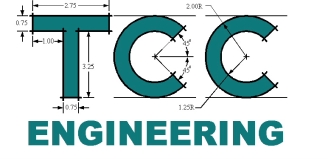
EGR 262   
Fundamental Circuits Laboratory   
Lab Manual

Instructor: Paul Gordy  
Office: H-115   
Phone: 822-7175   
Email: PGordy@tcc.edu



Tidewater Community College

ACKNOWLEDGEMENT

This lab manual used for EGR 262 – Fundamental Circuits Lab at Tidewater Community College is a revised version of the lab manual used for the course ECE 287 – Fundamental Circuits Laboratory offered at Old Dominion University. The material is used with the permission of Dr. Vishnu Lakdawala, Department of Electrical and Computer Engineering, Old Dominion University. Additionally, Old Dominion University adapted the material from Notre Dame University as indicated below.

Paul Gordy

Engineering Program Head

Tidewater Community College

ACKNOWLEDGEMENT

This lab manual is a revised version based on original material from “Introduction to Electrical Engineering  
Lab Manual” offered at Department of Electrical Engineering, Univ. of Notre Dame for EE 224 course.   
We wish to gratefully acknowledge Prof. Mike Lemmon, Dept. of Electrical Engineering, Univ. of Notre   
Dame for all owing us to adopt the laboratory and providing the necessary information.

Dr. Vishnu K. Lakdawala

Indrajeet Kaylankar

Satyanadh Gundimada

**Contents**

Chapter 1. Breadboarding Circuits 7

1. Objective 7

2. Parts List 7

3. Background 7

4. Tasks 18

5. What you should have learned 20

Chapter 2. MicroStamp11 Familiarization 23

1. Objective 23

2. Parts List 23

3. Background 23

4. Tasks 32

5. What you should have learned 33

Chapter 3. Lights and Switches -Hardware 35

1. Objective 35

2. Parts List 35

3. Background 35

4. Tasks 43

5. What you should have learned 44

Chapter 4. Lights and Switches -real time 45

1. Objective 45

2. Parts List 45

3. Background 45

4. Tasks 47

5. What you should have learned 48

Chapter 5. Digital to Analog Conversion 49

1. Objective 49

2. Parts List 49

3. Background 49

4. Tasks 51

5. What you should have learned 52

Chapter 6. Analog-to-Digital Conversion -Part 1 -Hardware 53

1. Objective 53

2. Parts List 53

3. Background 53

4. Tasks 61

5. What you should have learned 63

Chapter 7. Analog-to-Digital Conversion -Part 2 -Software 65

1. Objective 65

2. Parts List: 65

3. Background 65

4. Tasks 66

5. What you should have learned 67

**Contents**

Chapter 8. Pulse Width Modulation 69

1. Objective 69

2. Parts List 69

3. Background 69

4. Tasks 79

5. What you should have learned 80

Chapter 9. Digital-to-Analog Conversion Revisited 81

1. Objective 81

2. Parts List 81

3. Background 81

4. Tasks 87

5. What you should have learned 88

Chapter 10. Getting Power off the Wall 89

1. Objective 89

2. Parts List 89

3. Background 89

4. What is an Alternating Voltage? 89

5. What is a Transformer? 91

6. What is a Rectiﬁer Circuit? 92

7. Tasks 95

8. What you should have learned 96

Chapter 11. Serial Interfaces 97

1. Objective 97

2. Parts List 97

3. Background 97

4. Tasks 104

5. What you should have learned 105

Chapter 12. Analog to Digital Conversion Revisited -Time Multiplexing 107

1. Objective 107

2. Parts List 107

3. Background 107

4. Tasks 110

5. What you should have learned 111

Chapter 13. APPENDIX: C-language Programming for the MicroStamp11 113

1. Tutorial Introduction 113

2. Variables 114

3. Operators 115

4. Flow Control 119

5. Functions and Program Structure 123

**Lab Book Format**

The grading of your labs will be based on successful demonstration of your working system to the teaching assistant (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 provided on the course website. Lab books are an individual eﬀort, the data gathering is a group eﬀort.

Each lab report must have 3 parts: 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. 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 instructor 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 students have completed their predictive work before going on to the in-lab tasks. Once a instructor 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 instructor 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 is available on the course website.

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

**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 multimeter* (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 left-hand 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 labeled with a letter. In ﬁgure 1, there are three nodes with the labels *a*, *b*, and *c*. The left-hand 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 right-hand picture. In this picture, you’ll see that the circuit element is drawn as an *ar*c and all of the nodes are simply drawn as a point. The right-hand representation is called a *graph*. In a graph, the circuit elements are always represented as *arc*s (also called *branches*) and the terminals are always represented as points or *nodes* of the graph.

*Electrical Circuit Graph*

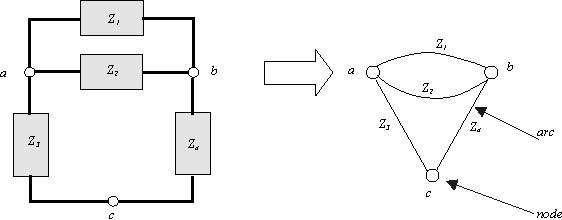


Figure 1. A simple electrical circuit and its graph

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

* The *voltag*e across each arc and
* The *curren*t through each arc.

*Circui*t *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 × 1018 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,



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 *resistanc*e of the device. This resistance is measured in units of *volt*s *pe*r *ampere* or *ohms* (denoted by the Greek symbol Ω). In mathematical terms, this relationship is written as

(3.1) *v*(*t*)= *Ri*(*t*)

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 left-hand picture in Figure 2. The right-hand 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*.

R

i

**+**

v

b

a

**-**

Carbon Resistor.jpg

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.

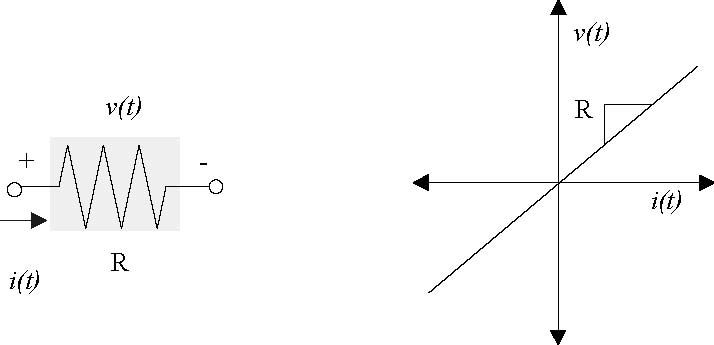


Figure 3. A linear resistor and its I-V characteristic

Two special types of resistors are the *shor*t *circui*t and *ope*n *circuit*. We deﬁne a *short circuit* as a two-terminal 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 left-hand 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 right-hand 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 left-hand 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*.



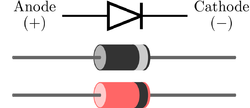
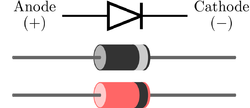
c

symbol for trim pot

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 left-hand 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 right-hand 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.

i(t)

+ v(t) -

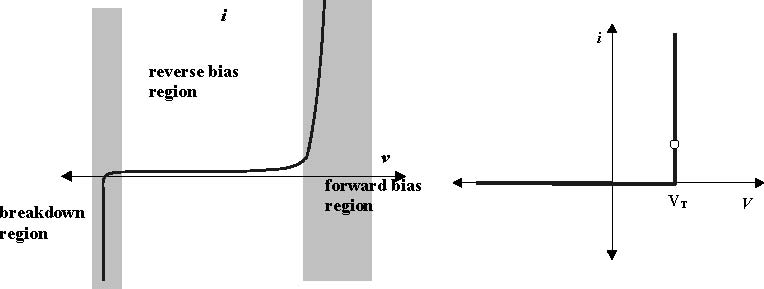
Figure 5. Diode

When the voltage v is positive and greater than a minimum threshold voltage VT, 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 *revers*e *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



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 left-hand 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.



Diode’s VI Curve Diode’s idealized VI Curve

Figure 6. Diode’s IV characteristic

The left-hand 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 right-hand 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 < VT*. 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.

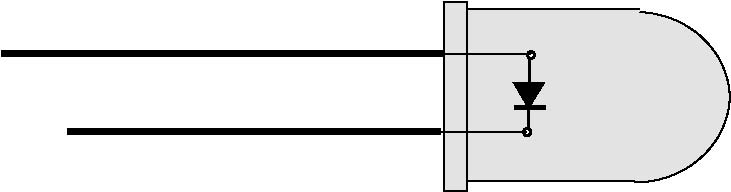
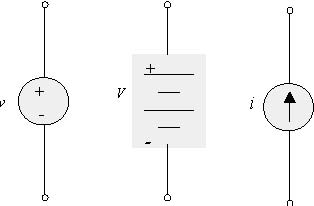


Figure 7. Light Emitting Diode

**3.4. What is an independent source?** Resistors are examples of so-called *passive* devices. We call them passive because they always dissipate energy. *Activ*e 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 left-hand symbol depicts an independent voltage source. The symbol is a circle with the voltage polarities marked on them and the voltage value V. The right-hand 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).



|  |  |  |
| --- | --- | --- |
| independent  voltage  source | constant  voltage  source | independent  current  source |

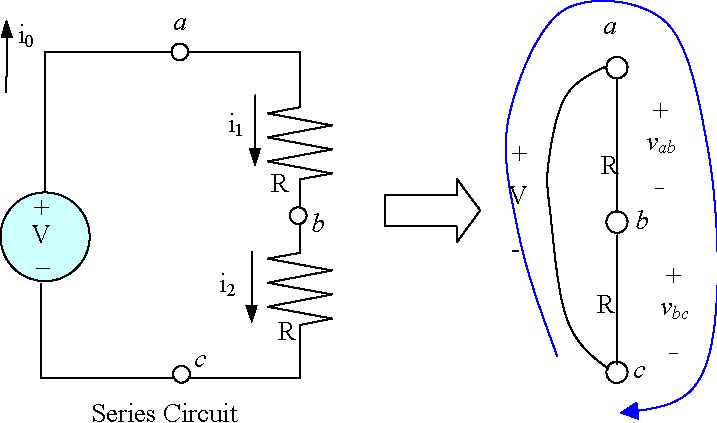
Figure 8. Independent Voltage and Current Sources

**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 left-hand picture is the circuit diagram and the right-hand 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 *curren*t *la*w 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 *Voltag*e *La*w *(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.***



Circuit Graph

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*) → (*b*, *c*) → (*c*, *a*)

The voltages obtained by traversing this loop are

*vab, vbc,* 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 − *va*b − *vb*c =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:

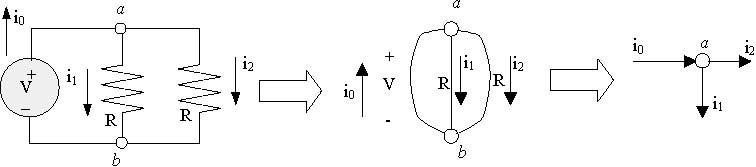
***The algebraic sum of current at any node is zero.***

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 right-hand 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

*i0 − i1 − i2 =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.



Parallel Circuit Circuit Graph KCL at Node *a*

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

i0 = i1

i1 = i2

i2 = i0

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 *va*c = 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

*va*b = *i*1R

*vb*c = *i*2R

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 *va*b = *vb*c = *i*0*R*.

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

V − *va*b − *vb*c =0

Using our previous results, this equation can be rewritten as

V − *i*0R − *i*0R =0

which is an algebraic equation in a single unknown quantity *i*0. We can now solve for *i*0 to deduce that



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/*2*R*, 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 *va*b = *vb*c = *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 *μ*Stamp11 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 *solderles*s *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. Please also refer to Figure 3 of experiment 2 on page 27.

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.

----------------------------------------------------------------------------------------------------------------- modified by PG

SK-10 Solderless Breadboard (or equivalent)

A B

# Internal Connections on the SK-10 Solderless Breadboard

## Notes:

## 1) Lines indicate which holes are connected under the breadboard.

2) To connect two or more wires together, plug them in the same row of holes.

3) Holes A and B are connected on some breadboards (as well as the similar holes on the other horizontal rows).

**Example**: Connect the following circuit using the SK-10 solderless breadboard.

+

\_

2.2 k

5.6 k

3.3 k

1.0 k

1.5 k

10 V

Jumper

Jumper

+

\_

Connections to

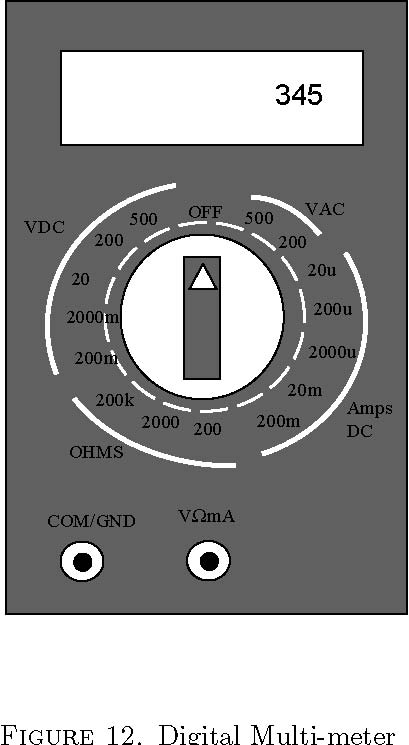
10 V power supply

-------------------------------------------------------------------------------------------------------------------------

Figure 11. Solderless Breadboard

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

In the middle of DMM’s faceplate you will ﬁnd a large 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 sub-modes 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.

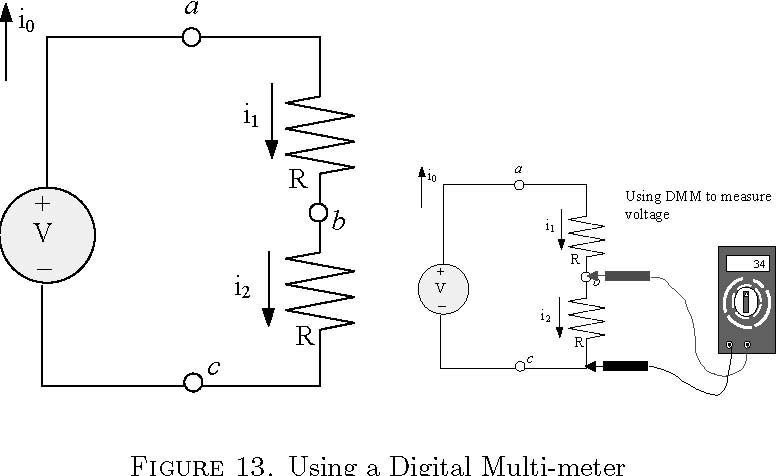


The actual measurements are made with two probes. One probe is black and the other is red. The black probe is usually connected to the COM/GND jack on the front of the DMM. The red probe is usually connected to the other jack (V-Ω-*mA*). The black probe is usually taken to be the reference or ground 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 left-hand picture shows the circuit schematic and the right-hand picture shows where to connect the DMM to measure the voltage across the second resistor.

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 left-hand drawing in the ﬁgure is the circuit and we want to measure the current ﬂowing through the second resistor *R*2. The right-hand 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 right-hand 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



Using a DMM to measure current

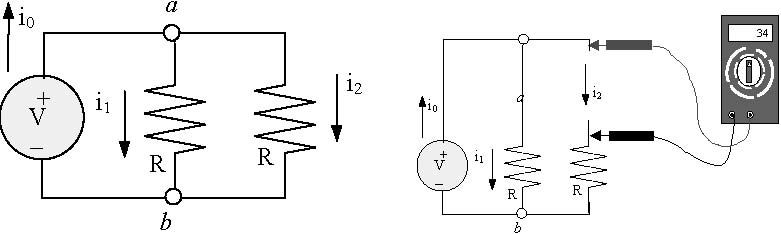


Figure 14. Using a Digital Multimeter

the DMM’s display. Note that it is important that the resistor actually be removed from the circuit. If you 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.

**4. Tasks (--- modified by PG)**

**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 5 kΩ. Since the second resistor’s value is variable, the second resistor’s value is denoted by the variable *R*.

Before coming to the lab you should do the following:

(1) Draw a *labeled* schematic diagram of the circuit.

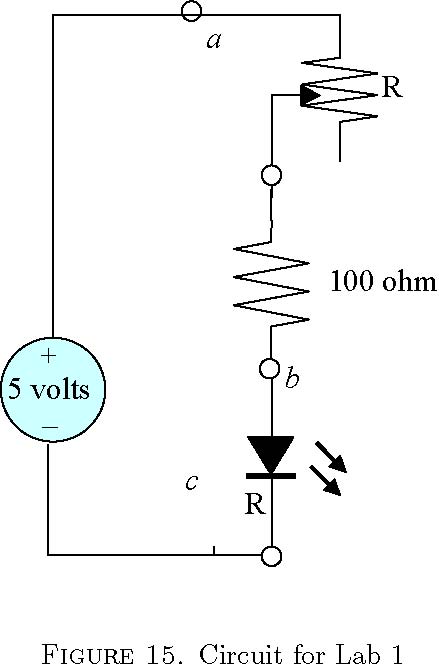
(2) Include an explanation of how the circuit works.

(3) Draw a picture showing how you plan to breadboard the circuit.

(4) Draw a labeled schematic diagram with the diode replaced by a diode model (use the values of Vo and Ro suggested by the instructor).

(5) Derive an expression for the current going through the diode as a function of the variable resistance, R (using the diode model).

(6) Plot the current through the LED as a function of the variable resistance R for at least 10 values of R from 0 to 5000 ohms. Use Excel or MATLAB for all graphs in this course. Graphs must always be properly formatted. Include both the table of values used and the graph in your notebook. **Always** include sample formulas with tables of calculations in Excel.



Ask the instructor 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, voltage, and resistance measurements on the circuit. Remember that to use the ammeter you will need to actually break the circuit and place the ammeter in series with the resistors. The variable resistor must be removed from the circuit each time you measure its resistance. 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 (be sure to remove it from the circuit to measure its value). You can change this resistance by turning the knob on the component. For at least 10 diﬀerent resistances, measure the current going through the diode and the voltage across the diode. Also observe the brightness of the diode. Make qualitative observations about the LED’s brightness as a function of current.

**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* (on the same graph). Your expected values should be recalculated using the exact resistance values measured in lab. Show the table of values used to generate the graph and include % error between expected and measured current.

(2) A graph of diode current versus diode voltage. Discuss how well your diode model matches this graph. Show the table of values used to generate this graphs as well. What values of Vo and Ro are implied by your graph?

(3) 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.

(3) How to use Ohm’s law, KVL, and KCL to predict the voltage and currents in a branch of a resistive network.

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. (1) MicroStamp11 with docking module
2. (2) USB cable
3. (3) 10 kΩ potentiometer, 100 Ω resistor, LED
4. (4) Hookup wires and breadboard
5. (4) Computer with ICC11 compiler, MicroLoad, and PuTTY

**3. Background**

In this lab the student will write and compile a C-language program that will then be downloaded and executed on a *microcontroller* known as the *MicroStamp11*. To complete the lab, the student will need to *communicat*e with the MicroStamp 11 using the personal computer’s (PC) USB port. Note that power (+5V and ground connections) are also provided to the MicroStamp 11 through the USB 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 *star*t executing the program and demonstrate that the program works as expected.

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

To see how a microcontroller diﬀers from a regular microcomputer, let’s ﬁrst look at the architecture of a regular computer. All computer systems are characterized by similar subsystems. The left-hand 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 *CP*U *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.

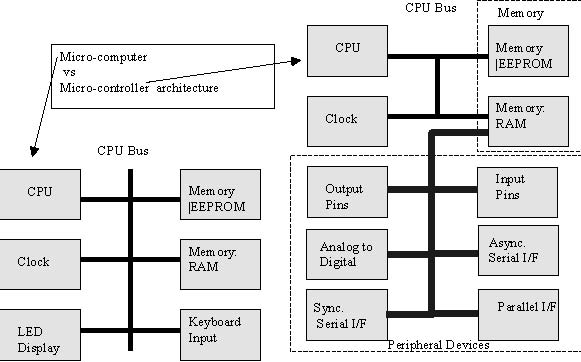


Figure 1. Block diagrams for microcomputer and microcontroller

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 *wor*d (depending upon the actual microcomputer being used).

There are two diﬀerent types of *memory*. These two types are called read-only memory (ROM) or *random* *acces*s *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 microcontroller such as the MicroStamp11 is a microcomputer that has been specially designed to speed up its access to its I/O devices. Most microcomputers access their I/O devices over the CPU bus. Since the CPU bus is also used by other computer subsystems, a microcomputer 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 microcontroller modiﬁes the CPU bus so that peripheral devices are directly mapped into the computer’s RAM. In other words, a microcontroller’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

right-hand drawing in ﬁgure 1 shows the architecture of the microcontroller. Note that this system has a much richer set of I/O devices than the standard microcomputer (shown in the left-hand 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 microcontroller module. Physically, the MicroStamp11 module is a small 1 by 2 inch printed circuit board (PCB) that is built around the Motorola 68HC11 microcontroller integrated circuit (IC). In addition to containing the microcontroller chip, the module holds a modest amount of circuitry devoted to power and memory management. The left-hand 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.

PA7 1

PA6 2

PA5 3

PA4 4

PA3 5

PA2 6

PA1 7

PA0 8

Reset\* 9

Vin 10

20 PD0/RXD

19 PD1/TXD

18 PD2/MISO

17 PD3/MOSI

16 PD4/SCK

15 PD5/SS

14 XIRQ\*

13 IRQ\*

12 +5 VDC

11 GND

MicroStamp 11 Pinout

The Master

74HC595 Pinout

The Slave

14 SER

12 RCLK

11 SRCLK

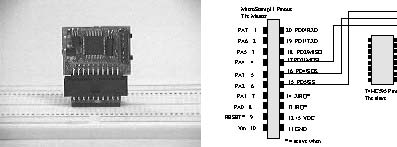


Figure 2. MicroStamp11 Module and its pinout

------------------------------------------------------------------------------------------------------------------Modified by PG

**3.3. How is power provided to the MicroStamp11?** 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. 5V is provided to the MicroStamp11 via the USB connection. This 5V supply can be connected to the power rails on the breadboard and then the MicroStamp11 can be powered by connecting the power rails to pins 11 and 12 on the MicroStamp11 as illustrated below.

5V from USB to power rails on breadboard

20 11

MicroStamp11

1 10

V G + D R G T

**USB**

**Interface**

11

MicroStamp11

1 10

5V from power rails to the MicroStamp11

------------------------------------------------------------------------------------------------------------------------------------

Microcontrollers are *digital synchronous* devices. By *synchronou*s we mean that all instructions are executed in synchrony with a hardware clock. The speed of the microcontroller’s hardware clock is set by an external *crystal*. The crystal is the silver cylinder on the right-hand 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.

You will also find two sliding switches on the module. These switches are used to control the operational *mod*e of the device. For our purposes there are two operational modes of interest: the *boo*t mode and *ru*n mode. MicroStamp11 programs are written and compiled on a personal computer (PC) and then downloaded into the MicroStamp11 module over the PC’s USB port using a special *loade*r 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 right-hand 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 labeled with one or two names. The precise function of these pins will be explained as needed in the labs. Finally, the MicroStamp11 module has a “reset” button that allows you to restart a program that has been stored in the MicroStamp11.

------------------------------------------------------------------------------------------------------------------Modified by PG

**3.4. How does one communicate with the MicroStamp11?** You will need to load a program into the *μ*Stamp11’s EEPROM before it can do anything. To load such a program, raises an obvious question; how do you communicate with the *μ*Stamp11? Unlike your personal computer, the *μ*Stamp11 does not have a keyboard or terminal associated with it. The *μ*Stamp11 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 *μ*Stamp11, it must therefore be done through an intermediary. That intermediary is your personal computer.

Your personal computer communicates with the *μ*Stamp11 through its USB port. The human user communicates with the serial port through a *termina*l *progra*m such as PuTTY. PuTTY is a freeware program, so you can download it at home if you wish. It has also been installed on the computers in lab.

------------------------------------------------------------------------------------------------------------------------------------

**3.5. What are kernel functions?** An *operatin*g *syste*m (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 *kerne*l *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 USB port. The programs are C-language programs and we’re assuming the student is already familiar 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 PCs have the utility Microload that is used to download the executable ﬁle into Microstamp11.

------------------------------------------------------------------------------------------------------------------Modified by PG

As an example let’s consider creating a simple C-language program. Before starting, be sure that the files kernel.c and vector.c have been copied into the C:\icc11\include folder (drive and path may vary). These files can be downloaded from the course website.

------------------------------------------------------------------------------------------------------------------------------------

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 *kerne*l *function*s 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.

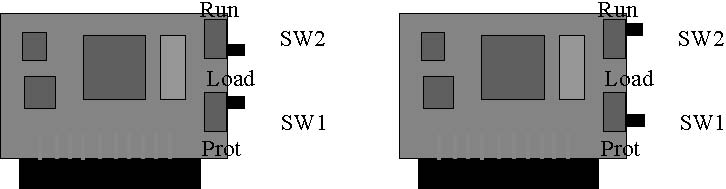
Note that the arguments for the function OutChar() are CR and LF. These two arguments are macros deﬁned in kernel.c. These macros associate the logical names CR and LF with the ASCII character byte generating a ”carriage return” and ”line feed”. So the end eﬀect of the instructions

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 PuTTY 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. 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.



BOOT mode: RUN mode:   
SW2 position to “Load” SW2 position to “Run”   
SW1 position to “Load” SW1 position to “Prot”

Figure 5. Switch Position for MicroStamp11

Once in boot mode, the microstamp11 is ready for program download. There is a windows based download program on the lab machines called MicroLoad. To download your program using this program, start the

program and click the large Load button. You will be presented with a ﬁle open dialog box. Choose the ﬁle you wish to download (in this case hello.s19) and click the Open button. The program will then ask you to slide the switches together on the MicroStamp11 and push reset. Do this and click OK. Your program should then be downloaded. When the utility Microload is ﬁnished then 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 PuTTY window. Your PuTTY program should be conﬁgured for serial communications using the assigned COM port at 38000 baud with 8 data bits and 1 stop bit. (See the instructor’s handout: “Running Your First Program on the MicroStamp11”) You then press the RESET button and your program should start executing. The PuTTY 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 *μ*Stamp11, 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 *μ*Stamp11 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 pinout for the *μ*Stamp11 (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 microcontroller’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 *μ*Stamp11 by simply pressing this reset button.

**4. Tasks (--- modified)**

**4.1. Pre-lab Tasks:**

(1) Include a picture of Breadboard Layout (with the MicroStamp11 and USB module).

(2) Explain how to program the MicroStamp11, including:

* Detailed step-by-step instructions (in your own words) for compiling a program using the ICC11
* Detailed step-by-step instructions (in your own words) for downloading a program using MicroLoad
* Detailed step-by-step instructions (in your own words) for running the program using PuTTY

(3) Write a C-language calculator program for the MicroStamp11. Your program should read two ***unsigned*** integers from the PuTTY, adds these integers together, and then write the answer to the PuTTY. After doing this the program should wait until a new addition problem is entered from the PuTTY. ***Note:*** ***Begin each program with a section of comments including your name, course number, lab number, filename of program (Lab1.c, for example), and a brief description of the program.*** ***All programs should include plenty of comments.***

(4) Your pre-lab writeup should have a listing of this program as well as an explanation of how the program works.

(5) List the min and max values (in decimal form) for signed integers and unsigned integers.

**4.2. In-lab Tasks:**

(1) No additional connections are required for this lab, but ask ***the instructor to verify that your USB and power connections are correct before proceeding.***

(2) Detailed handwritten comments (like a diary) of all activities during lab.

(3) Follow the instructions in the handout ***“Running Your First Program on the MicroStamp11”***

(4) Run your program and verify that it is working properly. Include at least 4 examples printed from the terminal program (PuTTy), including one with a result that is greater than 65535. Also test your program with negative numbers as inputs and discuss the result.

**4.3. Post-Lab Tasks:**

(1) Demonstrate the functionality of your program to the instructor and have the instructor 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.

(4) Explain the answer for the example that was greater than 65535.

(5) Discuss how well the program works and any limitations to the program.

**5. What you should have learned**

At the end of this lab, the student should be able to download a program to the *μ*Stamp11 and then start the program.

CHAPTER 3

**Lights and Switches -Hardware**

**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 lab has been broken into two parts. The ﬁrst part (this chapter) focuses on the hardware side of the system and the second part (next chapter) focuses on the real-time software needed by the lab.

**2. Parts List**

Italicized parts were used in the previous lab.

1. *(1) wire kit and breadboard*
2. *(2) MicroStamp11, docking module, and USB cable*
3. *(3) Computer with ICC11 compiler, MicroLoad, and PuTTY*
4. (4) one 7-segment LED display (LSD3221-11 or similar)
5. (5) seven 220 ohm resistors
6. (6) one 10 k-ohm resistor
7. (7) one button

**3. Background**

In this lab the student will build circuits that interface the MicroStamp11’s *input/outpu*t *port*s to *ligh*t *emittin*g *diode*s and *buttons*. In this particular lab, the student will then use some new *kerne*l *function*s 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 microcontroller 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 *logica*l *stat*e of a pin is said to be *hig*h 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 *port*s 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).

PA7 1

PA6 2

PA5 3

PA4 4

PA3 5

PA2 6

PA1 7

PA0 8

Reset\* 9

Vin 10

20 PD0/RXD

19 PD1/TXD

18 PD2/MISO

17 PD3/MOSI

16 PD4/SCK

15 PD5/SS

14 XIRQ\*

13 IRQ\*

12 +5 VDC

11 GND

MicroStamp 11 Pinout

The Master

74HC595 Pinout

The Slave

14 SER

12 RCLK

11 SRCLK

Figure 1. MicroStamp11 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 Microcontrollers 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 *input*s 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 *directio*n *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 *i*th bit in DDRD is set high, then the *i*th pin on PORTD has its directional state set to output. If the *i*th bit in DDRD is low (0), then the *i*th 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 *i*th 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.

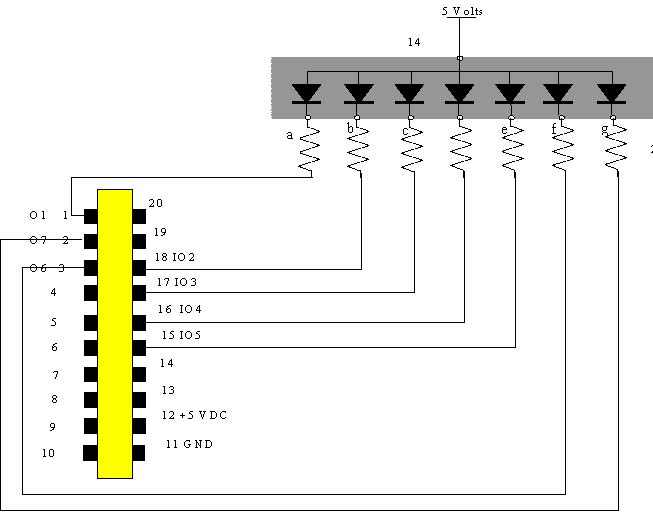
**3.2. What is a 7-segment LED?**  An LED is a *light* *emitting* *diode*. A diode is a two-terminal semi-conductor device that behaves something like an electronic valve. When the diode is forward biased, then the diode can conduct a substantial current and the LED emits lights. The intensity of this 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 Microcontroller’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 220 ohm resistor in order to limit the current load on the *μ*Stamp11 to a safe level. One possible connection that uses pins 1-3 and 15-18 on the *μ*Stamp11 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 Microcontroller. 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 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.



220 Ω

\

Figure 2A: 7-Segment Display connected to the MicroStamp11

7

6

3

2

1

8

9

10

11

14

Common

Anode

e

13

b

d

g

NC

c

f

a

DP

No

Pin

Common

Anode

No

Pin

No

Pin

Figure 2B: Common Anode 7-segment Display

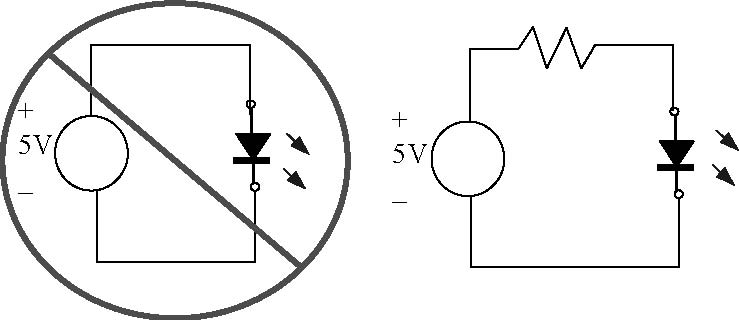
(common anode connection to Vcc)

**3.3. How does one connect a button to an I/O port?** The preceding section showed how you might connect the *μ*Stamp11 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 right-hand 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 pairs. It is recommended that you use the DMM to check which pairs of terminals are electrically connected when the button is pushed.

220 Ω



Dangerous diode The “acceptable” way

connection can result of connecting a

in large currents. diode.

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



a

b

a

b

top

side

b

a

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.

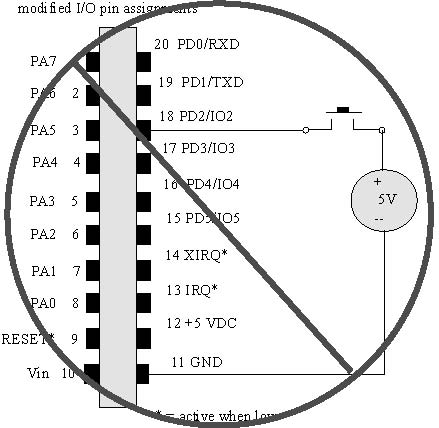
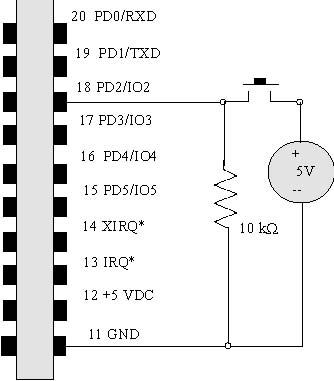
In this case, when the switch is closed, then we will have a small current going through the 10 k-ohm resistor and the potential between the input pin and GND is 5 volts. When the switch is opened, all of the charge left on the input pin will drain away to ground through this resistor and the potential at the input pin will be *pulled-dow*n to ground.

MicroStamp11 Directions with

modified I/O pin assignments

MicroStamp11 Directions with

modified I/O pin assignments



PA7 1

PA6 2

PA5 3

PA4 4

PA3 5

PA2 6

PA1 7

PA0 8

Reset\* 9

Vin 10

Figure 5. Use of pull-down resistor

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 (--- modified)**

**4.1. Pre-lab Tasks:**

(1) Draw a schematic diagram including the button circuit and the 7-segment display circuit (include pin numbers for the 7-segment display).

(2) Explain how button circuit works.

(3) Diagram identifying segments on a 7-segment display and a truth table indicating which segments are lit for each decimal digit.

(4) This lab will use common-anode 7-segment displays. Discuss the difference between common-anode displays and common-cathode displays. Include diagrams.

(5) Pinout and part number for the 7-segment display.

(6) Draw a diagram showing which segments are lit for each of the ten decimal values.

(7) Explain how the LED circuit works.

(8) Write a C-language program such that when the user presses a button an unsigned integer will be incremented (mod-10) and the value of the integer will be displayed both on PuTTY and on a 7-segment display. In particular:

A) when the button connected to PA0 is pressed, the integer should be incremented

B) include a delay (pause) to allow for switch debounce

C) the value of the integer should be displayed on PuTTY

D) a function named **display\_digit** should be called to diplay the integer value on a 7-segment display. The function should accept an integer between 0 and 9 as input and then use the clear\_pin and set\_pin kernel functions to display that digit on your LED display. Outputs O1-O7 should be connected to segments a-g on the 7-segment display. A partial example of such a function is provided below. This function will be used in several later programs in this course.

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;  
// similar for case 2 through case 9

}

}

(9) Remember to begin all programs with a section of comments including your name, course number, lab number, filename of program (Lab1.c, for example), and a brief description of the program. All programs should include plenty of comments.

**4.2. In-lab Tasks:**

(1) Detailed handwritten comments (like a diary) of all activities during lab. For example, if your program did not successfully compile or download, discuss what you did to correct the problem.

(2) Breadboard of button and 7-segment display circuit checked by instructor.

(3) Run the program which includes a delay used for debouncing. The display should operate perfectly. Print a sample of the results from the terminal program.

(4) Remove the delay from the program above. The display should now have problems with switch debounce. Press the button (somewhat quickly) ten times and record the sequence of numbers seen on the 7-segment display. Calculate the average number of bounces that occurred. Print out a sample of the results the terminal program. Can you tell that there was a switch debounce problem looking at the results from the terminal program? Explain.

**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) Discuss the problem of switch debounce and how it was handled in this lab.

(3) Demonstrate the functionality of your completed system to the instructor.

**5. What you should have learned**

After completing this lab you should know

* how to drive a7-segment display
* how to connect a push-button to the *μ*Stamp11
* how to write a C-program that reads to and writes from the I/O pins of the *μ*Stamp11.

CHAPTER 4

**Lights and Switches -real time**

**1. Objective**

In this lab, the student will rewrite the program written in the preceding lab. The new program will count the number of times a button is pushed in a 10 second interval and then display that number on a 7-segment LED display. This particular lab shows how a microcontroller can be used to build systems that control program execution in *real-time*.

**2. Parts List**

1. *(1) wire kit and breadboard*
2. *(2) MicroStamp11, docking module, and USB cable*
3. *(3) Computer with ICC11 compiler, MicroLoad, and PuTTY*
4. *(4) one 7-segment LED display (LSD3221-11 or similar)*
5. *(5) seven 220 ohm resistors*
6. *(6) one 10 k-ohm resistor*
7. *(7) one button*

**3. Background**

The program written in the preceding lab counted the number of times a button was pushed after the system was reset. In this lab, the student will rewrite the program so that it counts the number of button pushes over a 10-second interval and then displays that number on the 7-segment display (modulo 10). What is new here is that the microcontroller must precisely measure the 10-second interval and stop counting after that interval has expired. In other words, your program must be able to keep track of a timed deadline and must stop executing its count function when that deadline is over. Computer systems whose program executions satisfy a timed deadline are called *real-tim*e systems. Your particular system will need to use the MicroStamp11’s *kerne*l *function*s to accurately measure time intervals. Your system will also need to *debounc*e the switches in order to keep an accurate count of the button pushes.

**3.1. What is a real-time system?** In many cases, a computing system executes a program without any consideration for *ho*w *lon*g it takes that program to run. There are, however, numerous applications where program execution is time sensitive. If a microcomputer is used to deploy an airbag, then it is crucial that the microcomputer deploys the airbag as soon as a collision is detected. Another example occurs in automotive engine control systems. In this case, the microcontroller periodically samples engine performance and adjusts engine operation on the basis of these samples. For such schemes to work, however, it is crucial that the sampling be done at regular time intervals and this means that the microcomputer must complete processing of the sampled data within a speciﬁed time interval. Computing systems whose programs either respond to external events in a timely manner or complete program execution within a speciﬁed time interval are referred to as *real-time* systems. Real-time systems are characterized by the fact that some aspect of the program (initial response or execution time) satisﬁes a time *deadline*.

The MicroStamp11 can be used as the heart of a real-time embedded system. The module has a number of internal registers that can *interrupt* a program’s execution in response to either external events (such as an input pin going high) or to events generated by the hardware clock. You will take a closer look at these ”interrupts” in a later lab. In this lab, we provide a number of high-level kernel functions that use hardware interrupts to allow you to control the MicroStamp11’s timed response.

**3.2. What is button debouncing?** The preceding 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.

**3.3. 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 (assuminga1m-sec clocktick).

value = count (3, 1000);

**4. Tasks (--- modified)**

**4.1. Pre-lab Tasks:**

(1) Write a C-language program (Program 4A - One Second Timer) to increment and display a count (mod-10) in one-second increments on a 7-segment display after a button is pressed. Display the count on the terminal program (PuTTY) as well. Include the program listing (with many comments).

(2) Write a description of how Program 4A works.

(3) Write another C-language program (Program 4B - Ten Second Button Counter) that counts the number of times a button is pushed over a 10-second interval. Your program should do the following:

* The timed interval starts with a button push.
* The system will count the number of times the button is pushed over a 10 second interval.
* The system should stop the count when the 10 second time interval has expired.
* The program should display all digits of the count on the 7-segment LED display. The most signiﬁcant digit of the count should be displayed ﬁrst and the next signiﬁcant digit should be displayed after a button push (prompt the user).
* After all digits have been displayed, the program should return to the start and wait for a button push to start the next timed interval.
* The count should be displayed on PuTTY as well.
* Give the user clear prompts for all inputs.
* Include the program listing in your notebook (with many comments).

(4) Write a description of how Program 4B works.

(5) Include a schematic diagram breadboard layout.

The following code segment is a ”partial” listing for program 4B that you can use as a starting point.

void main(void){  
init();  
while(1){  
 pause(1000); // 1 second pause for standard mode – use 3250 for Turbo mode  
 button(6);  
 icount=count(6,10000); // replace 10000 with 32500 for Turbo mode  
 display\_digit(icount);  
}}

**4.2. In-lab Tasks:**

(1) Compile and download Program 4A and demonstrate it to the instructor. Record any changes made to the program. Include a printout from the terminal program.

(2) Use an online stopwatch program to determine the amount of time required for your program/circuit to display 60 counts. Record this time. If the time is not within 1 second of being correct (60 seconds), adjust the length of pause used in your program. Make a table showing the values used with the pause statement and the corresponding times.

(3) Compile and download Program 4B and demonstrate it to the instructor. Record any changes made to the program.

(4) Experimentally verify that Program 4B works correctly. Record (in a table) the number of button presses versus the displayed value for several cases, including the following number of button presses: 0, 1, 2, 5, 10, 15, 20, 25, and the highest number that you can press. Include a printout from the terminal program.

(5) Include detailed handwritten comments (like a diary) of all activities during lab.

**4.3. Post-Lab Tasks:**

(1) Include a final program listing for each program. Highlight any changes made to the original program listing and discuss the changes.

(2) Discuss how well each program performed.

**5. What you should have learned**

After completing this lab you should know:

* what a *real-tim*e system is
* how to use the kernel functions to pause program execution
* how to debounce a switch

CHAPTER 5

**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 take 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. *(1) wire kit and breadboard*
2. *(2) MicroStamp11, docking module, and USB cable*
3. *(3) Computer with ICC11 compiler, MicroLoad, and PuTTY*
4. *(4) one 7-segment LED display (LSD3221-11)*
5. *(5) seven 220 ohm resistors*
6. (6) two 10 k-ohm resistor
7. (7) two buttons
8. (8) two R ohm resistors (use two 2.2 kΩ resistors) --- mod
9. (9) four 2R ohm resistors (use eight 2.2 kΩ resistors) --- mod

**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 × 22)+(1 × 21)+ (0 × 20)=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.

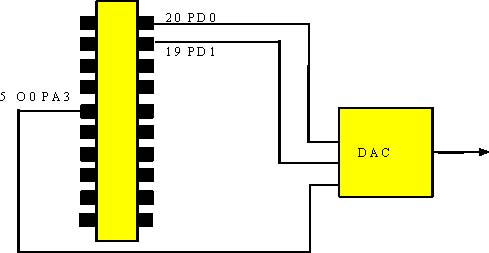


Figure 1. The DAC’s block diagram

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

If we wish to generate a voltage that lies somewhere between the low ground voltage level (zero) and the ”high” voltage level (5V), 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



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.

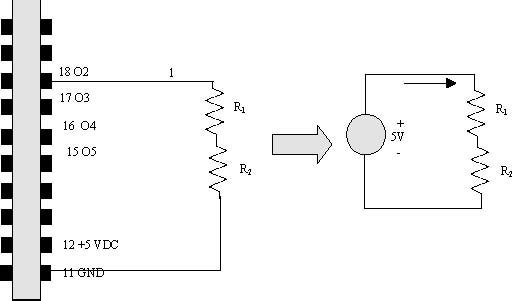
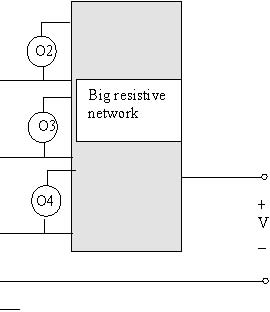
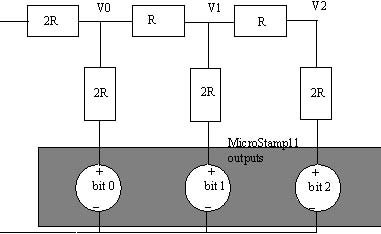


Figure 2. Voltage divider: a preliminary DAC design

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 *R*2R *ladde*r *network*. A schematic drawing of the R2R ladder network is shown in the right-hand side of ﬁgure 3. This network requires two diﬀerent values of R and 2R ohms. You can choose R to be 2.2 kΩ. The independent voltage sources shown in the ﬁgure 3 are the MicroStamp11’s output pins.

+

Vout

\_

GND

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 (--- modified)**

**4.1. Pre-lab Tasks:**

(1) Draw a schematic showing the MicroStamp11 connected to the following:

A) A 2 button-circuit that connects to pins PA0 and PA1, respectively. Note that you should be able to reuse the single button you already assembled in the previous lab.

B) An R2R ladder network (using only 2.2 kΩ resistors) that is driven by the 3 output pin PA3, PD0, and PD1.

C) The 7-segment display and current-limiting resistors used in previous labs.

(2) Show a breadboard layout for the schematic above.

(3) Determine the expected output voltage V2 (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. Use superposition as shown in the class notes.

(4) Summarize the results above in a table of binary inputs (000 – 111) versus analog output (V2).

(5) Plot the predicted voltage V2 as a function of the binary value of the input (shown on the display) using a ***stair-step graph***. See the class notes for an example of a stair-step graph.

(continued)

(6) Write a C-language program that:

A) Increments a ***signed integer*** variable when the PA0 button is pressed and decrements an unsigned integer variable when the PA1 button is pressed.

B) Uses the ***display\_digit*** function from previous labs to display the integer variable (modulo 8) on the 7-segment display.

C) Calls a function (that you must write), ***binary\_output***, to output the appropriate binary values to the R2R ladder network to produce the output voltage V2.

D) Includes the command ***disable\_sci()*** in order to use pins 19-20 (PD1 & PD0) for outputs instead of for serial communication. Note that we will not be able to communicate to PuTTY after executing this command.

E) Include the modified kernel function, ***kernel5.c***, as discussed in the class notes.

(7) Print out a few lines from kernel5.c and highlight the change made to line 437.

(8) Write an explanation of how the program works.

(9) Explain why the function disable\_sci( ) was used.

The main program might look something like:

**#include “kernel5.c” // modify kernel.c to comment out the instruction wait\_pin(pin); //in the button function (perhaps line 437)**

**.**

**.**

**void main(void){**

**int count;**

**init( );**

**disable\_sci( ); //disable serial comm so that pins 19-20 can be used as outputs**

**count = 0;**

**while(1){**

**if (read\_pin(6) == 1 && read\_pin(7) == 0) //increment button pressed**

**{button(6); //use modified button command in kernel5.c**

**count++;} //increment counter**

**if (read\_pin(6) == 0 && read\_pin(7) == 1) //increment button pressed**

**{button(7); //use modified button command in kernel5.c**

**count--;} //decrement counter**

**…**

**… (call function display\_digit)**

**… (call function binary\_output)**

**… (add instructions for mod-8 operation)**

**}}**

The binary\_ output function might look something like:

**void binary\_output(count){**

**DDRD |= bit(0); // set the direction of PD0 to 1 (output)**

**??? // set the direction of PD1 to 1 (output)**

**switch(count){**

**case 0: // b2b1b0 = 000**

**PORTD &=^ bit(0); // b2 = 0**

**PORTD &=^ bit(1); // b1 = 0**

**clearpin(0); // b0 = 0**

**break;**

**case 1: // b2b1b0 = 001**

**(etc)**

**4.2. In-lab Tasks:**

(1) Build the circuit consisting of the two button inputs and the R2R ladder network using only 2.2 kΩ resistors. Ask the instructor to check your circuit before proceeding.

(3) Download and test your program to verify that the display properly increments and decrements (mod-8) and that the expected voltages are produces by the R2R ladder network (measure V2 using a DMM).

(4) Record the output voltage V2 measured with a DMM for each of the 8 possible inputs.

(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) Final program listing. Highlight any changes made to the original program listing and discuss the changes.

(2) Stair-step graph of measured analog voltages (V2) versus possible binary inputs.

(3) Table comparing predicted output voltages (V2) to measured output voltages (V2). Include % error.

(4) Assessment of how well the DAC works. Discuss reasons for the error in the measured values of the output analog voltage (V2).

(5) Demonstrate the functionality of your ﬁnal system to the instructor.

**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
* have acquired more experience in programming the MicroStamp11 and breadboarding circuits

CHAPTER 6

**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, *Vr*, generated by your DAC against a reference voltage, *V* and to return a high logical voltage of 5 volts if *V > Vr* and return zero volts otherwise.

**2. Parts List**

1. *(1) wire kit and breadboard*
2. *(2) MicroStamp11, docking module, and USB cable*
3. *(3) Computer with ICC11 compiler, MicroLoad, and PuTTY*
4. *(4) one 7-segment LED display (LSD3221-11)*
5. *(5) seven 220 ohm resistors*
6. *(6) two 10 k-ohm resistor*
7. *(7) two buttons*
8. *(8) two R ohm resistors (use two 2.2 kΩ resistors) --- mod*
9. *(9) four 2R ohm resistors (use eight 2.2 kΩ resistors) --- mod*
10. (10) one LM660 quad op-amp IC
11. (11) two resistors (*R*) for op-amp buﬀer
12. (12) two 1N4007 diodes
13. (13) one 10 k-ohm trim potentiometer
14. (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, *Vr*, that is compared against the voltage, *V*, you wish to convert into a 3 bit digit. The *referenc*e *voltag*e V will be generated by a potentiometer. In this lab you will use diodes and *operationa*l *ampliﬁer*s to build circuits that *compar*e the two voltages, *buﬀe*r your R2R ladder network from this new circuitry, and then *clam*p 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

VCC (5 V)

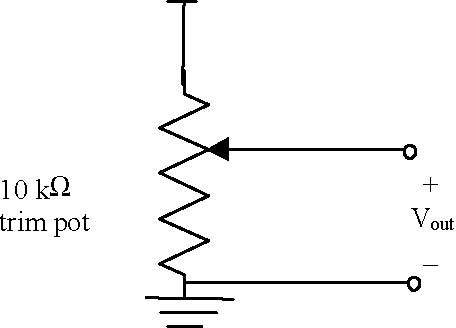


Figure 1. Reference voltage source is supplied by a potentiometer

**3.2. What is an operational ampliﬁer?** A voltage *ampliﬁe*r is a special circuit that accepts an input voltage, *V*in and outputs a voltage, *V*out = *AV*in that is proportional to the input voltage. The proportionality factor A is called the *gai*n of the ampliﬁer. If *A*> 1, then the ampliﬁer actually does *amplif*y the input voltage. If *A<*1, then the ampliﬁer *attenuate*s the input voltage.

An *operationa*l *ampliﬁe*r 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*out = *A*(*V*+ *− V−*) and such that *A* is a 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 *suppl*y *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 105 − 107 . In addition to this the op-amp circuitry is designed so that



Vout = A

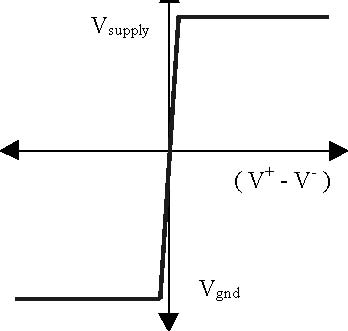
Vgnd

Vsupply

Vout = A(V+ - V-)

V+

V-



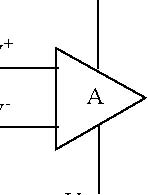


Figure 2. Opamp symbol

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 *dependen*t 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.

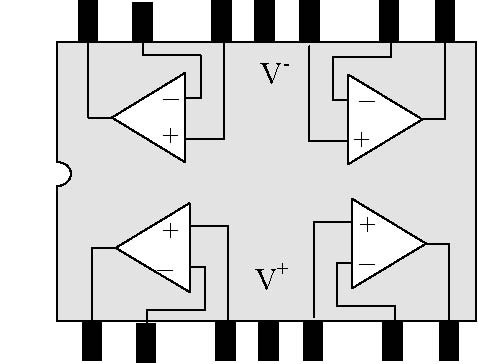
Dependent voltage sources are a very good approximation for the op-amp’s behavior. In other words, the op-amp is a circuit that has been *engineere*d to be well approximated by an idealized circuit element . This means that we can use op-amp models in a reliable manner to predict the behavior of op-amp 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 *suppl*y *voltage*s 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 *±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 *±V*s volts). By quad, we mean that there are 4 op-amps on a single chip. The pinout for the LM660 is shown below in ﬁgure 3.

14 13 12 11 10 9 8



1 2 3 4 5 6 7

Figure 3. Pin out for LMC660 Quad Op-amp

**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 *isolate*s or *separate*s 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 *sourc*e to drive other circuits. The problem is that if we were to attach another circuit to our DAC, then we would be changing the *R*2R ladder network and hence would change the

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.

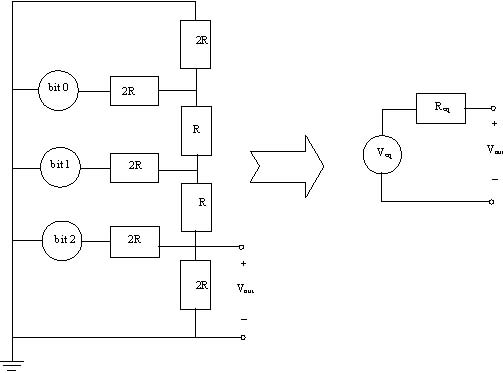


Figure 4. Thevenin equivalent of the DAC network

Assuming that the Thevenin equivalent voltage *V*eq and resistance *R*eq are known, then we can go ahead and 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

(3.1) 

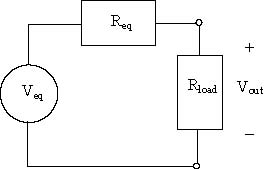


Figure 5. Loaded Thevenin equivalent circuit

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*load*/*(*R*load + *R*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 kΩ and let’s assume that *R*load equals 8 Ω. This rather low load resistance is common for some devices such as audio speakers. The ratio is now readily seen to be



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*load*/*(*R*load + *R*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 *buﬀerin*g *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 modeled 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*in = *V*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.

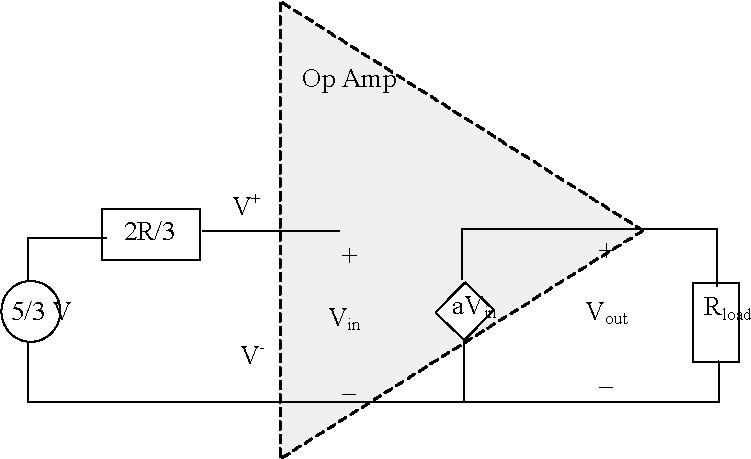


Figure 6. Circuit diagram for DAC buﬀered by unity gain ampliﬁer

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.

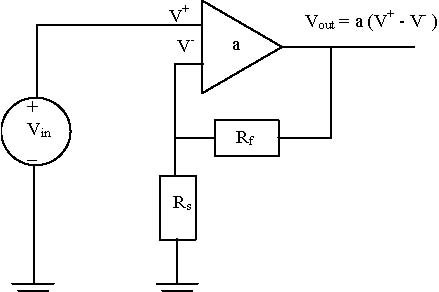
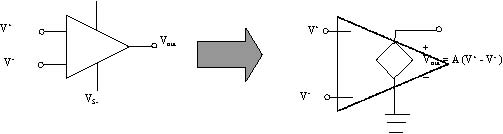


Figure 7. Non-inverting op-amp connection

**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 (105*−*107), 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.

VS+



V+

V-

V+

V-

VS-

Vout

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”.

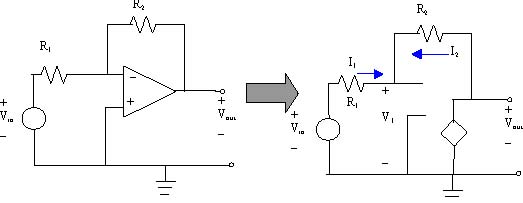


Figure 9. Inverting Op-amp Feedback Circuit

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*out = *−AV*1

where *V*1 is the voltage from the negative terminal to ground. Because A is large, on the order of 106, 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= Vin/R1.

We also know that the current, *I*2, through the feedback resistor, *R*2, must be *I*2 = Vout/R2. 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



which simplifies to



In other words, the output voltage is proportional 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 106. 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 *cli*p 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 *clam*p *circui*t is a special type of circuit that is used to limit or *clam*p 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. If more than 5 volts is applied to the input of the circuit, the diode gets forward biased and hence the voltage at the node *Vout* is the sum of the diode voltage drop and the 5V supply. Since the voltage is more than 5V at this node, a voltage divider circuit is put in place to make the output voltage equal to 5V. If the input voltage is less than 5V then the voltage at node *Vout* is equal to the applied voltage. Thus the output voltage will be always less than or equal to 5V.

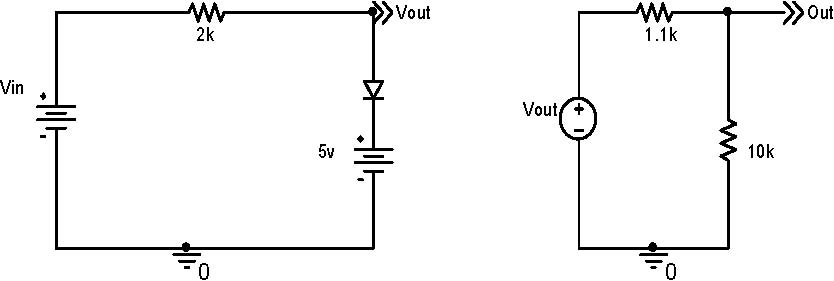


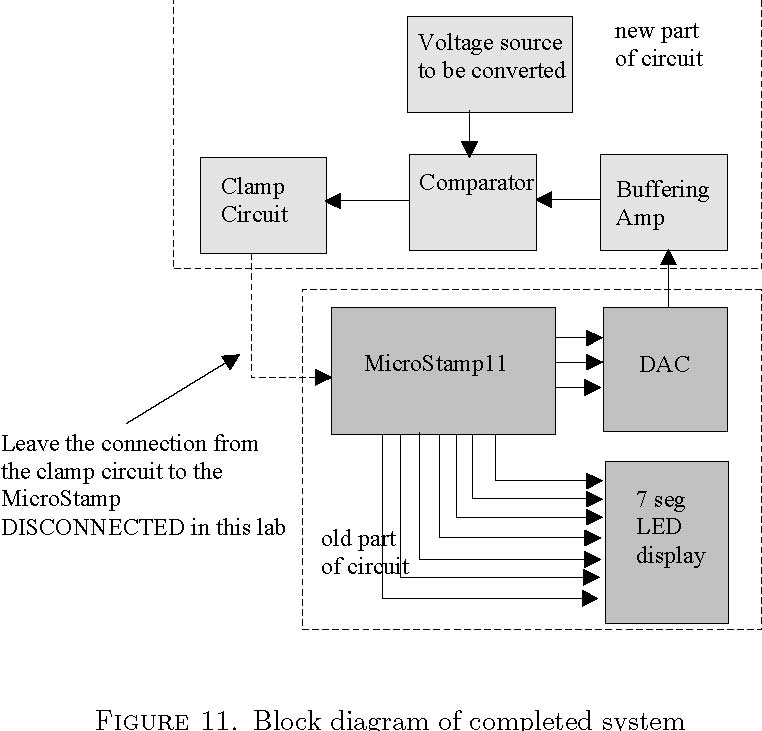
Figure 10. Clamp Circuit

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
* 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.



**4. Tasks (--- modified)**

**4.1. Pre-lab Tasks:**

(1) Draw the schematic of a unity-gain buffer. Explain how the circuit works. Write an expression for the output voltage.

(2) Draw the schematic of a comparator circuit that compares a reference voltage (0-9V) generated by a 10 kΩ potentiometer to the input voltage generated by your buﬀer circuit. Your circuit should generate a voltage that is either 0 or 9 volts. Explain how the circuit works. Write an expression for the output voltage.

(3) Draw the schematic of a clamp circuit that clamps the comparator’s voltage to either zero or ﬁve volts. Explain how the circuit works. Write an expression for the output voltage.

(4) Show the calculation for the resistance in the clamp circuit so that the maximum current drawn from the source is around 1 mA.

(5) Show a complete schematic including the MicroStamp11, R2R ladder network, buffer, comparator, and clamp circuit.

(6) Draw the breadboard layout for the schematic above.

**4.2. In-lab Tasks:**

(1) Note: It is best to build and test one circuit at a time rather than building the entire circuit.

Build the buffer circuit and connect it to the DAC you built earlier. Test your circuit by measuring the buffer output voltage versus the buffer input voltage from the DAC for all 8 possible cases. See sample table in lab notes.

(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 (80 total measured values). See sample table in lab notes.

(3) Build the clamp circuit and connect it to the comparator. (The thumbwheel potentiometer seems to work best here.) Adjust the potentiometer on the output of the clamp circuit to exactly 5V when the output of the comparator circuit is 9V and then put a piece of tape on the potentiometer so that it will not be accidentally changed. 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 (80 total measured values). See sample table in lab notes.

(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. See sample table in lab notes.

(5) Describe what happened in the lab.

(6) Demonstrate your circuit to the instructor. The instructor 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 input buffer voltage for each of the 8 cases. Assess how well the buffer works.

(2) Plot the experimental data for the comparator circuit. Note: Use either 8 graphs of Vcomp vs. Vref (1 for each DAC input) or else use one 3D graph (80 points total). Assess how well the comparator circuit works.

(3) Plot the experimental data for the clamp circuit. Note: Use either 8 graphs of Vclamp vs. Vref (1 for each DAC input) or else use one 3D graph (80 points total). Assess how well the clamp circuit works.

**5. What you should have learned**

At the end of this lab you should know:

* how to design a 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

CHAPTER 7

**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 *algorith*m 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 log2 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 log2 8 = 3 guesses.

**3.2. What is a binary search?** The ”guessing game” is an example of a *binar*y *searc*h 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 *binar*y *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*lower ≤ *V*in ≤ *V*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



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

**4. Tasks (--- modified)**

**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 basic structure of your program, therefore, should be as outlined below:

#include "kernel.c"

#include "vector.c"

void main(void){  
init();

disable\_sci( );  
while(1){

for(i=0; i < N; i++){  
guess = (upper + lower)/2;  
binary\_output(guess);  
if((PORTA & bit(2)) != 0) {upper = guess;}

else{lower = guess;}

}  
display\_digit(guess);  
}

(2) Explain how the program works.

(3) Include a schematic diagram.

(4) Plot a stair-step graph of predicted digital outputs (to be displayed on the 7-segment display) as a function of applied reference voltage. Use the data table of threshold voltages from last lab.

**4.2. In-lab Tasks:**

(1) Verify that the circuit/program from Lab 6 is still functioning correctly and that the clamper circuit produces outputs between 0 and 5 volts (the potentiometer should still be taped so that the 5V value does not change). It is important that this voltage be correct before feeding it back to the MicroStamp11 as an input.

(2) If the previous step was correctly verified, close the loop between the clamp circuit and the MicroStamp11 by connecting the clamp circuit’s output to pin PA2.

(3) Compile and download your program to the MicroStamp11.

(4) Test your program to see if it is working correctly. As you vary the reference voltage (potentiometer) from 0 to 5V, the 7-segment display should change from 0 to 7.

(5) Record the output value on your 7-segment display as the reference voltage varies from 0 to 5V in 0.2V increments.

(6) Demonstrate the functionality of your system to the instructor.

**4.3. Post-Lab Tasks:**

(1) Print a final program listing. Highlight any changes made to the original program listing and discuss the changes.

(2) Graph of digital output (shown on 7-segment display) versus applied reference voltage (stairstep graph using the 26 measured data points).

(3) Comparison of graphs from pre-lab and post-lab.

(4) Assessment of how well the ADC works.

**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

CHAPTER 8

**Pulse Width Modulation**

**1. Objective**

The purpose of this lab is to have the student rewrite the output compare interrupt handler in kernel.c to make the MicroStamp11 generate a pulse width modulated (PWM) waveform with a period of about 4 msec and whose duty cycle is programmable. This PWM generator will be used in the next lab to simplify and enhance the functionality of the DAC module you built earlier.

**2. Parts List**

Same as lab 6’s parts list.

**3. Background**

This lab rewrites the kernel’s *interrupt handler* for an *output compare event*. Your rewritten interrupt handler will generate a *pulse width modulated* (PWM) signal with a programmable duty cycle and 4 msec period. The PWM signal will be output on I/O pin PA4. You will measure the performance of your PWM signal generator using a piece of test equipment known as an *oscilloscope*.

**3.1. 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*: *→*. 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 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 *perio*d of the signal. If T is the smallest positive number such that *v*(*t*)= *v*(*t*+ *T*) (for all *t ∈ ℜ*), 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-widt*h *modulate*d signal is a *T*-periodic signal, *v*, if there exists 0 *<T*1 *<*T such that

(3.1) 

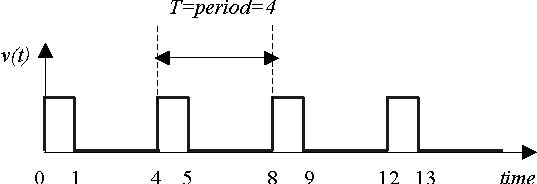
for t ∈ [0*,*T ]. We refer to the ratio *T*1*/*T as the *dut*y *cycl*e of the signal. We usually represent the duty cycle as a percentage. Equation 3.1 deﬁnes the values that v takes over a single fundamental period, T. Since v is T-periodic, we know that the pattern characterized in equation 3.1 will 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.

**3.2. What is an Interrupt Handler?** Let’s consider a program that the MicroStamp11 is executing. A program is a list of instructions that the microcontroller executes in a sequential manner. A *hardware event* is something special that happens in the microcontroller’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 microcontroller generates a *hardware interrupt*. The interrupt forces the microcontroller’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 microcontroller to the original program it was executing. So a hardware interrupt allows a microcontroller to interrupt an existing program and react to some external hardware event. This type of ﬂow control is illustrated in Figure 2.

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 microcontroller supervising the operation of various devices on the car’s dashboard. Usually, this microcontroller 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 microcontroller 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 *μ*Stamp11? 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 microcontroller stops everything and begins re-executing the program.

PROGRAM

Sequential

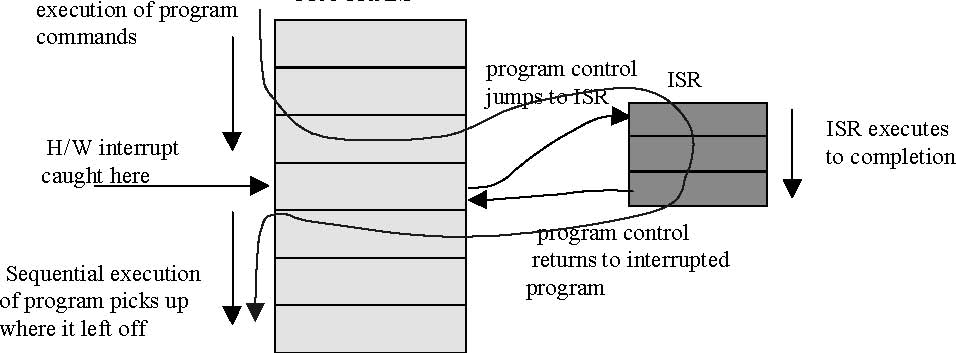


Figure 2. Control ﬂow in the presence of a hardware interrupt

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 *μ*Stamp11 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.

**3.3. What is an Output Compare Event?** The output compare event is a hardware event tied to the microcontroller’s *real-tim*e *clock*. The real-time clock on the *μ*Stamp11 is a hardware subsystem within the *μ*Stamp11 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 *μ*Stamp11’s clock is running at a 9.83 MHz rate.

|  |  |  |
| --- | --- | --- |
| PR1 | PR0 | TCNT clock rate |
| 0 | 0 | 407 ns |
| 0 | 1 | 1.628 μs |
| 1 | 0 | 3.255 μs |
| 1 | 1 | 6.511 μs |

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 *outpu*t *compar*e *register*. When the value in TCNT matches the number in the output compare register, we trigger an *output-compar*e *event*. The Motorola 68HC11 microcontroller 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 microcontroller’s output pins.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| TFLG1 |  |  |  |  |  |  |  |
| OC1F | OC2F | OC3F | OC4F | OC5F | IC1F | IC2F | IC3F |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| TMSK1 |  |  |  |  |  |  |  |
| OC1I | OC2I | OC3I | OC4I | OC5I | IC1I | IC2I | IC3I |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| TCTL1 |  |  |  |  |  |  |  |
| OM2 | OL2 | OM3 | OL3 | OM4 | OL4 | OM5 | OL5 |

Figure 3. Hardware Registers used by Output compare interrupt

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

* the interrupt is *armed*.
  1. *Armin*g an interrupt means to enable the source of the interrupt. We *ar*m 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
  1. *Enablin*g the interrupt means that the software pays attention to the interrupt. We *enabl*e all interrupts by clearing the I bit in the condition code register of the microcontroller. This bit is usually cleared in the init() function using the assembly command ***cli***.
* and any previous interrupts are acknowledged.

*Acknowledgin*g an interrupt means that we tell the system that a previously received interrupt has been serviced. We *acknowledg*e 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 *instal*l 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 *OM*x and *OL*x 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 microcontroller 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.

**3.4. 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. (1) initialization of any global variables and counters in the ISR
2. (2) arming the interrupt
3. (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 ns from the current time (128 *μ*s).

**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.

**3.5. 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 microcomputer 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.

To get the student started, therefore, we’re showing the 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.

**3.6. 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 multimeter (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. This lab uses a Tektronix analog scope. The control panel of this scope is shown in ﬁgure 4

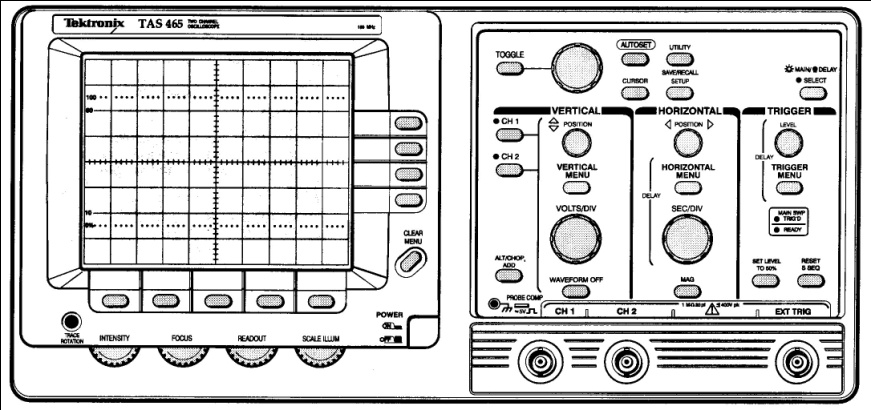


Figure 4. Tektronix Analog Oscilloscope Control Panel

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 as well as the front panel of the Tektronix scope that you’ll probably be using. Note that there are essentially three important types of controls that you need to be able to use. These controls are the

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

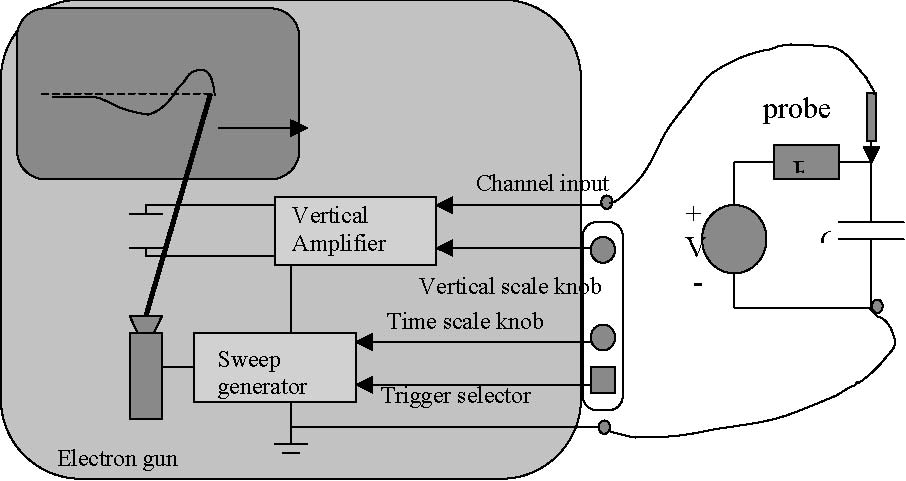


Figure 5. Diagram of the internal subsystems in an analog oscilloscope

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*.

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.

In recent years a big advance in oscilloscope technology occurred. This advance was the *digita*l *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 scope can do many things that analog scope 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.

**4. Tasks (--- modified)**

**4.1. Pre-lab Tasks:**

(1) Explain how the initial OC4han interrupt handler works (50 percent duty cycle PWM).

(2) Program Listings: Make three modifications to kernel.c and write a main program as follows. Save the modified kernel.c as kernel8.c.

A) Init(): Modify init( ) for a clock period of 407 ns. Highlight all changes and explain how it works. Add comments to all new or changed code. See class notes for more details.

B) kernel8.c: Modify kernel.c as follows: Define unsigned integers ***highticks*** and ***lowticks*** and write a new function named ***setpwm(D)*** that will convert the duty cycle, D, into the appropriate number of highticks and lowticks based on a 9828 time tick period. Highlight all changes and explain how it works. Add comments to all new or changed code. See class notes for more details.

C) OC4han(): Modify the OC4 interrupt handler so it generates a PWM signal on PA4 that has approximately a 4 msec period (9828 time tick period) and a programmable duty cycle. In other word, use highticks and lowticks rather than the fixed 1024 ticks in the current function. Highlight all changes and explain how it works. See class notes for more details.

(D) Main Program: Write a C-language program that prompts the user to enter a duty cycle D (0 to 100) and then calls the function setpwm. Include the new kernel function (kernel8.c). Include many comments.

**4.2. In-lab Tasks:**

(1) Remove the R2R ladder network, buffer circuit, comparator circuit, and clamp circuit from your breadboard. Leave the 7-segment display connected.

(2) Compile and download your main program written in the pre-lab task. Be sure that kernel8.c has been saved in the include folder for the ICC11 compiler.

(3) View the output on PA4 on the oscilloscope. Verify that your program is working correctly. As you enter different values for D, you should see the period of the waveform change on the oscilloscope.

(4) Capture oscilloscope images on the computer in lab to capture each of the 11 oscilloscope screens (for D = 1%, 10%, 20%, …, 80%, 90%, 99%). Add cursors to each screen to measure TH. Verify that the period is 4 ms (or f = 250 Hz). Print each of them and tape them into your lab notebook. Show the input value of D next to each screen capture. Also show the calculated value of D using the measured value of TH. See a sample screen capture in the class notes.

(5) Description of what happened during the in-lab task.

**4.3. Post-Lab Tasks:**

(1) Form a table comparing the duty cycle input to your program to the duty cycle measured on the oscilloscope.

(2) Assess the performance of your program.

(3) Final program listings for any programs that were modified during lab. Highlight any changes made to the original program listings and discuss the changes.

(4) Demonstrate the functionality of your completed system to the instructor.

**5. What you should have learned**

After completing this lab you should know:

* What an interrupt handler is and how to write one for the MicroStamp11,
* How to use the output compare interrupt to generate a PWM signal
* How to use an oscilloscope

CHAPTER 9

**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. *(1) wire kit and breadboard*
2. *(2) MicroStamp11, docking module, and USB cable*
3. *(3) Computer with ICC11 compiler, MicroLoad, and PuTTY*
4. *(4) one 7-segment LED display (LSD3221-11)*
5. *(5) seven 220 ohm resistors*
6. *(6) two 10 k-ohm resistor*
7. *(7) two buttons*
8. *(8) one LM660 quad op-amp IC*
9. *(9) two resistors (R) for op-amp buﬀer*
10. *(10) two 1n4007 diodes*
11. *(11) one 10 k-ohm trim potentiometer*
12. *(12) two additional resistors for diode clamp circuit*
13. (13) additional resistor for RC network
14. (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. Microcontrollers, however, can do more than manipulate digital data. Microcontroller must have internal clocks that precisely orchestrate the various digital computations performed by the device. In other words, the microcontroller can detect and generate timed events with a great deal of precision. This is precisely what you took advantage of in the preceding lab when you used the MicroStamp11 to generate a pulse width modulated (PWM) signal.

In this lab you will build a DAC that uses the PWM signal you generated in the previous lab to drive a simple circuit consisting of a capacitor and resistor. This circuit is called an *R*C *circuit*. The RC circuit’s *respons*e *t*o *th*e *PW*M *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.

**3.1. What is an RC Circuit?** An *R*C 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,



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 *capacitanc*e 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 *cerami*c 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 1. 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

1. Another type of capacitor is the *electrolyti*c 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 1). 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 1.

standard (ceramic) electrolytic capacitor capacitor

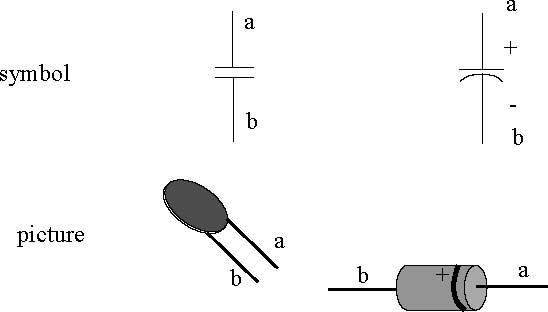


Figure 1. Symbols and drawings of capacitors

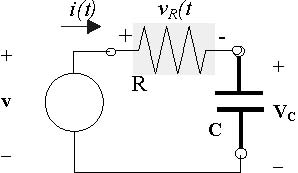
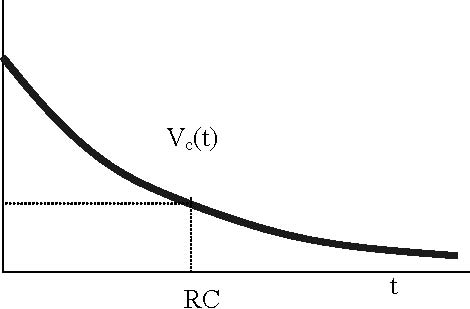
An *R*C circuit is a particularly simple network containing a capacitor. The *R*C circuit consists of an independent voltage source in series with a resistor, *R*, and a capacitor *C*. The schematic diagram for this circuit is shown in Figure 2. Analyzing this circuit means determining the voltage over the capacitor, *vc*(*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 *natura*l *respons*e and *ste*p *response*. The derivation of these particular response equations is done in the lecture component of the course.

Figure 2. RC circuit

*Natura*l *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 ≥ 0). This particular solution is called the *natura*l *respons*e of the *R*C circuit and it can be shown to

|  |
| --- |
| have the form  (3.1) |
|  |

It is valuable to plot the general shape of the natural response in equation 3.1. Note that the voltage has a time dependency that is an exponential function of time. This exponential function, *e−t/R*C 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 ≥ 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 3.



V0

0.368V0

0

Figure 3. Natural Response

Note that the expression, *RC*, has units of time. We generally refer to *R*C as the *tim*e *constan*t of the circuit. In fact, at time t = *RC*, we know that the voltage is *v*c(nat)(*RC*)= *V*0*e−1* ≈ 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)(3*RC*)= *V*0*e−3* ≈ 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 *R*C *constan*t of our circuit.

Standard capacitor values are on the order of *μ*F (a large capacitor) to pF. If we were to use a 1 k-ohm resistor in series with a 1 *μ*F capacitor, the RC constant would be *R*C = 1000 × 10*−6* = 1 ms. In this case, our source-free circuit would discharge the capacitor in about 4-5 ms. If we were to use an even smaller capacitor, let’s say about 100 pF, then this discharge time would be even shorter. In particular, for a 100 pF capacitor in series with a 1 k-ohm resistor, we would expect a time constant of *R*C =103 × 100 × 10*−*12 =10*−7* sec. This is one tenth of a microsecond. So in this case we would discharge a capacitor in about half a microsecond, a very short time interval.

*Ste*p *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,



where *u*(*t*) is a unit step function. The capacitor’s response to this particular “step” input is called the *ste*p *respons*e of the RC circuit. The step response can be shown to have the following form,

(3.2) 

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,

(3.3) *vc*(*t*)= V (1 − e *−t/RC*)*u*(*t*)

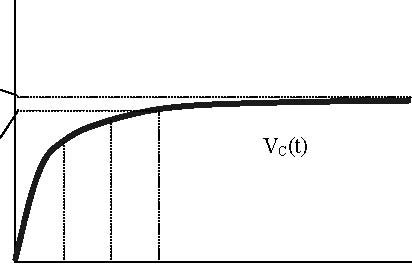
for all *t*. Figure 4 plots this function for t ≥ 0. This ﬁgure shows that the initial voltage over the capacitor

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(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.

If we do not neglect the initial charge on the capacitor, then the circuit’s response is given by equation 3.2. Notice that this equation is simply the sum of equation 3.3 and the natural response in equation 3.1. So we can simply sum the two responses shown in Figure 3 and 4 to obtain a plot of the system’s total response.

Figure 5 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 5. 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.

**3.2. What is the RC circuit’s response to a PWM signal?** In this section, we discuss the RC circuit’s (ﬁgure 2) 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 6. Over a single period, [0*,*T], the



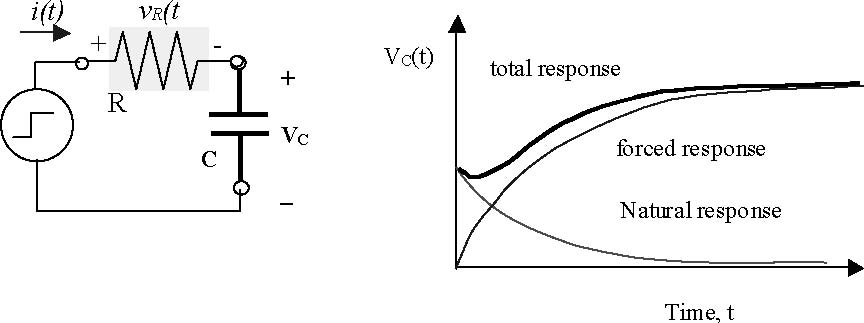
V

0.95V

0 RC 2RC 3RC

Figure 4. Step Response assuming uncharged initial capacitor

vR(t)



**+**

**v(t)**

**\_**

Figure 5. Total Response of RC Circuit

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.

*T=period=4*

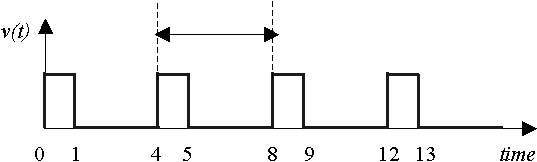


Figure 6. Pulse Width Modulated Signal

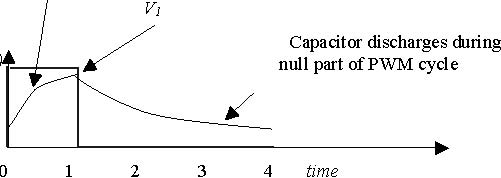
During the “charging” phase, we can think of the RC circuit as being driven by a step function of magnitude V volts. If we assume that the capacitor has an initial voltage of *V*0 at the beginning of the charging phase (time *t* = 0), then the circuit’s response is simply given by equation 3.2 for *t* ∈ [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* ∈ [*T*1*,T*] is given by equation 3.1 of the form *V*1*e−*(*t−T*1)*/R*C for *t* ∈ [*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 7 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 7. 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*.

Capacitor charges during

positive part of PWM cycle



Capacitor charges during

null part of PWM cycle

vc(t)

Vo

Steady state response

exhibits “ripple” about a

steady state value.

*T=period=4*

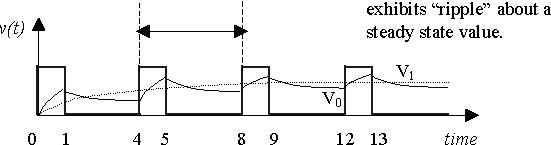


Figure 7. Response to PWM signal over a single period

The steady state region shown in Figure 7 is usually characterized by two “ﬁgures of merit”. The ﬁrst “ﬁgure of merit” is the *mea*n *voltage*, *Vm*, of the steady state response and it is given by the equation



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



We often specify the ripple as a percentage in which *V*r is normalized by the steady state voltage *V*m. For example if *V*m = 10 volts and *V*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 7, 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*m must vary in a linear manner with the commanded voltage and its ripple, *Vr*, should be 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.

**4. Tasks (--- modified)**

**4.1. Pre-lab Tasks:**

(1) Consider an RC circuit whose input is a PWM signal with a period T and pulse width *T*1 . Derive an equation for the signal’s steady state value, Vm, and an equation for the ripple voltage, Vr, as a function of *R*, *C*, T1, and T (i.e., show the derivation of Eq. 5 and Eq. 6 in the class notes.) Show all steps.

(2) Assume that the PWM signal has a 50 percent duty cycle, *R*C is 10 ms, and V = 5V. Substitute these values into Eq. 5 and Eq. 6 for Vm and Vr to find expressions for Vm and Vr as a function of T (i.e., show the derivation of Eq. 7 and Eq. 8 in the class notes.)

(3) Use Eq. 7 and Eq. 8 to calculate Vm , Vr and % ripple as T varies from 0.1ms to 3 ms (see sample table in class notes). If we want the maximum ripple voltage to be 5%, what value of T should be used? Highlight this result in the table (it should be T = 2 ms). Add an explanation as to the significance of this highlighted line in the table.

(4) Assume that RC = 10 ms, V = 5V, and T = 2ms (so that T1 = DT = 0.002T). Substitute these values into Eq. 5 to find an expression for Vm in terms of D (i.e., derive Eq. 9).

(5) Calculate the frequency, f, for T = 2 ms.

(6) Form a table calculating Vm as D varies from 0% to 100% in increments of 5%. Also graph Vm versus D. See sample table and graph in class notes. What does this plot suggest about the relationship between the duty cycle, D, and the average steady-state capacitor voltage?

(7) If RC = 10 ms, form a table of 5 or more possible choices of R and C values using the following constraints:

A) 1 kΩ < R < 10 kΩ

B) Use standard 5% resistor values (see table in Pinouts document on course website)

C) Use capacitor values available in lab (see table in Pinouts document on course website)

(8) Select one of the combinations of RC values in the table above (and highlight it) and draw a schematic of the RC circuit connected to the MicroStamp11.

(9) Program Listing: Modify the main program you wrote in the preceding lab as follows:

A) Prompt the user to enter a digital value (0-63) representing a 6-bit digital input.

B) Convert the digital value to a duty cycle (perhaps D = x\*100/63) before calling the setpwm function.

(10) Program Listing: Modify kernel8.c (and save as kernel9.c) changing T from 4 ms to 2 ms (i.e., change number of clock ticks from 9828 to 4914).

(11) Explanation of how the program and circuit work.

**4.2. In-lab Tasks:**

(1) Measure R with an ohmmeter and C with an impedance bridge and record their values. Calculate RC and verify that it is within 20% of the designed 10 ms value.

(2) Build the RC circuit and attach it to pin PA4.

(3) Verify that the circuit is working properly. As the input varies from 0 to 63, the output voltage should vary from 0V to 5V. Note that the output may not be correct for 0 and 63, but should work correctly for 1-62.

(4) Create a table for measuring the analog output, Vm, for 64 digital input values (shown in both decimal and binary form) to be entered using the keyboard. Use a DMM (set to measure DC voltages) to measure the analog voltage output by your DAC circuit for digital inputs of 0 to 63. See sample table in class notes.

(5) Use the oscilloscope to capture an image of the analog output voltage, Vm, for at least 10 different digital inputs between 1 and 62. Add cursors to the max and min points on the waveform to measure the ripple voltage.

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. See a sample screen capture in the class notes.

(6) Label each screen capture with the digital input and the calculated ripple voltage.

(7) Include a printout from the terminal program showing how the user interacts with the program.

(8) Description of what happened during in-lab task.

**4.3. Post-Lab Tasks:**

(1) Include a listing of your ﬁnal program in the lab book and explain how it works.

(2) Plot the measured analog voltages for digital inputs 1-62. Is the graph linear? Should it be?

(3) Plot % ripple (calculated from the 10 oscilloscope screen captures) versus digital input. What is the max % ripple? Does this match the design specifications?

(4) Discuss the performance of the DAC.

(5) Compare the DAC constructed in this lab to the one built using an R2R ladder network in Lab 5.

**5. 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,

CHAPTER 10

**Getting Power oﬀ the Wall**

**1. Objective**

Throughout 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) and 5 volts DC from a 120 alternating (AC) voltage source. You could then use this power supply to drive your system.

**2. Parts List**

Italicized parts were used in the previous lab.

1. *(1) wire kit and breadboard*
2. *(2) MicroStamp11, docking module, and USB cable*
3. (3) four 1N4007 diodes
4. (4) LM7805 5Vvoltage regulator with heat sink
5. (5) 120-12 volt wall transformer (AC adapter)
6. (6) 100 *μ*F electrolytic capacitor
7. (7) 1N4733 5.1V Zener Diode
8. (8) 0.1 *μ*F capacitor
9. (9) 100 ohm and 1000 ohm resistors
10. (10) 10 k-ohm potentiometer

**3. Background**

This lab has the student build a circuit that converts a 120 volt *alternatin*g *voltag*e intoa5and 12 volt DC (direct current) voltage level. In order to do this, you will need to use a *transforme*r 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-wav*e *rectiﬁe*r circuit. The 12 volts delivered by the rectiﬁer circuit must then be stepped down to a ﬁxed ﬁve volt level using a *voltag*e *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 *generato*r and a conceptual drawing of this device is shown in Figure 1.

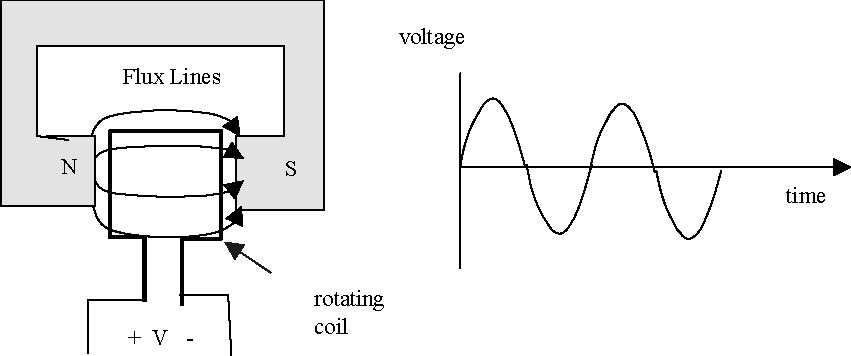


Figure 1. A generator and the voltage it generates

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 *alternatin*g current or AC.

The equation for a waveform of this type is:

(4.1) *v*(*t*)= A cos(*ω*t + *φ*)

where A is the *amplitude*, ω is the *frequency*, and φ is the *phas*e. 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 *cycle*s *pe*r *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

(4.2) A cos(*ω*t + *φ*)= A cos(*ω*(t + T )+ *φ*)

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

(4.3) A cos(*ω*(t + T )+ *φ*)= A cos(*ω*t +2π + *φ*)

Clearly this occurs if *ω*T =2π or rather

(4.4) T = 2*π/*ω

is the fundamental period of this sinusoidal function.

The *siz*e 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

(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 *direc*t *curren*t 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 *transforme*r 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.

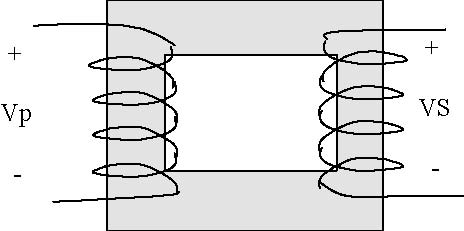


Figure 2. Transformer

An ideal transformer obeys a simple relationship. The sinusoidal voltage on the secondary side, *vs*(*t*), will be proportional to the sinusoidal voltage on the primary side, *vp*(*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 *turn*s *ratio*. In equation form this means that

*vs*(*t*)= *nvp*(*t*)

where n is the turns ratio, *vs*(*t*) is the voltage waveform over the secondary coil, and *vp*(*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-transforme*r (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.

Special plug providing

12 AC volts

two wires coming out

from transformer’s

secondary

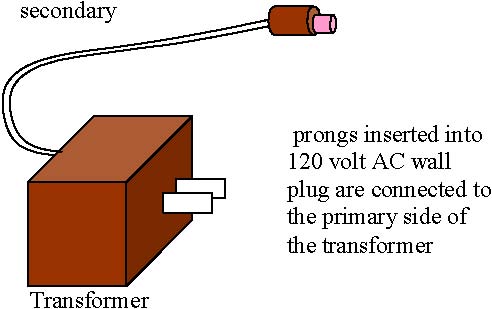


Figure 3. Wall Transformer

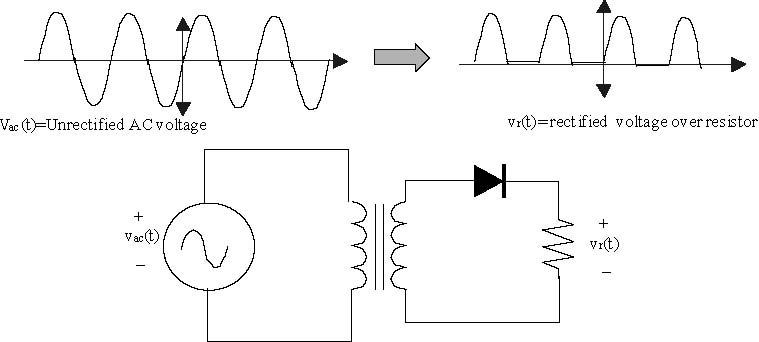
**6. What is a Rectiﬁer Circuit?**

Now that we’ve *steppe*d *dow*n the AC voltages to a level that is more in line with the voltage requirements of the *μ*Stamp11, 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-wav*e 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 *Va*c 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 *Va*c 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.

Looking at the output voltage, *vr*(*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.



Transformer

Figure 4. Half-wave rectiﬁer

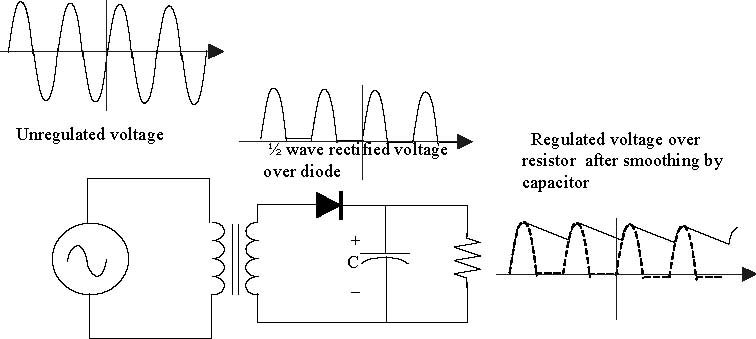
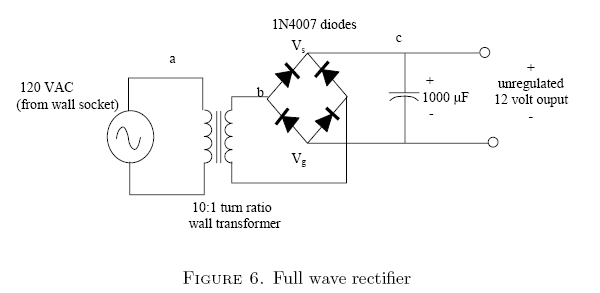


Figure 5. Half-wave rectiﬁer with capacitor

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 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 *electrolyti*c 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 *μ*F 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-wav*e *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 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.

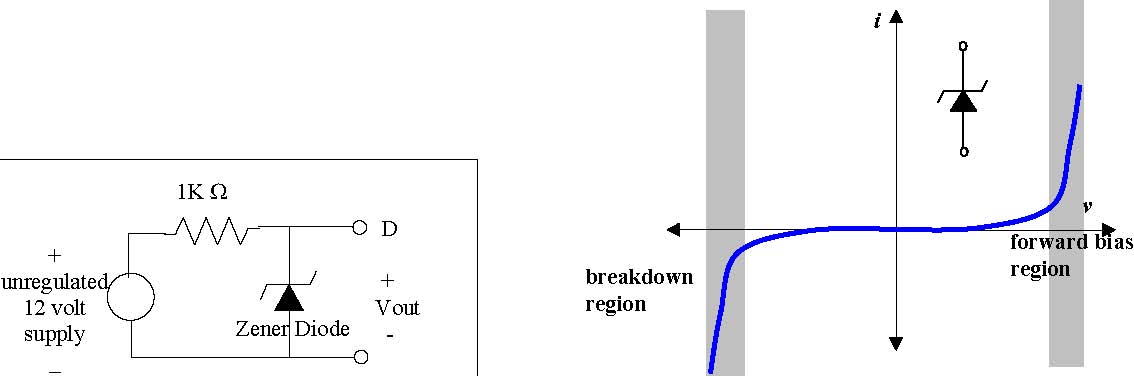
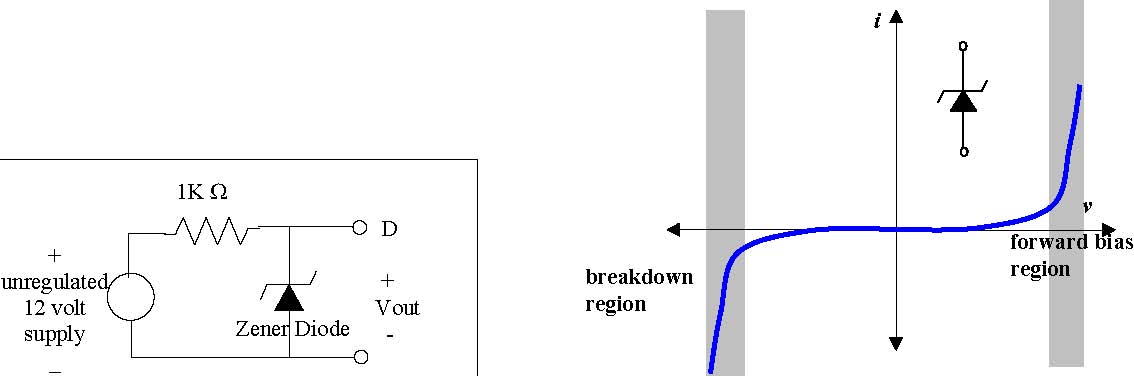
+

100 μF

\_

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 *Vg*. Your MicroStamp11, however, requires a 5 volt supply. We can step down this 12 voltage to a 5 V voltage in several ways. One method is to use a Zener diode to clamp the voltage at 5volts. A Zener diode is a diode whose breakdown voltage 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.

Breakdown voltages in Zener diode are set to specific values.

Figure 7. Zener Diode Voltage Regulator

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 right-hand 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.

In connecting your voltage regulator be sure to put a 0.1 *μ*F 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.

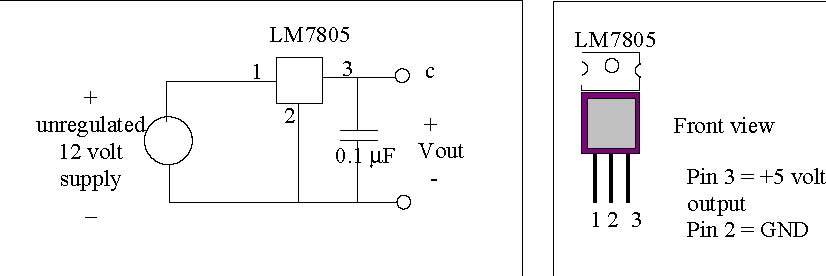
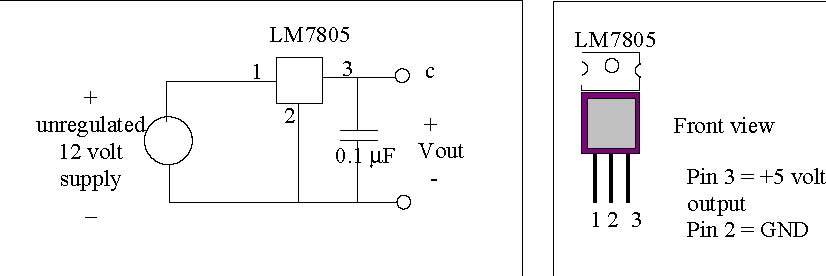
 

Figure 8. LM7805 Votlage Regulation Circuit

**7. Tasks (--- modified)**

**7.1. Pre-lab Tasks:**

(1) Sketch of power supply waveforms at node a, node b, node c (without the capacitor) and node c (with the capacitor) in Figure 6.

(2) Explain in your own words how your power supply works.

(3) Schematic of the Zener diode regulator power supply, including the AC input, transformer, full-wave rectifier, filter capacitor, Zener-diode regulator (with 1kΩ resistor between filter capacitor and a 1N4733A (5.1V) Zener diode) and a load resistance. The load resistance should consist of a 100 Ω resistor and a 10 kΩ potentiometer in series (to make sure that the load resistance never drops below 100 Ω).

(4) Calculate the predicted output voltage, Vout, of the Zener diode regulator if the unregulated voltage is 20V (not 12V as shown in the lab guide) and Rload = 100 Ω , 200 Ω , 300 Ω , 400 Ω , 500 Ω , 1kΩ, 2kΩ, … , 10kΩ. Show a sample calculation for Vout. Also calculate the power dissipated by Rload for each case and tabulate the results. What is the max power dissipation seen in this table? Graph Vout versus Rload. If the thumbwheel potentiometer used in lab has a max power rating of ¾ W, will the max rating be exceeded?

(5) Schematic of 7805 regulator power supply, including the AC input, transformer, full-wave rectifier, filter capacitor, 7805, and a load resistance. The load resistance should consist of a 100 Ω resistor and a 10 kΩ potentiometer in series.

(6) Show the predicted output voltage, Vout, of the 7805 regulator if the unregulated voltage is 20V (not 12V as shown in the lab guide) and Rload = 100 Ω , 200 Ω , 300 Ω , 400 Ω , 500 Ω , 1kΩ, 2kΩ, … , 10kΩ. Explain how the predicted voltages were determined. Also calculate the power dissipated by Rload for each case. What is the max power dissipation seen in this table? Graph Vout versus Rload. If the thumbwheel potentiometer used in lab has a max power rating of ¾ W, will the max rating be exceeded?

**7.2. In-lab Tasks:**

(1) Build the power supply circuit ﬁrst without the voltage regulator and 100 μF capacitor. Use the oscilloscope to view the output of the transformer at node b. Use the ***Measurement*** feature on the oscilloscope to find the ***maximum voltage.*** Capture the image.

(2) Use the oscilloscope to view the output of the full-wave rectifier at node c (with no capacitor). Use the ***Measurement*** feature on the oscilloscope to find the ***maximum voltage***. Capture the image.

(3) Measure the value of the capacitor (before adding it to your circuit) on an impedance bridge and record the value.

(4) Now add the capacitor to your circuit and use the oscilloscope to view the voltage waveform at node c. Use the ***Measurement*** feature on the oscilloscope to find the ***average voltage*** and the ***ripple voltage***. Capture the image.

(5) Add the Zener-diode regulator circuit and 10 kΩ load resistor and measure the output voltage as a function of load resistance for Rload = 100 Ω , 200 Ω , 300 Ω , 400 Ω , 500 Ω , 1kΩ, 2kΩ, … , 10kΩ.

(7) Remove the Zener-diode circuit and add the 7805 voltage regulator circuit and 10 kΩ load resistor and measure the output voltage as a function of load resistance for Rload = 100 Ω , 200 Ω , 300 Ω , 400 Ω , 500 Ω , 1kΩ, 2kΩ, … , 10kΩ.

**7.3. Post-Lab Tasks:**

(1) Discuss the waveform captured showing the output of the transformer. Was it as expected?

(2) Discuss the waveform captured showing the output of the full-wave rectifier without the filter capacitor. Was it as expected?

(3) Discuss the waveform captured showing the output of the full-wave rectifier with the filter capacitor. Was it as expected?

(4) Create a table comparing Vout (predicted) and Vout(measured) for each value of Rload for the Zener diode regulator circuit. Include % error. Also graph Vout (predicted) versus Rload and Vout(measured) versus Rload (on the same graph and include a legend). Discuss the results and explain any differences.

(5) Create a table comparing Vout (predicted) and Vout(measured) for each value of Rload for the 7805 regulator circuit. Include % error. Also graph Vout (predicted) versus Rload and Vout(measured) versus Rload (on the same graph and include a legend). Discuss the results and explain any differences.

(6) Compare the two power supplies built and tested in lab. Which is better?

(7) The power supply you built provides 5 volts for your MicroStamp11. However, the op-amps used in Labs 6-7 require 9 volts. Explain how you could modify your power supply to power your op-amps.

**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.

CHAPTER 11

**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 *seria*l *communicatio*n *interfac*e 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. *(1) wire kit and breadboard*
2. *(2) MicroStamp11, docking module, and USB cable*
3. *(3) Computer with ICC11 compiler, MicroLoad, and PuTTY*
4. *(4) one 7-segment LED display (LSD3221-11)*
5. (5) seven 100 ohm resistors
6. (6) one ULN2003 Darlington array driver IC
7. (7) 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 *seria*l *interfac*e between the MicroStamp11 and the display. In a serial interface, the Microcontrollers 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 *shif*t *registe*r 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 *synchronou*s *seria*l *(SPI*) *subsyste*m to drive a *shif*t *registe*r that will serve as our serial-to-parallel device.

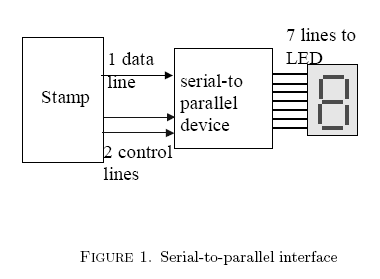


Figure 1. Serial-to-parallel interface

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 *drive*r 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.

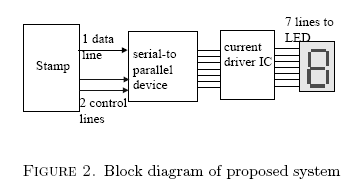


Figure 2. Block diagram of proposed system

**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.

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

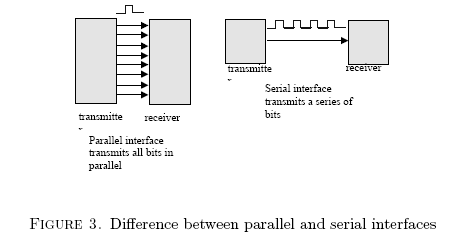


Figure 3. Diﬀerence between parallel and serial interfaces

speak to each other. Serial communication protocols assume that bits are transmitted in *serie*s down a single channel. A serial *protoco*l 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 non-divisible 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.

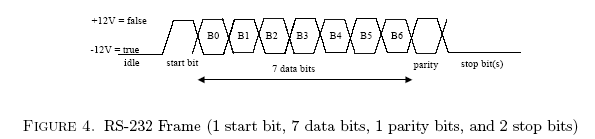


Figure 4. RS-232 Frame (1 start bit, 7 data bits, 1 parity bits, and 2 stop bits)

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

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 microcontroller 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 sometimes 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.

master slave

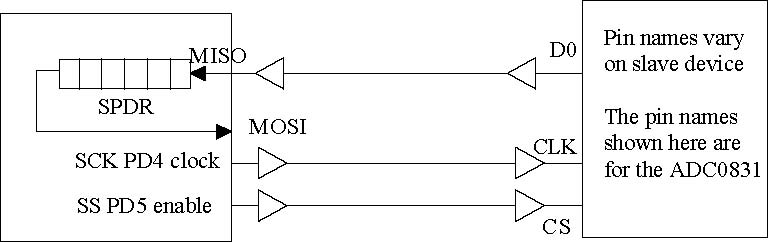


Figure 5. MicroStamp11’s SPI subsystem

Using the SPI subsystem to output data to the shift register is relatively easy with the *kerne*l *functio*n 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.

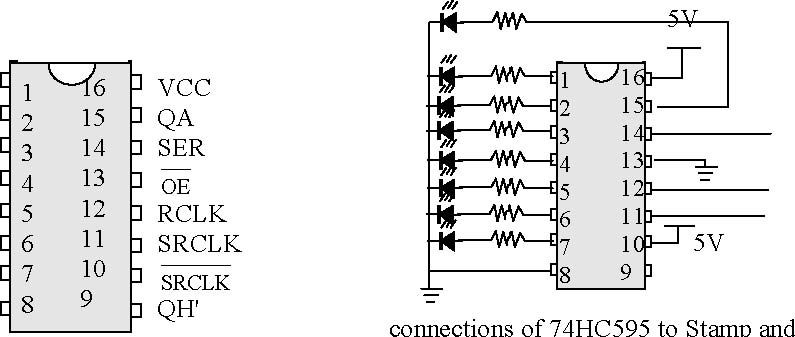
**Usage**: ddata=shiftin(SPI\_62kHz);

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

logical name transfer frequency

SPI\_1MHz 1MHz  
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.



QB

QC

QD

QE

QF

QG

QH

GND

7 segment 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 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

RCLK – PD5(SS) SRCLK -PD4(SCK) SER – PD3(MOSI)

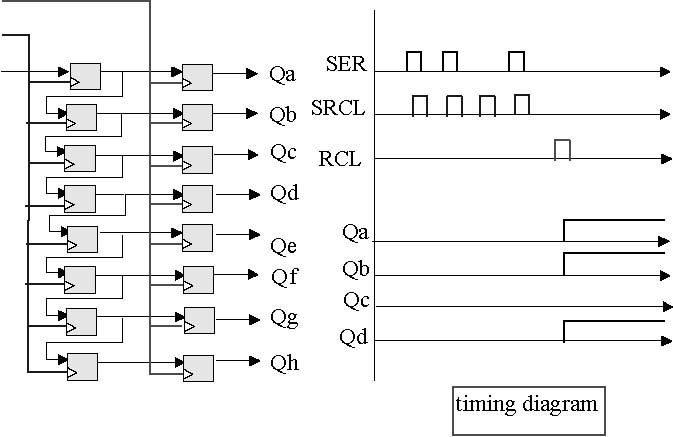


Figure 7. Timing diagram for the serial-to-parallel chip

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.

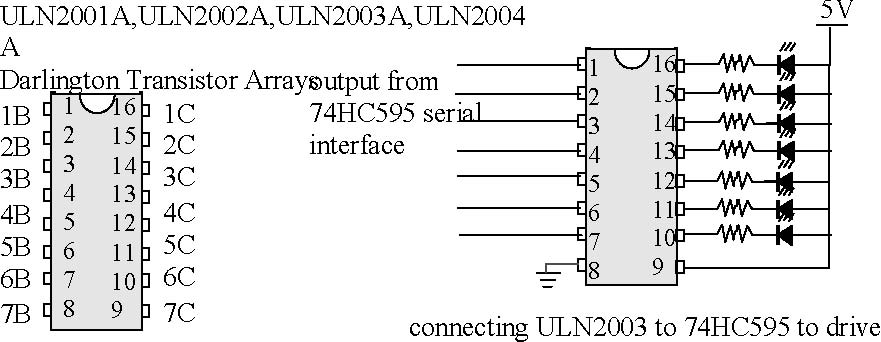
In your design, you will need to connect the Microstamp11’s data pin (MOSI) to the serial line ping (SER) on the latch. The clock signal is produced by the Microstamp on pin SCK. You need to connect SCK to pin SRCLK (source clock) in order to clock the data into the latch. Finally, you will need to use 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 7little 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 *μ*Stamp11 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 pinout 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 left-hand 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 *μ*Stamp11, 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.



a 7-segment LED display

Figure 8. Darlington array drivers

This approach to interfacing isolates the *μ*Stamp11 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 microcontroller. 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 labeled schematic diagram of the modiﬁed LED display driver circuitry and draw a picture of the breadboard layout you plan to use.

(2) Rewrite the program in your previous lab 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.

**4.2. In-lab Tasks:**

(1) Explanation of what happened in the lab.

(2) Current measured on each of the LED segments BEFORE changing display circuitry.

(3) Current measured on each of the LED segments AFTER changing display circuitry.

(4) Sketch of oscilloscope traces for serial line’s waveform for each of the possible display values (62.5 kHz)

(5) Sketch of oscilloscope traces for faster serial rates.

**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 instructor.

**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

CHAPTER 12

**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-multiplexe*d two-digit display

**2. Parts List**

1. *(1) wire kit and breadboard*
2. *(2) MicroStamp11, docking module, and USB cable*
3. *(3) Computer with ICC11 compiler, MicroLoad, and PuTTY*
4. (4) two 7-segment LED display (LSD3221-11)

(5) seven 100 k-ohm resistors

1. (6) two 2N4401 transistors
2. (7) two 5 k-ohm resistors
3. *(8) two 10 k-ohm resistor*
4. *(9) two buttons*
5. *(10) one LM660 quad op-amp IC*
6. *(11) two resistors (R) for op-amp buﬀer*
7. *(12) two 1n4007 diodes*
8. *(13) one 10 k-ohm trim potentiometer*
9. *(14) two additional resistors for diode clamp circuit*
10. *(15) one ULN2003 Darlington array driver IC*
11. *(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 *tim*e 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 microcontroller 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.

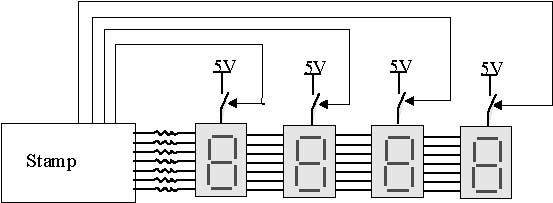


Figure 1. Multiplexed Display

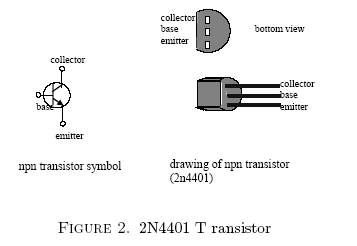
The proper operation of a multiplexed display depends upon a feature of human visual perception known as *ﬂicke*r *fusion*. If a light is ﬂashed quickly enough, individual ﬂashes become imperceptible and the illusion of a steady light is created. This is the basis for movies, television, and ﬂuorescent 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.

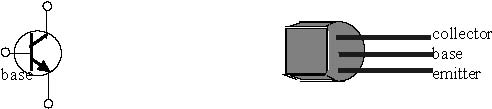
Multiplexed displays only show one digit at a time, but by cycling through all digits repetitively and cycling very fast, a multi-digit display is perceived. To drive a particular digit, its common anode is connected to the appropriate supply voltage and the segments are driven as needed for the desired 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 characterizes 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 left-hand 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 right-hand side of ﬁgure 2.

Collector 



Emitter

npn transistor symbol drawing of npn transistor (2N4401)

Figure 2. 2N4401 Transistor

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

Current limiting

resistor 5 kΩ 12V

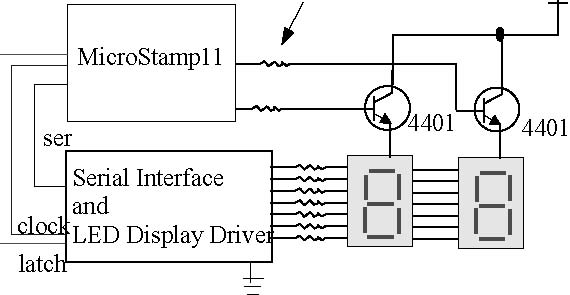


Figure 3. Schematic of Display Circuit

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 *bloc*k 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 instructor 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 instructor. The instructor 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

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 *typ*e of variable i (line 3). The program contains two *ﬂow-contro*l 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.

|  |  |  |
| --- | --- | --- |
| 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

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 *globa*l scope because they can be referenced by multiple functions. A variable whose declaration appears within a function only has scope within that function. Such variables are said to have *loca*l 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**

*Operator*s 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 *expressio*n 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 right-hand 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 *precedenc*e 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 *relationa*l *operators*. Relational operators take arithmetic variables as arguments and return a *logica*l *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

|  |  |
| --- | --- |
| operation | meaning |
| < | less than |
| > | greater than |
| <= | less than or equal to |
| >= | 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 *bitwis*e operators. A bitwise operator acts on the *bit*s 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 *additio*n of two 8-bit integers. In Microcontrollers, however, we often wish to have precise control over individual *bit*s 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 *binar*y operators we can also have *unar*y 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 ith 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 *se*t 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 left-hand 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 *cas*e 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.

*Continu*e and *brea*k 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. (1) We wish to make the overall software system easy to understand;
2. (2) We wish to minimize the coupling or interactions between modules;
3. (3) We wish to group together I/O port accesses to similar devices;
4. (4) We wish to minimize the size (maximize the number) of modules;
5. (5) Modules should be able to be tested independently;
6. (6) We should be able to replace/upgrade one module without aﬀecting the others;
7. (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 Compound Statement. 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.