic diagram.
2	Schematic diagram of reference voltage generator and comparator.
3	Design and analysis of clamp circuit and schematic diagram
2	Breadboard layout of the combined buffer/comparator/clamp circuit.

10 Pts	**IN-LAB TASKS**
2	Experimental data from DAC/Buffer combination (open loop)
2	Experimental data from DAC/Buffer/Comparator combination (open loop)
2	Experimental data from DAC/Buffer/Comparator/Clamp combination (open loop)
2	Make a table of threshold voltages.
2	Explanation of what happened in the lab.

8 Pts	**POST-LAB TASKS**
2	Graph of experimental data from DAC/Buffer combo. Assessment of circuits performance
3	Graph of experimental data for DAC/Buffer/Comparator combo. Assessment of circuits performance
3	Graph of experimental data for DAC, Buffer, Comparator, Clamp combo. Assessment of circuits performance.

2 Pts	**DEMONSTRATION ..... STOP if not checked off**
2	Successful demonstration of completed system.

GO ON to next lab if completed.

Name _____ .

Lab Group Number _____ GRADE _____ out of 30

TA signature _____ Date _____ .

CHAPTER 6

Analog-to-Digital Conversion - Part 2 - Software

1. Objective

The ADC you will finish building in this lab uses the circuitry of the previous lab to drive a simple binary search algorithm. The purpose of this lab is to have the student write, compile, and test this binary search algorithm and thereby complete the construction of a 3-bit successive approximation ADC.

2. Parts List:

Parts list is the same as lab 6's parts list.

3. Background

The test circuitry from the previous lab provides a TRUE/FALSE declaration that the reference voltage V_r generated by the DAC is greater than the voltage V you wish to convert. We can use this simple TRUE/FALSE response to drive an *algorithm* that systematically determines the 3-bit binary representation of the voltage V. The algorithm we'll use is called a *binary search* that might be seen as a formalization of a simple *guessing game*. ADC's based on this binary search technique are sometimes called successive approximation ADC's.

3.1. What is the Guessing Game? The game is two-person game called "Guess the Number". The first player thinks of an integer within a known range. The second player tries to guess the number. If the guess is incorrect, then the first player tells the second player whether the guess was too high or too low. Eventually, the second player guesses the correct number. The second player's score equals the number of guesses he made. The players then reverse their roles and repeat the game. The winner is the player who gets the correct answer with the fewest guesses.

The key strategy in this game is to generate a clever guess. If, for example, the second player knows the number is between 0 and 100, then a reasonable first guess is 50. This choice evenly splits the range, giving you the maximum amount of information about the next guess. If the first player says the guess is too low, then the second player splits the reduced range and guesses 75. If the player says the guess is too high, then the optimal guess is 25. It can be shown that by splitting the remaining range in half after each guess, it will, at worst, take the second player no more than $\log_2 n$ guesses to find the unknown number where n is the initial range. So if the unknown number lies between 0 and 7, then it can be guessed in no more than $\log_2 8 = 3$ guesses.

3.2. What is a binary search? The "guessing game" is an example of a *binary search* algorithm. To see how we can use it in our situation, let's assume that the potentiometer is the first player. Remember that the potentiometer generates a reference voltage that we want to convert into a binary number. So this reference voltage is the "unknown" number that the second player has to guess. The second player is the MicroStamp11. It needs to generate a guess as to what the voltage should be and the circuitry you built in the previous lab declares whether that guess is too high or too low.

You can formalize the sequence of guesses generated by the second player using a computer algorithm known as a *binary search*. We start with an assumed upper and lower bound on the input voltage, V_{lower} and V_{upper}. We assume that the unknown voltage, V_{in} satisfies the inequalities,

$$V_{\text{lower}} \leq V_{\text{in}} \leq V_{\text{upper}}$$

The MicroStamp11 now generates a guess voltage that lies halfway between the upper and lower unknown voltage. In other words, the MicroStamp11 guesses that

$$V_{\text{guess}} = \frac{V_{\text{upper}} + V_{\text{lower}}}{2}$$

This guess is again compared to V_{in}. If $V_{\text{guess}} > V_{\text{in}}$, then the MicroStamp11 will set $V_{\text{lower}} = V_{\text{guess}}$. In other words, we reset the lower bound of our uncertainty to our previous guess. If $V_{\text{guess}} < V_{\text{in}}$ then the MicroStamp11 resets the upper bound to our previous guess (i.e., $V_{\text{upper}} = V_{\text{guess}}$). After a finite number of guesses, the binary search algorithm terminates within a specified neighborhood of the unknown voltage Vin.

4. Tasks

4.1. Pre-lab Tasks:

(1) Write a C-language MicroStamp11 program that implements the binary search described above. Your program should run within an infinite `while` loop. Within that loop your program should execute a `for` loop that outputs a guess voltage, V_{guess}, then reads input pin PA2 and finally generates a new guess voltage according to the binary search algorithm. You can assume that PA2 is a binary signal declaring whether the guess was too high or too low. The `for` loop should be executed until the algorithm converges to the correct voltage level. Once the `for` loop has been completed, your program should display the guessed value on the 7-segment LED. The structure of your program, therefore, should be structured as outlined below:

```
#include"kernel.c"

void main(void){
  init();
  while(1){
    for(i=0;i<N;i++){
      guess = (upper+lower)/2;
      output_voltage(guess);
      if(read_pin(PA2)==TRUE){
        upper = guess;
      }else{
        lower = guess;
      }
```

```
    }
    display_digit(guess);
  }
} #include "vector.c"
```

The above listing only outlines the basic structure of the algorithm. You will need to modify the preceding listing to obtain a working program. Be sure to include a listing of your preliminary program in the lab-book. Also explain how the program works.

(2) Plot the voltage values that your program will converge to as a function of the reference voltages. *hint: use the threshold values found in the last lab.*
(3) Show a listing of your preliminary algorithm and the plot to the TA before starting the In-lab tasks.

4.2. In-lab Tasks:

(1) Make sure that the output from the circuitry you constructed in the previous lab produces outputs between 0 and 5 volts. Have the TA double check your final breadboard. After this has been done you can close the loop between the clamp circuit and the MicroStamp11 by connecting the clamp circuit's output to pin PA2.
(2) Write, compile, and download your program to the MicroStamp11.
(3) Debug your program and rewrite the program, if needed, until it works as expected.
(4) Record the output value on your LED display for at least 30 different reference voltages. Plot the displayed LED data as a function of the reference voltages.
(5) Your lab book should have a description of what happened during your In-lab task.

4.3. Post-Lab Tasks:

(1) The post-lab section of your notebook should include a listing of the final program you came up with. Explain how the final program works.
(2) Compare your in-lab test data and your pre-lab predictions. Comment on how well these two match. Graph the LED display data versus the actual reference voltage.
(3) Compare your initial pre-lab program and the final program you came up with. Discuss the differences between the two programs and why you needed to make these changes.
(4) Demonstrate the functionality of your system to the TA.

5. What you should have learned

At the conclusion of this lab you should have learned how to

- Write a simple binary search algorithm
- Debug and fix your program
- Understand comparator concepts

6. Grading sheet

LAB 6 - Analog to Digital Conversion - Software

7 Pts	**PRE-LAB TASKS**	
3	Program Listing and explanation of how it works.	
3	Graph of predicted LED outputs as a function of applied reference voltage and explanation of how this graph was obtained. Use the data table of threshold voltages from last lab.	
1	TAs check of the pre-lab program listings completeness.	

6 Pts	**IN-LAB TASKS**	
1	TA check of your final circuit before closing the loop between clamp circuit and MicroStamp11.	
3	Explanation of what happened in the lab.	
2	Experimental data recording the LED display for at least 30 different reference voltage levels.	

10 Pts	**POST-LAB TASKS**	
3	Final Program Listing and an explanation of how this program works.	
2	Discussion of how your final program listing differs from the pre-lab listing and why the modifications were necessary.	
2	Graph of LED display data versus actual reference voltage.	
3	Comparison of the predicted and actual voltage conversions. Assessment of how well the ADC works.	

2 Pts	**DEMONSTRATION ..... STOP if not checked off**	
2	Successful demonstration of completed system.	

GO ON to next lab if completed.

Name _____ .

Lab Group Number _____ GRADE _____ out of 25

TA signature _____ Date _____ .

CHAPTER 7

Digital-to-Analog Conversion Revisited

1. Objective

In one of the previous labs you designed and built a DAC that used 3 output pins and only provided 3-bits of precision. This lab builds a different DAC that requires only a single output pin and provides 6-bits of precision. Your system should read the commanded voltage level from a terminal window on a PC and a 3-bit version of your commanded voltage should be output to the single digit LED display.

2. Parts List

Italicized parts were used in previous lab.

(1) *wire kit and breadboard*
(2) *MicroStamp11, docking module, and serial cable*
(3) *Laptop with ICC11 compiler and PMS91 loader*
(4) *one 7-segment LED display (LSD3221-11)*
(5) *seven 2.2 k-ohm resistors*
(6) *two 10 k-ohm resistor*
(7) *two buttons*
(8) *one LM660 quad op-amp IC*
(9) *two resistors (R) for op-amp buffer*
(10) *two 1n4007 diodes*
(11) *one 10 k-ohm trim potentiometer*
(12) *two additional resistors for diode clamp circuit*
(13) **additional resistor for RC network**
(14) capacitor for RC network.

3. Background

This lab has the student build an improved DAC that uses fewer I/O pins and provides greater precision than the DAC built around the R2R ladder network. These improvements are obtained by thinking of using "time" as a control variable.

The MicroStamp11 is a digital device and it is because of its digital nature that we needed so many I/O pins in the earlier DAC design. Micro-controllers, however, can do more than manipulate digital data. Micro-controller must have internal clocks that precisely orchestrate the various digital computations performed by the device. In other words, the micro-controller can detect and generate timed events with a great deal of precision.

In this lab you will build a DAC that uses a PWM signal to drive a simple circuit consisting of a capacitor and resistor. This circuit is called an *RC circuit*. The RC circuit's *response to the PWM input signal* is a time-varying signal whose average value is proportional to the duty cycle of the PWM signal. You should be studying capacitive circuits in your lecture course right now.

4. Pulse Width Modulation

4.1. What is Pulse Width Modulation? When we look at something like a circuit, we characterize its behavior by determining the node voltages and branch currents. But if these voltages and currents are time-varying, then we can no longer use a single "number" to characterize the circuit's behavior, we must use a "function" that we'll refer to as a "signal". We define a *signal* as a function that maps *time* onto some real number. So, for instance, the voltage function v is a rule that associates a time t with an actual voltage measurement $v(t)$. The value that v takes at a time t is denoted as $v(t)$. Since both time and voltage are real numbers, we can denote the voltage function using the notation $v : \Re \rightarrow \Re$. This notation says that v maps the real line back into itself.

We say a signal, v is *periodic* if there exists a positive time T such that $v(t) = v(t+T)$ for all $t \in \Re$. In other words, at any moment, t, in time, the value of v ($v(t)$) will always be repeated some specified time interval T in the future. We refer to T as the *period* of the signal. If T is the smallest positive number such that $v(t) = v(t+T)$ (for all $t \in \Re$), then we refer to T as the signal's *fundamental period*. If T is the period of a periodic signal v, we often refer to v as being T-periodic.

A *pulse-width modulated* signal is a T-periodic signal, v, if there exists $0 < T_1 < T$ such that

$$v(t) = \begin{cases} 1 & 0 \leq t < T_1 \\ 0 & T_1 \leq t < T \end{cases} \tag{4.1}$$

for $t \in [0, T]$. We refer to the ratio T_1/T as the *duty cycle* of the signal. We usually represent the duty cycle as a percentage. Equation 4.1 defines the values that v takes over a single fundamental period, T. Since v is T-periodic, we know that the pattern characterized in equation 4.1 will repeat itself at regular intervals of duration T. Figure 1 shows a pulse-width modulated signal whose duty cycle is 25%.

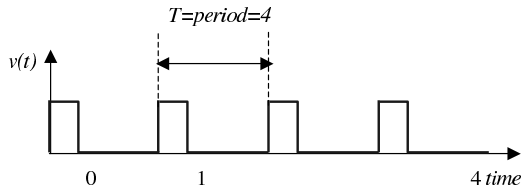

FIGURE 1. Pulse Width Modulated Signal

This lab asks you to modify one of the output compare event interrupt handlers in the kernel so that pin PA4 generates a PWM signal whose duty cycle can be set from within the `main` program. In order to complete this lab you need to learn what an output compare event is and what an interrupt handler is.

4.2. What is an Interrupt Handler? Let's consider a program that the MicroStamp11 is executing. A program is a list of instructions that the micro-controller executes in a sequential manner. A *hardware event* is something special that happens in the micro-controller's hardware. An example of such an event is the RESET that occurs when pin 9 on the MicroStamp11 is set to ground.

When an event occurs the micro-controller generates a *hardware interrupt*. The interrupt forces the micro-controller's program counter to jump to a specific address in program memory. This special memory address is called the *interrupt vector*. At this memory location we install a special function known as an *interrupt service routine* (ISR) which is also known as an *interrupt handler*. So upon generating a hardware interrupt, program execution jumps to the interrupt handler and executes the code in that handler. When the handler is done, then program control returns the micro-controller to the original program it was executing. So a hardware interrupt allows a micro-controller to interrupt an existing program and react to some external hardware event. This type of flow control is illustrated in figure 2.

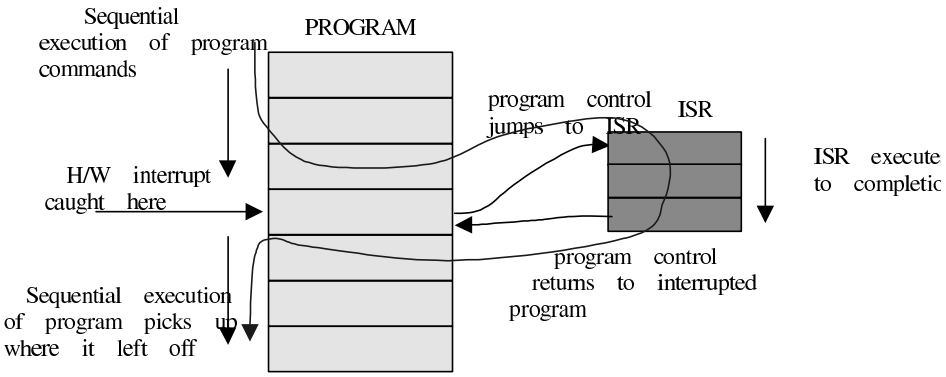

FIGURE 2. Control flow in the presence of a hardware interrupt

Interrupt service routines are used to execute extremely important pieces of code in response to critical events. In an automotive system, for example, we might have a micro-controller supervising the operation of various devices on the car's dashboard. Usually, this micro-controller would be concerned with making sure the electronic gauges on the dash are displaying the correct information. If, however, the car is in a collision, then these display functions are much less important than, say, the deployment of an airbag. So, when this "collision" event occurred, we would want the micro-controller to interrupt its usual tasks in order to execute the "deploy airbag" task.

In our case, of course, our hardware events are not as dramatic as "deploy airbag". What type of hardware events are of interest to the μStamp11? You have already used some of these events. We've already mentioned the RESET interrupt that is tied to pin 9. When this pin goes low, it generates a hardware interrupt that forces program execution to jump to the interrupt vector `0xFFFE`. This memory location is the default starting address defined in `vector.c`. So when pin 9 goes low, the micro-controller stops everything and begins re-executing the program.

A table of some of the other hardware interrupts along with their interrupt vectors are found in the following table.

interrupt vector	interrupt source
0xFFFE	Power on, Reset
0xFFFA	Watchdog timer failure (COP)
0xFFF0	real time interrupt (RTIF)
0xFFEE	timer input capture 1, (IC1)
0xFFEC	timer input capture 2, (IC2)
0xFFEA	timer input capture 3, (IC3)
0xFFE8	timer output compare 1, (OC1)
0xFFE6	timer output compare 2, (OC2)
0xFFE4	timer output compare 3, (OC3)
0xFFE2	timer output compare 4, (OC4)
0xFFE0	timer output compare 5, (OC5)
0xFFDE	timer overflow (TOF)
0xFFD8	SPI serial transfer complete (SPIF)
0xFFD6	SCI events (RDRF, TDRE)
0xFFF8	illegal opcode trap
0xFFF6	software interrupt (SWI)

This table is only a partial list of the hardware events that can be used to interrupt program execution. In using the μStamp11 to generate a PWM signal, we'll only need to focus on one of the hardware events; the output compare event (`OC4`). This is the same hardware event that is used by the kernel function `pause`.

4.3. What is an Output Compare Event? The output compare event is a hardware event tied to the micro-controller's *real-time clock*. The real-time clock on the μStamp11 is a hardware subsystem within the μStamp11 that provides a very precise and steady time reference. In particular, the clock increments a hardware register whose logical name is `TCNT`. `TCNT` is a 16-bit unsigned counter. It is incremented at a rate that is determined by two bits in a control register `TMSK2`.

The rate at which TCNT is incremented is determined by the bits `PR1` (`0x02`) and `PR0` (`0x01`) in `TMSK2`. The following commands

```
TMSK2 &= ~0x01;
TMSK2 &= ~0x02;
```

clear the `PR0` and `PR1` bits in `TMSK2` and causes `TCNT` to be incremented once every 500 nanoseconds. Other update rates are shown in the following table under the assumption that the μStamp11's clock is running at an 9.83 Mhz rate.

PR1	PR0	TCNT clock rate
0	0	407 nsec
0	1	1.628 μ-sec
1	0	3.255 μ-sec
1	1	6.511 μ-sec

The counter `TCNT` cannot be reset or stopped by the user. So to generate timing events, we compare the value in `TCNT` against another number that is held in an *output compare register*. When the value in `TCNT` matches the number in the output compare register, we trigger an *output-compare event*. The Motorola 68HC11 micro-controller has 5 different output compare registers so it is possible to trigger 5 different

output compare events . These registers have the logical names `TOC1`, `TOC2`, `TOC3`, `TOC4`, and `TOC5`. The exact addresses of these registers will be found in the include file `hc11.h`.

Figure 3 shows the three registers used by the output compare interrupt. These three registers are `TMSK1`, `TFLG1`, and `TCTL1`. The register `TMSK1` is a control register that is used to "arm" the interrupt. The register `TFLG1` is a status register that can be used to "acknowledge" the servicing of a caught interrupt. Register `TCTL1` is used to modify the way in which the output compare interrupt interacts with the micro-controller's output pins.

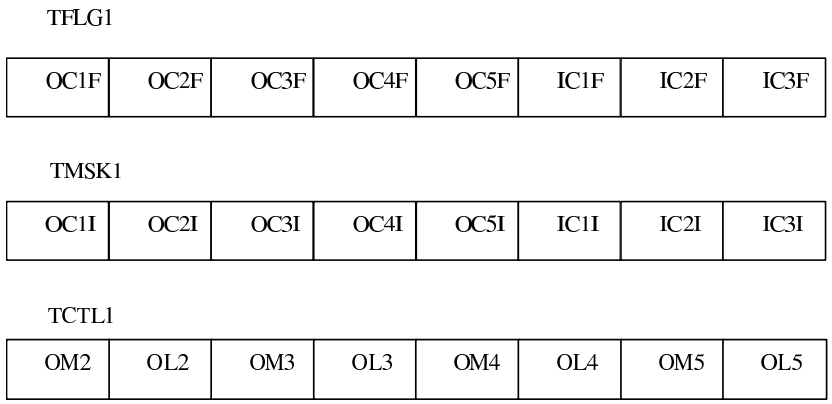

FIGURE 3. Hardware Registers used by Output compare interrupt

Output compare events are generated in the micro-controller's hardware. This event will result in a hardware interrupt (also called an output compare interrupt) being generated if:

- the interrupt is *armed*.
  Arming an interrupt means to enable the source of the interrupt. We *arm* an interrupt by setting the appropriate bit in a hardware register. Output compare interrupts are armed by setting bits in the register `TMSK1`. The output compare 4 (OC4) interrupt, for example, is enabled by setting bit `OC4I` in register `TMSK1` to 1 (see figure 3).
- the interrupt is *enabled*
  Enabling the interrupt means that the software pays attention to the interrupt. We *enable* all interrupts by clearing the I bit in the condition code register of the micro-controller. This bit is usually cleared in the `init()` function using the assembly command `cli`.
- and any previous interrupts are *acknowledged*.
  Acknowledging an interrupt means that we tell the system that a previously received interrupt has been serviced. We *acknowledge* an interrupt by setting the appropriate bit in the status register `TFLG1`. When an interrupt is caught by the software, an appropriate bit in `TFLG1` is cleared (set to zero). This status register allows us to explicitly determine the source of the interrupt. If, however, we want our system to catch the next interrupt, we must explicitly acknowledge that the interrupt was caught and this acknowledgement is performed by setting the appropriate bit in `TFLG1` to one. So, for example, we acknowledge the `OC4` interrupt by setting bit `OC4F` in `TFLG1` to 1 (see figure 3).

When the interrupt is caught by the software, program execution jumps to the interrupt vector and begins executing the instructions at that location. The user can *install* a special interrupt service routine (ISR) at this interrupt vector through the use of special compiler directives. We'll discuss how you go about writing an ISR in the next section.

The output compare event can also be used to effect specific output pins. The `TCTL1` register determines what effect the OC2, OC3, OC4, and OC5 events will have on the output pin. The layout for the TCTL1 register is given in figure 3. The logical names for the bits in this register are OMx and OLx where x takes values between 2 and 5. The following table itemizes the effect that the bits in `TCTL1` have on the output pins.

OMx	OLx	Effect when TOCx=TCNT
0	0	Does not effect OCx
0	1	toggle OCx
1	0	clear OCx (set to zero)
1	1	set OCx (set to 1)

The OC events OC2, OC3, and OC4 are tied to pins PA6, PA5, and PA4, respectively, on PORTA. Recall that these pins only have the direction state of "output". The other output-compare event, OC5, is tied to pin PA3 on PORTA. This is a bidirectional pin and this means that to use TCTL1 to effect PA3, we'll need to set its direction state to output. The OC1 event uses the output pin somewhat differently than OC2-OC5 and we won't discuss its use in this course. Output-compare events OC2-OC5, however, have easily defined functions. Namely that when the event OCx occurs, the state of the output pin PAx changes from 0 to 1 or vice versa, depending upon how the bits OMx and OLx are set. This can be extremely useful if we are attempting to have the micro-controller generate output voltages quickly in response to output compare events. Because the pin state changes are handled in hardware, this effect can be manifested very quickly. If we were to attempt to do the same thing in software, it could potentially take a long time for the ISR to execute and this can dramatically destabilize a program's real-time behavior.

4.4. How is an OC Interrupt Handler Written? You write an ISR as you would write a program function. ISR's, however, must be treated somewhat differently by the compiler and we must therefore have a way to tell the compiler that a specific function is an ISR and not a regular function. Basically, there are three things we need to do in writing an ISR. These things are:

- initialize the interrupt handler
- declare the ISR function
- define the ISR's interrupt vector

We now discuss each of these steps below:

Initialization: Initialization of a hardware interrupt is done in the function `init()`. The basic things that must be done are

(1) initialization of any global variables and counters in the ISR
(2) arming the interrupt
(3) acknowledging any previously caught interrupts

The following source code shows the listing for a simplified `init()` function. This function is called at the start of any program you write.

```
void init(void){
  asm(" sei");
  CONFIG = 0x04;
  BAUD=BAUD38K;
  SCCR1 = 0x00;
  SCCR2 = 0x0C;

  _Time=0;
  TMSK2 = 0x0D;
  TMSK1 |= OC4I;
  TFLG1 |= OC4F;
  TOC4 = TCNT + 256;
  asm(" cli");
}
```

The first and last line of this function are assembly language instructions that disable/enable interrupt handling. Essentially, the first instruction `asm("sei")` sets the I bit in the condition code register, thereby disabling any interrupts. This prevents us from catching an interrupt while we're trying to initialize the system. The last command `asm(" cli")` clears the I bit, thereby re-enabling interrupt handling.

The next 4 instructions after the `sei` instruction are used to disable the watchdog timer (`CONFIG=0x04`) and to setup the asynchronous serial interface (SCI). Setting up the SCI involves declaring the agreed upon baud rate (`BAUD=BAUD38K`) and setting the appropriate values in the SCI's control registers, `SCCR1` and `SCCR2`.

The following code segment from `init()` is specific to the initialization of the output compare 4 interrupt.

```
_Time = 0;
TMSK2 = 0x0D;
TMSK1 |= OC4I;
TFLG1 |= OC4F;
TOC4 = TCNT + 256;
```

The first instruction zeros a global variable `_Time`. This global variable is a counter that keeps track of the number of times `OC4han` has been executed. It will be incremented each time `OC4han` is executed. Due to its global nature, this variable is accessible by all functions in your program. So all of your functions can use `_Time` as a variable holding the current real-time. The second instruction (`TMSK2 = 0x0D`) sets the rate at which `TCNT` is incremented. In particular, this choice will increment `TCNT` once very 407 nanoseconds. The instruction `TMSK1 |= OC4` arms the output compare 4 interrupt by setting bit `OC4I`. The instruction `TFLG1 |= OC4F` acknowledges any previously received OC4 interrupts, thereby paving the way for your program to catch the next OC4 event. Finally, we set the output compare register `TOC4` to the next time we want the OC4 event to occur. This new deadline is obtained by taking the current value of `TCNT` and adding 256 to it. So the next OC4 event should occur 256×500 nanoseconds from the current time (128 μseconds).

Interrupt Handler: The interrupt handler's source code will also be found in `kernel.c`. ISR's must be handled differently than regular functions. In particular, the compiler needs to ensure that the return from an ISR is handled by the `rti` assembly command.

You write the ISR as if it were an ordinary function. But you need to alert the compiler that this function is an interrupt handler. This alert is provided through a pre-compiler direction known as a `pragma`. The actual code in `kernel.c` is shown below:

```
#pragma interrupt_handler OC4han()
void OC4han(void){
  TOC4 = TOC4 + 256;
   _Time = _Time+1;
   TFLG1 |= OC4;
}
```

The first line

```
#pragma interrupt_handler OC4han()
```

tells the compiler that the following function is to be treated like an interrupt handler. The body of `OC4han` has only three statements. In general, we want interrupt handlers to be extremely short. In particular, the line

```
TOC4 = TOC4 + 256;
```

resets the output compare register `TOC4` to the next time we want this interrupt to be generated. The next line of code

```
_Time = _Time + 1;
```

increments a global variable `_Time`. The final line of `OC4han()` is

```
TFLG1 |= OC4;
```

This line acknowledges the interrupt by setting the appropriate bit in `TFLG1` to one.

Interrupt Vector: The final thing we need to do is declare the interrupt vector associated with our interrupt handler. This is accomplished using another `pragma`. The code you would need to provide is:

```
extern void OC4han();
#pragma abs_address:0xffe2;
void(* OC4_handler[])()={ OC4han };
#pragma end_abs_address
```

The program used here tells the compiler to install the interrupt handler at the absolute address `0xFFE2`. This absolute address is the interrupt vector for output compare 4 event (refer back to the earlier table of interrupt vectors). With these lines of code we've tied our ISR `OC4han` to the OC4 event.

4.5. How do I rewrite OC4han to generate a PWM signal? This question, of course, is the heart of this lab. While you've been provided the basic information needed to rewrite the OC4 interrupt handler, the task may still be very difficult to complete. The difficulty arises from the fact that the PWM signal you

are being asked to write has a period of 2 msec. This is extremely fast for a micro-computer running off of an 8 MHz clock. It is sufficiently fast that you run the risk of having your interrupt handler interrupted by a hardware event.

As you learned above, interrupt handlers are subroutines that are executed when a hardware interrupt event occurs. It is important, however, that the interruption be as short as possible, otherwise the main program may not execute correctly. As an example, let's consider a hardware event (such as an output compare event) that is generated in a periodic manner. If the interrupt handler is not sufficiently short, then the length of time required to execute the interrupt handler may be longer than the hardware event's period. If this occurs, then the interrupt handler's execution will be interrupted by the next hardware event. This interruption of the interrupt handler can occur over and over again so that the system never returns from the interrupt handler to the main program. If this occurs, then the system may appear to be deadlocked or else other erratic behavior may occur which can be very difficult to debug.

This is precisely the problem that we face in rewriting the OC4 handler to generate a PWM signal. To generate the PWM signal we need to generate output compare events at time intervals with less than 2 msec duration. If the OC4 handler is implemented in an inefficient manner by using more instructions than absolutely necessary it is highly likely that the OC4 handler will not be completed before the next OC4 event occurs. The bottom line is that we must be very careful to write an OC4 handler that is as short as possible. This is, in fact, a good rule to follow in writing any interrupt handler.

Below is a listing for an OC4 handler that generates a PWM signal on PA4 with a 50 percent duty cycle. The student will need to modify this handler in order to complete the lab.

```
#pragma interrupt_handler OC4han()
void OC4han(void){
  if(TCTL1==0x08){
    TOC4=TOC4+1024;
    TCTL1 = 0x0C;
    _Time = _Time + 1;
  }else{
    TOC4 = TOC4 + 1024;
    TCTL1 = 0x08;
  }
  TFLG1 |= OC4;
}
extern void OC4han();
#pragma abs_address:0xffe2;
void (*OC4_handler[])()={ OC4han };
#pragma end_abs_address
```

This OC4 interrupt handler generates a 50 percent duty cycle PWM signal that has a period of 2048 hardware time ticks. The interrupt handler takes advantage of the fact that the occurrence of an OC4 interrupt can be programmed to immediately change the logical state of the output pin PA4. Recall that this is accomplished by setting the appropriate bits in the control register `TCTL1`.

4.6. What is an Oscilloscope? The waveform generated by your PWM signal generator is a time-varying waveform. To verify the functionality of your PWM signal generator, you must be able to plot the

signal's voltage level as a function of time. A digital multi-meter (DMM) cannot be used to do this, you need another piece of electronic test equipment that is called on *oscilloscope*.

An oscilloscope is an instrument that provides a plot for a time-varying input signal. The control panel of a Tektronix analog scope is shown in figure 4.

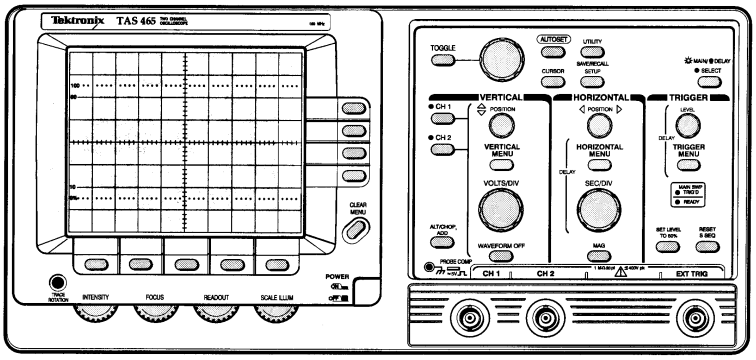

FIGURE 4. Tektronix Analog Oscilloscope Control Panel

An analog oscilloscope consists of a cathode ray tube (CRT), sweep generator and a vertical amplifier as shown in figure 5. The sweep generator causes an electron beam in the CRT to sweep horizontally across a phosphorescent screen. The vertical amplifier moves the beam in a vertical direction in response to an applied input signal. The horizontal and vertical movement of the electron beam traces out the input signal's time variations. This trace appears on the oscilloscope's screen.

Figure 5 shows the internal components of the oscilloscope. Note that there are essentially three important types of controls that you need to be able to use. These controls are the

(1) vertical scale (volts/div)
(2) time scale (msec/div)
(3) trigger

The vertical scale controls the gain of the vertical amplifier. It determines how far the electron beam will move (in vertical direction) in response to an applied voltage. The time scale determines how quickly the electron beam sweeps the screen and the trigger determines when the electron beam starts its sweep.

Unfortunately, CRT's are not like a piece of paper. When you write on the screen, the phosphor will emit light for only a short time. The picture, therefore, must be constantly refreshed. You have absolutely no hope of this happening unless the waveform you are trying to look at is constantly repeating (we refer to such waveforms as being *periodic*). Many interesting waveforms are periodic. With a repeating waveform you have some hope that the same image will be continuously refreshed so that your slow eyes can perceive it.

The earliest scopes had a knob that would let you adjust the sweep frequency in an attempt to match the frequency of the input signal. As you might guess, this made using a scope rather tricky. Fortunately, a much better system soon emerged, called the *triggered sweep*.

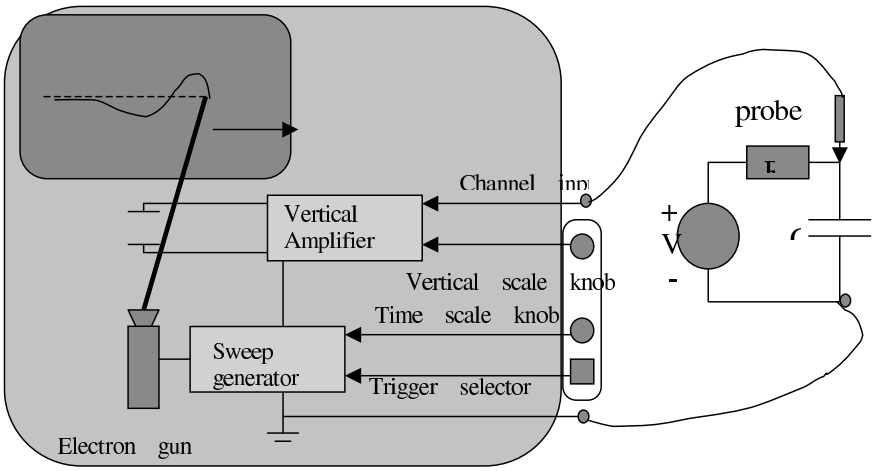

FIGURE 5. Diagram of the internal subsystems in an analog oscilloscope

The idea behind a triggered sweep was to try to get the multiple sweeps to start at the same place in the waveform each time. This way the pictures would all line up. When the scope sees the input go past a certain level, it would trigger the horizontal sweep. Soon after a sweep was complete, the circuit would re-arm and await another trigger event.

This system, of course, had limitations. Complex waveforms, for example, that have multiple waves in a single period can easily confuse the trigger circuit. To trigger the circuit, all of the crossings of the trigger point will look the same so it will choose them randomly. Often there is another signal somewhere in the circuit with the same period, but without the extra bumps. Most scope will allow you to trigger the sweep for one channel from a different channel so you can synchronize to the cleaner signal.

The trigger is characterized by at least three parameters. These parameters are

(1) The trigger event
(2) The trigger channel
(3) The trigger level

The trigger event is usually specified as either a rising (falling) edge. This means that the oscilloscope begins its sweep when the voltage change is rising (falling). The trigger channel determines which of the scope's input channel can be used as the source of the trigger signal. The trigger level determines how large of a rising or falling edge will cause the scope to trigger.

The next big advance in oscilloscope technology was the *digital scope*. A digital storage oscilloscope (DSO) is basically a computer optimized for data acquisition. At the heart of a DSO is one or more high speed analog-to-digital converters. These ADC sample analog voltages and display them on a computer screen. Digital scopes can do many things that analog scopes cannot do. First of all the digital scope digitizes the wave form and can hold it on the screen forever. So you can take a snapshot of a particular signal and then study it at your leisure.

The initial set-up for a digital scope can be intimidating to the beginning student because the control panel looks so complicated. A set-up procedure for the scope has been provided below (see figure 6) If you follow this set-up you should be able to set-up the scope properly.

EE224 Lab 8 - Digital Scope Setup

1. Press the ***Save/Recall*** button.
2. Press the soft key on the bottom of the screen labeled "Recall Factory setup"
3. Press the soft key on the side of the screen labeled " OK Confirm factory Init"
4. Hook Channel 1 probe to your signal.
5. Press the ***AutoSet*** button on the right hand side of the scope.
6. Verify that the horizontal scale is set to 100uS/division.
7. Adjust the horizontal position knob so that the trigger point is one division awayfrom the left side of the display.
8. Adjust the vertical position knob so that the ground reference is two divisions from the bottom of the display.
9. Change the vertical scale to 1V/division.
10. Your screen should now look something like the figure below.

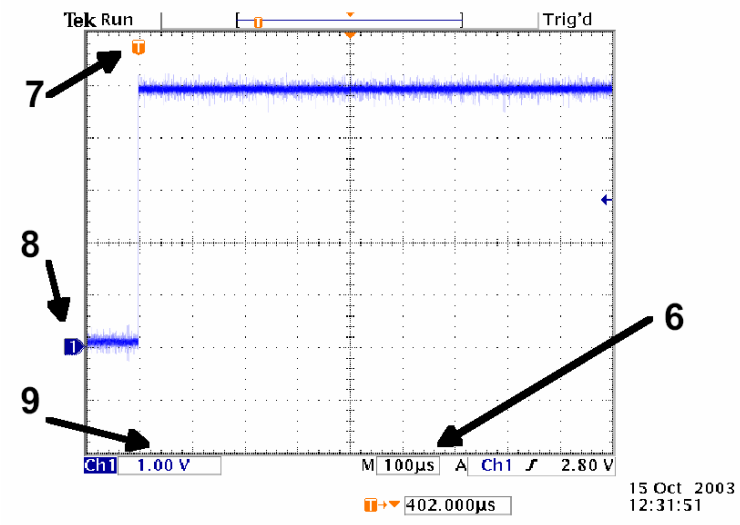

FIGURE 6. Oscilloscope Setup

What you've just read is a rather hasty introduction to oscilloscopes, but this should be enough to get you started. In this lab, you'll need to use the oscilloscope to capture PWM signals generated by your system. You will need to use the captured trace to measure the duty cycle and period of the PWM signal.

4.7. What is an RC Circuit? An RC circuit contains a single resistor, R and a single capacitor C. From your course textbook you should already know that a capacitor is a two-terminal device whose voltage, $v(t)$, and current, $i(t)$, satisfy the following relationship,

$$i(t) = C\frac{dv(t)}{dt}$$

This equation says that the current flowing through a capacitor is proportional to the rate at which voltage changes across the device's terminals. The proportionality constant, C is called the device's *capacitance* and it is measured in units called *farads*.

Capacitors come in a variety of forms. One of the most common types of capacitors is a *ceramic* capacitor. A ceramic capacitor is shaped like a disk with two leads coming out of it. A picture of the schematic symbol of the capacitor is shown in figure 7. This symbol consists of two bars (representing the capacitor's two plates) with two leads coming out of them. A picture of a representative ceramic capacitor is also shown in figure 7. Another type of capacitor is the *electrolytic* capacitor. The symbol for an electrolytic capacitor has one of its plates curved and the top plate is marked with a plus sign (see figure 7). Electrolytic capacitors are constructed using a paper soaked in an electrolyte. This fabrication method gives enormous capacitances in a very small volume. But it also results in the capacitor being *polarized*. In other words, the capacitor only works with one polarity of voltage. If you reverse the polarity, hydrogen can disassociate from the internal anode of the capacitor and this hydrogen can explode. Electrolytic capacitors always have their polarity clearly marked, often with a bunch of negative signs pointed at the negative terminal. A picture of an electrolytic capacitor is shown in figure 7.

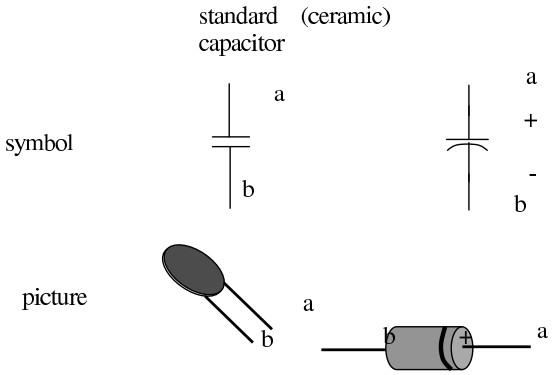

FIGURE 7. Symbols and drawings of capacitors

An RC circuit is a particularly simple network containing a capacitor. The RC circuit consists of an independent voltage source in series with a resistor, R, and a capacitor C. The schematic diagram for this circuit is shown in figure 8. Analyzing this circuit means determining the voltage over the capacitor, $v_c(t)$, (as a function of time). The exact solution, of course, depends on two things. These two things are the initial voltage over the capacitor, V_0, and the input voltage, $v(t)$, generated by the independent source. In the remainder of this section we state two specific solutions known as the *natural response* and *step response*. The derivation of these particular response equations is done in the lecture component of the course.

Natural Response: The first specific solution we'll consider is the voltage over the capacitor under the assumption that the capacitor's initial voltage is V_0 and the applied input voltage is zero (i.e., $v(t) = 0$ for all $t \geq 0$). This particular solution is called the *natural response* of the RC circuit and it can be shown to have the form

$$v_c^{(\text{nat})}(t) = V_0 e^{-t/RC} \tag{4.2}$$

for $t \geq 0$.

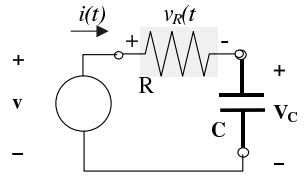

FIGURE 8. RC circuit

It is valuable to plot the general shape of the natural response in equation 4.2. Note that the voltage has a time dependency that is an exponential function of time. This exponential function, $e^{-t/RC}$ has a negative exponent so that as t increases, the function's value decreases in a monotone (non-increasing) manner to zero. In other words, if we consider $v_c^{(nat)}(t)$ for $t \geq 0$, we expect it to start (at time 0) at the voltage V_0 and then to taper off to zero as t increases. This particular relationship is shown in figure 9.

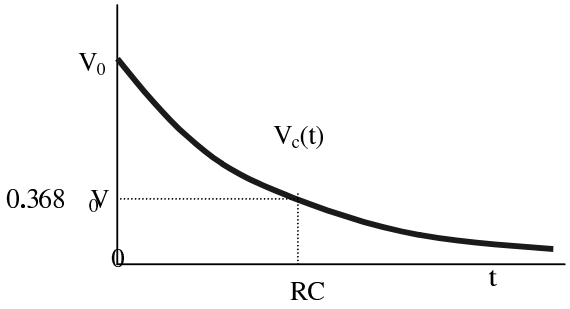

FIGURE 9. Natural Response

Note that the expression, RC, has units of time. We generally refer to RC as the *time constant* of the circuit. In fact, at time $t = RC$, we know that the voltage is $v_c^{(nat)}(RC) = V_0 e^{-1} \approx 0.368 V_0$. This means that after one "time constant", the initial voltage on the capacitor has decayed to about one third of its initial value. After three time constants, we expect $v_c^{(nat)}(3RC) = V_0 e^{-3} \approx 0.05 V_0$. This is, of course, a very small number and it means that after 4-5 time constants, the voltage over the capacitor is essentially zero. The time it takes to finish discharging the capacitor is determined by our choice for the resistors R and C. In other words, the discharge time for the capacitor is determined by the *RC constant* of our circuit.

Standard capacitor values are on the order of μF (a large capacitor) to pico-farads. If we were to use a 1 k-ohm resistor in series with a 1 μF capacitor, the RC constant would be $RC = 1000 \times 10^{-6} = 1$ m-sec. In this case, our source-free circuit would discharge the capacitor in about 4-5 milli-seconds. If we were to use an even smaller capacitor, let's say about 100 pico-farad, then this discharge time would be even shorter. In particular, for a 100 pico-farad capacitor in series with a 1 k-ohm resistor, we would expect a time constant of $RC = 10^3 \times 100 \times 10^{-12} = 10^{-7}$ sec. This is one tenth of a micro-second. So in this case we would discharge a capacitor in about half a micro-second, a very very short time interval.

Step Response: The second specific solution we'll consider is the voltage over the capacitor under the assumption that the capacitor's initial voltage is V_0 and the applied input voltage is a step function of magnitude V. In other words,

$$v(t) = \begin{cases} V & t \geq 0 \\ 0 & t < 0 \end{cases} = Vu(t)$$

where $u(t)$ is a unit step function. The capacitor's response to this particular "step" input is called the *step response* of the RC circuit. The step response can be shown to have the following form,

$$v_c^{(\text{step})}(t) = \left(V_0 e^{-t/RC} + V(1 - e^{-t/RC})\right) u(t) \tag{4.3}$$

Let's assume that $V_0 = 0$ so that the capacitor is initially uncharged. In this case the step response takes the following simplified form,

$$v_c(t) = V(1 - e^{-t/RC})u(t) \tag{4.4}$$

for all t. Figure 10 plots this function for $t \geq 0$. This figure shows that the initial voltage over the capacitor is zero and then grows in a non-decreasing (monotone) manner until it approaches a steady state voltage of V volts. The rate at which $v_c^{(\text{step})}(t)$ approaches the steady state voltage is determined by the time constant RC. On the basis of our discussion for the natural response, we expect the capacitor to be fully charged to within 5 percent of its full charge (V) within three time constants. After 4-5 time constants, the capacitor should be completely charged to V volts for all practical purposes.

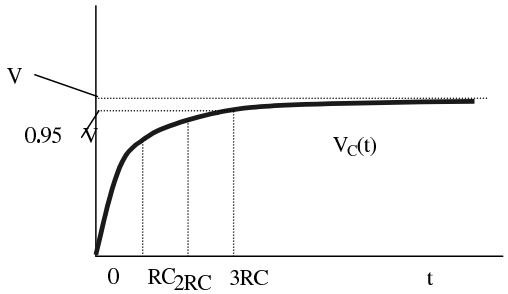

FIGURE 10. Step Response assuming uncharged initial capacitor

If we do not neglect the initial charge on the capacitor, then the circuit's response is given by equation 4.3. Notice that this equation is simply the sum of equation 4.4 and the natural response in equation 4.2. So we can simply sum the two responses shown in figure 9 and 10 to obtain a plot of the system's total response.

Figure 11 illustrates how these individual parts of the response are combined to form the total response. One of the lighter lines represents the forced response to a step input. The other decreasing light line represents the natural response to an initial voltage on the capacitor. The total response is simply obtained by taking their sum which is shown by the dark trace in figure 11. What we see in this figure is that as time goes to infinity, the initial charge on the capacitor dies away and the total response converges to the steady state voltage V.

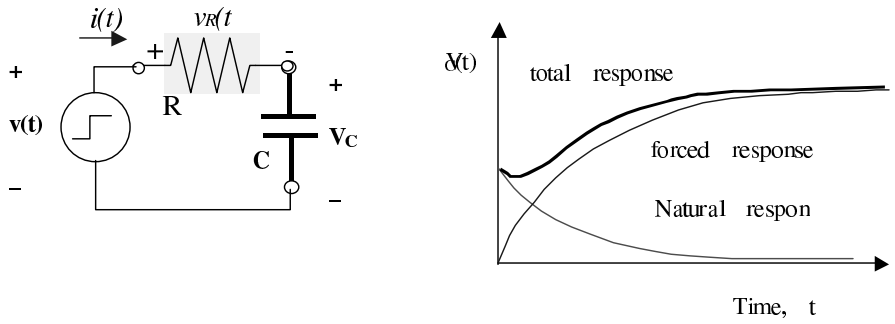

FIGURE 11. Total Response of RC Circuit

4.8. What is the RC circuit's response to a PWM signal? In this section, we discuss the RC circuit's (figure 8) response to an input signal that is a pulse width modulated signal of known period and duty cycle. The pulse width modulated input signal is shown in figure 12. Over a single period, $[0,T]$, the input voltage to the circuit has two distinct parts. There is the "charging" part from $[0,T_1]$ during which the applied voltage is V. In this interval, the capacitor is being charged by the external voltage source. The second part is the "discharging" part from $[T_1,T]$. During this interval the applied voltage is zero and so the capacitor is discharging through its resistor.

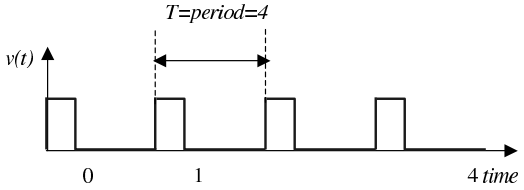

FIGURE 12. Pulse Width Modulated Signal

During the "charging" phase, we can think of the RC circuit as being driven by a step function of magnitude V volts. If we assume that the capacitor has an initial voltage of V_0 at the beginning of the charging phase (time $t=0$), then the circuit's response is simply given by equation 4.3 for $t \in [0,T_1]$.

During the "discharge" phase, there is no external voltage being applied to the RC circuit. This means that the system response is due solely to the capacitor voltage that was present at time T_1 after the charging period. As a result, the capacitor's voltage over the time interval $[T_1,T]$ is simply the RC circuit's natural response. This means, of course, that the capacitor voltage for $t \in [T_1,T]$ is given by equation 4.2 of the form $V_1 e^{-(t-T_1)/RC}$ for $t \in [T_1,T]$, where the initial voltage, V_1, is the voltage on the capacitor at time $t=T_1$.

The top drawing in figure 13 illustrates the output signal we expect from a PWM signal driving an RC circuit over an interval from $[0,T]$. For times beyond this interval, we expect to see the waveform shown in the bottom drawing in figure 13. In this drawing we assume that the capacitor is initially uncharged. As

our circuit cycles through its charge and discharge phases, the voltage over the capacitor follows a saw-tooth trajectory that eventually reaches a steady state regime. In this steady-state region, the capacitor on the voltage zigzags between V_0 and V_1 volts. The exact value of these steady state voltages is dependent on the period T and the duty cycle T_1/T.

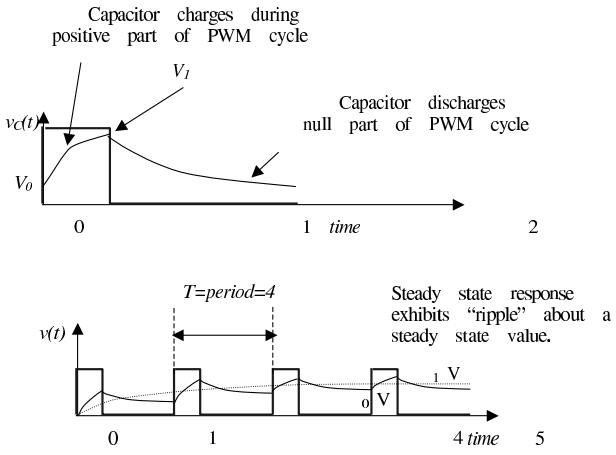

FIGURE 13. Response to PWM signal over a single period

The steady state region shown in figure 13 is usually characterized by two "figures of merit". The first "figure of merit" is the *mean voltage*, V_m, of the steady state response and it is given by the equation

$$V_{\rm m} = \frac{V_1 + V_0}{2}$$

where V_1 and V_0 are the maximum and minimum voltages over the steady state region, respectively. The other "figure of merit" is the *ripple*. The ripple measures the peak variation in the steady state region and it is given by the equation

$$V_{\rm r} = \frac{V_1 - V_0}{2}$$

We often specify the ripple as a percentage in which $V_{\rm r}$ is normalized by the steady state voltage $V_{\rm m}$. For example if $V_{\rm m} = 10$ volts and $V_{\rm r} = 1$ volt, then the ripple would be 10%.

In this lab you will be using the output of the RC network as the analog voltage generated by a digital-to-analog converter (DAC). As you can see in figure 13, this analog voltage is not really constant, it has a mean value and a small ripple. So the performance of the RC-DAC can be characterized by these two figures of merit. If our DAC performs well, then its mean voltage $V_{\rm m}$ must vary in a linear manner with the commanded voltage and its ripple, V_r, should be very very small. In return for accepting a small ripple, we gain some important benefits. In the first place the DAC only needs to use a single output line and the precision of the DAC increases significantly (from 3 to 6 bits).

The reason for the "increased" precision is that we are no longer using the output lines to encode the digital number we want to convert. Instead, we are using a time-varying signal (the PWM signal) to encode the voltage we wish to convert. We don't get something for nothing. In return for this enhanced DAC, we must settle for a small ripple on the converted voltage and our DAC's response time to changes in the requested voltage will be governed by our circuit's RC time constant.

5. Tasks

5.1. Pre-lab Tasks:

(1) Consider an RC circuit whose input is a PWM signal with a period T and duty cycle T_1/T. Derive an equation for the signal's steady state value as a function of R, C, and the PWM signal's duty cycle and period.
(2) Derive an equation for the signal's ripple as a function of R, C, and the PWM signal's duty cycle and period.
(3) Assume that the PWM signal has a 50 percent duty cycle, RC is 1 milli-second. Use the preceding equations to plot the ripple and steady state value of the capacitor's voltage as a function of the PWM signal's period, T. Use this plot to select a range of periods for which the capacitor's voltage ripple is less than 5 percent. Use the largest value of T ensuring a 5 percent ripple and plot the average capacitor voltage as a function of the PWM signal's duty cycle. What does this plot suggest about the relationship between the duty cycle, T_1/T, and the average steady-state capacitor voltage?
(4) Before starting the In-lab task, show your pre-lab work to the TA so we can verify the correctness of your analysis and design.

5.2. In-lab Tasks:

(1) Complete Oscilloscope tutorial
(2) Remove the R2R ladder network from your breadboard and disconnect the clamp circuit's output from the MicroStamp11. You will be rebuilding parts of this circuit in later labs and you won't need the R2R ladder network anymore. The rest of your breadboard you can reuse.
(3) Build the RC circuit and attach it's input to pin PA4.
(4) Obtain the files *lab7pwmdac.c* and *kernel7.c* from the lab website.
(5) Modify the *lab7pwmdac.c* program you obtained from the lab website to ensure it works with your *display digit* function. This program accepts a 6-bit integer representing the requested output value of your DAC, outputs a 3-bit version of this request to the 7-segment LED display, and sets the pwm output. Modify *kernel7.c* to set the PWM signal's period equal to the period (T) you determined in task 3 of the pre-lab tasks. Look for the variable *pwmtime* and set it to the appropriate number. If you are not sure what number to use, re-read the PWM section in this chapter.
(6) Use a DMM (set to measure DC voltages) to measure the voltage output by your DAC circuit for at least 16 different commanded voltages.
(7) Use the oscilloscope to measure the steady state voltage and value of the RC circuit's response (ripple voltage) for at least 16 different commanded voltages. Please note that it may be difficult to stably trigger the scope directly from the RC circuit's output. It is recommended that you use channel 1 to display the RC circuit's output and that you use channel 2 to display the PWM signal. Be sure to trigger the scope from channel 2 as the PWM signal provides a more reliable triggering signal.
(8) Description of what happened during in-lab task.

5.3. Post-Lab Tasks:

(1) Include a listing of your final program in the lab book and explain how it works.
(2) Plot the DMM and oscilloscope measurements versus commanded voltages. Assess how well your new DAC works.

(3) Plot the ripple voltage versus commanded voltages.
(4) Explain why this new design is an enhancement over the earlier DAC you built.
(5) Demonstrate the functionality of your system to the TA.

6. What you should have learned

After completing this lab you should know

- how to analyze an RC circuit's response to a PWM signal,
- how to use an RC circuit to build an improved digital-to-analog computer,

7. Grading sheet

LAB 7 - Digital to Analog Conversion Revisited

15 Pts	**PRE-LAB TASKS**
5	Analysis deriving the steady-state voltage of PWM driven RC circuit.
5	Analysis deriving the ripple voltage of PWM driven RC circuit.
2	Design of RC circuit, breadboard layout, and explanation of design.
2	Graph of predicted ripple and steady state voltage assuming a 2 msec PWM period as a function of requested duty cycle. These predictions use the RC circuit parameters you designed in the pre-lab.
1	TAs verification of your analysis and conclusions.

10 Pts	**IN-LAB TASKS**
1	TA check of your oscilloscope set up.
3	Explanation of what happened in the lab.
2	Experimental measurement of DAC output using DMM.
2	Experimental oscilloscope measurement of steady state voltage. for at least 16 different commanded voltages.
2	Experimental oscilloscope measurement of ripple for at least 16 different commanded voltages.

8 Pts	**POST-LAB TASKS**
2	Final Program Listing and an explanation of how this program works.
2	Graph of measured voltages (oscilloscope and DMM) versus commanded voltages
2	Graph of ripple as a function of commanded voltages.
2	Comparison with your pre-lab predictions and assessment of DAC performance relative to your earlier DAC system.

2 Pts	**DEMONSTRATION ..... STOP if not checked off**
2	Successful demonstration of completed system.

GO ON to next lab if completed.

Name _____ .

Lab Group Number _____ GRADE _____ out of 35

TA signature _____ Date _____ .

CHAPTER 8

Getting Power off the Wall

1. Objective

Through out all of these labs you've used the bench power supplies to provide power to your circuits. In this lab, you'll build a power supply that delivers 0 volts (ground), 5 volts, and 12 volts DC from a 120 alternating (AC) voltage source. You will then use this power supply to drive your system.

2. Parts List

Italicized parts were used in the previous lab.

(1) *wire kit and breadboard*
(2) *MicroStamp11, docking module, and serial cable*
(3) *Laptop with ICC11 compiler and PMS91 loader*
(4) *one 7-segment LED display (LSD3221-11)*
(5) *seven 2.2 k-ohm resistors*
(6) *two 10 k-ohm resistor*
(7) *two buttons*
(8) *one LM660 quad op-amp IC*
(9) *two resistors (R) for op-amp buffer*
(10) *two 1n4007 diodes*
(11) *one 10 k-ohm trim potentiometer*
(12) *two additional resistors for diode clamp circuit*
(13) *additional resistor for RC network*
(14) *capacitor for RC network.*
(15) **four 1n4007 diodes**
(16) **LM7805 voltage regulator with heat sink**
(17) **120-12 volt wall transformer**
(18) **1000 μF electrolytic capacitor**
(19) **Zener Diode and 100-400 ohm resistor**
(20) **0.1 μF ceramic capacitor**

3. Background

This lab has the student build a circuit that converts a 120 volt *alternating voltage* into a 5 and 12 volt DC (direct current) voltage level. In order to do this, you will need to use a *transformer* to reduce the alternating voltage from 120 to 12 volts. This 12 voltage alternating voltage must then be converted to a 12

volt constant (DC) voltage through a special diode-capacitor circuit called a *full-wave rectifier* circuit. The 12 volts delivered by the rectifier circuit must then be stepped down to a fixed five volt level using a *voltage regulator*.

4. What is an Alternating Voltage?

Faraday's law of induction provides a basis for converting mechanical energy into electrical energy. The basic idea is to move a coil of wire relative to a magnetic field. This motion will generate a current in the wire. Such a device is called a *generator* and a conceptual drawing of this device is shown in figure 1.

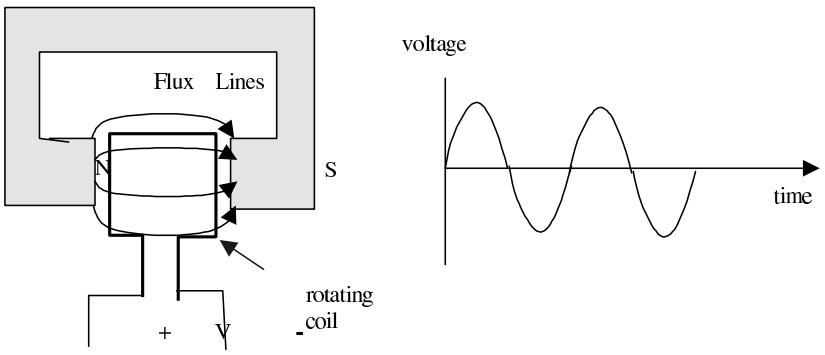

FIGURE 1. A generator and the voltage it generates

To make things simple, the coil is usually made to rotate within the field. As the coil rotates, it cuts through the flux lines, generating a voltage across the coil's terminals. When the face of the coil is parallel to the field, it cuts rapidly through the flux lines. But when the coil has turned 90 degrees and is perpendicular to the field lines, then the motion of the coil is tangential to the field and no voltage is produced. As the coil turns past this point, it cuts through the field in the opposite direction, generating a negative voltage. The end result of this chain of events is that the voltage produced by the generator varies as the cosine of the angle as shown below. This sinusoidal waveform is referred to as an *alternating* current or AC.

The equation for a waveform of this type is:

$$v(t) = A\cos(\omega t + \phi) \tag{4.1}$$

where A is the *amplitude*, ω is the *frequency*, and ϕ is the *phase* . Since $v(t)$ is a time-varying voltage signal, A has units of volts. The frequency has units of radians per second. Phase is measured in radians. We often measure frequency in a related unit of *cycles per second*. A cycle corresponds to 2π radians.

The sinusoidal waveform in equation 4.1 is a **periodic** waveform. A signal v is periodic if and only if there exists $T > 0$ such that $v(t) = v(t+T)$ for all t. To see if a sinusoidal waveform is periodic we therefore need to find T such that

$$A\cos(\omega t + \phi) = A\cos(\omega(t+T) + \phi) \tag{4.2}$$

In particular, we know that the cosine function repeats every 2π radians so we need to find T such that

$$A\cos(\omega(t+T) + \phi) = A\cos(\omega t + 2\pi + \phi) \tag{4.3}$$

Clearly this occurs if $\omega T = 2\pi$ or rather

$$T = \frac{2\pi}{\omega} \tag{4.4}$$

is the **fundamental period** of this sinusoidal function.

The *size* of a sine wave can be measured in a variety of ways. We may, for instance, use the waveform's amplitude (A) to specify the waveform's size. Another measure of "size" is the signal's **root mean square** or **rms** strength

$$\text{RMS} = \left[\int_0^{\omega/2\pi} (A\cos(\omega t + \phi))^2 \, dt\right]^{1/2} \tag{4.5}$$

Since generators naturally produce sine waves, these waveforms play an important role in electrical engineering. It also turns out that sine waves also provide an efficient way of transporting electrical energy over a long distance. This is part of the reason why AC voltages are used in international power grids and, of course, this is why your wall socket provides a 120 volts (rms) AC voltage at 60 Hz.

In contrast to AC voltages, batteries provide a *direct current* or DC voltage. DC voltages are constant over time. In order to obtain DC voltages from an AC wall socket we're going to have to find some way of **regulating** the AC power source.

5. What is a Transformer?

A *transformer* is a device where two or more coils share a common magnetic field. Figure 2 is a drawing of a transformer. In this drawing you see two coils; a primary and a secondary coil. Both coils are wrapped around a ferro-magnetic core. The idea is that the primary coil takes in a time-varying voltage and creates a time-varying magnetic field. The ferro-magnetic core channels this magnetic field through the secondary coil. The secondary coil converts this time-varying magnetic field back into a time-varying voltage.

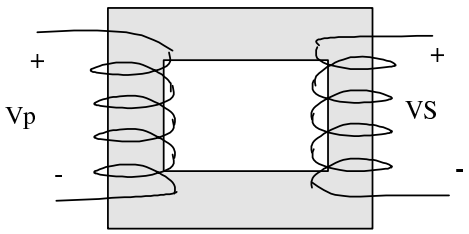

FIGURE 2. Transformer

An ideal transformer obeys a simple relationship. The sinusoidal voltage on the secondary side, $v_s(t)$, will be proportional to the sinusoidal voltage on the primary side, $v_p(t)$. The constant of proportionality, n, is determined by the ratio of the number of turns on the secondary side to the number of turns on the primary side. This constant is commonly called the *turns ratio*. In equation form this means that

$$v_s(t) = nv_p(t)$$

where n is the turns ratio, $v_s(t)$ is the voltage waveform over the secondary coil, and $v_p(t)$ is the voltage waveform over the primary coil.

For our purposes, we will use a transformer that converts the 120 VAC available at the wall socket to a 12 VAC signal. You should have such a *wall-transformer* (see figure 3 in your kit. The transformer plugs directly into the wall socket and the wires coming out of the transformer terminate in a connector that you can plug into a special socket that is in your kit. The wall transformer looks like a very "fat" and "heavy" wall plug. It is fat and heavy because it has a fat and heavy transformer inside of it. The transformer has been designed to convert a 120 VAC input into a 12 VAC signal. So this means that your transformer has a 10:1 turns ratio.

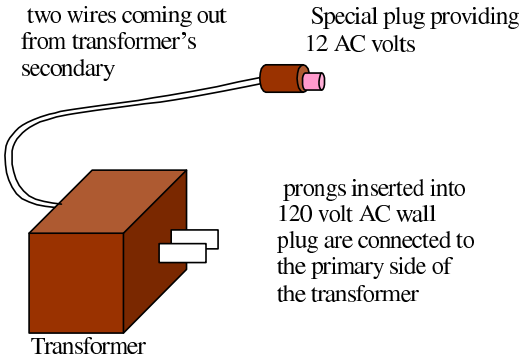

FIGURE 3. Wall Transformer

6. What is a Rectifier Circuit?

Now that we've *stepped down* the AC voltages to a level that is more in line with the voltage requirements of the μStamp11, we are left with the problem of converting a 12 volt AC signal into our desired 5 volt DC power supply. We'll approach this in two steps. First we'll convert the AC voltage into a DC voltage via a process known as *rectification*. Then we'll step down this 12 volt DC voltage down to 5 volts using the **voltage regulator** . This section briefly talks about the rectification process.

The simplest possible circuit for converting AC into DC is a *half-wave* rectifier. This circuit consists of a single diode that only allows current to flow in one direction. A possible circuit is shown below in figure 4. In this figure, you'll find the AC power source connected to the primary side of a transformer. Note the symbol we use for the transformer. The secondary terminals of this transformer are then connected to a diode and resistor in series.

The operation of this circuit is straightforward. When V_{ac} is in the positive part of its cycle, a positive voltage is produced on the secondary side of the transformer. This voltage forward biases the diode and the diode begins passing current. As a result most of the voltage drops across the load. When V_{ac} is negative, then the secondary side also has a negative voltage. The diode is then reverse biased and ceases to pass current. As a result, the voltage drop over the load is zero. The voltage waveform over the load resistor therefore looks as shown in figure 4. Only the positive side of the sinusoidal cycle is present and the negative side has been clamped off by the diode.

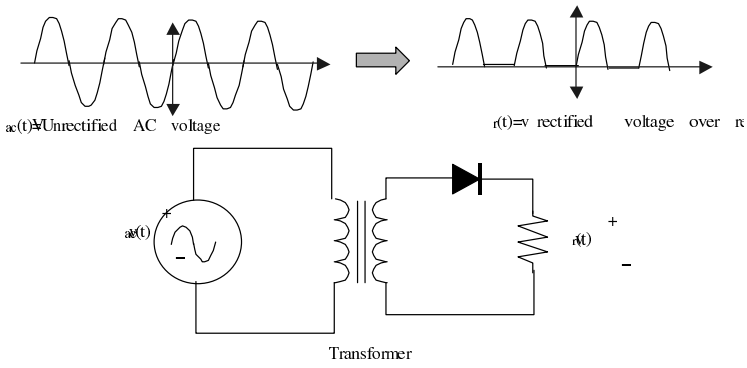

FIGURE 4. Half-wave rectifier

Looking at the output voltage,$v_r(t)$, you should note that it resembles the output of the battery in that it is always positive. Unfortunately, this positive waveform is rather "bumpy" and we need to find a way to smooth it out. The RC circuit shown in figure 5 is used to smooth out these bumps. In this circuit, we've added a large capacitor in parallel with the load resistance. The capacitor can store energy during the times when the voltage over the load is positive. When the load voltage is clamped to zero, our capacitor can then slowly release its stored energy, thereby smoothing out the voltage over the load.

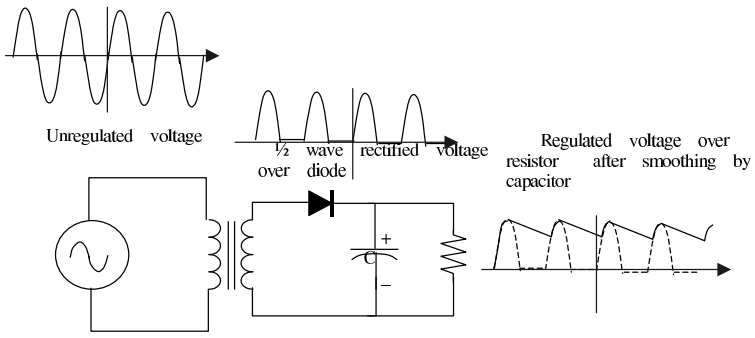

FIGURE 5. Half-wave rectifier with capacitor

What happens in this circuit is that the diode turns on when the voltage on the cap is about 0.7 volts (the threshold voltage for the diode) below that coming out of the transformer. Meanwhile the load discharges the cap with our standard RC time constant. The circuit must be carefully designed so that the time-constant is much longer than the AC cycle time. Even so, the cap will probably lose some voltage over the idle time between pulses and this loss will result in voltage *ripple*. The resulting waveforms are shown below in figure 5.

There is something else new in this circuit. Notice how the bottom plate of the capacitor is shown with a curve and the top plate is marked with a plus sign. This is because special capacitors are required to get a high capacitance in a small space. In particular, you'll be using *electrolytic* capacitors. Such capacitors are

constructed using a paper soaked in an electrolyte. This fabrication method gives enormous capacitances in a very small volume. But it also results in the capacitor being *polarized*. In other words, the capacitor only works with one polarity of voltage. If you reverse the polarity, hydrogen can disassociate from the internal anode of the capacitor and this hydrogen can explode. Electrolytic capacitors always have their polarity clearly marked, often with a bunch of negative signs pointed at the negative terminal. You should have a 1000 μF capacitor in your parts kits that you can use in your power supply circuit.

While the half-wave rectifier has the virtue of simplicity, it lacks efficiency because we are throwing away the negative side of the waveform. A better solution would be to use the power in both sides of the waveform. Circuits that do this are called *full-wave rectifiers*. In particular, you can use the following circuit shown in figure 6 to build the full-wave rectifier. The left-hand side of this circuit is the full wave bridge. This part of the circuit consists of four specially arranged diodes. The output of the full wave rectifier is is, essentially a 12 volt DC supply. There will be a small ripple on this supply, but you won't really be able to notice it even if you look at the waveform using the oscilloscope.

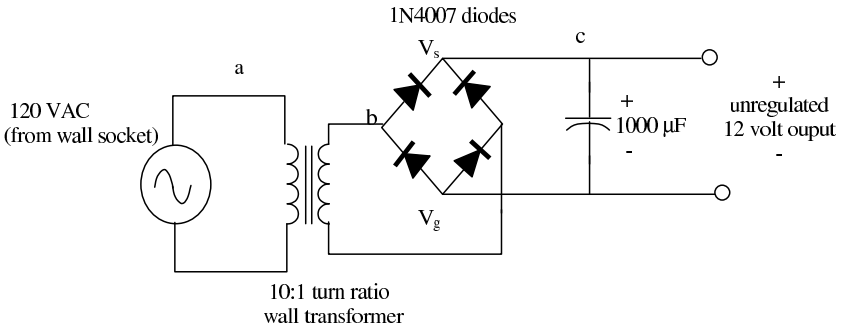

FIGURE 6. Full wave rectifier

The circuit shown in figure 6 generates a DC voltage of 12-volts and ground across the two terminals marked V_s and V_g. Your MicroStamp11, however, requires a 5 volt supply. We can step down this 12 voltage voltage to a 5 voltage voltage in several ways. One method is to use a Zener diode to clamp the voltage at 5 volts. A zener diode is a diode whose breakdown voltages has been designed to sit at a specific voltage level. The circuit shown in figure 7 performs this function. The resistor in series with the diode is used to limit the output current, typical values are on the order of 100-500 ohms.

Another way of stepping down the 12 voltage supply is to use a special three-terminal device called a **voltage regulator**. A voltage regulator is a special semiconductor device that has been specially designed to act as an ideal battery. The voltage regulator connections are shown on the righthand side of figure 8. As you can see the voltage regulator has 3 pins. Pin 1 (VIN) is connected to the positive battery terminal. Pin 2 (GND) is connected to ground (the negative terminal of your battery) and Pin 3 is the 5 volt regulated output. In your lab kit you'll find an LM7805 voltage regulator. You can use this to construct the regulator driven power supply for your system. Your lab kit also includes a metal heat sink. The heat sink should be attached to the regulator to help protect it from thermal overstress.

In connecting your voltage regulator be sure to put a 0.1 μF capacitor on the output end of your power supply. This capacitor helps remove voltage spikes from your power supply, for if you have a step change in the voltage, the capacitor acts as a short circuit to ground.

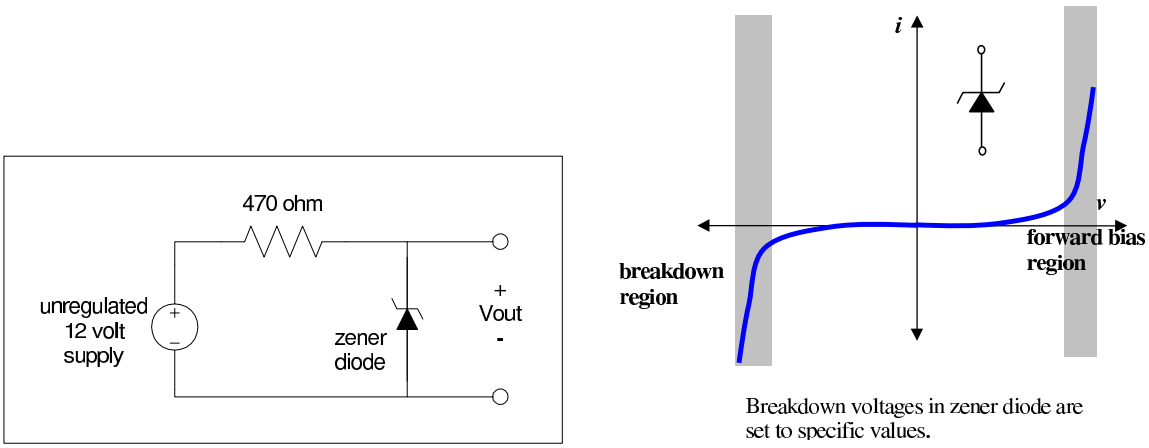

FIGURE 7. Zener Diode Voltage Regulator

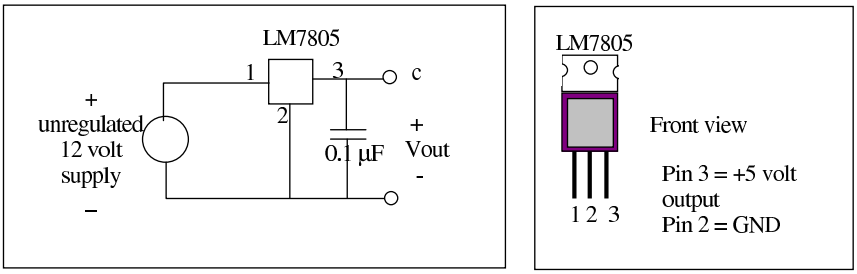

FIGURE 8. LM7805 Votlage Regulation Circuit

7. Tasks

7.1. Pre-lab Tasks:

(1) Sketch the time-varying waveforms you expect to see at nodes a, b, and c in figure 6.
(2) Explain in your own words how your power supply works.
(3) Draw the complete schematic diagrams for a power supply that uses the LM7805 to regulate the output voltage. Draw another complete schematic for the circuit that uses the Zener diode to regulate the output voltage.
(4) For both circuits, plot the delivered output voltage you expect your source to supply as a function of the load resistance.
(5) Draw a sketch of your breadboard layout for your power supply using the voltage regulator.

7.2. In-lab Tasks:

(1) Build the power supply circuit first without the voltage regulator and 1000 μF capacitor. Use the oscilloscope to view the voltage waveform at node c. Sketch this waveform in your lab book.

(2) Now add the capacitor to your circuit, use the oscilloscope to view the voltage waveform at node *c*. Again include a sketch of what you see in the lab notebook.
(3) Add the zener-diode regulator circuit and measure the output voltage as a function of load resistances between 10 kohms and 100 ohms. You should at least use resistance values of 10 kohm, 4.7 kohm, 1 kohm, 470 ohm, 100 ohms. Note that you'll need to use some resistors that have higher power ratings than you've used in the past.
(4) Remove the zener-diode circuit and add the voltage regulator circuit. Repeat the preceding task for this circuit.
(5) Now connect your power supply (using the voltage regulator) to your previous lab circuit. Have the TA check your final design. It should now run without having to be connected to be bench power supply.

7.3. Post-Lab Tasks:

(1) The power supply you built provides 5 volts for your MicroStamp11. Your op-amps, however, require 9 volts. Explain how you could use your power supply to power your op-amps.
(2) Compare the voltage-load characteristics you plotted in the Pre-Lab and In-lab tasks. Explain any differences between your Pre-Lab and In-lab results. Finally use the data you gathered in your In-lab task to determine a Thevenin equivalent circuit for the power supplies that use the voltage regulator and compare this against the Thevenin equivalent circuit for the zener-diode based power supply. This should include plots of the data. Which power supply is the better?

8. What you should have learned

After completing this lab you should know how a power supply converts alternating voltage levels into the DC voltages used in many electrical systems. The circuit uses diodes to rectify the time-varying voltage and then uses an RC circuit to convert it to a DC voltage level. You also learned what a Zener diode was. Finally you were shown how to use a semiconductor voltage regulator to step down a voltage to a safe 5 volt level.

9. Grading sheet

LAB 8 - Getting Power off the Wall

16 Pts	**PRE-LAB TASKS**
2	Sketch of power supply waveforms.
4	Explanation of circuit operation.
2	Schematic of zener-diode power supply.
2	Schematic of voltage-regulator power supply.
4	Sketch of predicted output voltage versus load resistance for both zener diode and voltage regulator supplies.
2	Sketch of breadboard layout of voltage regulator supply.

13 Pts	**IN-LAB TASKS**
5	Explanation of what happened in the lab.
2	Sketch of oscilloscope waveforms for rectified waveform w/o capacitor
2	Sketch of oscilloscope waveform for rectified waveform with capacitor
2	Experimental measurements of output voltage versus load resistance for zener-diode regulated power supply.
2	Experimental measurements of output voltage versus load resistance for voltage regulator power supply.

14 Pts	**POST-LAB TASKS**
2	Explanation of how you could power your op-amps from your power supply.
6	Comparison of predicted and experimentally determined voltage/load characteristics for both the zener-diode and voltage regulator power supplies This should include plots of the data.
6	Thevenin models for both power supplies

2 Pts	**DEMONSTRATION ..... STOP if not checked off**
2	Successful demonstration of completed system.

GO ON to next lab if completed.

Name _____ .

Lab Group Number _____ GRADE _____ out of 45

TA signature _____ Date _____ .

CHAPTER 9

Serial Interfaces

1. Objective

In this lab the student will enhance the 7-segment LED display designed in earlier labs so it uses fewer output pins and is brighter. This enhancement will be accomplished using a *serial communication interface* between the LED display and the MicroStamp11. The modified LED drive circuit only requires 3 output pins, rather than the seven pins used in the original design.

2. Parts List

(1) *wire kit and breadboard*
(2) *MicroStamp11, docking module, and serial cable*
(3) *Laptop with ICC11 compiler and PMS91 loader*
(4) *one 7-segment LED display (LSD3221-11)*
(5) **seven 100 ohm resistors**
(6) *two 10 k-ohm resistor*
(7) *two buttons*
(8) *one LM660 quad op-amp IC*
(9) *two resistors (R) for op-amp buffer*
(10) *two 1n4007 diodes*
(11) *one 10 k-ohm trim potentiometer*
(12) *two additional resistors for diode clamp circuit*
(13) **one ULN2003 Darlington array driver IC**
(14) one 74HC595 serial-to-parallel IC

3. Background

In the LED display you designed earlier, each LED is controlled by a separate I/O pin. Since our display has seven segments, this means that seven of the MicroStamp11's output lines are needed to drive the display. This is a problem because the MicroStamp11 only has eleven output lines. So our earlier design uses nearly all of the output lines to drive the display, thereby reducing the ability of the MicroStamp11 to interact with other peripheral devices.

One way to reduce the required number of output lines is to use a *serial interface* between the MicroStamp11 and the display. In a serial interface, the Micro-controllers sends a "series" of pulses, one after the other, down a single wire. Each of these pulses represents the value that one of the LED segments must take. A serial-to-parallel device called a *shift register* is used to transform this series of pulses into a constant signal on seven separate lines. We still have seven wires going to the LED display, but in this case, we only need

one wire (plus perhaps a couple of extra control lines) between the MicroStamp11 and the serial-to-parallel device.

The picture in figure 1 illustrates the proposed connections between the MicroStamp11 and the serial-to-parallel interface. The lab uses the *MicroStamp11's synchronous serial (SPI) subsystem* to drive a *shift register* that will serve as our serial-to-parallel device.

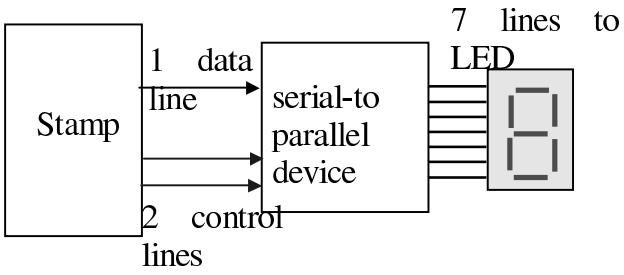

FIGURE 1. Serial-to-parallel interface

This lab also introduces another enhancement to the LED display. The shift register used in this lab is unable to source a great deal of current. So if we were to drive the LED display directly from the shift register (as shown in figure 1), then the lit LEDs would be very dim. In order to brighten up the display, we need to drive the LEDs with more current. In this lab, we will do this by using a special *driver* integrated circuit that implements an array of transistor current drivers. The driver IC will be connected in series between the shift register and the LED display as shown in figure 2.

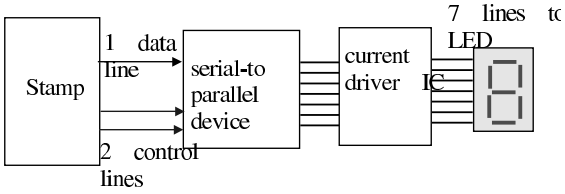

FIGURE 2. Block diagram of proposed system

3.1. What is a serial interface? A *serial interface* is a communication interface between two digital systems that transmits data as a *series* of voltage pulses down a wire. A "1" is represented by a high logical voltage and a "0" is represented by a low logical voltage. Essentially, the serial interface encodes the bits of a binary number by their "temporal" location on a wire rather than their "spatial" location within a set of wires. Encoding data bits by their "spatial" location is referred to as a *parallel interface* and encoding bits by their "temporal" location is referred to as a *serial interface*. Figure 3 graphically illustrates the difference between these two communication methods.

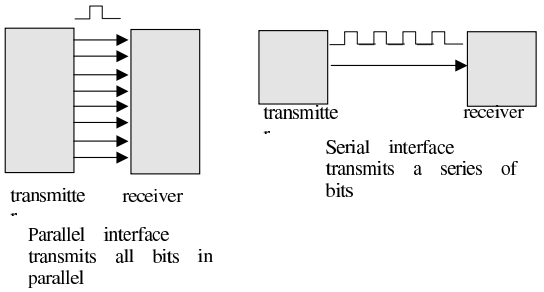

FIGURE 3. Difference between parallel and serial interfaces

A key issue with a serial interface is knowing where the data is on the wire. As an example, let's assume that the wire is initially at a low logical level. We'll refer to this as the *idle channel condition*. If we now transmit a string of zeros down the wire, how can we distinguish between the string of zeros and the idle channel condition?

The answer to our dilemma lies in creating a *protocol*. A protocol is an agreement between two parties about how the two parties should behave. A communication protocol is a protocol about how two parties should speak to each other. Serial communication protocols assume that bits are transmitted in *series* down a single channel. A serial *protocol* has to address the following issues

- How does the receiver know when to start looking for information?
- When should the receiver look at the channel for the information bits?
- What is the bit order? (MSB or LSB first)
- How does the receiver know when the transmission is complete?

These issues can be addressed in a variety of ways, but we can usually identify two distinct approaches. The first approach is embodied in *synchronous serial interfaces* (usually abbreviated as SPI) and the second is in *asynchronous serial interfaces* (usually abbreviated as SCI). Asynchronous serial links are commonly used to communicate between two computers. You used the SCI interface when you used `OutString` to write out characters to the PC's terminal window. The synchronous serial link (SPI) is used when you transmit data between devices that may not have an internal clock. The SPI interface is what you'll use in this lab because the parallel-to-serial shift register you're using has no internal clock.

Asynchronous (SCI) Serial Interface: In an *asynchronous serial interface* (SCI), data is transmitted in well-defined *frames*. A *frame* is a complete and nondivisible packet of bits. The frame includes both *information* (e.g., data) and *overhead* (e.g. control bits). In asynchronous serial protocols the frame often consists of a single start bit, seven or eight data bits, parity bits, and sometimes a stop bit. A representative timing diagram for a frame that might be used by an SCI interface is shown in figure 4. In this figure, the frame has one start bit, seven data bits, one parity bit, and one stop bit. Most of the bits in this frame are self-explanatory. The start bit is used to signal the beginning of a frame and the stop bit signals the end of the frame. The *parity* bit is a special bit that is used to detect transmission errors.

In an asynchronous serial interface, the reading of the data line is initiated by detecting the *start bit*. Upon detecting the start bit, the receiver then begins reading the "data" bits from the line at regular intervals

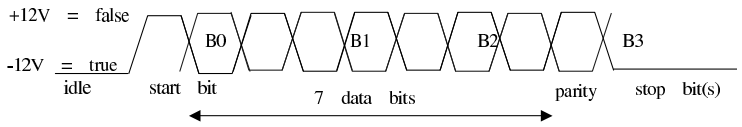

FIGURE 4. RS-232 Frame (1 start bit, 7 data bits, 1 parity bits, and 2 stop bits)

determined by the receiver's clock. This means, of course, that the transmitter and receiver must have a prior agreement on the rate at which data is to be transmitted.

The issue of "when" to look for data bits in the frame must be agreed upon prior to establishing the link. Asynchronous serial protocols usually require that information bits be transmitted at regular time intervals. For instance if we have a 2400 kbaud modem, then both receiver and transmitter know that they should look for information bits arriving at a rate of 2400 thousand bits per second.

The SCI interface is said to be *asynchronous* because both devices do not need to *synchronize* their clocks before communicating. The receiver simply waits for the start bit and then beginnings reading the data line at the agreed upon baud rate. What this means is that the transmitter can transmit a frame without waiting for the receiver to explicitly synchronize to the transmitter's clock. In other words the receiver can receive data in an *asynchronous manner* from the transmitter.

Synchronous Serial Interface: In a *synchronous serial interface*, the receiver has no internal clock. This means that the receiver cannot *independently* synchronize its reading of the data line with the transmitter's transmission rate. The receiver needs some help and that help comes in the form of a clock signal that is shared by the transmitter and receiver. The clock signal acts a control line that tells the receiver when to read from the data line. What this means is that the transmitter and receiver must synchronize their access to the data line in order to successfully transmit data.

SPI interfaces are used when the micro-controller has to transmit data to a device without an internal clock. This is precisely the situation that occurs when we use the MicroStamp11 to transmit data to the shift-register. The MicroStamp11 has an internal clock, but the shift-register has no clock. We usually think of the device with the clock as a *master* and the other device as a *slave*. So in our case the MicroStamp11 is the master and the shift-register is the slave. Typically the slave uses the master's clock to shift data into or out of the slave. This means that the SPI serial channel needs a minimum of two lines. The primary two lines are somtimes referred to as the *data* and *clock* lines. The data line actually has the data bits and the clock line carries clock pulses telling the slave when to read/write the data bits. The value of this approach is that the slave can be a rather simple, inexpensive, and low power device. The disadvantage is that the SPI interface will need control lines in addition to the data line. The SCI interface, on the other hand, only needs a single data line. In this lab, the SPI interface will need three lines (see figure 1); one data line, one clock line, and an additional line that is used to control the internal state of the shift register. Details on how to use the MicroStamp11's SPI subsystem are discussed in the next subsection.

3.2. How do I use the MicroStamp11's SPI subsystem? The MicroStamp11's SPI interface uses four pins. These are pins 15-18. They correspond to bits PD2 through PD5 on PORTD. The *clock* line comes out of pin PD4 with the logical name `SCK`. This line is a 50 percent duty cycle clock whose rate can be controlled by the programmer. There are two *data* lines. The master-out slave-in (MOSI) line is on pin PD3. It is used to clock data out to the slave device from the MicroStamp11. The master-in slave-out (MISO)

line is on pin PD2. This pin is used to clock data into the MicroStamp11 from the slave device. In addition to the clock and data lines, there is an additional control line with the logical name `SS` (slave select) This control line is on pin PD5. The slave-select (SS) pin is an optional control line that can be used when the channel is active. It is often used to signal the end or beginning of a transmission.

Figure 5 shows how the SPI interface is constructed. The data (MOSI/MISO) pins are connected to an 8-bit data register with logical name `SPDR`. When a data transfer operation is performed, this 8 bit register is serially shifted eight positions and the data is transmitted to or received from the slave. Figure 5 illustrates the pins and their connection to the `SPDR` buffer assuming that a serial IC (ADC0831) is clocking data into the MicroStamp11 over the MISO pin.

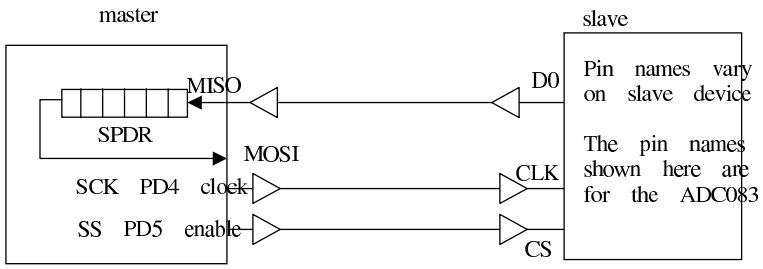

FIGURE 5. MicroStamp11's SPI subsystem

Using the SPI subsystem to output data to the shift register is relatively easy with the *kernel function* provided in the lab. The kernel function is `shiftout()`. This function clocks out an 8 bit frame at a specified rate over the MOSI line. This function is *blocking*, which means that the function will not return until the data has actually been transmitted by the MicroStamp11. We've also included a function `shiftin` that can be used to clock data into the MicroStamp11 from the slave device. A more detailed description of both kernel functions is provided below:

- `void shiftout(unsigned char data, unsigned char rate)`
  Description: This function clocks out a byte of data stored in `data` at the rate specified by `rate`. The function first sets PD3-5 to output and then sets these lines low. The data is then clocked out over pin PD3 using pin PD4 as the clock. After the data has been clocked out the function toggles line pin PD5 (slave-select) to signal to the slave that it is finished.
  Usage: The following code segment

```
digitdata = 0x3F;
shiftout(digitdata,SPI_62kHz);
```

  transmits the binary number 00111111 at 62.5 kHz.
- `unsigned char shiftin(unsigned char rate)`
  Description: This function clock in a byte of data and returns the data as an unsigned character. The data is clocked in at a rate specified in the function's `rate` argument. Prior to clocking in the data, the function toggles the slave select line (PD5) to inform the slave that it is ready to receive data.
  Usage: `ddata=shiftin(SPI_62kHz);`

For both functions the legal values for the `rate` argument are defined in the following table.

logical name	transfer frequency
`SPI_1MHz`	1 MHz
`SPI_500kHz`	500 kHz
`SPI_125kHz`	125 kHz
`SPI_62kHz`	62.5 kHz

3.3. What is a serial-to-parallel device? A serial-to-parallel device accepts a series of timed pulses and latches them onto a parallel array of output pins as shown in figure 1. This lab uses a rather simple integrated circuit (IC) known as an 8-bit serial-in/parallel-out shift register (74HC595) as the serial-to-parallel device. Figure 6 shows the pin out for the IC and shows how to connect the chip to the MicroStamp11 and the LED display.

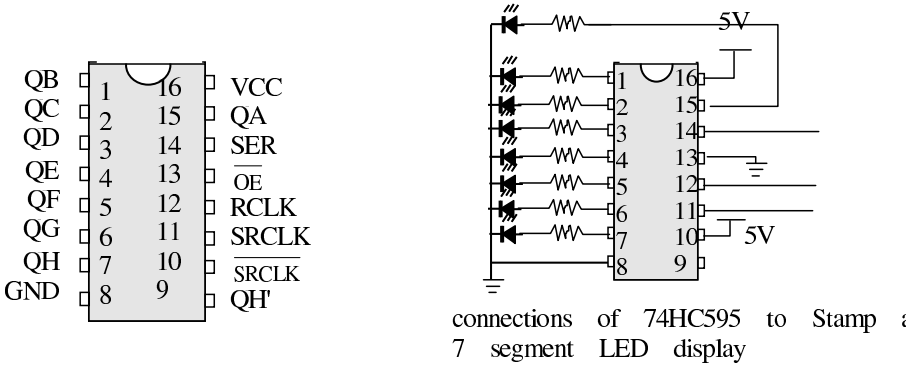

FIGURE 6. Serial-to-Parallel IC chip

The basic idea is to send a sequence of bits (i.e., positive pulses) down to the serial-to-parallel chip (74HC595). The 74HC595 will then convert this serial input into a parallel input and latch the output so it continues to drive the LED display until the next series of bits is received and latched. This operation, however, requires some degree of control. For instance, we need some way of telling the chip when the pulses are expected to occur and we need some way of telling the chip when to latch the data. This means that we need two *control* lines to the chip, in addition to the data line.

To understand how these control lines work, we need to take a closer look at how the 74HC595 functions. The 74HC595 consists of an interconnected set of simpler digital circuits known as *latches*. There are actually two banks (columns) of latches, each column consisting of 8 latches. Figure 7 shows these two columns. The first column of latches form something known as a *shift register*. This bank accepts the serial input and shifts each bit in the series of input pulses down into the column. The second column of latches is used to store what is in the first column. This second column is connected to the chip's output and is used to drive the LED display. This frees up the first column to receive another series of inputs, without disrupting the actual signals delivered to the 7-segment display.

Each latch is a digital circuit with two inputs and one output. The top input with the black arrowhead represents the data and the other input with the white arrowhead represents a control. The lines with the

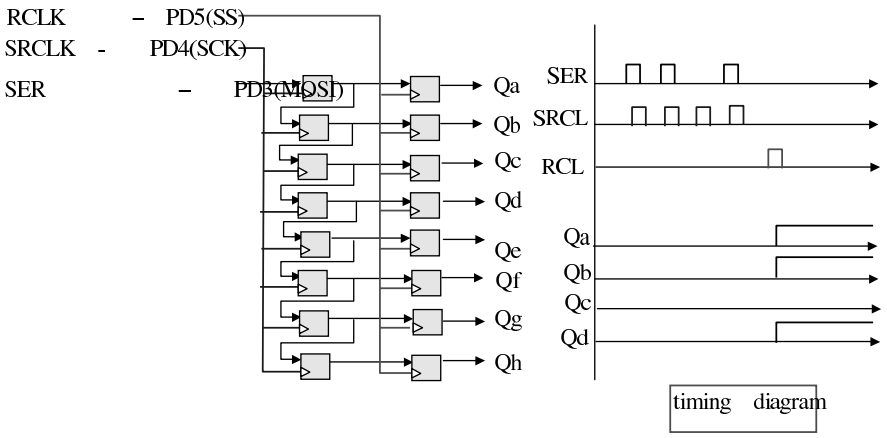

FIGURE 7. Timing diagram for the serial-to-parallel chip

black arrowhead are connected to the pin marked SER (serial input) and the lines with white arrowheads are connected to the pin marked SRCLK. This line is sometimes called the clock line. When the data line (SER) is high and the control (SRCLK) is high, then the output of the latch is also high until it is reset. When the data (SER) is high and the control (SRCLK) is low, then the output does not change from its previously latched value. The first column of latches are daisy chained together so the output of the top latch is the data input to the second latch. What we do, therefore, is input a bit string into SER along with a SRCLK signal. Each time SRCLK goes high, the data in a latch is shifted to the latch below it. We can therefore enter a string of 8 bits into the first column of latches by sending each bit along the SER line and then setting the SRCLK line high for each bit of data. The timing diagram shown in figure 7 shows how the bit string 1011 is shifted into the device. After the data has been shifted into the first column, we transfer this data in parallel to the second column of latches by setting the `RCLK` line high. This second column of latches will retain their value until we overwrite them with another `RCLK` signal.

In your design, you will need to connect the Microstamp11's data pin (MOSI) to the serial line ping (SER) on the latch. The clock signal is produced by the Microstamp on pin SCK. You need to connect SCK to pin SRCLK (source clock) in order to clock the data into the latch. Finally, you will need to use a control signal generated by the state select pin (SS) on the microstamp to latch the 74HC595's output. This means you will need to connect pin SS on the microstamp to pin RCLK on the latch.

Note that the kernel function `shiftout()` has been written to coordinate the clock signal on `SRCLK` with the latch signal generated by the slave select line (`SS`) connected to the shift register's `RCLK` pin. You might find it interesting to examine the source code of the `shiftout` function and see if you can explain how it works.

3.4. How can I make the LEDs brighter? The very last part of the project involves finding a *safe* way to drive the LED display. In our previous lab we used a rather high resistance value (2.2 k-ohm) to limit the current drawn by the seven segment display. The problem with this is that the display is dim since there is very little current passing through it. We can brighten the display by using a smaller resistor (say 100 ohms), but this would increase the total current drawn by the LED display by an order of magnitude.

This is too much current for the μStamp11 or the serial-to-parallel chip to source safely. We therefore need to find a better way of driving the LED display.

To interface our LED to the serial interface, we'll use a *high-voltage high-current Darlington transistor array* (ULN2003). This integrated circuit consists of seven so-called Darlington pairs (a special transistor circuit that can be used as a high current source). The particular chip we're using works well driving a circuit from ground.

The pin-out and connections for the ULN2003 to the display are shown in the figure 8. The ULN2003 is a 16 pin DIP. The inputs (pins 1-7) are all on the lefthand side of the chip and the outputs (pins 10-16) are on the right-hand side of the chip. To work effectively, we need to tie pin 8 to ground and pin 9 should be tied to a supply voltage above 5 volts. We've connected the LED's to the device output via current limiting resistors. Because the ULN2003 can source much more current than the 74HC595 or the μStamp11, we can safely use a much smaller resistor in order to make the displays much brighter. In my version of this circuit I used a 100 ohm resistor.

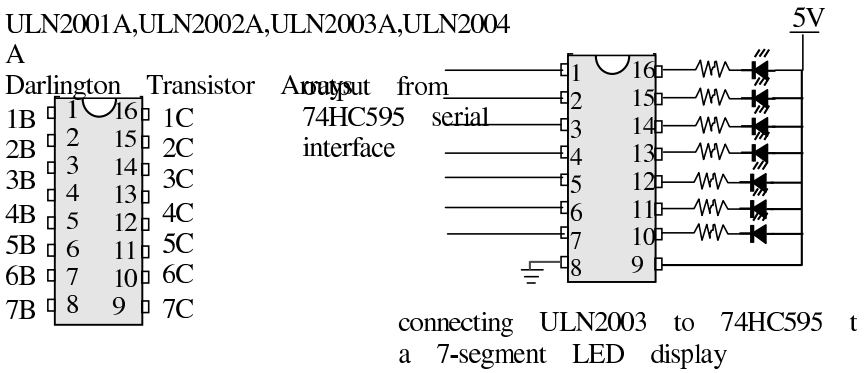

FIGURE 8. Darlington array drivers

This approach to interfacing isolates the μStamp11 from the actual devices it is driving. This can be very important if we are driving inductive loads such as motors. Due to the inductive nature of a motor, we can expect large current transients that can damage the micro-controller. For this reason we often use separate power supplies for motors and other peripheral devices in addition to driving these devices through buffering device such as the ULN2003.

4. Tasks

4.1. Pre-lab Tasks:

(1) Draw the labelled schematic diagram of the modified LED display driver circuitry and draw a picture of the breadboard layout you plan to use.
(2) Rewrite the final program from lab 7 so it uses the SPI interface to drive the LED display. You can assume that the SPI interface is running at 62.5 kHz.

(3) Draw a timing diagram for the signal you expect to come out of the MOSI line for each of the 10 possible display values. Make sure you include a clock signal at the top to reference the waveform against.

4.2. In-lab Tasks:

(1) Measure the current being drawn by one of the display segments for the system you built in the preceding lab.
(2) Modify your current breadboard to implement the LED display circuit you designed in the Pre-lab. Have the TA double check your wiring before starting to test your circuit.
(3) Compile and download your program into the MicroStamp11.
(4) Use the oscilloscope to display the signal coming out of the MOSI line for each of the 10 possible display values. It is recommended that you use the SS pin to trigger your scope.
(5) Measure the current being drawn by one of the display segment for your modified system.
(6) Repeat tasks 4 and 5 for faster SPI transmission rates. You only need to sketch the waveform for one of the numbers being transmitted.
(7) Description of what happened during the in-lab task.

4.3. Post-Lab Tasks:

(1) Compare the currents being drawn by each LED in your system. Determine whether the LED display is brighter.
(2) Compare your predicted MOSI waveforms against the waveforms you actually measured.
(3) Assess how fast you can drive your new LED display. Discuss what happens as the SPI rate increases.
(4) Demonstrate your project's functionality to the TA.
(5) Please provide a list of constructive recommendations concerning the whole course that can be incorporated into next year's labs.

5. What you should have learned

After completing this lab the student should know:

- the difference between serial and parallel interfaces,
- the difference is between synchronous and asynchronous serial interfaces,
- how the shift-register serial-to-parallel interface works,
- how to control the shift-register interface using the MicroStamp11's SPI subsystem,
- and how to safely drive LEDs using a high-current driver IC.

6. Grading sheet

LAB 9 - Serial Interfaces

10 Pts	**PRE-LAB TASKS**	
3	Schematic diagram of LED display circuitry, breadboard layout, and explanation of how the circuit works.	
3	Program listing of lab 7s program (modified to use the serial interface) and explanation of how the modified program works.	
3	Labelled drawing showing the expected oscilloscope waveform you expect to observe for each of the possible inputs to the serial LED display. (62.5kHz)	
1	TAs verification of your pre-lab work.	

15 Pts	**IN-LAB TASKS**	
3	Explanation of what happened in the lab.	
2	Current measured on one of the LED segments BEFORE changing display circuitry.	
2	Current measured on one of the LED segments AFTER changing display circuitry.	
3	Sketch of oscilloscope traces for serial lines waveform for each of the possible display values (62.5 kHz)	
5	Sketch of oscilloscope traces for faster serial rates.	

10 Pts	**POST-LAB TASKS**	
3	Table comparing LED currents before and after circuit change. Explanation of tables significance.	
3	Comparison of predicted and measured oscilloscope traces (62.5 kHz)	
2	Assessment of oscilloscope traces for faster serial rates.	
2	Constructive Comments about whole lab course.	

2 Pts	**DEMONSTRATION ..... STOP if not checked off**	
2	Successful demonstration of completed system.	

GO ON to next lab if completed.

Name _____ .

Lab Group Number _____ GRADE _____ out of 37

TA signature _____ Date _____ .

CHAPTER 10

Analog to Digital Conversion Revisited - Time Multiplexing

1. Objective

The last three labs enhanced specific subsystems of the ADC system you built earlier. These enhancements reduced the number of I/O lines required by your system and they increased the precision of your DAC. In this lab you will combine these improvements to complete a successive approximation ADC that converts an analog reference voltage into a 6-bit integer. In order to display this 6-bit integer on your LED display, you will need to expand your LED display from a single-digit display to a *time-multiplexed* two-digit display

2. Parts List

(1) *wire kit and breadboard*
(2) *MicroStamp11, docking module, and serial cable*
(3) *Laptop with ICC11 compiler and PMS91 loader*
(4) **two 7-segment LED display (LSD3221-11)**
(5) seven 100 k-ohm resistors
(6) **two 2n4401 transistors**
(7) **two 5 k-ohm resistors**
(8) *two 10 k-ohm resistor*
(9) *two buttons*
(10) *one LM660 quad op-amp IC*
(11) *two resistors (R) for op-amp buffer*
(12) *two 1n4007 diodes*
(13) *one 10 k-ohm trim potentiometer*
(14) *two additional resistors for diode clamp circuit*
(15) *one ULN2003 Darlington array driver IC*
(16) *one 74HC595 serial-to-parallel IC*

3. Background

In this lab the student will build an analog-to-digital converter that transforms a reference voltage into a 6-bit digital number. The converted number will then be displayed on the LED display. The problem we face is that our current display is a single-digit display that can only display voltage levels between 0 and 9, a precision of only 3 bits. Since our ADC provides more than 3-bits of precision, we'll need to modify our LED display into a two-digit display.

A two-digit display can be built using the same basic principles employed in the previous lab. In particular, we propose using *time* in a controlled manner so that the MicroStamp11 first drives one digit of the display and

then drives the other digit of the display. This approach is referred to as *time-multiplexing*. Time-multiplexing has the micro-controller switch back and forth between the multiple devices by using an *electronic switch*. In this lab the student will add an extra digit to the current display, use an electronic switch to alternately control each digit of the display, and then *complete* the hardware and software parts of the ADC. Finally, the student will test the ADC by verifying the correctness of the converted values for constant and time-varying reference voltages. For the last test, you will need to use another piece of electronic test equipment known as a *function generator*.

3.1. What is time multiplexing? Let's return to the design we used for a single digit display in lab 4. In that system, we used one MicroStamp11 output line to drive each LED segment. To display a particular number, we needed to set seven output lines at the appropriate level. If we now try to extend this approach to a dual digit display, then we would need to use 14 output lines to drive the display. Since the MicroStamp11 only has 11 output lines, we have a problem.

The usual solution is to build a multiplexed display. In a multiplexed display we connect all like segments of the digits together. In this way, no matter how many digits we have in our display, we only need the seven wires. To decide which display to turn on, we need a switch that connects the common anode terminal of the LED display to +5 volts. We would therefore need to have an additional wire for each digit of our display. A simple schematic diagram illustrating the ideas behind a multiplexed display is found in figure 1.

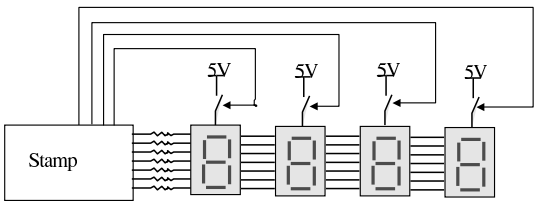

FIGURE 1. Multiplexed Display

The proper operation of a multiplexed display depends upon a feature of human visual perception known as *flicker fusion*. If a light is flashed quickly enough, individual flashes become imperceptible and the illusion of a steady light is created. This is the basis for movies, television, and fluorescent lighting. All of these devices flash quickly enough to give the illusion of steady light or movement. How fast must a light flash before flicker fusion occurs? As a general rule of thumb, any light flashing faster than 50 times per second will appear to be steady. But you will verify this number as one of the in-lab tasks.

Multiplexed displays only show one digit at a time, but by cycling through all digits repetitively and cycling very fast, a multi-digit display is perceived. To drive a particular digit, its common anode is connected to the appropriate supply voltage and the segments are driven as needed for the desired number in that place. The cycling through the digits must be done quickly enough for flicker fusion to occur and this is accomplished through the system's software.

3.2. What are electronic switches? From the preceding section, we saw that we can create a multiplexed display by switching on the individual digits one at a time. We can of course, use the MicroStamp11 to directly drive the common anode of the LED's, but this is a dangerous solution. Remember that we've designed the LED circuit so it draws a large current. This means that if we were to directly drive the LED's from the MicroStamp11, then we would probably draw more current than the MicroStamp11 could

safely source. In other words, we would probably destroy the MicroStamp11. So we need to find a way of controlling the switching process without actually using the MicroStamp11 to drive the LEDs. This can be done through a simple transistor switch.

In particular, we may use a 2N4401 general purpose transistor as a switch. A transistor is a three-terminal semiconductor device that can be used as either an *amplifier* or *switch*. The operational amplifier you used earlier is a very complex transistor circuit that has been specially designed to provide the high gain, high input resistance, and low output resistance that characterize an op-amp. Transistors also can be used as electronic switches. Such switches lie at the heart of all digital logic circuits such as the simple shift-register you used earlier as well as the MicroStamp11 itself. These multiples uses of the transistor make it one of the most significant technological developments of the 20th century. Its invention effectively enabled the information age that marked the beginning of the new millennium.

A transistor has three terminals. The earliest transistor technologies were based on creating a sandwich of n and p-type semi-conductor materials. These so-called *npn* or *pnp* bipolar junction transistors (BJT)'s are still used today. The 2N4401 is a BJT. The lefthand drawing in figure 2 shows the electronic symbol used for an *npn* BJT. From this figure you will see that the three terminals for the transistor are referred to as the *collector*, *emitter*, and *base* terminals. The physical device is extremely small. It is a small cylinder that has one side flattened. The flattened side is used to help determine which of the three leads is the base, emitter, and collector. A drawing of the physical device is shown on the righthand side of figure 2.

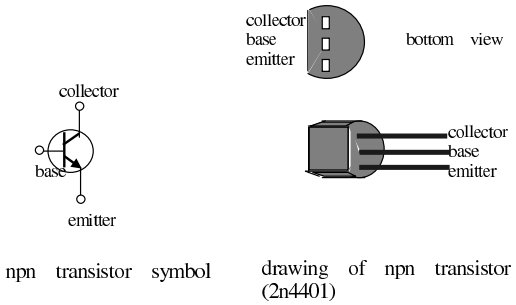

FIGURE 2. 2N4401 T ransistor

Transistors can be connected in a way that allows a relatively small base current to either switch on or off a relatively large collector current. Such a connection is shown in figure 3. In this figure, a 5 k-ohm resistor is connected in series to the base terminal of the transistor. The collector terminal is connected to the positive supply voltage of +9 volts and the emitter terminal is connected to the common anode of the 7-segment LED display. When the voltage level on the base terminal is low, then, the transistor switch is closed and a current flows from the nine volt supply, through the LED to ground. When the voltage level on the base terminal is high, then the switch is open and no current flows through the LED. The size of the emitter-collector current is controlled by the 100 ohm resistors of the LED driver circuit you designed in the last lab. Because of the small size of these resistors, the emitter-collector current is large when the transistor switch is closed and the LED's are bright. The base-emitter current is determined by the relatively large resistor (5 k-ohm) on the base terminal. Because of the large size of this resistor, the base-emitter current can be kept to low levels that can be tolerated by the MicroStamp11.

3.3. How to complete the design? Completing the hardware side of the ADC design is relatively easy. Most of the circuits have already been either constructed or discussed. From Lab 8, you constructed

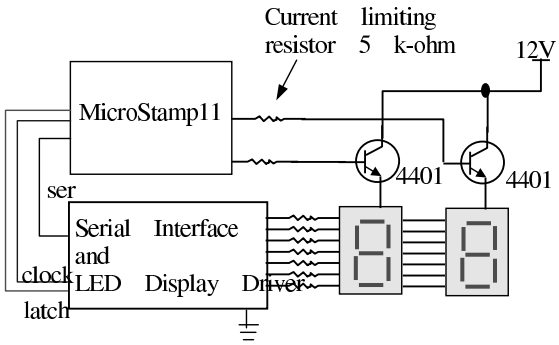

FIGURE 3. Schematic of Display Circuit

the RC-DAC. This will need to be connected to the ADC test circuitry you constructed in lab 5. After this is done, you will need to build the multiplexed LED display. Finally, you will need to modify the successive approximation ADC program you wrote in Lab 6 so that the program drives the RC-DAC (rather than the R2R ladder DAC), the binary search converges to a 6-bit digital number (instead of 3-bit), and the quantized number is used to drive the dual-digit (rather than single digit) display.

3.4. What is a function generator? A *function generator* is a piece of electronic test equipment that generates a variety of periodic voltage signals such as sine waves and square waves. There are three controls on the face of the device that allow the user to control 1) the type of waveform (sinusoid or square), 2) the frequency of the periodic signal, 3) and the amplitude of the signal. There are two terminals on the control panel. The black terminal is for ground and the red terminal is the desired voltage waveform.

4. Tasks

4.1. Pre-lab Tasks:

(1) Draw a schematic diagram of the dual digit display circuitry and draw a picture of your proposed breadboard layout. Explain how the time multiplexed display works.
(2) Draw a schematic diagram of your completed ADC circuit and draw a picture of your proposed breadboard layout. You need not redraw the display circuitry, simply denote it as a *block* within your drawing.
(3) Rewrite the ADC program you created earlier so it converts a constant reference voltage into a 6-bit digital number and outputs this number to the dual digit display you designed. Describe how this program differs from the original program.
(4) Predict the converted voltage as a function of the reference voltage and plot this relationship.

4.2. In-lab Tasks:

(1) Modify breadboard to implement the display circuitry you designed in the pre-lab and have the TA double check the correctness of the circuit before continuing.

(2) Make the last few connections that connect the RC-ADC and clamp circuit to the MicroStamp11. This should complete the hardware part of the ADC design. Compile and download your ADC program into the MicroStamp11.
(3) Record the display measurements for at least 32 different constant reference voltages.
(4) Use a function generator to generate a square wave reference voltage. Record the behavior of your display for a number of frequencies between one and 1000 Hz.
(5) Modify your program to change the display's refresh rate. You should examine at least 10 frequencies between 10 and 100 Hz. For each of the frequencies determine whether or not flicker-fusion occurs.
(6) Describe what happened during the In-lab task.

4.3. Post-Lab Tasks:

(1) Compare the quantized measurements you observed for in-lab task 3 against your pre-lab predictions. Assess how well your ADC converts constant reference voltages.
(2) For the observations and measurements you made for in-lab task 4, plot the maximum observed voltage as a function of frequency. Use frequency domain concepts to explain your circuit's observed behavior. This task essentially asks you to evaluate the frequency response of your ADC.
(3) Use the data for in-lab task 5 to determine lowest refresh rate, above which flicker-fusion occurs.
(4) Demonstrate the functionality of your completed system to the TA. The TA will also check the completeness and correctness of your lab book, before signing off on the lab.

5. What you should have learned

After completing this lab the student should know:

- how to construct a multiplexed display,
- how to use transistors as switches,
- the importance of using "time" to control a circuit's operation,
- and what is meant by a system's frequency response.

6. Grading sheet

LAB 10 - Analog to Digital Conversion Revisited Time Multiplexing

10 Pts	**PRE-LAB TASKS**
3	Schematic diagram of dual digit display circuitry, breadboard layout, and explanation of how the circuit works.
2	Schematic diagram of completed ADC hardware and breadboard layout.
2	Program listing for revised ADC program and explanation of how it works.
2	Analysis predicting the quantized voltage levels as a function of the reference voltage level.
1	TAs verification of your pre-lab work.

10 Pts	**IN-LAB TASKS**
3	Explanation of what happened in the lab.
1	TA check of breadboard prior to testing.
2	Observed LED display values for at least 32 different constant reference voltages.
2	ADC/square wave results General observations for at least 10 frequencies
2	ADC/square wave results table of maximum value versus frequency.

8 Pts	**POST-LAB TASKS**
2	Final Program listing and explanation of how it works, focusing on differences from pre-lab listing.
1	Graph of quantized display value versus reference voltage (constant case).
1	Assessment of ADC performance for constant reference voltages.
2	Graph of maximum quantized value versus frequency (square wave case)
2	Assessment of ADC performance for time-varying reference voltages.

2 Pts	**DEMONSTRATION ..... STOP if not checked off**
2	Successful demonstration of completed system.

GO ON to next lab if completed.

Name _____ .

Lab Group Number _____ GRADE _____ out of 30

TA signature _____ Date _____ .

CHAPTER 11

IR Communication Link

Preceding lab projects used serial communication protocols (SCI or SPI) to transfer data between computers and other digital devices. In both of these protocols, data was transmitted over a *wire*. The objective of the next couple of labs is to explore wireless communication using an **infra-red** or IR communication link. In this lab you'll construct the comm-link's hardware.

1. Lab Objective:

Build an IR communication link that transmits and receives a 38-40 kHz carrier wave. Your system should begin transmitting a 3-bit integer (0-7). You should be able to increment and decrement the integer you're sending through the use of increment and decrement buttons. You will need to view the received output signal to make sure that the data you sent was correctly received.

2. Communication Systems

The serial communication protocols you used in the last project transmitted a series of voltage pulses down a wire. Each pulse represented a *bit*. When you attached an oscilloscope to this wire and triggered correctly, then the observed trace might have looked something like the trace in figure 1.

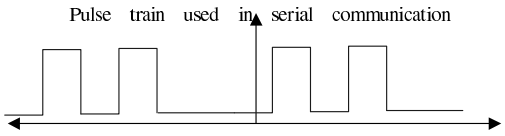

FIGURE 1. Signal waveform

There are, however, other ways of transmitting information besides toggling between zero and 5 volts. If we had *abstracted* the communication systems you used earlier to extract their essential conceptual components, we would end up with a block diagram something like that found in figure 2. In this figure, we find an information source, whose information is *transformed* into a signal that can be transmitted through a physical *channel*. On the other end of the channel, the received signal is transformed back into information that can be directly used by the *destination*.

The abstracted blocks in figure 2 (source, transmitter, channel, receiver, and destination) all have concrete implementations in the system you built in lab 8. The information source was the MicroStamp11 program

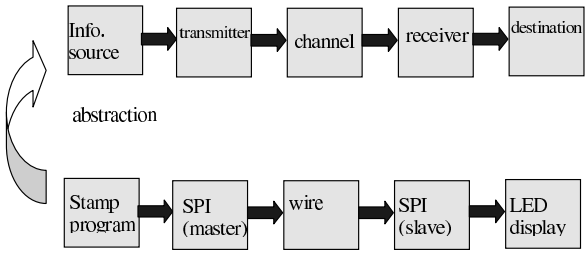

FIGURE 2. Communication System

used to increment and decrement a desired integer. The channel consists of the wires between the microcontroller and the slave device. Finally, the destination was that two digit LED display you built.

The block diagram shown in figure 2 represents a high level abstraction of a concrete communication system. But given this abstraction, we can substitute lab 8's realization for these blocks by other concrete realizations and thereby obtain a different type of communication system. The *different* type of communication system we'll consider in this lab is a *wireless* communication system.

3. The Channel

The channel is the medium through which we transmit information. There are essentially two types of channels. The "wired" channel consists of media such as copper wire, co-axial cables, and fiber optic cables. The "wireless" channel consists of media such as air, water, vacuum, or earth.

Lab 8 transmitted information through a copper "wired" channel. In this case, we simply applied a voltage at one end of the wire. The receiver at the other end of the wire would detect the potential drop. In a wireless channel, information propagation is more complex. In most cases, wireless channels rely on *wave dynamics* to transmit energy through the medium. As an example, we might consider a channel such as the atmosphere and the propagation of light across this channel. Light is an electro-magnetic (EM) wave. Essentially, this means that we have electrical and magnetic fields that pass energy back and forth between each other. The dynamics of this interaction are such that the wave propagates over a distance. Other wireless channels supporting either acoustic (air/water) or seismic (earth) waves rely on similar dynamical mechanisms for wave propagation

A channel is not an ideal medium for energy transmission. Channels often distort the signals that pass through them. The channel can add "noise" to the signal. In particular, we can think of the channel (such as the atmosphere) as a "system block". The input into the block is the transmitter's output signal and the output of the channel is then input into the receiver. If the channel can be represented as a linear system, then we may relate the channel's input $x(t)$ to its output, $y(t)$, through the following equation

$$Y(s) = H(s)X(s) + N(s) \tag{3.1}$$

where $X(s)$ and $Y(s)$ are the Laplace transforms of the input and output, respectively. $H(s)$, of course, is the *transfer function* for the channel and $N(s)$ is an additive noise term.

Recall from the last chapter that $|H(j\omega)|$ is network's frequency response. So, for instance, we could graph the spectrum of a wireless channel as a function of the frequency of the EM wave that's propagating over the

channel. One possible frequency response is shown below in figure 3. What you'll see is that the atmospheric channel's transfer function has several "holes" in it. The holes refer to places where the atmosphere absorbs EM radiation. Obviously, if we were to transmit a signal in a frequency range covered by one of these holes, then very little of the signal would reach the receiver. So the transmitter usually encodes the information signal onto a carrier wave whose frequency sits in one of the frequency "windows" in which the atmosphere is nearly transparent. One of these particular windows occurs in the **infra-red** (IR) range of the EM spectrum, which is why we're building an IR wireless link in this lab.

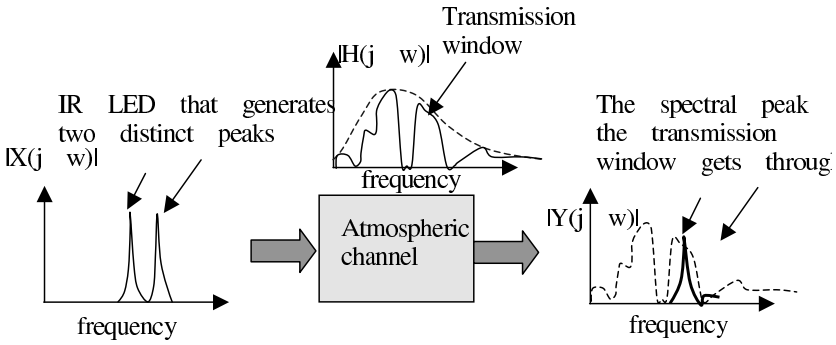

FIGURE 3. Atmospheric Channel's Frequency Response

4. Transmitter

The purpose of the transmitter is to transform the information we want to send into a signal that can be propagated by the channel. In the case of our wired copper channel, this means we want the information to be transformed into a modulated voltage level, something like the pulse train in figure 1. For a wireless channel, however, the transmitter needs to encode the information onto an EM wave that can be easily propagated. In this lab, this means that the transmitter needs to take the voltage pulses generated by the μStamp11 and it needs to put them onto an infra-red EM wave. We usually refer to this encoding as *modulation*.

Let $p(t)$ be a time-varying signal representing the bits of information we want to transmit over the channel. A possible waveform for $p(t)$ is shown in figure 1. The transmitter then uses this signal to modulate a sinusoidal (or square) wave with a frequency ω_c. We refer to this sinusoidal wave as the signal's **carrier wave**. For the wireless channel we're building, standard receiver structures usually assume a carrier wave with a frequency between $38 - 44$ kHz. The carrier wave is added to assist in subsequent signal processing at the receiver end. So the information that the transmitter generates has the form

$$x_1(t) = p(t)\sin(\omega_c t + \phi) \tag{4.1}$$

This type of encoding is referred to as **amplitude modulation** or AM because we're modulating the amplitude of the carrier wave. When $p(t)$ is digital data of the form shown in figure 1, we refer to this modulation scheme as **amplitude shift keying** or ASK.

The signal $x_1(t)$ in equation 4.1 is still an electrical signal that has been generated by the transmitter. The transmitter must now use this signal to amplitude modulate the IR beam we intend to transmit through the

channel. So the strength of the IR beam that is actually transmitted by the system has the functional form

$$x(t) = p(t)\sin(\omega_c t + \phi)\sin(\omega_0 t)$$

where ω_0 is the frequency of the IR beam, somewhere on the order of 1-500 TeraHz (10^{12}).

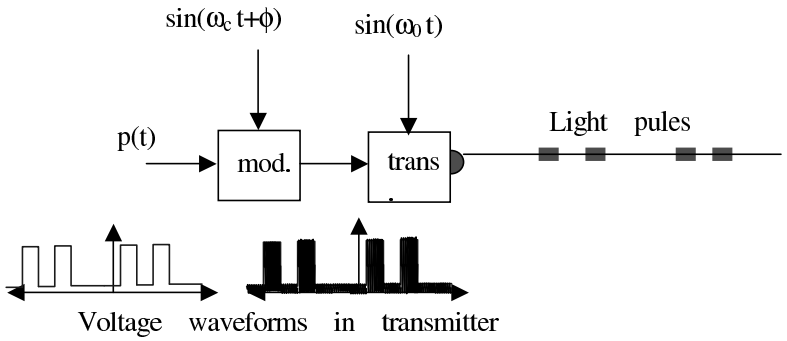

FIGURE 4. Block Diagram of Transmitter

Figure 4 illustrates the steps outlined above. The information signal $p(t)$ is mixed with the carrier wave in the block label `mod`. At the bottom of the figure, you'll see representative waveforms for the voltages in the transmitter that enter and exit the `mod` system block. The output of the modulator is then used by the `trans` (transducer) block in the system. The transducer block uses this voltage waveform to modulate the IR wave. This IR beam is then sent out into the channel (air) to be caught by the receiver.

In this lab, you'll need to build a circuit that performs the basic modulation and transducer functions shown in figure 4. A schematic diagram for the entire transmitter circuit is shown in figure 5. The modulator function is carried out by the 555 timer IC shown in the top part of the schematic. The transducer function is performed by the photo-diode circuit shown on the bottom part of the figure.

The 555 timer IC (TLC555) is a standard IC chip that is used to generate very precise clock pulses. It is also possible to use the μStamp11 to generate the clock pulses, but this puts a heavy burden on the micro-controller. Using the TLC555 IC relieves the micro-controller from this burden, thereby freeing it up to do more useful things. The clock signals generated by the TLC555 are shown below in figure 6. The frequency of this periodic waveform can be controlled by two external resistors and one capacitor. In particular, the positive pulse width generated by the TLC555 can be computed from the equation

$$T_+ = 0.69(R_1 + R_2)C$$

and the negative pulse width is

$$T_- = 0.69R_2C$$

Figure 6 illustrates the pin-out for the TLC555. You usually implement R_1 using a current limiting resistor and a trim pot (see figure 5). You can use the trim-pot shown in figure 5 to adjust the carrier frequency ω_c.

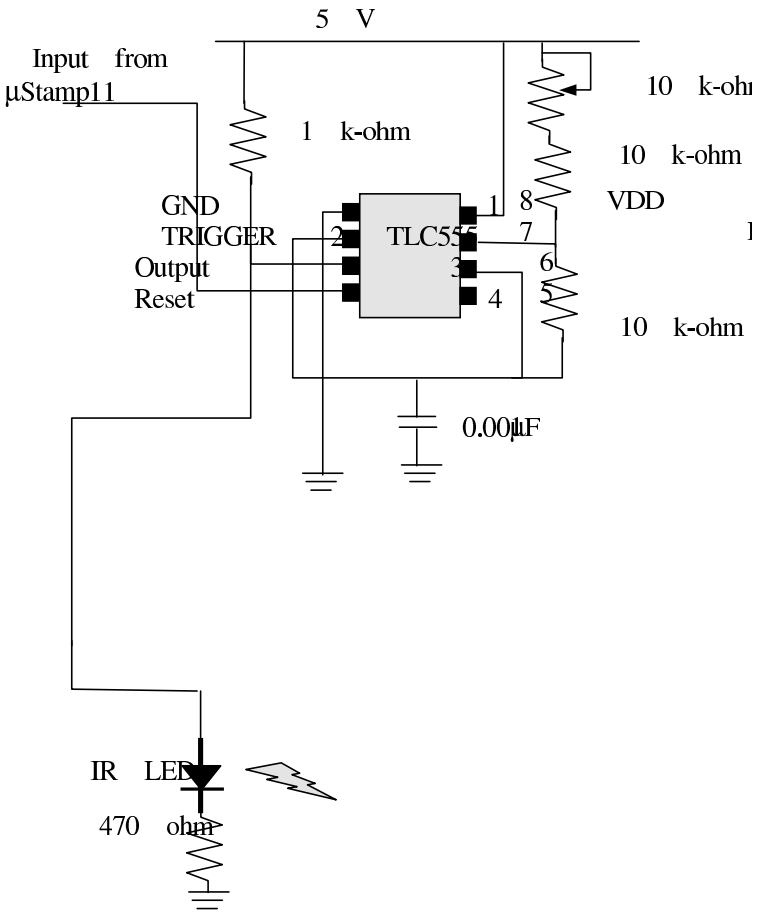

FIGURE 5. Schematic Diagram of Transmitter

5. Receiver

The purpose of the receiver is to translate the received IR pulses into a set of pulsed voltage levels representing information. We can think of the receiver as simply undoing what the transmitter has done as is shown in figure 7.

As can be seen in figure 7, our system consists of two parts. The `trans` subsystem translates the received IR energy into a voltage waveform. Ideally the `trans` subsystem removes the IR frequency from the received signal so we are left with the modulated carrier signal $x_1(t)$ that was originally generated by the transmitter. In practice, the channel may have introduced some distortion so that the received $x_1(t)$ may not look exactly like the transmitter's $x_1(t)$. The modulated carrier is then *demodulated* in the `demod` subsystem. This subsystem removes the carrier wave, so we are left with the original signal $p(t)$ that the transmitter started with.

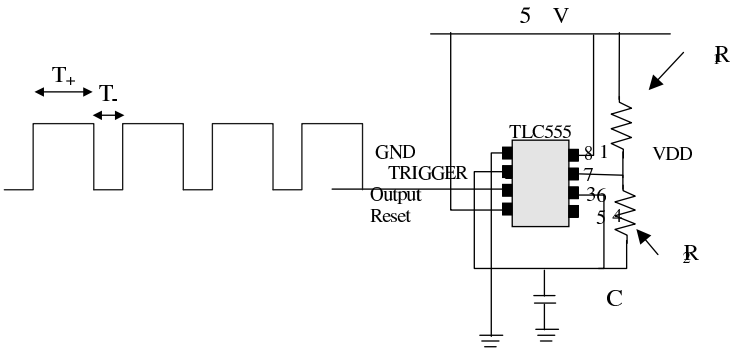

FIGURE 6. 555 Timer IC

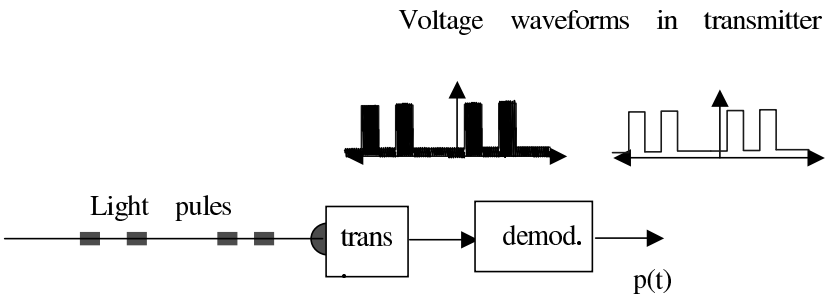

FIGURE 7. Block Diagram of Receiver

While the conceptual function of the `trans` and `demod` subsystems is straightforward, the actual implementation can be somewhat more involved. This is because the receiver's performance must be robust to distortions introduced by the channel. In practice, therefore, the `trans` and `demod` subsystems consist of additional subsystems. In figure 8 we show the block diagram for an integrated circuit (IC) that incorporates the photodetection, transducer, and demodulation functions into a single chip. Let's step through this block diagram.

The IR beam is received by a photodiode shown on the left hand side of figure 8. The output of the photodiode goes through a buffering amplifier which is then fed through a bandpass filter. The purpose of the bandpass filter is to only select out those frequencies in the neighborhood of the carrier wave. So if there are additional frequency components in the received signal (due perhaps to the channel), these are removed by the bandpass filter. The output of the bandpass filter goes through a demodulator. The demodulator consists essentially of a diode whose purpose is to rectify the signal. The rectified signal is sent through a lowpass filter to remove high frequency harmonics that may have been introduced by the rectification function. The final signal leaving the lowpass filter will not be a sharp set of pulses (due to distortion introduced by the channel and our demodulator's filter functions), so we introduce a comparator to "clean-up" the distorted pulses received by the module. The output of the modulator then feeds a single transistor current source.

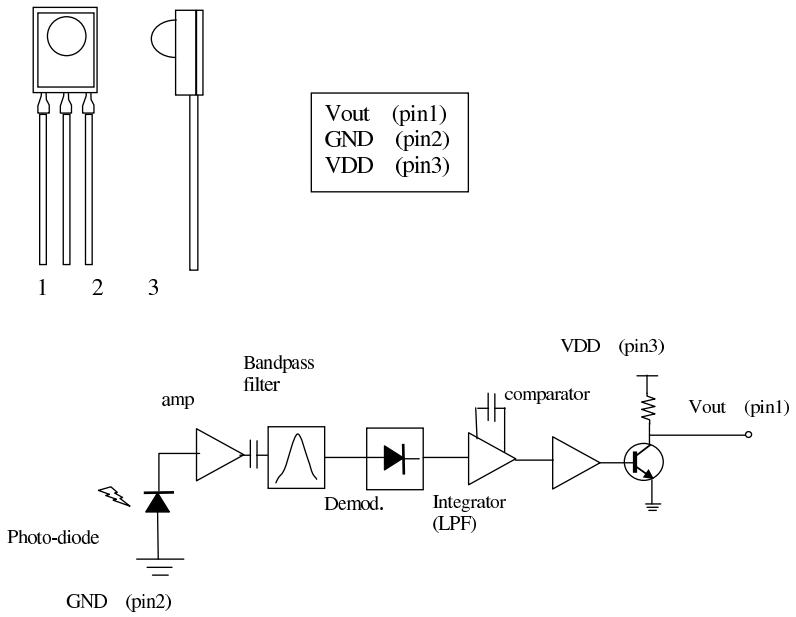

FIGURE 8. PNA4601M Receiver Module

The use of the PNA4601 receiver module makes the design of your IR link receiver very easy. Essentially, all you need to do is connect up the module to a current source and an LED. The basic circuit schematic for the receiver circuit (not including the LED) is shown below in figure 9.

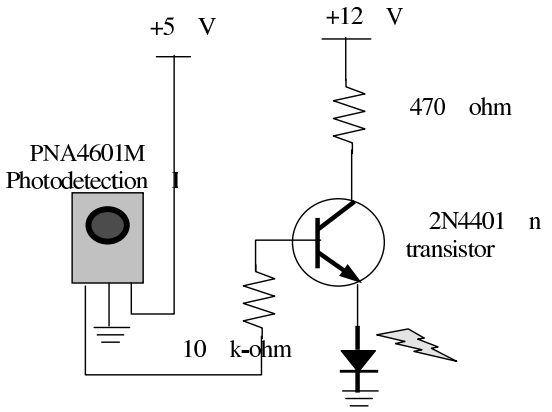

FIGURE 9. Receiver Schematic

6. What you need to do to complete this lab

As before you'll need to pass the pre-lab qualifying exam before you can start the lab. The schematic for most of the transmitter and receiver circuits have already been provided in this lab. Your schematics will need to indicate how you plant to integrate these circuits together with the μStamp11. You'll also need to provide a preliminary listing for your program. In addition to this, we'd like to you *sketch* the expected received signal at the output of the PN4601 module for each of the 8 integers you can send over the link. For example, if you were to send a `0x01`, then you'd expect the output waveform to have exactly one pulse. Be clear in explaining the scheme you used to transform the requested integer into the transmitted bit string.

We've provided a simple `kernel` for this lab that uses the `OC4han` interrupt handler to simply increment a global counter. The kernel updates this counter (`_Time`) once every 256 hardware time ticks. What you'll need to do with regard to the software is

- Display the requested 3-bit integer on one of the LED displays.
- Be able to increment and decrement the requested integer using external buttons.
- Transform the 3-bit integer into a string of pulses that are then transmitted over the wireless link. We recommend that the duration of your bits be about 10 ticks of the `_Time` variable (i.e. use the `pause` command).

In order to complete this lab you will need to demonstrate your circuit. In particular we want you to hook up the output of the PN4601 module to the oscilloscope in order to graphically display the signal being received by your system.

- Demonstrate that your LED activity indicator works. If the channel is active, then the LED should be blinking quickly. If the channel is inactive then the LED should be lit but not blinking.
- Demonstrate that the received signal changes as you increment and decrement the requested button.
- Demonstrate that the requested signal you sent matches the integer you requested to be sent. Do this for each one of the 8 possible integers you can send. Compare it to your pre-lab sketches and explain any differences.

CHAPTER 12

Wireless Serial Link

The preceding lab had you build a simple IR communication link. The only information you transmitted over that link, however, was a 1 or 0. In this lab, we'll try using the IR link for communication. In particular we want you to use the IR link to transmit and receive a 3-bit number. Your system will then display both transmitted and received data on your LED display.

1. Lab Objectives

Build a system that does the following:

(1) Use a pair of buttons to select a 3-bit (0-7) integer.
(2) Transmit that 3-bit integer as an amplitude shift keyed signal over an IR link to a neighbor's system. Display the transmitted integer on the lefthand digit of your LED display.
(3) Receive a 3-bit integer (ASK) from your neighbor and display the decoded value on the righthand digit of your LED display

2. Serial Communication Protocol

A **protocol** is an agreement between two parties about how the two parties should behave. A communication protocol is a protocol about how two parties should speak to each other. Serial communication protocols assume that bits are transmitted in **series** down a single channel. A serial *protocol* has to address the following issues

- How does the receiver know when to start looking for information?
- When should the receiver look at the channel for the information bits?
- What is the bit order? (MSB or LSB first)
- How does the receiver know when the transmission is complete?

The IR link you'll be constructing is an asynchronous serial link and this means that these four issues must be agreed upon by both transmitter and receiver prior to sending any information.

In general, serial links transmit data in distinct *packets* or *frames*. A frame is a set of bits that are transmitted sequentially by the transmitter. In general the frame consists of four types of bits; a start bit, data bits, parity bits, and stop bits. The start bit is used to signal the beginning of a frame. The stop bit is used to signal the end of a frame. The data is contained in the data bits and the parity bit is an extra bit that is often used to detect transmission errors. In this lab, you'll use the MicroStamp11 to transmit a frame that consists of only start, data, and stop bits. You won't be using parity checking to detect transmission errors.

In many serial protocols, the issue of how to detect the start of a transmitted frame uses a special bit known as a **start bit**. Both transmitter and receiver assume that the channel is initially **idle** (i.e. zero volt logical level). The beginning of a frame is then signalled by setting the channel high for a specified length of time and then setting it low again. Transmission of data bits usually commences a specified interval of time after the falling edge of the start bit.

The issue of "when" to look for data bits in the frame must be agreed upon prior to establishing the link. Asynchronous serial protocols usually require that information bits be transmitted at regular time intervals. For instance if we have a 2400 kbaud modem, then both receiver and transmitter know that they should look for information bits arriving at a rate of 2400 thousand bits per second.

One possible way of formulating the protocol is to require that the first data bit occur a fixed time interval after the falling edge of the start bit. As a concrete example, let's assume that the data bits are always an integral multiple of 10 microseconds after the falling edge of the start bit. In this case, the receiver simply needs to look at the channel every 10 microseconds after it has identified the falling edge of the start bit. If the channel is high when the receiver looks, then we record a 1 and if the channel is idle, then we record a zero.

Once you've recovered a series of bits, there is a natural question concerning the order of the transmitted data bits. In other words if each transmission consists of a byte, then is the first data bit received the most or least significant bit of that byte. The receiver and transmitter need to agree upon the bit order prior to transmission so the data bits are properly interpreted.

We can use one or more stop bits to signal the end of a transmitted frame. The stop bits are generated by setting the channel to idle for 1 or more time intervals of T duration. Another approach however, would simply count the number of received bits after the falling edge of the start bit.

To summarize, the serial protocol we recommend you use consists of the following agreements. In this protocol, the variable T is a time interval agreed upon by the receiver and transmitter.

- The transmitter signals the start of a transmission by setting the channel high for $T/4$ seconds. We'll refer to this pulse as the start-bit.
- The kth information bit's rising edge occurs $kT+T/2$ seconds after the falling edge of the start-bit. The kth information bit's falling edge occurs at kT seconds after the falling edge of the start-bit. k ranges from 0 to 2.
- Each transmission frame consists of one start bit, exactly 3 information bits (i.e., k ranges between 0 and 2), and two stop bits.
- The two stop bits may be generated by setting the channel to idle for $2T$ seconds.
- The first information bit after the start-bit is the LSB of the transmitted byte.

The timing diagram for this serial protocol is shown in figure 1.

3. Input Capture Interrupts

To complete this lab you'll need to find a way to detect the start bit of a frame. The start bit is known to be of duration $T/4$. Since we know all other data bits are of longer duration than this, we can reliably detect the beginning of a frame by looking for a short pulse of duration $T/4$ seconds. This means we will need to be able to measure pulse widths. Pulse widths can be measured using **input capture interrupts**. This section discusses the use of input capture interrupts on the MicroStamp11.

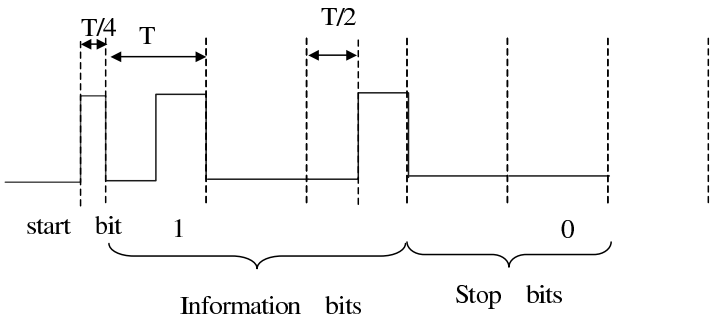

FIGURE 1. Wireless Serial Protocol Timing Diagram

The input capture (IC) event is a hardware event tied to the logical state of one of the input pins on PORTA. We begin by connecting an external TTL level signal to pins PA2, PA1, or PA0 on PORTA. Recall that these pins always have the "input" direction state. Pins PA0, PA1, and P2 are tied to IC events `IC3`, `IC2`, and `IC1`, respectively. If the logical voltage level on one of these pins changes, then the MicroStamp11 latches the current 16-bit value of the timer register `TCNT` into the pin's *input capture latch register*. The logical names for the latch registers associated with input capture events `IC1`, `IC2`, and `IC3` are `TIC1`, `TIC2`, and `TIC3`, respectively.

We can have the input capture event trigger an input capture interrupt if the input capture interrupt bit (`IC1I`, `IC2I`, or `IC3I`) is set in the control register `TMSK1`. So, for example, if we set pin `IC1I` in `TMSK1`, then the occurrence of an input capture event on pin `PA2` will do two things. First it will cause `TCNT`'s current value to be latched into register `TIC1`. Second, it will cause the program to jump to the interrupt vector associated with interrupt `IC1`. The first action essentially records the time when the input capture event occurred. The second action jumps to an ISR that performs the computations required to service the `IC1` interrupt.

Figure 2 shows the three control registers used by the input capture interrupts. These three registers have the logical names `TMSK1`, `TFLG1`, and `TCTL2`. The register `TMSK1` is a control register that is used to "arm the input capture interrupt. Arming the input capture interrupt `IC1`, for instance, is accomplished by setting bit `IC1I` in register `TMSK1`. The register `TFLG1` is a status register that can be used to "acknowledge" the servicing of a caught interrupt. We acknowledge a previously caught IC interrupt such as `IC1` by setting bit `IC1F` in register `TFLG1`. The final register `TCTL2` is used to define the "way" in which an input capture event is generated.

It is rather vague to say that the IC event occurs when the input pin's logical state changes, for it is unclear what we mean by a change of state. Does the voltage rising from 0 to 5 volts (a rising edge) or does the falling voltage (from 5 to 0 volts) constitute a "change of state"? In fact, we can identify three types of input capture events based on how we define a state change. We can trigger an event on the rising edge, the falling edge, or on either rising/falling edges. Which edge we choose to use for triggering the IC event can be specified by setting the appropriate bits in the `TCTL2` control register.

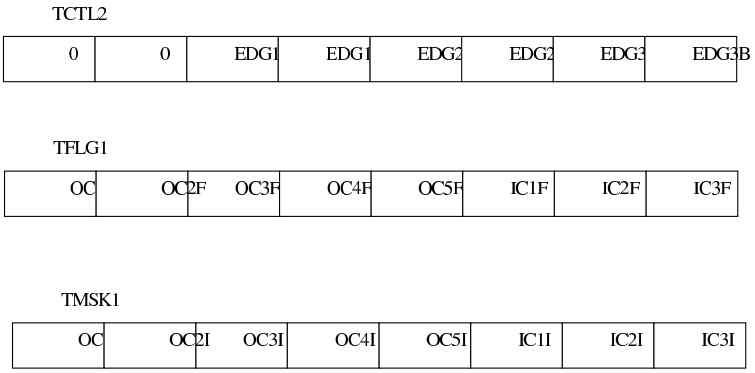

FIGURE 2. Registers for Input Capture Interrupts

The bits in `TCTL2` that define the edge events for `ICn` ($n = 1, 2$, or 3) have the logical names `EDGnB` and `EDGnA`. The effect that setting these bits have on defining the input capture event for the nth pin is given in the following table:

EDGnB	EDGnA	Active edge
0	0	none
0	1	capture on rising
1	0	capture on falling
1	1	capture on both rising and falling

Input capture events can be used to measure the width of a pulse. This is done by using an input capture event that triggers on the rising edge of a pulse. The ISR servicing this interrupt saves the time when the interrupt was received and then resets the interrupt so it triggers on the falling edge. When the IC interrupt is triggered the next time, we can then subtract the time in `TIC1` from the previously saved IC interrupt time to obtain the duration of the pulse. The following code segment shows how this might be done using the `IC1` interrupt.

```
unsigned int _start_pulse;
unsigned int _pulse_width;

#pragma interrupt_handler IC1han();
void IC1han(void){
  asm(" sei");
  if(TCTL2 == 0x10){
    TCTL2 = 0x20;
    _start_pulse = TIC1;
    TFLG1 |= IC1;
  }else{
    if(_start_pulse < TIC1){
     _pulse_width = TIC1 - _pulse_width;
    }else{
     _pulse_width = 65536 - _start_pulse + TIC1;
```

```
    }
    TCTL2 = 0x10;
    TFLG1 |= IC1;
  }
}

extern void IC1han();
#pragma abs_address:0xffee;
void (* IC1_handler[])() = { IC1han };
#pragma end_abs_address
```

The preceding ISR uses two globally declared variables, `_pulse_width` and `_start_pulse`. In the initialization routine `init()`, we would first arm the IC1 interrupt by setting the appropriate bit in register `TMSK1`. In this initialization routine we would also use the instruction `TCTL2 = 0x10` to trigger IC1 interrupts on the rising edge of the received pulse on pin `PA2`. The ISR first checks to see if the interrupt was triggered by a rising edge or a falling edge. If the interrupt was triggered by a rising edge, then we reset `TCTL2 = 0x20` to cause the next IC interrupt to occur on a falling edge. We then store the time `TIC1` when the first IC1 interrupt occurred in the global variable `_start_pulse`.

If the ISR is triggered on a falling edge, then we know that `_start_pulse` already contains the time when the rising edge of the pulse occurred. So all we need to do is difference `TIC1` from `_start_pulse` to get the pulse width `_pulse_width`. This computation, however, is complicated by the fact that `TIC1` is the value in a 16-bit counter `TCNT`. If this counter overflows after `_start_pulse`, then it is possible that `TIC1` is actually less than `_start_pulse`. So our ISR needs to check and see if an overflow occurred by checking to see if `TIC1` is greater than `_start_pulse`. If this condition holds, then we merely need to difference `_start_pulse` from `TIC1` to obtain `_pulse_width`. If an overflow has occurred then `TIC1` is less than `_start_pulse` and we'll need to add 65536 to the difference in order to get the correct value for `_pulse_duration`.

4. What you need to do to finish this lab

The only "construction" you'll need to do is connect the output of the PN4601 module to the IC1 pin. The real challenge lies in the program because it requires you to coordinate the operation of the input capture and output compare interrupts.

To give you a good starting place, we've modified the `kernel` for lab 10 to take care of the time-tick and transmission functions. In particular, the global variable `_Time` is now incremented by the interrupt handler `OC2han`. The `OC2` output compare is enabled by `init()` and remains enabled forever. It increments the global variable `_Time` once every 256 hardware time ticks.

The `OC4` interrupt handler has been rewritten to transmit a single frame containing the character stored in the global variable `sdata`. The frame is configured as indicated above in figure 1. In the kernel, we've assumed that the start bit lasts 1000 hardware time ticks, the data bit lasts 2000 hardware time ticks, and that there is a stop-bit of duration 4000 hardware time ticks. The `OC4` output compare interrupt is enabled from a new kernel function `xmit_data_pa7()`. The handler disables itself once it has completed transmitting a single frame of data. The handler assumes that the data to be transmitted is held in global variable `sdata`.

The `kernel` function called `xmit_data_pa7()` is used to initiate transmission of a single frame. This function has the prototype

```
void xmit_data_pa7(void);
```

The function takes the value in global variable `sdata` and enables the OC4 interrupt which is then used to control the transmission of a single frame from output pin PA7. You would use the function as shown in the following code.

```
void main(void){
  int istep;

  init();
  sdata=0;
  rdata=0;
  while(1){
    istep=butcontrol();
    sdata=(sdata+istep)%8;
    display(sdata,rdata);
    xmit_data_pa7();
  }
}
```

This is a particularly simple program. The global variable `sdata` contains the data to be transmitted and the global variable `rdata` contains the data to be received. The program consists of a single `while` loop that repeatedly checks the state of the buttons, increment or decrements the data to be transmitted, displays the received and transmitted data to the dual digit LED display, and then enables the `OC4` interrupt. The interrupt handler `OC4han()` then transmits what's in `sdata` out of pin `PA7`. You should be able to use this simple program, along with the kernel we've provided you to start your system.

In order to finish this program, you'll need to write a pair of interrupt handlers. You will first need to write an input capture interrupt that detects the start bit of a frame. Upon detecting the start bit, your input capture interrupt should then disable itself and then begin an output compare interrupt that periodically reads the state of the input data pin. This output compare interrupt will decode the frame and should store the result in a global variable, `rdata`. This variable represents the *received data* and can be displayed. Since you know that each frame of data consists of exactly eight bits, you know exactly when to stop looking for data bits in the frame. Once your output compare interrupt has finished reading the data bits, it should disarm itself and it should re-arm the input capture interrupt to allow the system to begin looking for start bits again.

A more specific guideline to writing these two interrupt handlers is given below.

- We recommend that you use the `IC1` input capture to detect the frame's start bit. Remember that the start bit (see figure 1) is half the size of a regular data bit. Your interrupt handler will need to measure the width of every received bit. Your interrupt handler `IC1` should be initially armed in `init()`. It will disarm itself after it has detected the start bit and it should then arm the output compare interrupt `OC3`. The input capture interrupt will then be re-armed once the OC3 interrupt has finished its work.

- We recommend that you use the `OC3` output compare interrupt to periodically read the input pin (IC1). Remember that there are 8 data bits that occur at regular intervals after the start bit. So you can use the `OC3` interrupt handler to periodically read the logical state of the IC1 input pin. Each time it reads the state, your interrupt handler should set the appropriate bit in a global variable like `rdata`. After the eighth bit has been read, you interrupt handler should disarm itself and then re-arm the `IC1` interrupt, so your system can begin looking for a start bit again.

The only code you need to write will be these two interrupt handlers. Your original `main` program probably needs little (if any) modification because all of the work associated with decoding a received data frame is done by the interrupt handlers.

In order to finish this lab, you'll need to demonstrate that your system transmits and receives the requested data. You should be able to demonstrate that if the data link is broken that you link will stop working. You should also be able to transmit information to other people across the room.

CHAPTER 13

APPENDIX:
C-language Programming for the MicroStamp11

1. Tutorial Introduction

We start by considering the following C-language program for the MicroStamp11.

```
line   statement
  0   #define PORTA   *(unsigned char volatile *)(0x00)
  1   #include "kernel.c"

  2   void main(void){
  3     int i;
  4     init();
  5     while(1){
  6       if(i==0){
  7        OutString("Hello World");
  8        PORTA ^= 0xff;
  9       }
 10       i++;
 11     }
 12   }
 13   #include "vector.c"
```

The preceding program executes an infinite `while` loop that increments an `int` variable `i`, calls the function `OutString()`, and toggles the logical state of `PORTA` every time `i` equals zero. The function `OutString` is a kernel library function that sends a character string to the MicroStamp11's asynchronous serial port.

This simple program illustrates a number of important C-language statements. Lines 0,1, and 13 are compiler directives called *pragma*'s. A pragma is a special instruction to the compiler. The pragma in line 0 replaces the string `PORTA` with the volatile address 0x00. The program in lines 1 and 13 ask the compiler to insert instructions contained in files `kernel.c` and `vector.c`, respectively.

C-language programs consist of modules or *functions*. The `main` function in the above program is declared by the statement

```
void main(void){  }
```

The first token is the type of the function's return value. The token within the parentheses is the type of the function's argument and the curly brackets mark off the body of the function. For a MicroStamp11 program, the `main` function doesn't return anything, so the type of the returned variable is `void`.

Within the `main` function you will first find a variable declaration that declares the *type* of variable `i` (line 3). The program contains two *flow-control* statements (lines 5-6), the `while` and `if` statement. This program also contains several expressions formed from a combination of variables and operators (lines 6,8, and 9). The main program also calls two functions `init` and `OutString` (lines 4 and 7) both of which are declared in the included file `kernel.c`.

The remaining sections of this program review some of the basic C-language constructions such as variables, operators, expressions, and functions.

2. Variables

A C-language program consists of *functions*. A function contains statements or *expressions* that are formed from *variables* and *operators*. The variables are basic data objects that are manipulated by the program. Expressions specify these manipulations by combining the variable with various unary or binary operators. This section focuses on *variables*.

As noted above, a variable is a basic data type that is used to store results of a computation. All variables must be *declared* in the C-programming language. The general format of a variable declaration is of the form,

```
<type> <variable>;
```

where `<variable>` is the *logical name* of the variable. Each variable is a location in memory and hence is simply a string of bits. By associating this variable with a *type*, we can control how these bits are interpreted. Line 3 of the tutorial program had the variable declaration

```
int i;
```

This statement declares that the variable with name `i` is of type `int`, a signed integer. By assigning the `int` type, this means that the variable `i` is stored as a 16-bit signed integer taking values between -32768 and 32767. If we had used the declaration

```
char i;
```

then `i` would be interpreted as an 8-bit unsigned integer taking values between 0 and 255. Table 1 lists the commonly used variable types and their interpretation.

The location of the declaration in a program effects the *scope* of the variable. A variable's scope is the part of the program that "knows" or "sees" the variable. If we declare a variable outside of a function, then the scope consists of all functions that occur after the declaration. Such variables are said to have *global* scope because they can be referenced by multiple functions. A variable whose declaration appears within a

Declaration	Comment	Range
unsigned char uc;	8-bit unsigned number	0 to +255
char c1,c2,c3;	three 8-bit signed number	-128 to 127
unsigned int ui;	16-bit unsigned number	0 to +65535
int i1,i2;	two 16-bit unsigned numbers	-32768 to +32767
unsigned short us;	16=bit unsigned number	0 to +65536
short s1,s2;	two 16-bit signed numbers	-32768 to +32767

FIGURE 1. Variable Declarations

function only has scope within that function. Such variables are said to have *local* scope. The following code segment illustrates variable declarations with global and local scope.

```
unsigned int _Time;

#define "kernel.c"
void main(void){
  int i;
  while(1){
    if(i==0) OutUDec(_Time);
    i++;
  }
}
#define "vector.c"
```

In this program, the variable `_Time` is declared as an unsigned 16-bit integer. Because the declaration appears outside of any functions in `kernel.c` and `main`, this variable is visible to any function in the program. In contrast, the variable `i` is a signed integer with local scope, since its declaration appears within the function `main`.

3. Operators

Operators are special characters that can modify the value of a variable. Some of the basic arithmetic binary operators are given in table 2. In this table you'll find binary operators such as `+`, `*`, `/`, and `=`. These operators are said to be binary because they accept two arguments. The syntax for using these binary operators is

```
<arg1> <op> <arg2>
```

As a concrete example, consider the expression `a+b`. In this case, the first argument (`arg1`) is `a`, the second argument (`arg2`) is `b` and the binary `op` is `+`, arithmetic addition.

An *expression* is formed by concatenating variables and operators. The following statements provide simple examples of expressions.

```
a=b+c+d;
a=b+c*d;
```

operation	meaning
`=`	assignment statement
`+`	addition
`-`	subtraction (negation)
`*`	multiply or pointer reference
`/`	divide
`%`	modulo, division remainder

FIGURE 2. Basic Arithmetic binary operators

The action of each of these states is to first evaluate the expression on the righthand side of the `=` sign and then to replace `a`'s current value with the evaluated expression. This means that all of the arithmetic operations such as `+` and `*` are executed before the assignment operation `=`. In other words, there is a natural *precedence* of operations in which assignment, `=`, has the lowest precedence.

There is also a precedence between different binary operations. For example, multiplies and divides are always performed before addition and multiplies. So that the statement, `a=b+c*d`, would first perform the multiplication, $c * d$, then perform the addition, and then perform the assignment. This natural precedence of operations is always followed unless specified otherwise by parentheses. We may, for example, force the addition to be executed prior to the multiplication by surrounding the addition with parentheses. So in the following statement, `a=(b+c)*d`, the addition `b+c` is executed first, then the multiplication, then the assignment.

In addition to the standard arithmetic binary operators we can define some *relational operators*. Relational operators take arithmetic variables as arguments and return a *logical value*. A variable with `logical` type has a value of 1/0 (true/false). The most commonly used relational operators are shown in figure 3

operator	meaning
`<`	less than
`>`	greater than
`>=`	greater than or equal to
`==`	equal to
`!=`	not equal to

FIGURE 3. Relational Operators

The following program segment illustrates the use of a relational operator.

```
void main(void){
  int i;
  while(1){
    if(i==0) OutString("HELLO");
    i++;
  }
}
```

This is part of the original tutorial program's `main` function. In this case, the relational operator `==` is used to return a true value if the condition `i==0` is satisfied and is false otherwise. In this case the `if` statement uses this logical value to decide whether or not the output the string `"HELLO"`.

A *logical* operator takes two logically valued variables and returns a logically valued result. Such operators are used in evaluating complex logical statements. A list of commonly used logical operators will be found in table 4. Consider the following statement,

```
(i==0)&&(j<1);
```

returns TRUE if variable `i` is zero AND variable `j` is less than one.

Operation	Meaning
`&&`	boolean and
`||`	boolean or

FIGURE 4. Logical Operators

Finally, we can introduce a set of binary *bitwise* operators. A bitwise operator acts on the *bits* of the binary variable. Remember that `char i` is interpreted as an unsigned 8-bit integer. So if we execute `i+j`, then this is interpreted as the *addition* of two 8-bit integers. In Micro-controllers, however, we often wish to have precise control over individual *bits* within a `char` variable. The bitwise operators allow us that type of control. Table 5 lists the most common bitwise operators.

Operation	Meaning
`|`	logical or
`&`	logical AND
`^`	logical exclusive or
`<<`	shift left
`>>`	shift right

FIGURE 5. Bitwise Binary Operators

In addition to *binary* operators we can also have *unary* operators. An expression with a unary operator takes the form

```
<op> <arg>
```

A commonly used unary operator is negation. So the statement `-i` takes the variable `i` and negates it. The argument is `i` and the operator is `-`. A table of common unary operators is given in table 6.

Bitwise binary operators are particularly important when setting, clearing, or toggling bits within a MicroStamp11 program. As an example, let's consider the code segment,

```
#define bit(i) (1<<(i))
char PORTA;
PORTA = PORTA | bit(2);
```

Operation	Meaning
-	negation
!	logical not (true to false, false to true)
~	1's complement (NOT)
++	increment
--	decrement

FIGURE 6. Unary Operators

```
PORTA = PORTA & (~bit(2));
PORTA = PORTA ^ bit(2);
```

The first line defines a macro `bit(i)` that takes the argument `i`. The statement replacing `bit(i)` is the `1 << (i)` which means that the string `0x01` is shifts left by `i` places. This means that that `bit(i)` is a binary number whose `i`th bit is one. This elementary bit string is then used in the following statements to modify the value of `PORTA`. The first statement takes the OR of `PORTA` and `bit(2)`. The end result of this is to *set* the 2nd bit in PORTA to one. The second statement takes the logical AND of `PORTA` with the 1's complement (NOT) of `bit(2)`. The end result of this action is to *clear* the 2nd bit in `PORTA`. The final statement takes the exclusive or (XOR) of `PORTA` and `bit(2)`. The end result of this action is to *toggle* the 2nd bit in PORTA.

Finally, it is useful to note that we can form some special two character operators by combining the assignment with a binary operator. Table 7 lists some of these commonly used operator pairs. These operator pairs are often used to simplify an expression. For example, the preceding statements that were used to set, clear, and toggle the 2nd bit in `PORTA` can also be written as

```
PORTA |= bit(2);
PORTA &= ~bit(2);
PORTA ^= bit(2);
```

The action of these operator pairs is to first execute the arithmetic/logical binary operation, and then to assign this result to the variable on the lefthand side of the expression.

operation	meaning
+=	add value to
-=	subtract value from
*=	multiply value to
/=	divide value to
|=	or value to
&=	and value to
^=	exclusive or value to
<<=	shift value left
>>=	shift value right
%=	modulo divide value to

FIGURE 7. Common Operator Pairs

4. Flow Control

Every procedural language provides statements for determining the flow of control within programs. Although declarations are a type of statement, in C the unqualified word statement usually refers to procedural statements rather than declarations.

In the C language, statements can be written only within the body of a function; more specifically, only within compound statements. The normal flow of control among statements is sequential, proceeding from one statement to the next. However, as we shall see, most of the statements in C are designed to alter this sequential flow so that algorithms of arbitrary complexity can be implemented. This is done with statements that control whether or not other statements execute and, if so, how many times. Furthermore, the ability to write compound statements permits the writing a sequence of statements wherever a single, possibly controlled, statement is allowed. These two features provide the necessary generality to implement any algorithm, and to do it in a structured way.

Simple Statements: The C language uses semicolons as statement terminators. A semicolon follows every simple (non-compound) statement, even the last one in a sequence. When one statement controls other statements, a terminator is applied only to the controlled statements. Thus we would write

```
if(x>5) x = 0; else ++x;
```

with two semicolons, not three. Perhaps one good way to remember this is to think of statements that control other statements as "super" statements that "contain" ordinary (simple and compound) statements. Then remember that only simple statements are terminated. This implies, as stated above, that compound statements are not terminated with semicolons. Thus

```
while(x < 5) {func(); ++x;}
```

is perfectly correct. Notice that each of the simple statements within the compound statement is terminated.

Compound Statements: The terms compound statement and block both refer to a collection of statements that are enclosed in braces to form a single unit. Compound statements have the form

```
{ObjectDeclaration?... Statement?... }
```

`ObjectDeclaration?...` is an optional set of local declarations. If present, C requires that they precede the statements; in other words, they must be written at the head of the block. `Statement?...` is a series of zero or more simple or compound statements. Notice that there is not a semicolon at the end of a block; the closing brace suffices to delimit the end. In this example the local variable temp is only defined within the inner compound statement.

```
void main(void){ int n1,n2;
    n1=1; n2=2;
    { int temp;
       temp=n1; n1=n2; n2=temp; /* switch n1,n2 */
    }
}
```

The power of compound statements derives from the fact that one may be placed anywhere the syntax calls for a statement. Thus any statement that controls other statements is able to control units of logic of any complexity.

When control passes into a compound statement, two things happen. First, space is reserved on the stack for the storage of local variables that are declared at the head of the block. Then the executable statements are processed.

One important limitation in C is that a block containing local declarations must be entered through its leading brace. This is because bypassing the head of a block effectively skips the logic that reserves space for local objects. Since the goto and switch statements (below) could violate this rule.

If Statement: *If* statements provide a non-iterative choice between alternate paths based on specified conditions. They have either of two forms

```
if ( ExpressionList ) Statement1
if ( ExpressionList ) Statement1 else Statement2
```

`ExpressionList` is a list of one or more expressions and Statement is any simple or compound statement. First, `ExpressionList` is evaluated and tested. If more than one expression is given, they are evaluated from left to right and the right-most expression is tested. If the result is true (non-zero), then the `Statement1` is executed and the `Statement2` (if present) is skipped. If it is false (zero), then `Statement1` is skipped and `Statement2` (if present) is executed. In this first example, the function `isGreater()` is executed if G2 is larger than 100.

```
if(G2 > 100) isGreater();
```

Complex conditional testing can be implemented using the relational and boolean operators as shown below:

```
if ((G2==G1)||(G4>G3)) True(); else False();
```

The Switch Statement: *Switch* statements provide a non-iterative choice between any number of paths based on specified conditions. They compare an expression to a set of constant values. Selected statements are then executed depending on which value, if any, matches the expression. Switch statements have the form

```
switch ( ExpressionList ) { Statement?...}
```

where `ExpressionList` is a list of one or more expressions. `Statement?...` represents the statements to be selected for execution. They are selected by means of case and default prefixes–special labels that are used only within switch statements. These prefixes locate points to which control jumps depending on the value of `ExpressionList`. They are to the switch statement what ordinary labels are to the goto statement. They may occur only within the braces that delimit the body of a switch statement.

The *case* prefix has the form

```
case ConstantExpression :
```

and the default prefix has the form

```
default:
```

The terminating colons are required; they heighten the analogy to ordinary statement labels. Any expression involving only numeric and character constants and operators is valid in the case prefix.

After evaluating `ExpressionList`, a search is made for the first matching case prefix. Control then goes directly to that point and proceeds normally from there. Other case prefixes and the default prefix have no effect once a case has been selected; control flows through them just as though they were not even there. If no matching case is found, control goes to the default prefix, if there is one. In the absence of a default prefix, the entire compound statement is ignored and control resumes with whatever follows the switch statement. Only one default prefix may be used with each switch.

If it is not desirable to have control proceed from the selected prefix all the way to the end of the switch block, break statements may be used to exit the block. Break statements have the form

```
break;
```

Some examples may help clarify these ideas. Assume Port A is specified as an output, and bits 3,2,1,0 are connected to a stepper motor. The switch statement will first read Port A and the data with 0x0F `(PORTA&0x0F)`. If the result is 5, then PortA is set to 6 and control is passed to the end of the switch (because of the break). Similarly for the other 3 possibilities

```
#define PORTA *(unsigned char volatile *)(0x0000) void
step(void){ /* turn stepper motor one step */
   switch (PORTA&0x0F) {
      case 0x05:
         PORTA=0x06; // 6 follows 5;
         break;
      case 0x06:
         PORTA=0x0A; // 10 follows 6;
         break;
      case 0x0A:
         PORTA=0x09; // 9 follows 10;
         break;
      case 0x09:
         PORTA=0x05; // 5 follows 9;
         break;
      default:
         PORTA=0x05; // start at 5
   }
}
```

The While Statement: The *while* statement is one of three statements that determine the repeated execution of a controlled statement. This statement alone is sufficient for all loop control needs. The other two merely provide an improved syntax and an execute-first feature. While statements have the form

```
while ( ExpressionList ) Statement
```

where `ExpressionList` is a list of one or more expressions and Statement is an simple or compound statement. If more than one expression is given, the right-most expression yields the value to be tested. First, `ExpressionList` is evaluated. If it yields true (non-zero), then Statement is executed and `ExpressionList` is evaluated again. As long as it yields true, Statement executes repeatedly. When it yields false, Statement is skipped, and control continues with whatever follows.

In the example

```
i = 5; while (i) array[--i] = 0;
```

elements 0 through 4 of array[ ] are set to zero. First i is set to 5. Then as long as it is not zero, the assignment statement is executed. With each execution i is decremented before being used as a subscript.

Continue and *break* statements are handy for use with the while statement (also helpful for the do and for loops). The continue statement has the form

```
continue;
```

It causes control to jump directly back to the top of the loop for the next evaluation of the controlling expression. If loop controlling statements are nested, then continue affects only the innermost surrounding statement. That is, the innermost loop statement containing the continue is the one that starts its next iteration.

The break statement (described earlier) may also be used to break out of loops. It causes control to pass on to whatever follows the loop controlling statement. If while (or any loop or switch) statements are nested, then break affects only the innermost statement containing the break. That is, it exits only one level of nesting.

The For Statement: The for statement also controls loops. It is really just an embellished while in which the three operations normally performed on loop-control variables (initialize, test, and modify) are brought together syntactically. It has the form

```
for ( ExpressionList? ; ExpressionList? ; ExpressionList? )
Statement
```

For statements are performed in the following steps:

The first `ExpressionList` is evaluated. This is done only once to initialize the control variable(s).

The second `ExpressionList` is evaluated to determine whether or not to perform Statement. If more than one expression is given, the right-most expression yields the value to be tested. If it yields false (zero), control passes on to whatever follows the for statement. But, if it yields true (non-zero), Statement executes.

The third `ExpressionList` is then evaluated to adjust the control variable(s) for the next pass, and the process goes back to step 2. E.g.,

```
for(J=100;J<1000;J++) { process();}
```

A five-element array is set to zero, could be written as

```
for (i = 4; i >= 0; --i) array[i] = 0;
```

or a little more efficiently as

```
for (i = 5; i; array[--i] = 0) ;
```

Any of the three expression lists may be omitted, but the semicolon separators must be kept. If the test expression is absent, the result is always true. Thus

```
for (;;) {...break;...}
```

will execute until the break is encountered.

As with the while statement, break and continue statements may be used with equivalent effects. A break statement makes control jump directly to whatever follows the for statement. And a continue skips whatever remains in the controlled block so that the third `ExpressionList` is evaluated, after which the second one is evaluated and tested. In other words, a continue has the same effect as transferring control directly to the end of the block controlled by the for.

The Return Statement: The return statement is used within a function to return control to the caller. Return statements are not always required since reaching the end of a function always implies a return. But they are required when it becomes necessary to return from interior points within a function or when a useful value is to be returned to the caller. Return statements have the form

```
return ExpressionList? ;
```

`ExpressionList?` is an optional list of expressions. If present, the last expression determines the value to be returned by the function. I f absent, the returned value is unpredictable.

5. Functions and Program Structure

We have been using functions throughout this document, but have put off formal presentation until now because of their immense importance. The key to effective software development is the appropriate division of a complex problem in modules. A module is a software task that takes inputs, operates in a well-defined way to create outputs. In C, functions are our way to create modules. A small module may be a single function. A medium-sized module may consist of a group of functions together with global data structures, collected in a single file. A large module may include multiple medium-sized modules. A hierarchical software system combines these software modules in either a top-down or bottom-up fashion. We can consider the following criteria when we decompose a software system into modules:

(1) We wish to make the overall software system easy to understand;
(2) We wish to minimize the coupling or interactions between modules;
(3) We wish to group together I/O port accesses to similar devices;
(4) We wish to minimize the size (maximize the number) of modules;
(5) Modules should be able to be tested independently;
(6) We should be able to replace/upgrade one module with effecting the others;
(7) We would like to reuse modules in other situations.

The term function in C is based on the concept of mathematical functions. In particular, a mathematical function is a well-defined operation that translates a set of input values into a set of output values. In C, a function translates a set of input values into a single output value. We will develop ways for our C functions

to return multiple output values and for a parameter to be both an input and an output parameter. As a simple example consider the function that converts temperature in degrees F into temperature in degrees C.

```
int FtoC(int TempF){
     int TempC;
     TempC=(5*(TempF-32))/9;   // conversion
return TempC;}
```

When the function's name is written in an expression, together with the values it needs, it represents the result that it produces. In other words, an operand in an expression may be written as a function name together with a set of values upon which the function operates. The resulting value, as determined by the function, replaces the function reference in the expression. For example, in the expression

```
FtoC(T+2)+4;    // T+2 degrees Fahrenheit plus 4 degrees
Centigrade
```

the term `FtoC(T+2)` names the function `FtoC` and supplies the variable `T` and the constant 2 from which `FtoC` derives a value, which is then added to 4. The expression effectively becomes

```
((5*((T+2)-32))/9)+4;
```

Although `FtoC(T+2)+4` returns the same result as `((5*((T+2)-32))/9)+4`, they are not identical. As will we see later in this chapter, the function call requires the parameter (T+2) to be passed on the stack and a subroutine call will be executed.

Function Declarations: Similar to the approach with variables, C differentiates between a function declaration and a function definition. A declaration specifies the syntax (name and input/output parameters), whereas a function definition specifies the actual program to be executed when the function is called. Many C programmers refer to function declaration as a prototype. Since the C compiler is essential a one-pass process (not including the preprocessor), a function must be declared (or defined) before it can be called. A function declaration begins with the type (format) of the return parameter. If there is no return parameter, then the type can be either specified as void or left blank. Next comes the function name, followed by the parameter list. In a function declaration we do not have to specify names for the input parameters, just their types. If there are no input parameters, then the type can be either specified as void or left blank. The following examples illustrate that the function declaration specifies the name of the function and the types of the function parameters.

```
//  declaration                 input           output
void Ritual(void);             // none           none
char InChar(void);             // none           8-bit
void OutChar(char);            // 8-bit          none
short InSDec(void);            // none           16-bit
void OutSDec(short);           // 16-bit         none
char Max(char,char);           // two 8-bit      8-bit
int EMax(int,int);             // two 16-bit     16-bit
void OutString(char*);         // pointer to 8-bit none
```

Normally we place function declarations in the header file. We should add comments that explain what the function does.

```
void InitSCI(void);   // Initialize 38400 bits/sec char
InChar(void);    // Reads in a character, gadfly void
OutChar(char);   // Output a character, gadfly char
```

Sometimes we wish to call a function that will be defined in another module. If we define a function as external, software in this file can call the function (because the compiler knows everything about the function except where it is), and the linker will resolve the unknown address later when the object codes are linked.

Function Definitions: The second way to declare a function is to fully describe it; that is, to define it. Obviously every function must be defined somewhere. So if we organize our source code in a bottom up fashion, we would place the lowest level functions first, followed by the function that calls these low level functions. It is possible to define large project in C without ever using a standard declaration (function prototype). On the other hand, most programmers like the top-down approach illustrated in the following example. This example includes three modules: the LCD interface, the COP functions, and some Timer routines. Notice the function names are chosen to reflect the module in which they are defined. If you are a C++ programmer, consider the similarities between this C function call LCDclear() and a C++ LCD class and a call to a member function LCD.clear(). The *.H files contain function declarations and the *.C files contain the implementations.

```
#include "HC12.H"
#include "LCD12.H"
#include "COP12.H"
#include "Timer.H"
void main(void){
   char letter; int n=0;
   COPinit(); // Enable TOF interrupt to make COP happy
   LCDinit();
   TimerInit()
   LCDString("Adapt812 LCD");
   TimerMsWait(1000);
   LCDclear();
   letter='a'-1;
   while(1){
      if (letter=='z')
         letter='a';
      else
         letter++;
      LCDputchar(letter);
      TimerMsWait(250);
      if(++n==16){
         n=0;
         LCDclear();
      }
   }
```

```
}
 #include "LCD12.C"
 #include "COP12.C"
 #include "Timer.C"
#include "VECTORS.C"
```

C function definitions have the following form

```
type Name(parameter list){ CompoundStatement };
```

Just like the function declaration, we begin the definition with its type. The type specifies the function return parameter. If there is no return parameter we can use void or leave it blank. Name is the name of the function. The parameter list is a list of zero or more names for the arguments that will be received by the function when it is called. Both the type and name of each input parameter is required.

The last, and most important, part of the function definition above is CompoundStatement. This is where the action occurs. Since compound statements may contain local declarations, simple statements, and other compound statements, it follows that functions may implement algorithms of any complexity and may be written in a structured style. Nesting of compound statements is permitted without limit.

As an example of a function definition consider

```
int add3(int z1, int z2, int z3){ int y;
    y=z1+z2+z3;
    return(y);}
```

Here is a function named add3 which takes three input arguments.

Function Calls: A function is called by writing its name followed by a parenthesized list of argument expressions. The general form is

```
Name (parameter list)
```

where Name is the name of the function to be called. The parameter list specifies the particular input parameters used in this call. Notice that each input parameter is in fact an expression. It may be as simple as a variable name or a constant, or it may be arbitrarily complex, including perhaps other function calls. Whatever the case, the resulting value is pushed onto the stack where it is passed to the called function.

C programs evaluate arguments from left to right, pushing them onto the stack in that order. As we will see later, the ICC11 and ICC12 compilers allocate the stack space for the parameters at the start of the program that will make the function call. Then the values are stored into the pre-allocated stack position before it calls the function. On return, the return parameter is located in Reg D. The input parameters are removed from the stack at the end of the program.

When the called function receives control, it refers to the first actual argument using the name of the first formal argument. The second formal argument refers to the second actual argument, and so on. In other words, actual and formal arguments are matched by position in their respective lists. Extreme care must be taken to ensure that these lists have the same number and type of arguments.

It was mentioned earlier, that function calls appear in expressions. But, since expressions are legal statements, and since expressions may consist of only a function call, it follows that a function call may be written as a complete statement. Thus the statement

```
add3(--counter,time+5,3);
```

is legal. It calls add3(), passing it three arguments –counter, time+5, and 3. Since this call is not part of a larger expression, the value that add3() returns will be ignored. As a better example, consider

```
y=add3(--counter,time+5,3);
```

which is also an expression. It calls add3() with the same arguments as before but this time it assigns the returned value to y. It is a mistake to use an assignment statement like the above with a function that does not return an output parameter.