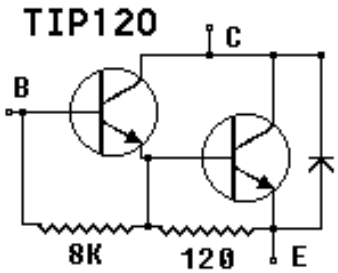
Controlling The Real World With Computers

Experiment 5 - Controlling Motors

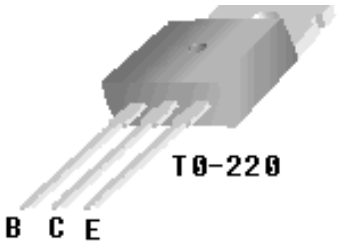
More current is needed for things such as motors, relays and solenoids. Even a small motor draws more current than the PPI or a small transistor can provide. A power device is called for.

One such device is the TIP120, a darlington transistor (seeTIP120.PDF):



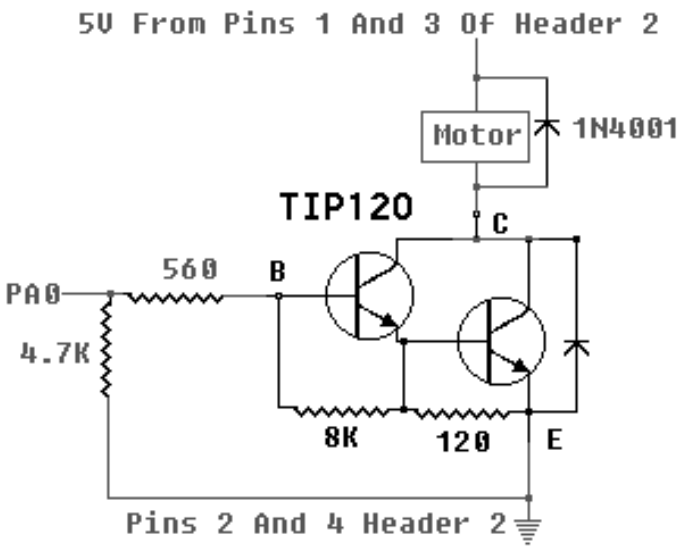
A darlington is actually two transistors. Notice that the emitter of the first transistor is connected to the base of the second. That causes their current gains to be multiplied, giving the TIP120 an hfe of about 1000. The base to emitter resistors are built into the package.

The TO-220 package has a metal tab with a hole in it which can be used to bolt the TIP120 to a piece of metal called a heat sink, used to pull heat away from the package. The tab is connected to the collectors of the transistors though, so if the heat sink is grounded the tab must be insulated from the heat sink but still conduct heat to it. That's accomplished with special materials made for the job. A heat sink is not needed for this experiment. Notice that the positions of the Base, Collector and Emitter are not the same as they were on the PN2222. The diode connected from collector to emitter will be discussed shortly.



Let's say we want to power an electric motor that needs about 1 Amp, such as the Radio Shack 273-223. With a current gain of 1000, we need only drive the base with 1ma (1ma * 1000 = 1A). The full 2.5ma capability of the 82C55 will be used however, in order to make certain plenty of current is delivered to the motor. The voltage between the base and emitter is about two diode drops, or about 1.4V. If we say we can provide 3V, our resistor has 3 - 1.4 = 1.6V across it. The value of the resistor would then be:
R = V/I = 1.6/2.5ma = 640
The closest standard value is 620 ohms, but 560 ohms is easier to find.
The power is: P = 1.6^2/560 = 4.57mw so any normal rating is OK.
Re-calculate the current:
I = V/R = 1.6/560 = 2.857ma plus a little

This circuit should deliver the full 2.5ma to the base of the the TIP120. That times 1000 is 2.5 amps. The TIP120 is being asked to provide only half of its maximum 5 amps current. A 4.7K pulldown resistor from the output of the PPI to ground is added to make certain the PPI line goes low when it's supposed to:

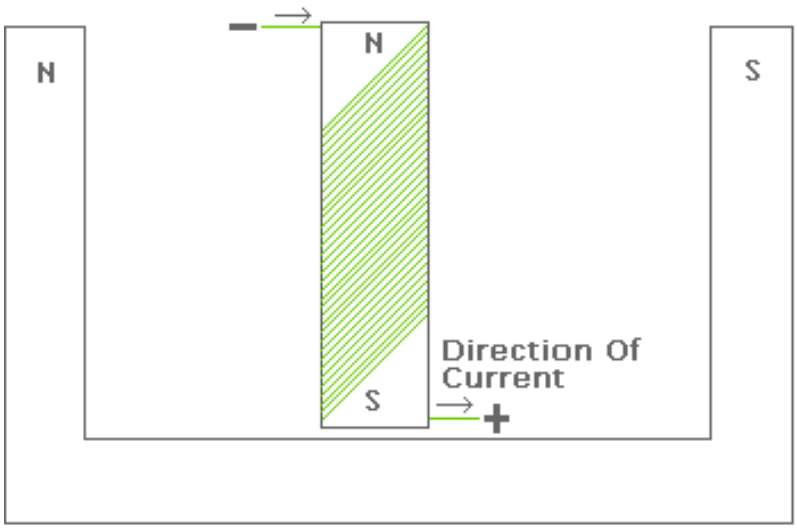


NOTE: Please be sure to read the Warranty And Disclaimer before working with the hardware!

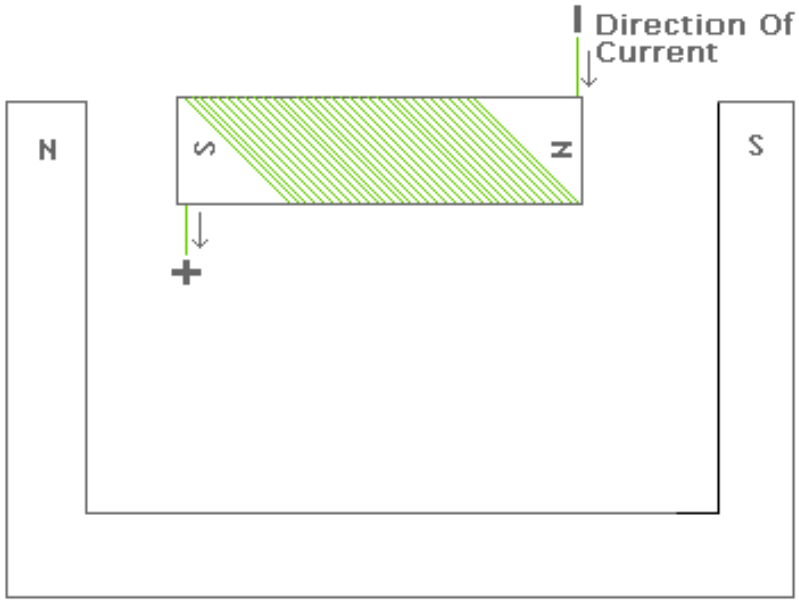
A short discussion of magnetism and motors will be helpful to explain the need for the diodes in the circuit. Incidentally, diodes such as these handle high powers and are usually called **rectifiers**. A 1N4001 for example, can handle 1 amp at 50V or 50W (although I wouldn't advise pushing one that hard), compared to the 1N4148, which is rated at only about 500mw maximum.

Just about everybody played with magnets in grade school. From that experience it was discovered that opposite poles of magnets attract and like poles repel. We were told later that an electromagnet could be formed by running electrical current through a wire wrapped around a piece of iron. By using a direct current, such as that from a battery or the 5V power supply of a computer, the piece of iron will have fixed North and South poles just like a permanent magnet. It will be attracted to and repelled from a permanent magnet in the same manner as another permanent magnet. A motor can be constructed using that knowledge.

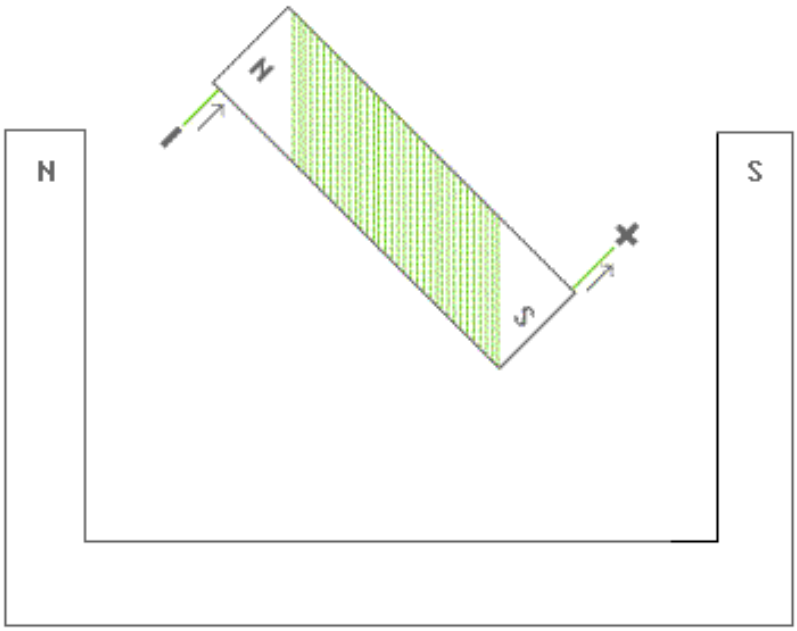
I will cover only the basic ideas of motor operation here. A more detailed explanation can be found in How Stuff Works. Notice what will happen when an electromagnet is placed inside a permanent magnet (wrap the fingers of your left hand around the electromagnet in the direction of the current and your thumb will point to North). The electromagnet is mounted on an imaginary shaft to allow it to rotate:



The opposite poles will attract, and the like poles will repel, so the electromagnet will rotate:



Now turn off the power long enough to let the electromagnet coast a little, then turn it back on, but with the polarity reversed. The whole process starts over:



The rotating electromagnet is called the armature. The switch used to reverse polarity is formed by brushes which connect to the coil by sliding over curved pieces of metal. Together the curved pieces of metal are called the commutator. They are bonded to an insulator on the shaft going through the armature.

In addition to producing mechanical movement, magnetism can also produce voltage. If a wire moves at right angles through a magnetic field or a magnetic field moves across a wire, a voltage will be produced in the wire. The magnitude of the voltage depends on the velocity of the wire or field, among other things. A magnet is formed when voltage is applied to the motor. A magnetic field exists around the armature just as it would around a permanent magnet. When the voltage is turned off, the magnetic field rapidly collapses. Since the field collapses in the opposite direction from which it was initiated, the voltage is reversed. It can be quite high compared to the powering voltage; enough to damage components connected to it.

We are providing one side of the motor with 5 volts. The TIP120 will take the other side near ground (actually about 1 volt above ground). Note that the 1N4001 will be reverse biased when the motor is turned on since its cathode is more positive than its anode. (Forgot about cathodes and anodes? See How To Read A Schematic.)

A relatively large voltage spike of short duration will occur when the motor is turned off that is opposite in polarity to the

applied voltage. The TIP120 side will be significantly more positive than the 5V side. This will forward bias the 1N4001 which will short out the spike, protecting the TIP120 and other circuitry. I measured peaks as high as 100 volts across the Radio Shack motor. **Please do not leave out spike protection rectifiers!** The diode that is built into the TIP120 also helps by shorting negative spikes to ground. Another term for voltage is **Electromotive Force** or EMF. The large reverse voltage spike seen when power is removed from a coil is often called **Back EMF**.

Now add the following to the bottom of digital.c:

```
void motor(long on, long off)
{
  int x;
  long y,z;

  printf("ON  ");
  outp(ppi_porta, 0xff);
  for(y=0L; y<on; y++);

  printf("OFF ");
  outp(ppi_porta, 0);
  for(z=0L; z<off; z++);
}

// turn port a on
void portaon(void)
{
  outp(ppi_porta, 0xff);
}

// turn port a off
void portaoff(void)
{
  outp(ppi_porta, 0);
}

// set up the ppi according to the dictates of the mode argument
void set_up_ppi(int mode)
{
  unsigned control = base + 0x23;
  int command;

  mode>>=6; // shift the mode value to the right 6 places

  command = (mode & 0x0c) << 1; // shift bits 2 and 3 into positions 4 and 5
  command += (mode & 3); // add in bits 0 and 2
  command |= 0x80; // OR in bit 7 for PPI set up

  outp(control, command); // set according to mode command

} // end set_up_ppi()

// end digital.c
```

The motor(..) function turns the motor on then off for the counts you send it. The functions portaon() and portaoff() turn Port A on and off. The set_up_ppi(..) function works the same as before except that it now accepts the new enumeration values as inputs, so it shifts the mode value right 6 places as required.

Now try the following to test it:

```
// experi5a.c

#include <conio.h>
#include <stdio.h>
#include <bios.h>

// include header with constants
#include "constant.h"

// include header with external declarations
#include "extern.h"

void main(void)
{
  int x,y,r,c;

  get_port(); // get the port number and establish register locations

  // make A an output and B and C inputs
  set_up_ppi(Aout_CUin_Bin_CLin); // uses the new enumeration

  while(!kbhit()) // stay in loop until key is hit
  {
    motor(1000000L, 1000000L); // motor on, off times
  } // end while(!kbhit())

  portaoff();

} // end experi5a.c
```

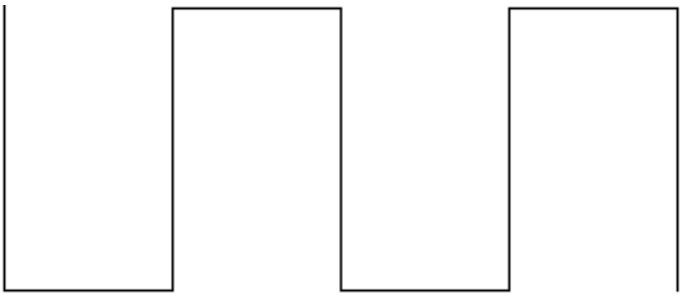
Notice the change in the while() loop. It works just like the previous loops that have been used, except that the test for a pressed key is now part of the while() function.

The extern.h file now contains the external declarations:

```
// external prototypes
extern void set_up_ppi(int mode);
extern void get_port(void);
extern int is_closure(int closurenumber);
extern void motor(long on, long off);
extern void portaoff(void);
extern void portaon(void);
```

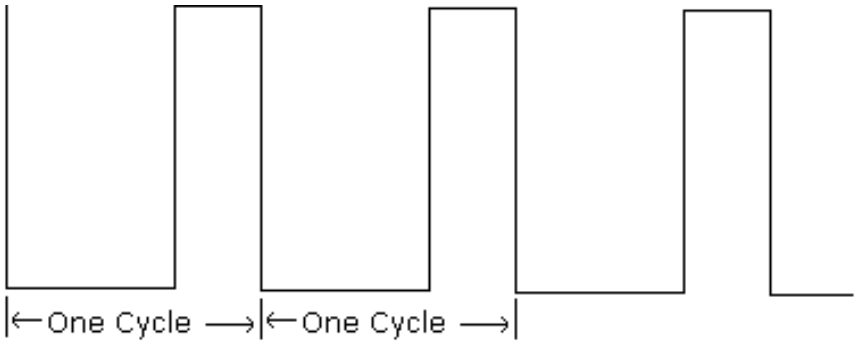
The constant.h file now contains the new PPI setup enumeration from Experiment 4 .

Back to childhood (nice place). Did you ever turn a bicycle upside-down and spin its wheels? If not, go outside and do it; we'll wait. It's right up there with mud puddles -- well close, anyway. Most kids spin the wheel by slapping it. The more slaps per unit of time, the faster it turns. Also, the longer the hand is in contact with the tire, the faster it turns. This is how something called **Pulse Width Modulation** (PWM) works. Pulsing the motor with the full voltage and current produces the same magnetic force in the armature, but for a shorter time. Thus, the force provided by the motor is about the same. Hit the tire or pulse the motor only half the time however, and the *average* voltage is cut in half. The motor will turn at half its full-voltage speed but still supply the same force. The signal to the TIP120 from the PPI would look something like this for a 50% **duty cycle**, which is to say it's on half of the time:



The motor will run at half its normal speed, providing the pulses are fast enough to take advantage of the motor's moving inertia. If you try to slow the shaft down with your fingers however, you will find that the motor has about the same force (torque) as it does at full speed.

Here's another one. If the transistor is on for 30% of a cycle, the duty cycle is 30% and the motor will run at about 30% of its full speed:



Now add this at the bottom of digital.c:

```
void motor2(long on, long off)
{
  int x;
  long y,z;

  outp(ppi_porta, 0xff);
  for(y=0L; y<on; y++);

  outp(ppi_porta, 0);
  for(z=0L; z<off; z++);
}
```

Notice it's the same as motor(..), but without the print statements. Now put this in something called experi5b.c:

```
// experi5b.c

#include <conio.h>
#include <stdio.h>
#include <bios.h>

// include header with constants
#include "constant.h"

// include header with external declarations
#include "extern.h"

void main(void)
{
  long on,off=5000L,r,c;

  get_port(); // get the port number and establish register locations

  // make A an output and B and C inputs
  set_up_ppi(Aout_CUin_Bin_CLin); // uses the new enumeration

  while(1) // stay in loop forever
  {
    for(on=1000L; on<=10000L; on+=100L)
    {
      // "keyboard hit" is a function that checks
      // to see if a key has been pressed.
      if(kbhit())
        break;// A key was pressed -- break out of the loop
      printf("on=%5ld off=%5ld total = %12.6f -- on = %9.6f percent of total\n"
      ,on,off,(double)on+(double)off,100.0*((double)on/((double)on+(double)off)));
      motor(on, off);
    }
    if(on<10000L)
      break;// A key was pressed -- break out of the loop

    for(on=10000L; on>1000L; on-=100L)
    {
      // "keyboard hit" is a function that checks
      // to see if a key has been pressed.
      if(kbhit())
        break;// A key was pressed -- break out of the loop
      printf("on=%5ld off=%5ld total = %12.6f -- on = %9.6f percent of total\n"
      ,on,off,(double)on+(double)off,100.0*((double)on/((double)on+(double)off)));
      motor(on, off);
    }
    if(on>1000L)
      break;// A key was pressed -- break out of the loop

  } // end while(1)
```

```
  portaoff();

} // end experi5b.c
```

Be sure to add the motor2(..) prototype to extern.h.

Notice the calls to printf(..) above. The %5ld parts print the **l**ong **d**ecimal numbers a minimum of 5 characters wide. Note also the references to **double**. A double is a floating point number -- a number with a decimal that can change position according to need. The term **(double)on** is what is called a **cast**. It forces the **on** variable to look like a double so floating point calculations can be performed with it. Casting to double is used because decimal fractions are needed to show the percent of total time. Here, it's percent = 100*(on/(on + off)). Since the on and off variables are longs, they are cast to doubles to do the calculations.

The basic formatting string in printf() for floating point is %f. If there is a number after the %, it says to print a minimum of that many digits, or spaces if there are not that many. If there is then a decimal and a number after the decimal, the number indicates the precision -- the number of places to print after the decimal place. Thus, %12.6f means print a floating point number using a minimum of 12 places, and 6 digits precision to the right of the decimal place.

You might need to play around with the numbers a little to get the motor to work right. Again, a lot depends on the speed of your computer. That problem will be handled in the next experiment. For now, the off time has been fixed at 5000. The on time is first varied from 1000 through 10000 in steps of 100. Thus, it starts with a duty cycle of 100 * (1000/6000), or about 16%, and goes up to 100 * (10000/15000), or about 67%. The second loop goes the other direction. Your motor should start slow, speed up, slow down again then repeat the cycle until you press a key.

Previous: Experiment 4 - The Multiple Closure Problem And Basic Outputs With The PPI
Next: Experiment 6 More Precise Control Of Motors

Controlling The Real World With Computers

Experiment 6 - More Precise Control Of Motors

Experiment 5 provided the basics of controlling motors, including a little on PWM. The goal in this experiment is to increase the usefulness of the system by configuring it in such a way as to permit a program to turn a bit on or off using simple function calls, and to do so with minimum memory usage and time-consuming overhead. Something like this:

TurnOn(WarningLED);
TurnOff(MainMotor);
TurnOn(LeftArmMotor);

We will then look into ways to automate the control of outputs by using a timer built into the computer.

Let's take another look at pointers since they will be used in the final code. Consider the following. It wouldn't hurt to highlight, copy, compile, run and play with it to get a feel for what's going on:

```
// experi6a.c

#include <malloc.h>
#include <conio.h>
#include <stdio.h>
#include <bios.h>

void main(void)
{
  int x,y,*ptr1,*ptr2;

  // cause ptr1 to point to the memory location of x
  ptr1 = &x;

  // notice that x is being set equal to 5 AFTER ptr1 is made to point to x
  x = 5;

  // the * dereferences ptr1, which is to say it provides the value pointed to
  printf("ptr1 points to x's address: x = %d *prt1 = %d\n",x,*ptr1);

  y = 10;
  ptr2 = &y;

  printf("ptr2 points to y's address: y = %d *prt2 = %d\n",y,*ptr2);

  ptr2 = &x;

  printf("both ptr1 and ptr2 point to x's address: x = %d y = %d *prt1 = %d *prt2 =
%d\n"
  ,x,y,*ptr1,*ptr2);
```

```
  *ptr2 = 123;

  printf("dereferenced ptr2 changed to 123: x = %d y = %d *prt1 = %d *prt2 = %d\n"
  ,x,y,*ptr1,*ptr2);

} // end experi6a.c
```

This is the output:

```
ptr1 points to x's address: x = 5 *prt1 = 5
ptr2 points to y's address: y = 10 *prt2 = 10
both ptr1 and ptr2 point to x's address: x = 5 y = 10 *prt1 = 5 *prt2 = 5
dereferenced ptr2 changed to 123: x = 123 y = 10 *prt1 = 123 *prt2 = 123
```

Notice that ptr1 is made to point to x's address before x is set equal to 5, and yet the dereferenced ptr1 also provides 5 (see Experiment 4 if you don't remember dereferencing). The same process applies when ptr2 is given the address of y and y is set to 10.

Finally, the program points both ptr1 and ptr2 to the address location of x. It then sets the dereferenced ptr2 to 123. Notice that x, the dereferenced ptr1 and the dereferenced ptr2 are now all equal to 123. Also note that no change was made directly to x. The change was made by changing the value pointed to by ptr2, which happens to be x and the value pointed to by ptr1. All values show the same because both pointers point to x. When the contents of the memory location are changed, all three get changed. That's the capability that's needed; the ability to control any output bit on any port by any of several control locations.

For basic on/off operation we need the port address, access to the current port value and something that will allow us to change an individual bit to turn it on or to turn it off, such as a couple of masks. It would be nice to be able to package all four elements in a single object so they could be treated as a unit. That's possible with something called a **structure**:

```
struct OC
{
  int PortAddress;
  char onmask;
  char offmask;
  int *PortData;
};
```

The name does not have to be OC. I used it because it's symbolic of Output Control. In fact, it just describes an object. It can't be used until you declare something as being an OC. That's done almost the same way you would declare anything else, such as "int x;." All you add is the **struct** keyword:
struct OC OutputControl;

OutputControl is now an **instance** of OC. You can refer to its **members** with a dot. For example:
OutputControl.PortAddress = 0x123;

Now try this piece of code:

```
// experi6b.c

#include <malloc.h>
#include <conio.h>
#include <stdio.h>
#include <bios.h>

struct OC
{
  int PortAddress;
  char onmask;
  char offmask;
  int *PortData;
};

void main(void)
{
  int pa=0x345,m=0xd,pd;
  struct OC OutputControl;

  OutputControl.PortAddress = pa;
  OutputControl.omask = m;
  OutputControl.PortData = &pd;

  pd = 12345;

  printf("pa = %#X m = %#X pd = %d\n",pa,m,pd);

  printf("OutputControl.PortAddress = %#X\nOutputControl.onmask =
%#X\n*OutputControl.PortData = %d\n"
  ,OutputControl.PortAddress,OutputControl.onmask,*OutputControl.PortData);


} // end experi6b.c
```

And the output:

```
pa = 0X345 m = 0XD pd = 12345
OutputControl.PortAddress = 0X345
OutputControl.onmask = 0XD
*OutputControl.PortData = 12345
```

A small note -- notice what the %#X does for you?

Everything works just like you'd expect, including the dereferenced pointer. The only change is that you use a dot to pick elements out of the object you have declared.

There are a maximum of 24 possible output lines on the board (which you can buy here -- just in case you were wondering). The OC structure allows us to establish the port address that will be used, mask off a bit and access the port data by means of a pointer.

Since we are going to need up to 24 such control structures, we could declare an array of them almost the same way we would any other data type:
struct OC OutputControl[24];

If we wanted to reference the 17th structure's address member for example, we would simply do something like this:
OutputControl[16].PortAddress = 0X345;
Remember, everything in C is zero-based, so the 17th structure is at [16]. The problem with using an array of structures however, is that it takes up memory that might never be used. You already know that each character uses 1 byte and that an integer uses 2 bytes in a DOS 16-bit system. A pointer needs 4 bytes. That makes the whole structure 8 bytes. At 8 bytes each, 24 of the OC structures need 192 bytes. That's not much for a multi-megabyte system, but it can get to be a problem with memory-starved control and embedded systems. In addition, the OC structure will be expanded before we get through, so it would be nice if we could use only the amount of memory that we need as we need it. Again, pointers come to the rescue. Rather than declare an array of structures, just declare an array of pointers to the structures:
struct OC *OutputControl[24];

This array reserves only enough memory for the pointers that will be used to point to the memory locations of the structures. Thus, it takes up 4 * 24 = 96 bytes. Try the following to illustrate the point. Here, the **sizeof(..)** operator will be used. As might be expected, it provides the size of an object:

```
// experi6c.c

#include <malloc.h>
#include <conio.h>
#include <stdio.h>
#include <bios.h>

struct OC
{
  int PortAddress;
  char onmask;
  char offmask;
  int *PortData;
};

void main(void)
{
  struct OC oc;
  struct OC *poc;
  struct OC OutputControl[24];
  struct OC *pOutputControl[24];

  printf("size of char = %d\n",sizeof(char));
  printf("size of int = %d\n",sizeof(int));
  printf("size of double = %d\n",sizeof(float));
  printf("size of double = %d\n",sizeof(double));
  printf("size of oc = %d\n",sizeof(oc));
  printf("size of poc = %d\n",sizeof(poc));
  printf("size of OutputControl = %d\n",sizeof(OutputControl));
  printf("size of pOutputControl = %d\n",sizeof(pOutputControl));

} // end experi6c.c
```

And the output:

```
size of char = 1
size of int = 2
size of double = 4
size of double = 8
size of oc = 8
size of poc = 4
size of OutputControl = 192
size of pOutputControl = 96
```

These object sizes are true only for the system the program was compiled and run on. The numbers will be accurate for most 16-bit DOS systems and compilers, but could be different for your system and compiler. Notice that the single oc structure has a size of 8 bytes, whereas the size of pointer poc is only 4 bytes. That's the reason the array of 24 pointers is 96 bytes smaller than the array of 24 structures. The difference will become even more significant as the size of the structure increases. Note that the pointer poc could have been given the address of oc the same way integer pointers were given the address of an integer:
poc = &oc;

The members of a pointer to a structure are referenced with "->". Thus, poc would reference the PortAddress thus:
poc->PortAddress = 0x345;
and dereferencing the pointer element looks like this:
*poc->PortData = 2;

That means we can do this -- try it, it's good for you:

```
// experi6d.c

#include <malloc.h>
#include <conio.h>
#include <stdio.h>
#include <bios.h>

struct OC
{
  int PortAddress;
  char onmask;
  char offmask;
  int *PortData;
};

void main(void)
{
  struct OC oc;
  struct OC *poc;
  struct OC OutputControl[24];
  struct OC *pOutputControl[24];

  int x = 1000;

  oc.PortData = &x;

  poc = &oc;
```

```
  oc.PortAddress = 100;

  poc->PortAddress = 10;

  printf("poc->PortAddress = %d\noc.PortAddress = %d\nx = %d\n*poc->PortData =
%d\n*oc.PortData = %d\n"
  ,poc->PortAddress
  ,oc.PortAddress
  ,x
  ,*poc->PortData
  ,*oc.PortData);

} // end experi6d.c
```

and get this as an output:

```
poc->PortAddress = 10
oc.PortAddress = 10
x = 1000
*poc->PortData = 1000
*oc.PortData = 1000
```

Notice that oc.PortAddress was set to 100, but got changed to 10. Changing poc->PortAddress didn't change poc->PortAddress alone. It also changed oc.PortAddress.

Also, since oc.PortData was made to point to the memory location of x, and since x had already been made equal to 1000, both *oc.PortData and *poc->PortData became 1000.

Let's take a look at x as it would appear in memory. Since 1000 is equal to the HEX value 03E8 and x is a two-byte integer, it would look like the following in memory. Note that we really don't care what the actual address location is. Lower memory in the table below is toward the left, and upper memory is toward the right. Notice that the bytes are reversed from the left-to-right order we might expect. This is typical of Intel processors:

				E8	03									

When oc.PortData is made to point to x, it points to the memory location containing E8. When oc.PortData is dereferenced, the value residing in that memory location, along with the value in the next location, are returned as a single integer; the 03E8 value (1000) is returned. If x is changed to another value, the same memory location is referenced, so the changed information is obtained.

The process also goes the other way. If the dereferenced oc.PortData is set equal to 100 (= 0x0064), then the above memory locations will change as well, and x will be made equal to 100:

				64	00									

The best way to understand what's going on is to play with the numbers then compile and run the program.

An array of pointers to OC structures is a good start, but that's all it is. It is still necessary to make the pointers point to an area in memory that can hold the structures. We need a way to make some room -- to do some memory allocation. No problem, we'll use **malloc(..)** . That's its job. The prototype for malloc(..) is:
void *malloc(size_t *size*);

The void simply means malloc(..) returns a pointer that is of any type you wish. It returns NULL if there is not enough memory. The size_t type is the same type that sizeof(..) returns. It's usually a long. Such types are usually declared using **typdef** . For example, size_t might be defined as follows:
typedef long size_t;

You can define virtually anything you like this way. Maybe you'd like to use the word integer in the place of int. OK:
typedef int integer;

How about OC? You would define it this way:

```
typedef struct
{
  int PortAddress;
  char onmask;
  char offmask;
  int *PortData;
} OC;
```

Now you can declare an instance without the struct keyword:
OC oc;

I like the non-typedef method because it reminds me that it's a structure that's being used. If you like typedef, go for it.

Back to the subject. Let's say that an array of pointers to OC structures has been defined like this:

struct OC *OutputControl[24];

All you would need to do to reserve memory for the 15th location is this:

OutputControl[14] = malloc(sizeof(struct OC));

Malloc will return the address of a block of memory for the 15th place that is the size of OC, providing the memory is available. A NULL will be returned if it's not.

Just as it is necessary to allocate memory for the structures, it is also necessary to free the memory when it is no longer being used. That is done by calling the free(..) function. **Always free memory after you are through with it.** Not doing so will give the computer severe heartburn. The following example frees the same memory that was allocated above:
free(OutputControl[14]);

The following enumeration will be used to set up each node of the array as needed. The most significant two bits [4:3] will be used to determine what port is to be used, and the least significant three bits [2:0] will determine the bit to be used in the port. It should be added to constant.h:

```
enum OutSetNums
{
  PA0, //  0 00 000 Port A Bit 0
  PA1, //  1 00 001 Port A Bit 1
  PA2, //  2 00 010 Port A Bit 2
  PA3, //  3 00 011 Port A Bit 3
  PA4, //  4 00 100 Port A Bit 4
  PA5, //  5 00 101 Port A Bit 5
  PA6, //  6 00 110 Port A Bit 6
  PA7, //  7 00 111 Port A Bit 7
  PB0, //  8 01 000 Port B Bit 0
  PB1, //  9 01 001 Port B Bit 1
```

```
    PB2, // 10 01 010 Port B Bit 2
    PB3, // 11 01 011 Port B Bit 3
    PB4, // 12 01 100 Port B Bit 4
    PB5, // 13 01 101 Port B Bit 5
    PB6, // 14 01 110 Port B Bit 6
    PB7, // 15 01 111 Port B Bit 7
    PC0, // 16 10 000 Port C Bit 0
    PC1, // 17 10 001 Port C Bit 1
    PC2, // 18 10 010 Port C Bit 2
    PC3, // 19 10 011 Port C Bit 3
    PC4, // 20 10 100 Port C Bit 4
    PC5, // 21 10 101 Port C Bit 5
    PC6, // 22 10 110 Port C Bit 6
    PC7  // 23 10 111 Port C Bit 7
  };
```

Put the structure definitions somewhere close to the variable declarations in digital. Near the top will be fine:

```


// digital.c

// The following are known only to the functions in this file.
// They can't be modified or even accessed by anything outside this
// file except through funtions in this file designed to provide access.

struct OC
{
  int PortAddress;
  char onmask;
  char offmask;
  int *PortData;
};

struct OC *OutputControl[24];

unsigned base;
unsigned switch_port;
unsigned ppi_porta;
unsigned ppi_portb;
unsigned ppi_portc;

int porta_val = 0xa; // port values are set for test
int porta_mask;

int portb_val = 0xb;
int portb_mask;

int portc_val = 0xc;
int portc_mask;
```

The port values are pre-set for test purposes only. If we are pointing to porta_val for example, the value read should be 0XA.

Now add the following configure and free routines to digital.c and their prototypes to extern.h. Notice the slight change to set_up_ppi() as well:

```
// configure the array number location to a port
// and bit number dictated by portselect
int ConfigureOutput(int arraynumber, int portselect)
{
  int x;

  if(arraynumber < 0 || arraynumber > 23)
    return 0; // illegal number

  if(portselect < 0 || portselect > 23)
    return 0; // illegal number

  if(OutputControl[arraynumber] == NULL)
  {
    if((OutputControl[arraynumber] = malloc(sizeof(struct OC))) == NULL)
    {
      printf("Not enough memory\n");
      return 0;
    }
  }

  x = portselect >> 3; // the port number is in bits 3 and 4

  switch(x) // 0 = Port A, 1 = Port B, 2 = Port C
  {
    case 0:
    OutputControl[arraynumber]->PortAddress = ppi_porta; // address for Port A
    OutputControl[arraynumber]->PortData = &porta_val;   // point to Port A data
value
    break;

    case 1:
    OutputControl[arraynumber]->PortAddress = ppi_portb; // address for Port B
    OutputControl[arraynumber]->PortData = &portb_val;   // point to Port B data
value
    break;

    case 2:
    OutputControl[arraynumber]->PortAddress = ppi_portc; // address for Port C
    OutputControl[arraynumber]->PortData = &portc_val;   // point to Port C data
value
    break;
  }

  OutputControl[arraynumber]->onmask = 1 << (portselect & 7); // shift by bits[2:0]
  OutputControl[arraynumber]->offmask = ~OutputControl[arraynumber]->onmask;

// /* remove the double slash rem to skip debug print statements

  printf("arraynum=%02d port select=%02d ",arraynumber,portselect);

  printf("Addr=%#X "
```

```
  ,OutputControl[arraynumber]->PortAddress);

  printf("*Data=%X "
  ,*OutputControl[arraynumber]->Data);

  printf("onmsk=");
  for(x=128; x>0; x/=2)
  {
    if(x & OutputControl[arraynumber]->onmsk)
      printf("1");
    else printf("0");
  }

  printf(" offmsk=");
  for(x=128; x>0; x/=2)
  {
    if(x & OutputControl[arraynumber]->offmsk)
      printf("1");
    else printf("0");
  }
  printf("\n");

// remove the double slash rem to skip debug print statements */

  return 1;
}

// free the output control structures
void FreeOutputControl(void)
{
  int x;

  for(x=0; x<24; x++)
  {
    if(OutputControl[x] != NULL)
      free(OutputControl[x]);
  }
}

// set up the ppi according to the dictates of the mode argument
void set_up_ppi(int mode)
{
  unsigned control = base + 0x23;
  int command;

  // make certain control locations start at NULL
  for(command=0; command<24; command++)
    OutputControl[command] = NULL;

  command = (mode & 0x0c) << 1; // shift bits 2 and 3 into positions 4 and 5
  command += (mode & 3); // add in bits 0 and 2
  command |= 0x80; // OR in bit 7 for PPI set up

  outp(control, command); // set according to mode command
```

```
} // end set_up_ppi()
```

Let's take a look at ConfigureOutput(.....). You have already seen a lot of this type of code. The first thing it does is reject any output or port select number that's less than 0 or greater than 23 since that's what will be used as the index into the array of structure pointers, as well as to determine what port and bit to use. If OutputControl at the index location is NULL then memory is allocated. Be sure to add the loop in set_up_ppi(..) that sets all locations to NULL in the first place so you can be sure this will work.

Next, x is set to portselect shifted to the right 3 places. This will put bits 3 and 4 in bit locations 0 and 1. The original 3 bits will go in the "bit bucket" -- they will disappear as far as x is concerned. Remember however, that nothing will happen to portselect. It can and will be used later. The resulting x value will range from 0 to 2, representing Port A through Port C. The switch statement sets the appropriate port address and causes the data pointer to point to the correct data value variable.

The final thing to do is to set up the masks. The on mask is generated by shifting a 1 to the left by the bit number. This value can range from 0 to 7 and is in bits [2:0] of portselect. The maximum value the three bits can produce = 1 + 2 + 4 = 7, so portselect is ANDed with all 3 bits, or 7. The off mask is the inverted version of the on mask. It would be a very good idea to take another look at the boolean and data lines sections if you don't understand those last two sentences.

Here is extern.h:

```
// extern.h

// external prototypes
extern int ConfigureOutput(extern int arraynumber, extern int portselect);
extern int TurnOn(int arraynumber);
extern int TurnOff(int arraynumber);
extern int is_closure(int input);
extern void set_up_ppi(int mode);
extern void blinker(long on, long off);
extern void btoa(void);
extern void motor(long on, long off);
extern void motor2(long on, long off);
extern void portaon(void);
extern void portaoff(void);
extern void portbon(void);
extern void portboff(void);
extern void portcon(void);
extern void portcoff(void);

// end extern.h
```

Use experi6e.c to test ConfigureOutput(..):

```
// experi6e.c

#include <malloc.h>
#include <conio.h>
#include <stdio.h>
#include <bios.h>

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

void main(void)
{
  int x,p,y;

  get_port(); // get the port number and establish register locations

  // make everything an output
  set_up_ppi(Aout_CUout_Bout_CLout);

  for(x=-1,p=PC7; p>=PA0; x++,p--)
  {
    if(!ConfigureOutput(x,p))
      printf("Error for arraynum = %d port select = %d\n",x,p);
  }

  for(x=0,p=PA0-1; p<=PC7+1; x++,p++)
  {
    if(!ConfigureOutput(x,p))
      printf("Error for arraynum = %d port select = %d\n",x,p);
  }

  // don't forget to free memory!
  FreeOutputControl();

} // end experi6e.c
```

The x variable is used for the arraynum and p is used for port select. The arraynum goes up while port select goes down in the first loop, and they both travel up in the second. Errors are introduced to test the function.

This is the output you should see:

```
Error for arraynum = -1 port select = 23
arraynum=00 port select=22 Addr=0X262 *Data=C onmsk=01000000 offmsk=10111111
arraynum=01 port select=21 Addr=0X262 *Data=C onmsk=00100000 offmsk=11011111
arraynum=02 port select=20 Addr=0X262 *Data=C onmsk=00010000 offmsk=11101111
arraynum=03 port select=19 Addr=0X262 *Data=C onmsk=00001000 offmsk=11110111
arraynum=04 port select=18 Addr=0X262 *Data=C onmsk=00000100 offmsk=11111011
arraynum=05 port select=17 Addr=0X262 *Data=C onmsk=00000010 offmsk=11111101
arraynum=06 port select=16 Addr=0X262 *Data=C onmsk=00000001 offmsk=11111110
arraynum=07 port select=15 Addr=0X261 *Data=B onmsk=10000000 offmsk=01111111
arraynum=08 port select=14 Addr=0X261 *Data=B onmsk=01000000 offmsk=10111111
arraynum=09 port select=13 Addr=0X261 *Data=B onmsk=00100000 offmsk=11011111
arraynum=10 port select=12 Addr=0X261 *Data=B onmsk=00010000 offmsk=11101111
arraynum=11 port select=11 Addr=0X261 *Data=B onmsk=00001000 offmsk=11110111
arraynum=12 port select=10 Addr=0X261 *Data=B onmsk=00000100 offmsk=11111011
arraynum=13 port select=09 Addr=0X261 *Data=B onmsk=00000010 offmsk=11111101
arraynum=14 port select=08 Addr=0X261 *Data=B onmsk=00000001 offmsk=11111110
arraynum=15 port select=07 Addr=0X260 *Data=A onmsk=10000000 offmsk=01111111
arraynum=16 port select=06 Addr=0X260 *Data=A onmsk=01000000 offmsk=10111111
arraynum=17 port select=05 Addr=0X260 *Data=A onmsk=00100000 offmsk=11011111
arraynum=18 port select=04 Addr=0X260 *Data=A onmsk=00010000 offmsk=11101111
arraynum=19 port select=03 Addr=0X260 *Data=A onmsk=00001000 offmsk=11110111
arraynum=20 port select=02 Addr=0X260 *Data=A onmsk=00000100 offmsk=11111011
arraynum=21 port select=01 Addr=0X260 *Data=A onmsk=00000010 offmsk=11111101
arraynum=22 port select=00 Addr=0X260 *Data=A onmsk=00000001 offmsk=11111110
Error for arraynum = 0 port select = -1
arraynum=01 port select=00 Addr=0X260 *Data=A onmsk=00000001 offmsk=11111110
arraynum=02 port select=01 Addr=0X260 *Data=A onmsk=00000010 offmsk=11111101
arraynum=03 port select=02 Addr=0X260 *Data=A onmsk=00000100 offmsk=11111011
arraynum=04 port select=03 Addr=0X260 *Data=A onmsk=00001000 offmsk=11110111
arraynum=05 port select=04 Addr=0X260 *Data=A onmsk=00010000 offmsk=11101111
arraynum=06 port select=05 Addr=0X260 *Data=A onmsk=00100000 offmsk=11011111
arraynum=07 port select=06 Addr=0X260 *Data=A onmsk=01000000 offmsk=10111111
arraynum=08 port select=07 Addr=0X260 *Data=A onmsk=10000000 offmsk=01111111
arraynum=09 port select=08 Addr=0X261 *Data=B onmsk=00000001 offmsk=11111110
arraynum=10 port select=09 Addr=0X261 *Data=B onmsk=00000010 offmsk=11111101
arraynum=11 port select=10 Addr=0X261 *Data=B onmsk=00000100 offmsk=11111011
arraynum=12 port select=11 Addr=0X261 *Data=B onmsk=00001000 offmsk=11110111
arraynum=13 port select=12 Addr=0X261 *Data=B onmsk=00010000 offmsk=11101111
arraynum=14 port select=13 Addr=0X261 *Data=B onmsk=00100000 offmsk=11011111
arraynum=15 port select=14 Addr=0X261 *Data=B onmsk=01000000 offmsk=10111111
arraynum=16 port select=15 Addr=0X261 *Data=B onmsk=10000000 offmsk=01111111
arraynum=17 port select=16 Addr=0X262 *Data=C onmsk=00000001 offmsk=11111110
arraynum=18 port select=17 Addr=0X262 *Data=C onmsk=00000010 offmsk=11111101
arraynum=19 port select=18 Addr=0X262 *Data=C onmsk=00000100 offmsk=11111011
arraynum=20 port select=19 Addr=0X262 *Data=C onmsk=00001000 offmsk=11110111
arraynum=21 port select=20 Addr=0X262 *Data=C onmsk=00010000 offmsk=11101111
arraynum=22 port select=21 Addr=0X262 *Data=C onmsk=00100000 offmsk=11011111
arraynum=23 port select=22 Addr=0X262 *Data=C onmsk=01000000 offmsk=10111111
Error for arraynum = 24 port select = 23
Error for arraynum = 25 port select = 24
```

Setting up output control ahead of time in this manner provides for faster action when in control mode because no decisions

have to be made. They are all made during setup. For example, to turn on an output just do the following:

```
int TurnOn(int arraynumber)
{
  if(OutputControl[arraynumber] == NULL)
    return 0; // node not set up

  // keep existing bits and OR this one in
  *OutputControl[arraynumber]->PortData |= OutputControl[arraynumber]->onmask;

  // put the result in this node's port register
  outp(OutputControl[arraynumber]->PortAddress,
*OutputControl[arraynumber]->PortData);

  return 1;
}
```

And to turn it off:

```
int TurnOff(int arraynumber)
{
  if(OutputControl[arraynumber] == NULL)
    return 0; // node not set up

  // keep existing bits but remove this one
  *OutputControl[arraynumber]->PortData &= OutputControl[arraynumber]->offmask;

  // put the result in this node's port register
  outp(OutputControl[arraynumber]->PortAddress,
*OutputControl[arraynumber]->PortData);

  return 1;
}
```

Taking a closer look:

```
                        Store to the Port Data variable,

      through the        at location     its current value                  with the
on
 dereferenced pointer    arraynumber,          ORed                          mask
value.
          |                   |                  |                              |
  *OutputControl        [arraynumber]->PortData |=
OutputControl[arraynumber]->onmask;

                      Then output it to the port at arraynumber

  outp(OutputControl[arraynumber]->PortAddress,
*OutputControl[arraynumber]->PortData);
```

Turning off a bit works the same way except that the port data value is ANDed with the off mask in order to turn off the desired bit. It is important to note that up to 8 nodes can access a single port data variable. That's what the pointers do for us. Each manipulates only the bit it uses to turn on or off its particular line.

Now add the On/Off routines to digital.c. You will need to change the starting values of the port data variables to 0 so things will work correctly:

```
// digital.c

#include <stdio.h>

// The following are known only to the functions in this file.
// They can't be modified or even accessed by anything outside this
// file except through funtions in this file designed to provide access.

struct OC
{
  int PortAddress;
  char onmask;
  char offmask;
  int *PortData;
};

struct OC *OutputControl[24];

unsigned base;
unsigned switch_port;
unsigned ppi_porta;
unsigned ppi_portb;
unsigned ppi_portc;

int porta_val = 0;
int porta_mask;

int portb_val = 0;
int portb_mask;

int portc_val = 0;
int portc_mask;

// configure the array number location to a port
// and bit number dictated by portselect
int ConfigureOutput(int arraynumber, int portselect)
{
  int x;

  if(arraynumber < 0 || arraynumber > 23)
  {
    printf("arraynumber error --- %d\n",arraynumber);
    return 0; // illegal number
  }

  if(portselect < 0 || portselect > 23)
  {
```

```
    printf("portselect error --- %d\n",portselect);
    return 0; // illegal number
  }

  if(OutputControl[arraynumber] == NULL)
  {
    if((OutputControl[arraynumber] = malloc(sizeof(struct OC))) == NULL)
    {
      printf("Not enough memory\n");
      return 0;
    }
  }

  x = portselect >> 3; // the port number is in bits 3 and 4

  switch(x) // 0 = Port A, 1 = Port B, 2 = Port C
  {
    case 0:
    OutputControl[arraynumber]->PortAddress = ppi_porta; // address for Port A
    OutputControl[arraynumber]->PortData = &porta_val;   // point to Port A data
value
    break;

    case 1:
    OutputControl[arraynumber]->PortAddress = ppi_portb; // address for Port B
    OutputControl[arraynumber]->PortData = &portb_val;   // point to Port B data
value
    break;

    case 2:
    OutputControl[arraynumber]->PortAddress = ppi_portc; // address for Port C
    OutputControl[arraynumber]->PortData = &portc_val;   // point to Port C data
value
    break;
  }

  OutputControl[arraynumber]->onmask = 1 << (portselect & 7); // shift by bits[2:0]
  OutputControl[arraynumber]->offmask = ~OutputControl[arraynumber]->onmask;

// /* add double slash rem to print debug statements

  printf("arraynum=%02d port select=%02d ",arraynumber,portselect);

  printf("Addr=%#X "
  ,OutputControl[arraynumber]->PortAddress);

  printf("*Data=%X "
  ,*OutputControl[arraynumber]->PortData);

  printf("onmsk=");
  for(x=128; x>0; x/=2)
  {
    if(x & OutputControl[arraynumber]->onmask)
      printf("1");
```

```
    else printf("0");
  }

  printf(" offmsk=");
  for(x=128; x>0; x/=2)
  {
    if(x & OutputControl[arraynumber]->offmask)
      printf("1");
    else printf("0");
  }
  printf("\n");

// add double slash rem to print debug statements */

  return 1;
}


// free the output control structures
void FreeOutputControl(void)
{
  int x;

  for(x=0; x<24; x++)
  {
    if(OutputControl[x] != NULL)
      free(OutputControl[x]);
  }
}


// turn on an output node
int TurnOn(int arraynumber)
{
  if(OutputControl[arraynumber] == NULL)
  {
    printf("can't turn on -- location %d not set up\n",arraynumber);
    return 0; // node not set up
  }

  // keep existing bits and OR this one in
  *OutputControl[arraynumber]->PortData |= OutputControl[arraynumber]->onmask;

  // put the result in this node's port register
  outp(OutputControl[arraynumber]->PortAddress,
*OutputControl[arraynumber]->PortData);

  return 1;
}


// turn off an output node
int TurnOff(int arraynumber)
{
```

```
  if(OutputControl[arraynumber] == NULL)
  {
    printf("can't turn off -- location %d not set up\n",arraynumber);
    return 0; // node not set up
  }

  // keep existing bits but remove this one
  *OutputControl[arraynumber]->PortData &= OutputControl[arraynumber]->offmask;

  // put the result in this node's port register
  outp(OutputControl[arraynumber]->PortAddress,
*OutputControl[arraynumber]->PortData);

  return 1;
}

// set up the ppi according to the dictates of the mode argument
void set_up_ppi(int mode)
{
  unsigned control = base + 0x23;
  int command;

  // make certain control locations start at NULL
  for(command=0; command<24; command++)
    OutputControl[command] = NULL;

  command = (mode & 0x0c) << 1; // shift bits 2 and 3 into positions 4 and 5
  command += (mode & 3); // add in bits 0 and 2
  command |= 0x80; // OR in bit 7 for PPI set up

  outp(control, command); // set according to mode command

} // end set_up_ppi()
```

Use experi6f.c to test the routines:

NOTE: Please be sure to read the Warranty And Disclaimer before working with the hardware!

```


// experi6f.c

void waitalittlewhile(void);  // timer for test only

#include <malloc.h>
#include <conio.h>
#include <stdio.h>
#include <bios.h>

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"
```

```
enum
{
  WarningLED,
  MainMotor,
  LeftArmMotor
};

void main(void)
{
  int x,y,*ptr1,*ptr2;

  get_port(); // get the port number and establish register locations

  // make everthing an output
  set_up_ppi(Aout_CUout_Bout_CLout);

  if(!ConfigureOutput(WarningLED,PA0))
      printf("Error setting up Warning LED = %d port select = %d\n"
        ,WarningLED,PA0);

  if(!ConfigureOutput(MainMotor,PA1))
      printf("Error setting up Main Motor = %d port select = %d\n"
        ,MainMotor,PA1);

  if(!ConfigureOutput(LeftArmMotor,PA2))
      printf("Error setting up Left Arm Motor = %d port select = %d\n"
        ,LeftArmMotor,PA2);

  while(!kbhit())
  {
    if(!TurnOn(WarningLED))
      printf("Can't turn on the Warning LED\n");
    else printf("Turned on the Warning LED\n");
    waitalittlewhile();

    if(!TurnOff(WarningLED))
      printf("Can't turn off the Warning LED\n");
    else printf("Turned off the Warning LED\n");
    waitalittlewhile();

    if(!TurnOn(LeftArmMotor))
      printf("Can't turn on the Left Arm Motor\n");
    else printf("Turned on the Left Arm Motor\n");
    waitalittlewhile();

    if(!TurnOff(LeftArmMotor))
      printf("Can't turn off the Left Arm Motor\n");
    else printf("Turned off the Left Arm Motor\n");
    waitalittlewhile();

    if(!TurnOn(MainMotor))
      printf("Can't turn on the Main Motor\n");
    else printf("Turned on the Main Motor\n");
    waitalittlewhile();
```

```
    if(!TurnOff(MainMotor))
      printf("Can't turn off the Main Motor\n");
    else printf("Turned off the Main Motor\n");
    waitalittlewhile();
  }

  // don't forget to free memory!
  FreeOutputControl();

  portaoff();

} // end experi6f.c

void waitalittlewhile(void)
{
  long x;

  for(x=0L; x<100000L; x++);

} // end waitalittlewhile(..)
```

You will need to hook up some LEDs and/or motors using the additional circuitry in Experiment 5 to test the program. That would be a very good idea, since what follows depends on what has past (quick, somebody write that down!). Seriously though, you need to make certain ConfigureOutput(..) works as it should before moving forward, and you won't really know that until you are sure the proper port lines are activated when commanded to do so. The example only uses Port A. It would be a very good idea to try the others as well. You might need to play with the number in the delay loop a little to get it to work right on your computer. Take out the // remark signs so ConfigureOutput(..) won't print any debug information once you get it working properly. Speaking of bugs, Professor/Admiral Grace Murray Hopper found the first computer bug in the Mark II computer on September 9th, 1945 -- a moth got caught in one of the relays. (read more about this computer science pioneer)

The delay loop is still a problem. A more reliable way of timing events is needed. Fortunately, the computer has a built-in timer. It is a very accurate crystal-controlled device that produces an **interrupt** at reliable intervals.

An interrupt is analogous to the bell on a telephone. A telephone with no bell would require a person to periodically pick it up to see if anyone was on the other end, which would be debatably more annoying than having to answer its ring.

Devices such as the timer can interrupt the microprocessor to cause it to sevice the interrupt as required. Special functions called **interrupt handlers** or **interrupt service routines** (**ISR**) are set up for that purpose. Each interrupt is assigned a number which is a reference to the location of a pointer in the first 1024 bytes of memory in a PC, much like the index to the array of pointers we used earlier. The table is called an interrupt vector table or dispatch table.

Our only concern here will be the first 16 locations. They are used for hardware **interrupt requests** (**IRQ**). Each pointer uses four bytes. The following program will display the first 16 pointers or vectors of a machine. It first sets a pointer to character to NULL, which means it points to location 0 in memory. The for() loop starts both x and n at 0, but increments x by 4 while incrementing n by 1. Thus, n keeps track of the interrupt number while x keeps track of the memory location of the vector. The dereferenced ptr shows what is in memory.

```
// experi6g.c

#include <conio.h>
#include <stdio.h>

void main(void)
{
  int x,y,n;
  char *ptr;
  ptr = NULL;

  for(x=0,n=0; n<16; x+=4,n++)
  {
    printf("Number = %03d Memory Location = %04d Vector = ",n,x);
    for(y=0; y<2; y++)
    {
      printf("%02X",*ptr);
      ptr++;
    }
    printf(":");
    for(y=0; y<2; y++)
    {
      printf("%02X",*ptr);
      ptr++;
    }
    printf("\n");
  }

}

// experi6g.c
```

The program should produce the first 16 locations as shown below, although the pointer values will vary among machines. The pointers are found in memory in a segment:offset format. The segment/offset scheme is a way to address more memory than 16 bit registers would normally allow. It divides memory into 65536 16-byte segments or paragraphs, for a total of 65536 * 16 = 1048576 bytes. That means the starting point of the segment is easy to get -- just multiply the segment value by 16. The offset is the number of bytes into the segment. Thus, to determine where something actually is in memory, simply multiply the segment by 16 then add the offset.

The timer uses interrupt 8. It can be seen from the table below that the vector is at 3120:3032 for the machine the program was run on. Remember from above however, that the bytes are reversed in memory. Thus, the segment = 0x2031 and the offset = 0x3230. First, Multiply the segment by 16. That is the same as shifting it to the left 4 bits, which gives us the HEX segment number with a 0 stuck on the right end of it -- 0x20310 -- now add the offset:
0x20310 + 0x3230 = 0x23540 = 144704 decimal

```
Number = 000 Vector Table Location = 0000 Vector = 2449:643A
Number = 001 Vector Table Location = 0004 Vector = 2072:7438
Number = 002 Vector Table Location = 0008 Vector = 362D:2D64
Number = 003 Vector Table Location = 0012 Vector = 732E:6173
Number = 004 Vector Table Location = 0016 Vector = 2C76:2031
Number = 005 Vector Table Location = 0020 Vector = 2E31:2031
Number = 006 Vector Table Location = 0024 Vector = 3939:382F
Number = 007 Vector Table Location = 0028 Vector = 3038:2F32
Number = 008 Vector Table Location = 0032 Vector = 3120:3032 (timer interrupt)
Number = 009 Vector Table Location = 0036 Vector = 3A32:333A
Number = 010 Vector Table Location = 0040 Vector = 3535:2063
Number = 011 Vector Table Location = 0044 Vector = 6C79:6465
Number = 012 Vector Table Location = 0048 Vector = 2045:7870
Number = 013 Vector Table Location = 0052 Vector = 2024:0043
Number = 014 Vector Table Location = 0056 Vector = 204C:6962
Number = 015 Vector Table Location = 0060 Vector = 7261:7279
```

Ralf Browns Interrupt List will provide you with more detail on vectors if you are interested. The routine for 8 actually invokes interrupt 1C which can be used by programs other than the operating system. That would be us. A special routine is written to handle the interrupt. The compiler knows what it is by the **interrupt** keyword:

```
// this happens on a timer interrupt
interrupt new_timer()
{
  disable(); // disable interrupts

  timer_counter++; // increment the counter

  enable(); // enable interrupts

} // end interrupt new_timer()
```

Interrupt handlers have no return value and no arguments. Notice how little the routine does. Interrupt handlers should always do as little as possible due to the fact that they might prevent other important processes from taking place if they do too much. That's because they usually turn off interrupts before doing their work. The compiler I use uses the call to disable() to turn off interrupts, as do many others. You might have to check to find out what yours uses.

Here is the updated extern.h with a few things not yet discussed:

```
// extern.h

// external prototypes

// digital routines -- also put at the top of digital.c without "extern"
extern int ConfigureOutput(int arraynumber, int portselect);
extern int TurnOn(int arraynumber);
extern int TurnOff(int arraynumber);
extern int is_closure(int input);
extern void set_up_ppi(int mode);
```

```
extern void blinker(long on, long off);
extern void btoa(void);
extern void motor(long on, long off);
extern void motor2(long on, long off);
extern void portaon(void);
extern void portaoff(void);
extern void portbon(void);
extern void portboff(void);
extern void portcon(void);
extern void portcoff(void);

// timer routines -- also put at the top of timer.c without "extern"
extern double get_frequency(void);
extern long get_timer_counter(void);
extern void set_up_new_timer(void);
extern void wait(double seconds);
extern void restore_old_timer(void);

// end extern.h
```

The location of the new interrupt service routine must be placed in the vector table so the system can find it. The location of the old routine must also be saved so the vector can be restored when the program is through with it. A call to getvect(...) will get the pointer to the former location so it can be saved, and a call to setvect(...) will put the new one in the table. Since the former routine is probably not located in this program's segment, it is declared as a **far** pointer. A routine to get the value of the timer counter will also be added. Notice the two prototypes just above the timer routines in the following. One is the new interrupt handler. The other is a place holder for the old handler.

The following is the timer module. Notice that the prototypes contained in this module are declared here as well as in extern.h. Here however, they are not declared as externals, since they are not external to this file. The digital prototypes should be placed at the top of digital.c in a like manner. It helps keep the confusion down for the compiler. With the compiler I use, the timer test would not run properly until I included the prototypes in both places.

Notice how the far pointer to the old timer routine is declared. The name is surrounded by parenthesis due to the precedence problems that would occur without them (see Experiment 3). Also notice the call to disable() and enable() in almost all of the timer routines. They disable interrupts for the short time needed to perform various tasks, then enable interrupts. It's not a good idea to have an interrupt occur while you are trying to work with something that is or will become part of an interrupt routine.

Recall that we will be dealing with the vector information at location 0x1C in the vector table. To set up the new routine, the old timer routine location is saved in the old_timer pointer using getvect(0x1c). The new routine is then recorded in the vector table using setvect(0x1c, new_timer). Restoring the old routine is simply a matter of using setvect() with the old timer pointer.

```


// timer.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>
#include "extern.h"

long get_timer_counter(void);
void set_up_new_timer(void);
void wait(double seconds);
void restore_old_timer(void);
```

```
double get_frequency(void);

void interrupt new_timer(), interrupt (far *old_timer)();

unsigned long timer_counter;

// save the old vector, set up new vector, zero out counter
void set_up_new_timer(void)
{
  disable(); // turn off interrupts

  old_timer = getvect(0x1c);

  setvect(0x1c, new_timer);

  timer_counter = 0L;

  enable(); // turn interrupts back on
}

// restore former table entry and rate
void restore_old_timer()
{
  disable();

  setvect(0x1c, old_timer);

  enable();
}

// return the value of the counter to the caller
long get_timer_counter(void)
{
  return timer_counter;
}

// the interrupt handler
interrupt new_timer()
{
  disable();

  timer_counter++;

  enable();
}

// end timer.c
```

Now the new routine, rather than the old routine will be called when a timer interrupt occurs. All the new_timer() routine does is increment the timer counter which was set to 0 when the routine was placed in the vector table. It will do a little more latter, but should never do too much. It especially should not call such things as printf().

The reason is that many routines are not **reentrant** , which is to say they can't be reliably re-entered while they are in the middle of something. That is often because they use the same piece of memory each time for what they do. For example, something in main() could be using printf() when an interrupt occurs. If the ISR then uses printf(), the fixed-memory location

would be clobbered and the results would probably look like anything but what was expected. Similar situations can cause much worse things to happen. See this article by David K. Every if you are interested in more detail about reentrant code.

The test program is simplicity personified:

```
// experi6h.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>
#include "extern.h";

void main(void)
{
  set_up_new_timer();

  while(!kbhit())
  {
     printf("%ld\n",get_timer_counter());
  }

   restore_old_timer();
}
```

Now compile experi6h and timer and link them. The end product should be experi6h.exe. Run experi6h and you should see the timer counter changing. The only thing that's changing the variable however, is the timer interrupt service routine. Nothing in experi6h has access to the counter.

The timer in the PC is driven by a 1193180 Hz (cycles per second) clock. It looks a lot like the square wave in Experiment 5 when viewed on an oscilloscope . There is a 16-bit divider register in the timer that is set to 0. The register counts down from wherever it's set -- 0 in this case -- with the first pulse rolling the register over from 0 to 65535. The timer will issue an interrupt when the register reaches 0 (but not on the first, loaded 0). It will then re-load the register with the original value, which it retains internally. The result is to divide the clock signal by 65536, producing an effective output of 18.20648193 Hz, and one interrupt every .054925493 seconds, or about 55ms.

It would be nice to be able to set up the timer for any rate desired. No problem. Just set up the register with 1193180/frequency. Let's say 1000 Hz is desired so the system will produce 1000 interrupts per second. 1193180/1000 = 1193.180. Digital register can't be loaded with a floating point number, but 1193 gets pretty close since 1193180/1193 = 1000.15088 Hz. The port address for the timer register is 0x40. The same location is used for the ms and ls values. It's the order that determines what will go where -- least significant then most significant. The new set_up_new_timer() gets a **double** as an argument. That's the same as a float, but with higher precision. The argument tells set_up_new_timer() what frequency to use in setting up the timer. Another double called divideby is used to determine what the divider would be by setting it equal to the input frequency divided into 1193180. Since only the result less the fraction will be used, divideby is rounded by adding .5 to it. Any fraction >= .5 will cause the whole number part to move up by one. Recall that we need to end up with is a 16-bit number for the timer's counter register. The most signifcant portion is the upper 8 bits, and the least significant portion is the lower 8 bits. The ms portion is divideby shifted to the right 8 bits, with divideby cast as an unsigned integer. Forgotten what some of that is all about?:
most significant and least significant: Data Lines
shift operators: Experiment 1
casting: Experiment 5

The actual frequency that results is then calculated for access by other routines through get_frequency().

```
// timer.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>
#include "extern.h"

long get_timer_counter(void);
int set_up_new_timer(double freq);
void wait(double seconds);
void restore_old_timer(void);
double get_frequency(void);

void interrupt new_timer(), interrupt (far *old_timer)();

unsigned long timer_counter;
double frequency;

// save the old vector, set up new vector, zero out counter
// set up timer rate
int set_up_new_timer(double freq)
{
  unsigned ms,ls;
  double divideby;

  if(freq < (1193180.0/65536.0))
    return 0;  // can't go below this

  if(freq > 1193180.0)
    return 0;  // or above this

  divideby = 1193180.0/freq;

  divideby+=0.5; // causes a round above .5

  ms = (unsigned)divideby >> 8; // get upper 8 for ms
  ls = (unsigned)divideby & 0xff; // mask off lower 8 for ls

  frequency = 1193180.0/(double)((ms << 8) + ls);

  timer_counter = 0L;

  disable(); // turn off interrupts

  outp(0x40, ls); // least significant byte of timer count
  outp(0x40, ms);  // most significant byte of timer count

  old_timer = getvect(0x1c);

  setvect(0x1c, new_timer);

  enable(); // turn interrupts back on
```

```
  return 1;
}

// restore former table entry and rate
void restore_old_timer()
{
  disable();

  outp(0x40, 0); // least significant byte of timer count
  outp(0x40, 0);  // most significant byte of timer count

  setvect(0x1c, old_timer);

  enable();
}

// return the frequency to the caller
double get_frequency(void)
{
  return frequency;
}

// return the value of the counter to the caller
long get_timer_counter(void)
{
  return timer_counter;
}

// wait for seconds and/or fractions of a second
void wait(double seconds)
{
  long wait_count, start_count;

  if(!seconds)
    return;

  if(timer_counter < 0L)
    return;

  wait_count = (long)((seconds * frequency) + 0.5); // round at .5

  start_count = timer_counter;

  while((timer_counter - start_count) < wait_count);
}

// the interrupt handler
interrupt new_timer()
{
  disable();

  timer_counter++;
```

```
  enable();
}

// end timer.c
```

The wait() routine takes advantage of the new timer setup. It is sent a double argument called seconds, representing the number of seconds of delay desired, which can include decimal fractions. The number of counts to wait will be the number of interrupts per second times the number of seconds to wait. Since the number of interrupts per second is the frequency calculated during setup, the count needed is the number of seconds desired times the frequency. This is rounded then cast to a long variable called wait_time. Another long called start_count gets the current counter value timer_counter. From then on, the number of counts elapsed will be equal to timer_counter - start_count. A while loop waits for that difference to be >= wait_count. Please note that set_up_new_timer() **must** be called or none of this will work. Also, it would probably be a good idea not to try to set the frequency too high. My test machine didn't like it when I tried 10K Hz. It worked up to 5K Hz just fine.

The new extern.h:

```
// extern.h

// external prototypes

// digital routines -- also put at the top of digital.c without "extern"
extern int ConfigureOutput(int arraynumber, int portselect);
extern int TurnOn(int arraynumber);
extern int TurnOff(int arraynumber);
extern int is_closure(int input);
extern void set_up_ppi(int mode);
extern void blinker(long on, long off);
extern void btoa(void);
extern void motor(long on, long off);
extern void motor2(long on, long off);
extern void portaon(void);
extern void portaoff(void);
extern void portbon(void);
extern void portboff(void);
extern void portcon(void);
extern void portcoff(void);

// timer routines -- also put at the top of timer.c without "extern"
extern double get_frequency(void);
extern long get_timer_counter(void);
extern int set_up_new_timer(double freq);
extern void wait(double seconds);
extern void restore_old_timer(void);

// end extern.h
```

The test program sets the timer up for 4K Hz then attempts a 60 second timer at 1 second intervals. It waits for a keystroke to start so a watch can be used to check accuracy. The call to getch() waits for a single character from the keyboard. It would be a very good idea to compile timer and experi6i, link them, then run this test since the routines will be used in the future:

```
// experi6i.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>
#include "extern.h"

void main(void)
{
  int x;

  set_up_new_timer(4000);

  printf("frequency = %f\n",get_frequency());

  printf("Press any key to begin test: ");
  getch(); // wait for key
  puts(""); // print a blank line

  for(x=0; x<60; x++)
  {
    if(kbhit())
      break;
    printf("x = %2d counter = %6ld\n",x,get_timer_counter());
    wait(1);
  }

  // don't forget this one -- your computer could freeze!!!
  restore_old_timer();
}

// end experi6i.c
```

The following test program combines output control with the interrupt-based timing routines. It is put together by linking the three object files after compiling. In MIX PowerC that would be:
pcl experi6j digital timer

This will produce an executible called experi6j.exe. Notice how the timer was set up. The desired interval is .1 second. In order to get good precision, the timer was set up to be about 100 times that fast. This was done by setting the timer close to 1000 Hz, but not exactly. The idea is to get close to the precision needed, but to try to avoid rounding in the counter calculations. To do that, divide the clock by the desired frequency, then use the nearest non-fractional number. For example, 1193180/1000 = 1193.180. Another one: the goal is to get close to 5K Hz. 1193180/5000 = 238.636. Use 1193180/239. Here, the closest non-fractional number to 1193.180 is 1000, so use 1193180/1000 as the argument for set_up_new_timer().

```
// experi6j.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

enum
{
  WarningLED,
  MainMotor,
  LeftArmMotor
};

void main(void)
{
  int x;

  get_port(); // get the port number and establish register locations

  // make everthing an output
  set_up_ppi(Aout_CUout_Bout_CLout);

  printf("1193180/1000 = %f\n"
  ,1193180/1000);

  set_up_new_timer(1193180/1000);

  printf("frequency = %f\n",get_frequency());

  if(!ConfigureOutput(WarningLED,PA0))
      printf("Error setting up Warning LED = %d port select = %d\n"
        ,WarningLED,PA0);

  if(!ConfigureOutput(MainMotor,PA1))
      printf("Error setting up Main Motor = %d port select = %d\n"
        ,MainMotor,PA1);

  if(!ConfigureOutput(LeftArmMotor,PA2))
      printf("Error setting up Left Arm Motor = %d port select = %d\n"
        ,LeftArmMotor,PA2);

  while(!kbhit())
  {
    if(!TurnOn(WarningLED))
      printf("Can't turn on the Warning LED\n");
    else printf("Turned on the Warning LED\n");
```

```
    wait(.1);

    if(!TurnOff(WarningLED))
      printf("Can't turn off the Warning LED\n");
    else printf("Turned off the Warning LED\n");
    wait(.1);

    if(!TurnOn(LeftArmMotor))
      printf("Can't turn on the Left Arm Motor\n");
    else printf("Turned on the Left Arm Motor\n");
    wait(.1);

    if(!TurnOff(LeftArmMotor))
      printf("Can't turn off the Left Arm Motor\n");
    else printf("Turned off the Left Arm Motor\n");
    wait(.1);

    if(!TurnOn(MainMotor))
      printf("Can't turn on the Main Motor\n");
    else printf("Turned on the Main Motor\n");
    wait(.1);

    if(!TurnOff(MainMotor))
      printf("Can't turn off the Main Motor\n");
    else printf("Turned off the Main Motor\n");
    wait(.1);
  }

  portaoff();

  // always free memory!
  FreeOutputControl();

  // always restore the old timer!
  restore_old_timer();
}

// end experi6j.c
```

The timer module can use the output control structure if it is provided with its description and declares it as an external. To do this, move the structure definition to outcont.h. A few more members have been added for future use:

```


// outcont.h

struct OC
{
  int PortAddress; // address of the port with this output line

  char onmask;     // mask that will turn this line on when
                   // ORed with the data value and stored back

  char offmask;    // mask that will turn this line off when
                   // ANDed with the data value and stored back
```

```
  int *PortData;   // pointer to the data value for the port

  long seton;      // on-time setting for this line
                   // a -1 means run continuously

  long setoff;     // off-time setting for this line

  long oncount;    // counts left for on time

  long offcount;   // counts left for off time
};

// end outcont.h
```

The digital.c file no longer contains the declaration for the OC structure. It now includes outcont.h which does that. It does however, declare an instance of the structure:

struct OC *OutputControl[24];

The timer.c file also includes outcont.h, but shows the instance of OC as external:

extern struct OC *OutputControl[24];

The OutputControl array of OC pointers can be viewed as being in digital and accessable to both the digital and timer modules. Here are the tops of each.

Digital -- OutputControl belongs to digital.c:

```


// digital.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>
#include "outcont.h" // defines output control structure

struct OC *OutputControl[24];

// prototypes
int ConfigureOutput(int arraynumber, int portselect);
int TurnOn(int arraynumber);
int TurnOff(int arraynumber);
int is_closure(int input);
void set_up_ppi(int mode);
void blinker(long on, long off);
void btoa(void);
void motor(long on, long off);
void motor2(long on, long off);
void portaon(void);
void portaoff(void);
void portbon(void);
void portboff(void);
void portcon(void);
void portcoff(void);
```

```
// The following are known only to the functions in this file.
// They can't be modified or even accessed by anything outside this
// file except through funtions in this file designed to provide access.
          .
          .
          .
          .
          .

// end digital.c
```

Timer -- OutputControl is external but can be accessed by timer.c:

```
// timer.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>
#include "outcont.h" // defines output control structure

// in digital.c
extern int ConfigureOutput(int arraynumber, int portselect);
extern struct OC *OutputControl[24];

// local prototypes
long get_timer_counter(void);
int set_up_new_timer(double freq);
void wait(double seconds);
void restore_old_timer(void);
double get_frequency(void);

void interrupt new_timer(), interrupt (far *old_timer)();

unsigned long timer_counter;
double frequency;
          .
          .
          .
          .
          .

// end timer.c
```

It would actually be a lot cleaner to put everything at the top of the files into corresponding header files, then include the header files. Everything above that goes at the top of digital.c would be put in digital.h, then #include "digital.h" entered at the top of digital.c Everything above that goes at the top of timer.c would be put in timer.h, then #include "timer.h" entered at the top of timer.c It makes no difference how you do it. It cleans up the C file to have the header, but it's easier to see what items are if they are declared at the top of the file being worked on.

Since the timer module now has access to the output control array, it can be used to set some of the variables for a node of the array. The following sets up pulse width modulation for a node. It first checks to make certain that offtime is reasonable, then calls ConfigureOutput(...). The seton member is set to -1 if the on time is less than 0, indicating the node stays on. Otherwise, seton is calculated to be the frequency (which is the number of interrupts per second) times the number of seconds desired. If ontime is greater than 0, the oncount member is set equal to the seton member, the setoff member is calculated and the node

is turned on. The setoff and oncount members are set to 0 if ontime is less than or equal to 0. Offcount is always set to 0:

```
// set up Pulse Width Modulation for an output
// input is the array number and the port select,
// along with the on time and off time in seconds
// on time and off time can be fractional seconds
int pwm(int arraynumber, int portselect, double ontime, double offtime)
{
  int x;

  if(offtime < 0)
    return 0;

  if(!ConfigureOutput(arraynumber, portselect))
    return 0;

  if(ontime < 0)
    OutputControl[arraynumber]->seton = -1L;
  else OutputControl[arraynumber]->seton = (long)((frequency * ontime) + 0.5);

  if(ontime > 0)
  {
    OutputControl[arraynumber]->oncount
      = OutputControl[arraynumber]->seton;

    OutputControl[arraynumber]->setoff
      = (long)((frequency * offtime) + 0.5);

    // keep existing bits and OR this one in
    *OutputControl[arraynumber]->PortData |= OutputControl[x]->onmask;

    // put the result in this node's port register
    outp(OutputControl[arraynumber]->PortAddress, *OutputControl[x]->PortData);
  }

  else
  {
    OutputControl[arraynumber]->setoff = 0L;
    OutputControl[arraynumber]->oncount = 0L;
  }

  OutputControl[arraynumber]->offcount = 0L;
}
```

This information is used by the updated timer interrupt service routine to determine when an output should be turned on or off.

The routine now runs through the 24 Output Control nodes. It skips all nodes that have not been set up or ones that run all the time.

Notice that the on count was set to seton in pwm(..), and that the off count was set to 0. Thus, the first time around, new_timer() will find that the on counter is greater than 0 and decrement it. If the on counter is made zero by the decrement, the node will be turned off and the off counter loaded with setoff. If, on the other hand, the off counter is greater than zero, it will be decremented. If it is made zero by the decrement, the node will be turned back on and the on counter loaded with seton again.

```
// the timer interrupt handler
interrupt new_timer()
{
  int x;

  disable();

  timer_counter++;

  for(x=0; x<24; x++)
  {
    if(OutputControl[x] == NULL) // not set up
      continue;

    if(OutputControl[x]->seton <= 0L) // stay on or not set
      continue;

    if(OutputControl[x]->oncount > 0L)
    {
      OutputControl[x]->oncount--;

      if(!OutputControl[x]->oncount)
      {
        // keep existing bits but remove this one
        *OutputControl[x]->PortData &= OutputControl[x]->offmask;

        // put the result in this node's port register
        outp(OutputControl[x]->PortAddress, *OutputControl[x]->PortData);

        OutputControl[x]->offcount = OutputControl[x]->setoff;
      }

    } // end if(OutputControl[x]->oncount > 0L)

    // note that this will not decrement as soon as set
    // above, but will wait until the next interrupt
    else if(OutputControl[x]->offcount > 0L)
    {
      OutputControl[x]->offcount--;

      if(!OutputControl[x]->offcount)
      {
        // keep existing bits and OR this one in
        *OutputControl[x]->PortData |= OutputControl[x]->onmask;

        // put the result in this node's port register
        outp(OutputControl[x]->PortAddress, *OutputControl[x]->PortData);

        OutputControl[x]->oncount = OutputControl[x]->seton;
      }

    } // end else if(OutputControl[x]->offcount > 0L)
```

```
  } // end for(x=0; x<24; x++)

  enable();
}
```

A node is continually turned on and off automatically in the new_timer() ISR. All main() has to do is call pwm(..) to set it up. It can then go about its business. On and off times can range from sub-milliseconds to years! That makes it good for everything from controlling the speed of a motor to flashing an LED to turning on and off such things as air-conditioning systems (providing output devices are added that can take the current and voltage).

Here is the test program. There is no need to copy it as it and the other files can be downloaded below:

```


// experi6k.c

// test
extern void show(void);

#include <dos.h>
#include <stdio.h>
#include <bios.h>

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

enum
{
  MainMotor,
  WarningLED,
  LeftArmMotor
};

void main(void)
{
  int x;
  double ontime;

  get_port(); // get the port number and establish register locations

  // make everthing an output
  set_up_ppi(Aout_CUout_Bout_CLout);

  printf("1193180/1000 = %f\n",1193180.0/1000.0);

  set_up_new_timer(1193180.0/1000.0);

  printf("frequency = %f\n",get_frequency());

  if(!pwm(MainMotor,PA0, .003, .002))
      printf("Error setting up Warning LED = %d port select = %d\n"
```

```
        ,WarningLED,PA0);

  if(!pwm(WarningLED,PA1, .03, .02))
      printf("Error setting up Main Motor = %d port select = %d\n"
        ,MainMotor,PA1);

  if(!pwm(LeftArmMotor,PA2, .3, .2))
      printf("Error setting up Left Arm Motor = %d port select = %d\n"
        ,LeftArmMotor,PA2);

  show();  // a little test routine in timer that shows contents of nodes

  printf("press any key to end test\n");

  getch();

  // don't forget to free memory!
  FreeOutputControl();

  portaoff();

  restore_old_timer();
}

// end experi6k.c
```

To make life a little easier, you can download all of the files for this experiment below. On most machines, just right-click then do a save-as:

experi6k.c	digital.c
digital.h	timer.c
timer.h	constant.h
extern.h	outcont.h

Now compile experi6k, timer and digital, then link them to form experi6k.exe. Notice the difference between this program and the previous ones? There is no infinite loop such as while(!kbhit()). Everything is set up, then there is a getch() which simply sits there and waits for a keystroke. Other activities could just as easily take place. The timer is taking care of pulse width modulation tasks in the background.

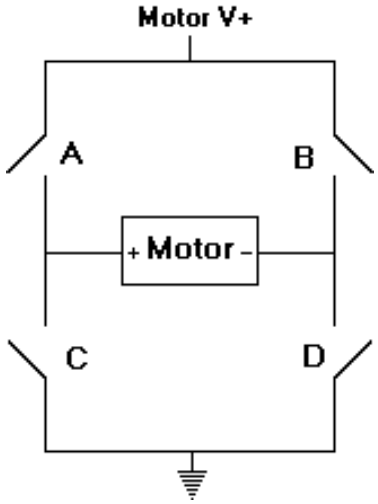
Using the proper interfacing and driver devices (see Experiment 5), hook up LEDs and/or motors to the outputs. Notice the difference in rates. The .3 seconds on and .2 seconds off for PA2 provides a normal-looking flasing LED cycle. PA1 on the other hand, is too fast for an indicator but could be used for motor speed control. PA0's 3ms on and 2ms off moves too fast to tell it's blinking at all, but works well for speed control. In other words, it's the choice of numbers that determines what an output will do.

Previous: Experiment 5 - Controlling Motors
Next: Experiment 7 - Bi-directional Control Of Motors And The H-Bridge
Problems, comments, ideas? Please e-mail me

Controlling The Real World With Computers

Experiment 7 - Bi-directional Control Of Motors And The H-Bridge

It would be useful to be able to run a motor in either direction. That's simple with a motor of the type that's been used in the experiments thus far. Reverse the DC power connections on such a motor and its shaft will rotate in the opposite direction. One way to control direction is to use switches arranged in the following manner:



This configuration is called an **H-Bridge** due to its shape. Let's say that the motor runs forward when its + terminal is connected to Motor V+ and its - terminal is connected to ground. It will run in reverse when the opposite is true. Turn on switch A and switch D and the motor will run forward. Turn on switch B and switch C and it will run in reverse. The following table shows all of the possibilities. A 1 means a switch is on, and a 0 means it's off:

A	B	C	D	State	A	B	C	D	State
0	0	0	0	**Off**	1	0	0	0	**Off**
0	0	0	1	**Off**	1	0	0	1	**Forward**
0	0	1	0	**Off**	1	0	1	0	**SHORT!!**
0	0	1	1	**Brake**	1	0	1	1	**SHORT!!**
0	1	0	0	**Off**	1	1	0	0	**Brake**
0	1	0	1	**SHORT!!**	1	1	0	1	**SHORT!!**
0	1	1	0	**Reverse**	1	1	1	0	**SHORT!!**
0	1	1	1	**SHORT!!**	1	1	1	1	**SHORT!!**

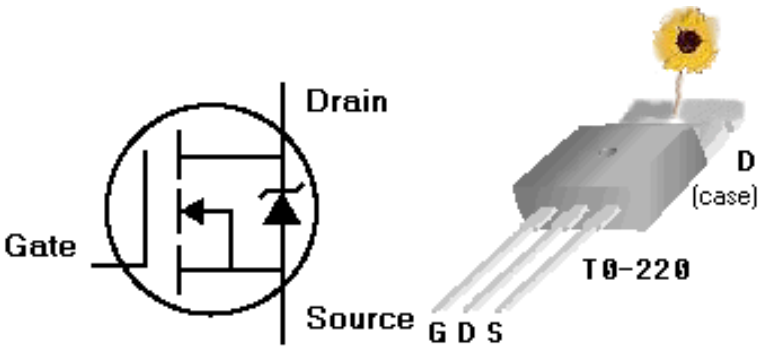
Only a few of the possibilities are needed. In fact, seven of the combinations must be avoided because they short out the power supply. We can use the forward and reverse, one of the offs and, optionally, one of the brakes.

The braking action might need a little explaining. A motor's armature will pass through the fields of the magnets when it is turning, as will the wire that's wound around the armature. That will induce an electrical current in the wire. Spin the shaft of

a motor and you can measure a voltage at its terminals; it has become a generator. Short the motor's terminals and you will feel it resist your attempts to spin the shaft. That's because the magnet that's formed by the current induced in the armature winding is opposite to that of the motor's field magnet. The opposite poles attract, resulting in a braking action. Both of the brake combinations short the terminals of the motor together.

There are a variety of devices that can be used for switches, including common mechanical switches, transistors and relays. The TIP120 will work for low currents, but a heat sink is needed for high current. The reason is that most transistors have a voltage drop between their collector and emitter of about .7 volts. Recall that power is P = I * V. With say, 5 amps and a .7V drop, the TIP120 will dissipate 5 * .7 = 3.5 watts. That's wasted power that could have been used by the load with a more efficient device.

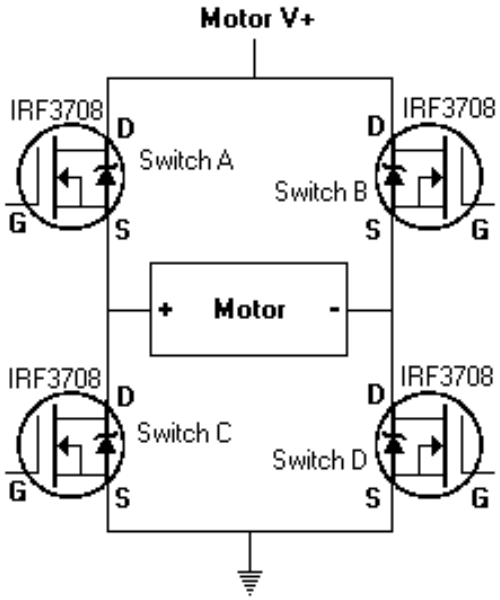
The switch to be used here is a **Metal Oxide Semiconductor Field Effect Transistor**, or **MOSFET**. MOSFETs come in two polarities called N and P channel. The one to be used here is an N channel device. A schematic representation and a picture showing the pin designations is below (the sunflower is almost never included). Notice that the Drain is also connected to the metal tab, making it necessary to insulate the MOSFET when using a heatsink:



An **enhancement mode** N channel MOSFET is turned on by making the gate more positive than the source. What is enhanced is the **conductivity** between source and drain. The unit of conductivity is the **mho**, which is simply ohm spelled backwards. The IRF3708 used here will turn on if the gate is about 2 volts more positive than the source (see IRF3708.PDF). The gate is not physically connected to the rest of the device. It influences the device by the fact that it is on one side of a very thin insulator, and works by means of the imposed field effect. In fact, the insulator is so thin that it can be ruptured by static voltages as weak as that from a piece of plastic. **Great care** should be taken when working with a MOSFET. The leads are sometimes shorted with a piece of wire or conductive foam when shipped. The shorting material should not be removed until the last moment when installing the device. Also, make sure you are grounded or have at least touched ground before touching a MOSFET.

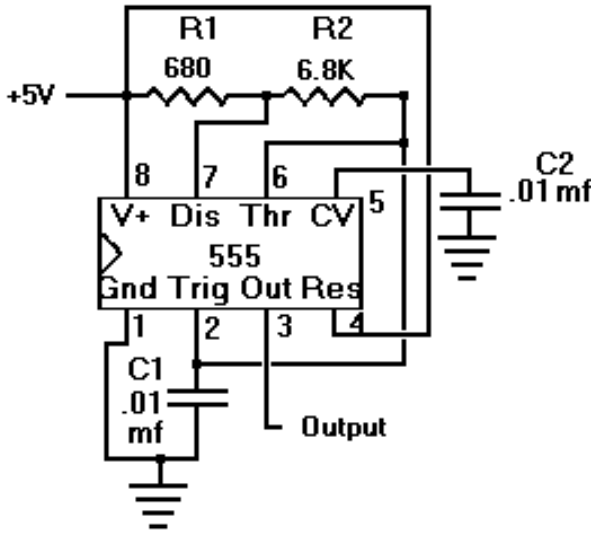
Now let's calculate power again. The resistance from Source to Drain of the IRF3708 is very high when it is off, but a maximum of only .029 ohms when it is on. Recall from How To Read A Schematic that V = I * R. That means we can replace the V in the power equation above with I * R since the two are equal. Thus,
$P = I * (I * R) = I^2 * R$.
If we use the same 5 amp current and .029 ohms we get:
$P = 5^2 * .029 = 25 * .029 = .725$ watts,
which means wasted power is reduced by almost 5 times.

The basic h-bridge looks like this using MOSFETs:



There is a bit of a problem when a MOSFET is used for Switch A or Switch B. The Drain is connected to the V+ for the motor and the Source to the motor. If the gate is then made to be 2 volts greater than the Source, the MOSFET will turn on. When that happens however, the voltage at the Source will increase until the gate is no longer 2 volts greater than the Source and the MOSFET will turn off. Another voltage source is needed that is always greater than Motor V+ by at least 2 volts.

Enter the 555 timer/oscillator (so called because it oscillates between a high and low level) (see LMC555.PDF):



The 555 timer has been around for about 30 years. A very good tutorial on how the "triple nickel" works has been written by Williamson Labs. Only the basic operation will be covered here. Positive 5V power goes to pin 8 and ground to pin 1. Pin 4 is a reset line that is active when it's low. It also gets +5V to disable it. Pin 5 is for an adjustment that is not used here. The capacitor connected to it helps keep down potential noise. The output is on pin 3.

In the circuit above, the output will be high as long as the voltage on pin 2 is less than 1/3 of V+ and will go low when pin 6 reaches 2/3 of V+. The following is the sequence of events for the circuit:

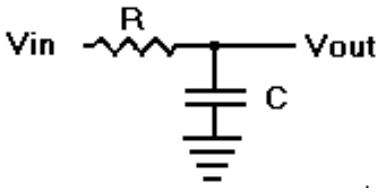
1. Pin 2 is less than 1/3 V+ when power is turned on, so the output on pin 3 starts out high.
2. The series value of R1 and R2 (R1 + R2) will start charging the capacitor connected to pins 2 and 6.
3. When the voltage on the capacitor reaches 2/3 of V+, the output on pin 3 will switch low. Pin 7, an open collector output, will also go low and begin discharging the capacitor through R2. Only R2 is used for discharge.
4. The capacitor will discharge until it reaches 1/3 of V+. The output will again go high, pin 7 will stop discharging, and the cycle will start over.

The output will be a series of pulses. The frequency of the pulse train will be:

F = 1/(.693 * C1 * (R1 + 2 R2))
Thus, F = 1/(.693 * 10^{-8} * (680 + (6.8K * 2))) = 10105.05212Hz Hz, or about 10K Hz.
(rusty on exponents? .01 = 10^{-2} so .01mf = 10^{-2} * 10^{-6} = 10^{-8} Farads -- just add the exponents to multiply)

Keeping R1 significantly smaller than R2 produces what will be close to a 50% duty-cycle square wave.

The characteristics of the timer depend on the **RC Time Constants** formed by R1, R2 and C1. The basic RC circuit looks like this:



The RC time constant for the above is simply R * C, or just RC. It takes RC seconds for Vout to reach about 63% of Vin.

The RC circuit is not linear, which is to say the voltage on C is not in direct proportion to time. For example, there will not be twice as much voltage on C in 2 seconds as there was in 1 second. That's because the rate at which the capacitor charges is in direct proportion to the current through R.

Let's say we start with Vin = 1 volt and that there is 0 volts on C. The current through R is I = V/R = 1/R initially since there is no voltage on C (see How To Read A Schematic). As the voltage on C increases however, it gets closer to Vin so the voltage drop across R decreases, and with it the current. As the current drops, the rate at which C charges also decreases.

A lot of things in nature work this way. A lot of water bursts forth initially when the dam breaks, but the rate decreases rapidly as the pressure falls. More water escapes, the pressure decreases, the rate falls and so on.

The voltage increases (or decreases when discharged through the resistor) on the capacitor **logarithmically**. The logarithm of a number is the power to which a base must be raised to get the number (see Data lines, bits, nibbles, bytes, words, binary and HEX).

For example, $\log_{10} 1000 = 3$ means that the base 10 must be raised to the power of 3 to get 1000.

The base e is commonly used for natural phenomena. It is equal to approximately 2.718281828. The special symbol **ln** is used for natural logarithms. ln 1000 = 6.907755279 says that $e^{6.907755279} = 1000$.

The voltage on a charging capacitor in an RC circuit in t seconds is shown by the charging equation:
$V_t = V_{in} * (1 - e^{-(t/RC)})$

Let's say t equals the RC time constant: t = RC. That makes the part in the parenthesis $1 - e^{-1} = .632120559$, which is where the 63% comes from. You should be able to find an e^x function on even the least expensive scientific calculator to confirm the above.

The 555 circuit alternately charges and discharges C1 between a maximum and minimum value. The discharge equation can be used to show the relationship of the voltages, RC and time:
$V_{min} = V_{max} * e^{-(t/RC)}$

Let's derive an equation that will tell us how much time it will take for charge and discharge. This can be done by isolating t to the left side of the equation:

1. Start with $V_{min} = V_{max} * e^{-(t/RC)}$
2. Divide both sides by V_{min}:
   $1 = (V_{max} * e^{-(t/RC)})/V_{min}$
3. Multiply both sides by $e^{(t/RC)}$:
   $e^{(t/RC)} = V_{max}/V_{min}$

(two rules here -- to multiply, add exponents, and anything raised to the power of 0 = 1)

4. Take the ln of both sides:
   $t/RC = \ln(V_{max}/V_{min})$
   (think, " to what power must e be raised to get $e^{(t/RC)}$?")
5. Multiply both sides by RC:
   $t = RC \ln(V_{max}/V_{min})$

In the case of the 555, V_{max} = 2/3 V and V_{min} = 1/3 V, so $V_{max}/V_{min} = 2$.
ln 2 = .693147181 (look familiar?).
Both resistors are used for charging so

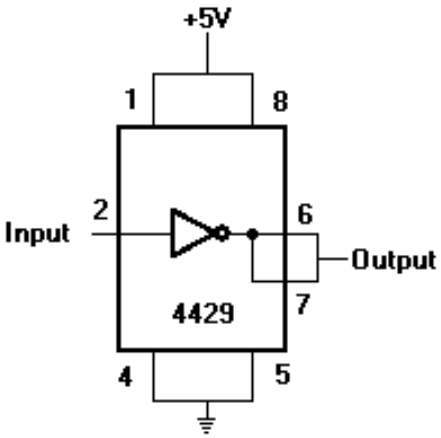
$t_{charge} = 7480 * 10^{-8} * \ln 2 = .000051847$ seconds

Discharge uses only R2:

$t_{discharge} = 6800 * 10^{-8} * \ln 2 = .000047134$ seconds

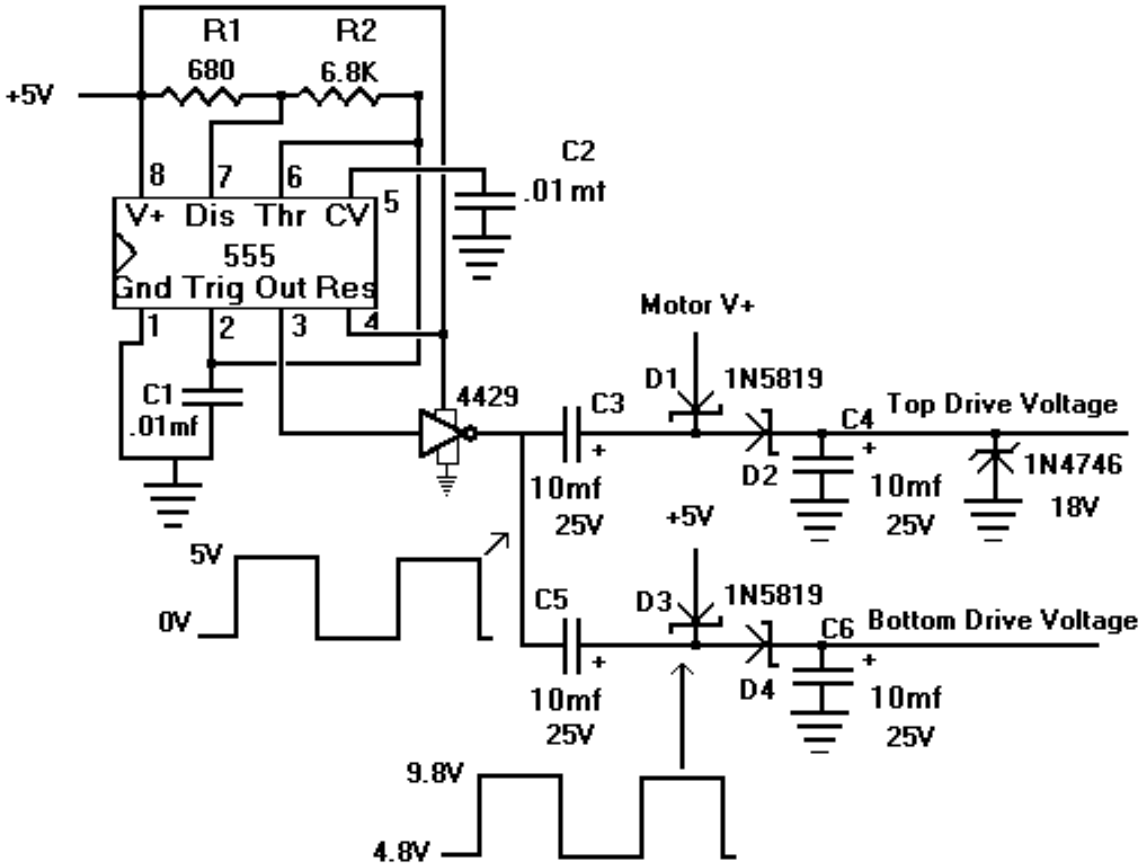
The time for a total cycle is the on time plus the off time:
= .000051847 + .000047134 = .000098981 seconds.
Note the almost 50% duty-cycle (on and off almost the same). The frequency is the inverse of the time for a cycle:
1/.000098981 = 10102.94821 Hz, which is more precise than when we rounded to .693. The target was 10K Hz. This is close enough.

This application will need more than the maximum 100ma current capability of the 555. A 4429 high current driver will provide the additional current (see TC4429.PDF):



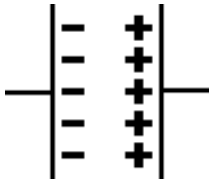
Please note that both pins 1 and 8 are used for power, and pins 4 and 5 for ground. Never use the device without using both pairs. It's also a good idea to use both 6 and 7 for the output. Also, be sure to place .1mf capacitors from power to ground near the device to absorb high current spikes.

A few more components can now be added to the 555 circuit to form a **voltage doubler**:

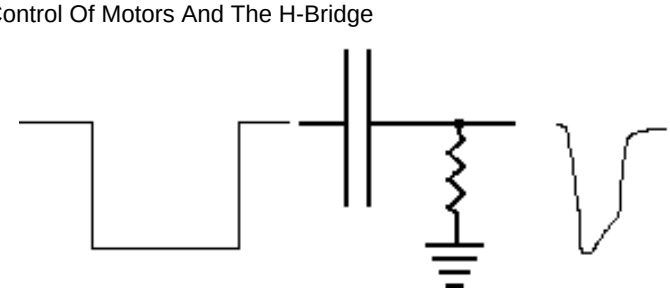


The 1N5819s are Schottky barrier rectifiers. They are used due to their lower-than-normal forward drop of .6V maximum, which falls to .4V or less under normal current loads. The output of the 4429 will swing from .025V to 4.975V with a 5 volt supply. I got the above waveforms from actual measurements on an oscilloscope. Notice that the voltage through D3 offsets the square wave (raises the whole thing) on the right side of C5 so that the top of the waveform (its **peak**) is near 10 volts. D4 directs everything to a second capacitor that filters the output. I measured 9.71 volts for the Bottom Drive Voltage with no load. A doubler is actually an almost-doubler. Note that C3, C4, C5 and C6 are all electrolytics and are polarized.

One of the secrets of the doubler is the fact that a capacitor such as C5 permits the pulse train from the oscillator to get through, but does not allow the DC voltage from D3 to get to the 4429. That's what allows the pulse train and the DC to add to each other, to then be usured out by D4 to the filter capacitors. C5 is functioning as a **blocking capacitor**. It functions to block pure DC, but lets AC or pulses through. Consider the following capacitor:



A negative charge has been produced on the left plate of the capacitor (see How to Read a Schematic). Since like charges repel, negatively charged electrons are pushed off of the other plate, rusulting in a positive charge. The capacitor can be said to have reached a point of equilibrium by the time the charges are balanced as in the above illustration. When the charge is first applied to the left plate and the electrons are driven off of the right plate however, whatever negative charge that was on the right plate will flow off of it, providing there is a path from the right plate to ground. A resistor or other load can supply the path. If the resistor is small enough to cause the capacitor to reach equilibrium before the next change in direction of the square wave, there will be only a pulse in response to the input. The width of the pulse is determined by the time it takes to reach equilibrium:



The above illustrates the **step response** of the circuit. C5 is large enough in the doubler circuit and the load light enough that the pulse train appears to get through unharmed. A careful reading will reveal however, that direct current won't appear to get through at all since it charges or discharges the capacitor only when power is first turned on. Equilibrium is reached early in the process, and the result is an offset of the pulse train.

Since C5 is large compared to the load and the width of the input pulses, the right side charges and discharges look virtually the same as the input on the left side. There will always be some distortion, but with large enough capacitance it will be minimal. Note that there is no negative input. The output of the 4429 is from 0 to 5 volts. When the pulse rises, a positive charge is pushed off of the right side of the capacitor. When the pulse again falls to 0, the left side will then be more negative than the right. This will **attract** the positive charge on the right, pulling it to 0 to complete the cycle.

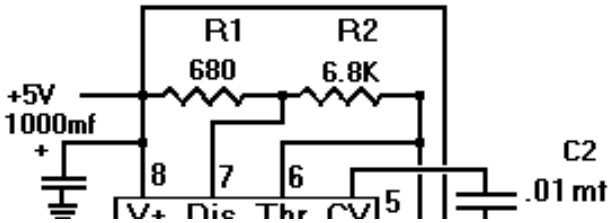
The bottom drive section is actually just sort of thrown in since the IRF3708 turns on at just 2 volts. Using almost 10 volts however, serves to lower the Drain to Source resistance even more. The data sheet says it falls to as little as .008 ohms with a Gate to Source voltage of 10 volts.

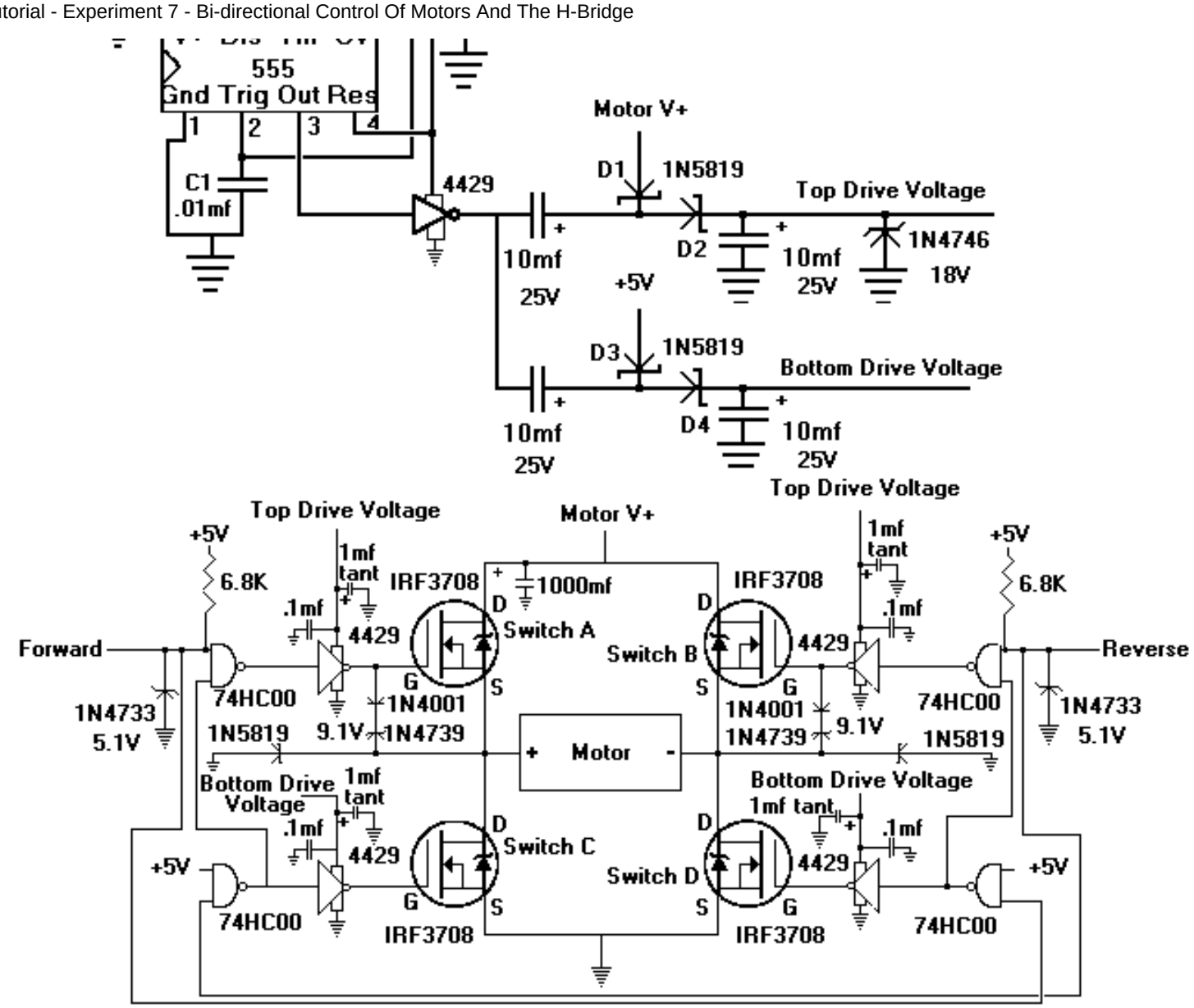
The circuit for the top MOSFETSs works in a similar manner, except that the square wave is offset by the motor voltage. Whatever the motor voltage is (up to 12 volts), the top Gate to Source voltage will always be about 5 volts greater. The 1N4746 zener keeps the voltage from being greater than 18 volts, which is the limit of the 4429s that will also be used to drive the MOSFETs.

4429s will be used to drive the MOSFETs because the MOSFETs have a high gate capacitance, and it is best to drive them full on or full off as quickly as possible. The idea is to spend as little time as possible in a moderate resistance region in order to minimize the power they consume. Fully off, the leakage current from Drain to Source is about 20 microamps. Since power is P = I * V, it's almost nothing in the off state. Fully on, the resistance is a maximum of .029 ohms. Since P = I^2 * R, the power is very little in the on state as well. The gate capacitance is 2417pf (**pf = picofarads = 10^{-12} Farad**) however, which calls for a high current to drive it. We can figure out how fast the MOSFET will be turned on or off using t = R * C from above. From Ohm's Law, R = V/I, so t = (V/I) * C, or t = (V * C)/I. From the 4429 data sheet it will be found that it can deliver a short-duration peak current of 6 Amps. The maximum voltage to a gate will be 10 volts. Using the gate capacitance above, t = (10 * 2417 * 10^{-12})/6 = about .04028 microseconds -- not bad at all.

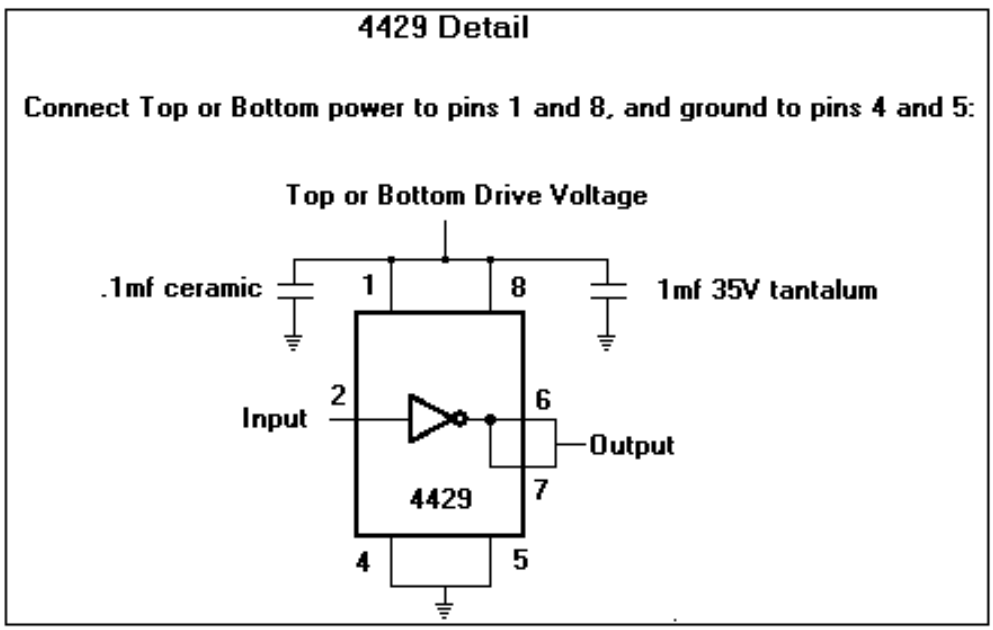
Back to the H-Bridge. Things can be simplified by saying we want to turn on A and D for forward, B and C for reverse and C and D for brake. Shorts can be avoided if forward and reverse disable each other. This can be done with a NAND gate (see the Boolean Logic section). The output of a two-input NAND gate will go low if both of its inputs are high. This permits us to use a NAND gate both as a gate and as an inverter. An inverter is formed by connecting one side of the gate to +5V. If the other side is then made high the output will be low. If it's low the output will be high. That's what in inverter does for a living.

The NAND gate used here is a 74HC00. It tolerates the high noise content of this type of circuit better than the standard 74LS00. One reason for that is that it requires its active high input voltage to be greater than the more common variety (compare 74HC00.PDF to 74LS00.PDF). The 6.8K pull-up resistors make certain the high voltage will be closer to 5 volts. The full circuit follows. Please be certain to connect the top and bottom drive voltages from the doubler to the h-bridge part of the circuit. The connections are left out of the schematic to make it easier to read:





NOTE: The 74HC00 gets +5V on pin 14 and ground on pin 7



Let me know if you would like me to offer the above as a kit with parts and a board by sending me an e-mail. I'll put one together if I get enough requests to pay for setting it up. The parts without a board can be ordered on the order page.

The combination of the 1N4001 rectifiers and 1N4739 zeners limits the gate to source voltage on the top switches to 10 volts (maximum is 12). The 1N5819 Schottky barrier rectifiers help keep negative pulses from backing up into the 4429 drivers, and the IN4001s block positive pulses. The 1N4733s help protect devices driving the inputs. Note that the MOSFETs already have zener protection built into them as well.

Be sure not to leave out any of the filter capacitors. Take special note of the fact that both a 1mf tantalum and a .1mf are connected to the power terminals of each 4429. The tantalums are used due to their low impedance at high frequencies. **Impedance** is expressed in ohms, and for a perfect capacitor is its reactance, which equals 1/(2 * pi * f) ohms, where f is frequency. Problem is, there is no such thing as a perfect capacitor. Tantalums are better here than electrolytics. The .1mf capacitors can be just about anything. Ceramic will work just fine. There is also a 1000mf electrolytic capacitor filter on +5V and on Motor +. There are all sorts of noises lurking around in motor control circuits. Leaving out filters can cause erratic operation or even destroy your 8255.

Notice what happens if Forward is made high. The forward signal takes one side of the Switch A NAND gate high. If the other side of the gate is also high, the output of the gate will be low, making the output of the 4429 high since it is an inverter, which will turn on MOSFET Switch A. If the other side of the NAND gate is low however, the output of the gate will be high, the output of the 4429 low and the switch will be off.

The Forward signal is also connected to an inverter formed with another NAND gate in the 74HC00. The output of the inverter goes to one side of the Reverse NAND gate, which disables Reverse. The inverted signal also goes to the 4429 for Switch D which turns it on. The Reverse MOSFET Switch B and Switch C are controlled in a like manner by the Reverse signal.

Now consider what happens if both Forward and Reverse are turned on. They will disable each other's top switches so that neither Switch A nor Switch B will be turned on. They will however, still turn on switches C and D which will short the motor out and provide a brake. The motor will be turned off but no braking action will take place when both Forward and Reverse are low.

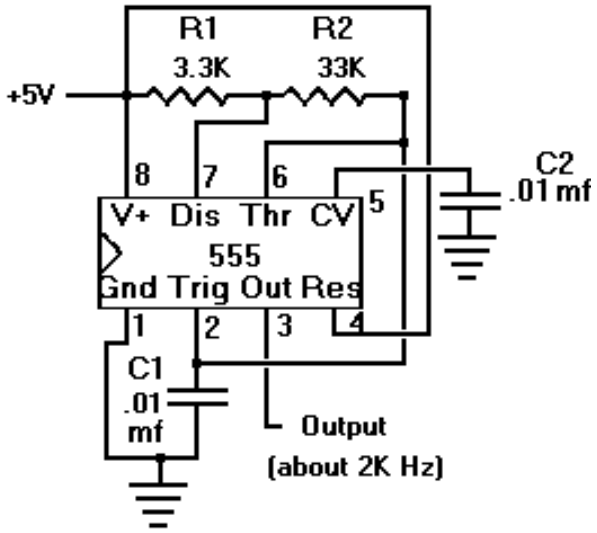
I have operated this circuit continuously for extended periods with a motor voltage of 11 volts and with 2 motors and 2 resistors in parallel (see How To Read A Schematic) to provide a load totaling 5 amps. The MOSFETs didn't even get warm with simple clip-on heatsinks while running this 55 watt load at 50% duty-cycle. It should be possible to operate significantly higher power loads with little need for additional heatsinking, and similar loads with no heatsinks at all.

The IRF3708 can take a maximum of 61 watts at high temperature according to the data sheet. Consider how much current about half that much power, 30 watts, translates to.

Recall that $P = I * V$ and $V = I * R$
so $P = I * (I * R) = I^2 * R$
divide both sides by R and switch sides,
$I^2 = P/R$
P = 30 and worst case R = .029 so
$I^2 = 30/.029$
$I^2 = 1034.482759$
Now take the square root of both sides and get:
I = 32.16337605 amps

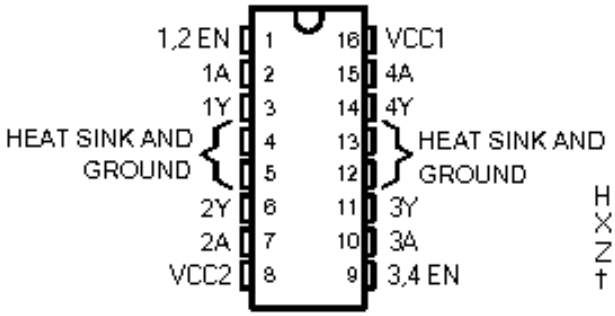
It would probably be a good idea to consider 30 amps the absolute maximum. It would also be a very good idea to use a good size heatsink with such high currents. After all, we are talking about driving a 360 watt load at 12 volts.

Testing the circuit before connecting it to the computer is strongly advised. I used the following circuit for testing:



Ground one of the inputs to the h-bridge and connect the above 2K Hz oscillator output to the other for a 50% duty-cycle test. Remove the oscillator, ground the other side, remove the ground from the previously grounded side and connect the oscillator to it to check the bridge for the opposite rotation of the motor. Run each direction for at least 15 minutes. The MOSFETs should get only barely warm with a 5 amp load, a 12 volt motor supply voltage and clip-on heatsinks. It would be a good idea to power the motor from other than the computer's power supply.

Many applications have no need to supply such high current devices. There are smaller h-bridges that require little additional circuitry. Consider the 754410 (see SN754410.PDF). The following is derived from the data sheet:

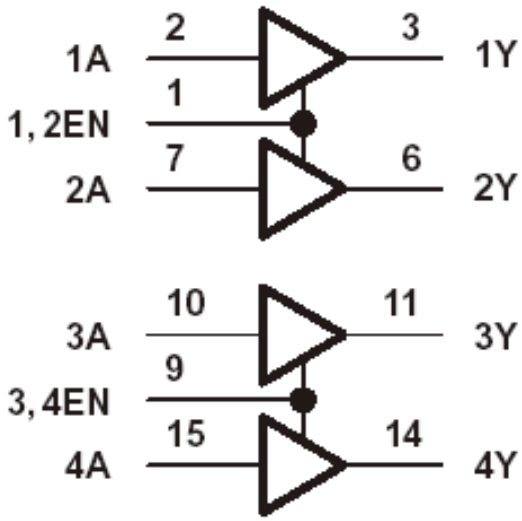


FUNCTION TABLE
(each driver)

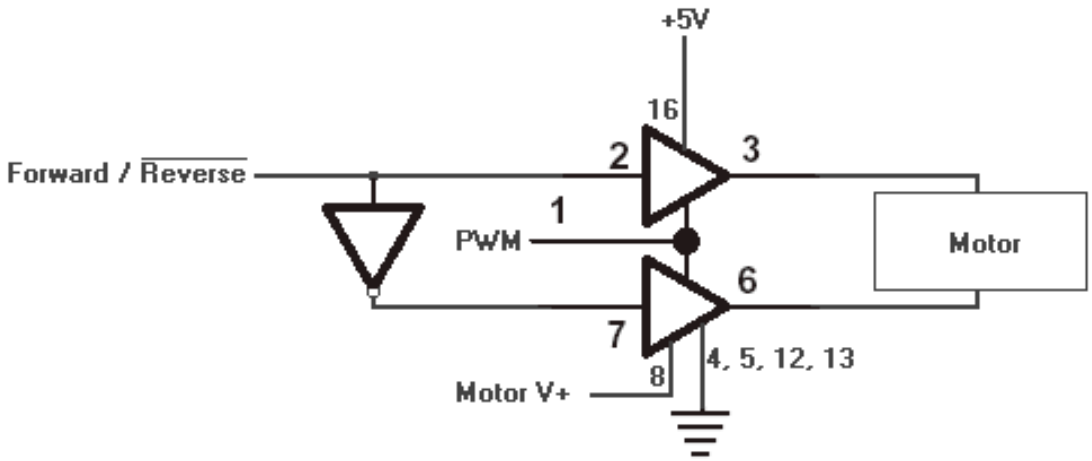
INPUTS †		OUTPUT
A	EN	Y
H	H	H
L	H	L
X	L	Z

H = High Level L = Low Level
X = Irrelevant
Z = Off
† Thermal shutdown turns off output

There are 4 sections inside, each with an input designated A, and an output designated Y:

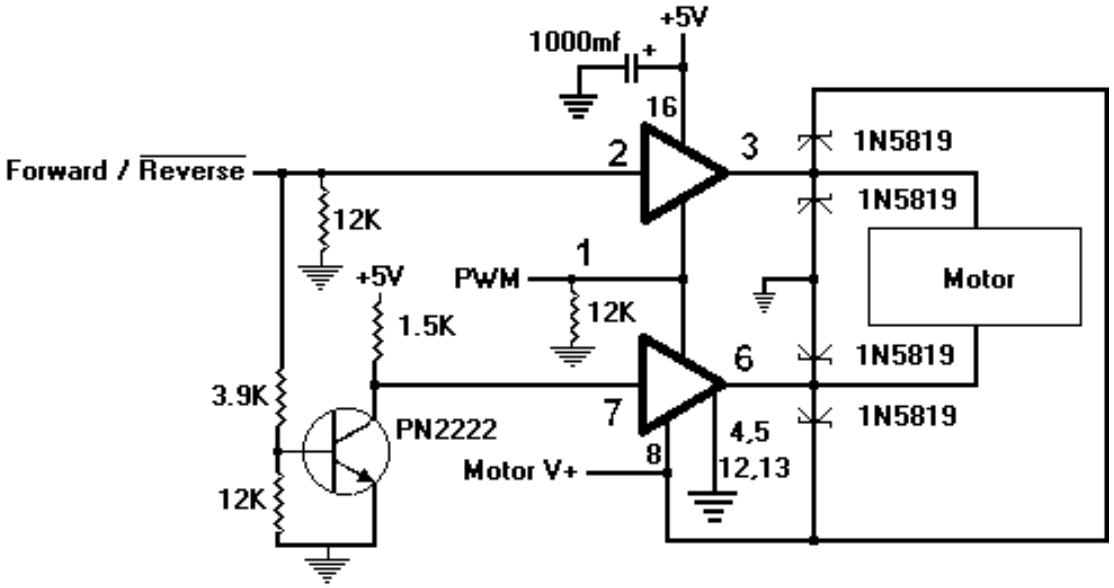


Each output is good for about 1 amp continuous output. The EN pins enable the outputs when high. The outputs are disconnected from what they are driving when the EN lines are low. An inverter can be added to provide bi-directional operation:



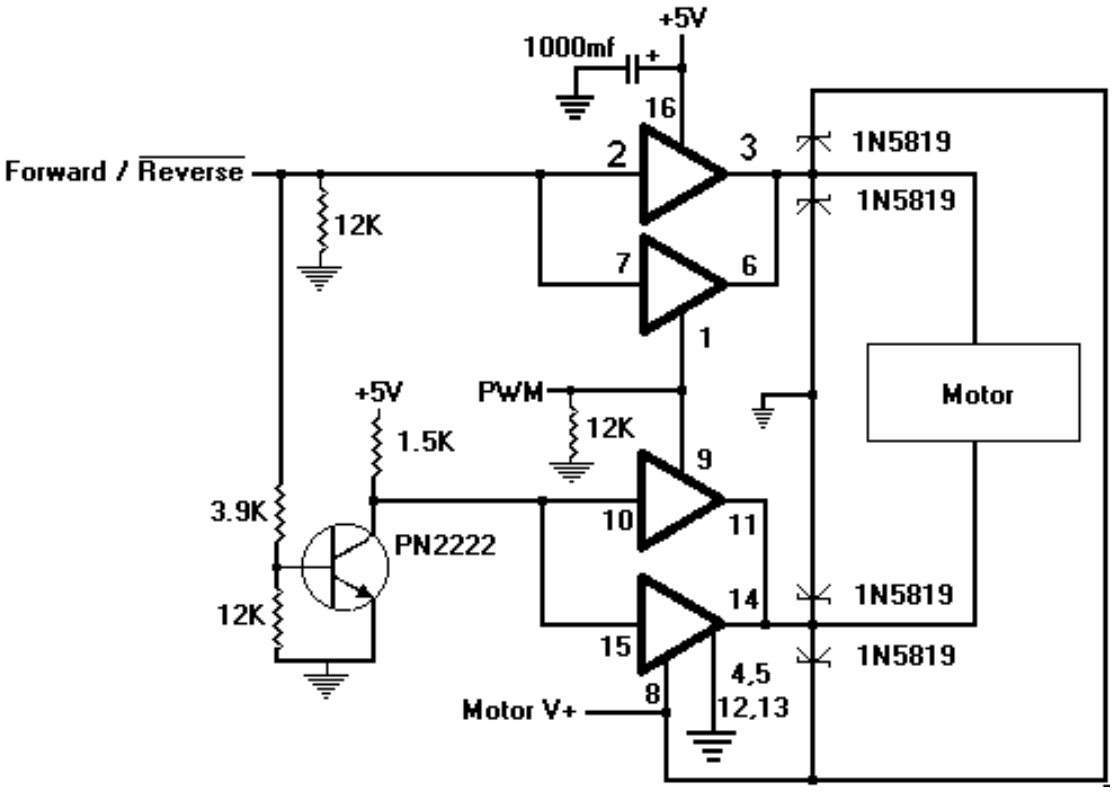
When the Forward/Reverse line is high the motor will rotate forward if the PWM line is also high. When the Forward/Reverse line is low (indicated by the bar over Reverse) the motor will rotate in reverse if the PWM line is high. Thus, the PWM line switches the motor on and off, providing pulses in the direction desired. The inverter could be one made out of a gate as above, or something such as the 74HC04, a HEX (means 6 here) inverter (see 74HC04.PDF).

Rather than use up room for a whole IC though, a single transistor can be used as an inverter. The PPI can provide a minimum of 3V and 2.5ma to the inverter. The inverter shown below only needs about .6ma input at 3 volts, and the 1.5K output resistor is small enough to restore the 2.5ma drive of the PPI. Stand the resistors on end for minimum space. The circuit with some additional protection rectifiers added looks like this:



I tested this circuit for extended periods using the .4 amp motor provided with Experimenter's Kit #1 which can be ordered, and the IC didn't even get warm without a heatsink. To duplicate the test, simple connect Motor V+ to +5V. Connect the 2K Hz test oscillator above to the PWM input. Disconnecting the oscillator should turn off the motor. Alternately connect and remove +5V from the Foward/Reverse line to check for proper direction control. The circuit will easily fit on the breadboard that can be ordered with Experimenter's Kit #1.

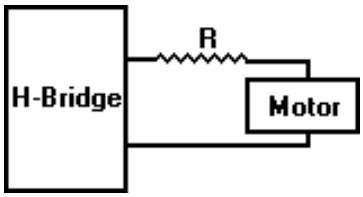
All sections can be used to increase output capability:



This one got a little too warm to touch with a 1.5 amp, 12 volt motor. That's because I forgot an important parameter of the chip. Its maximum power is only 2 watts. It didn't much like 18 watts! Actually, with the 50% duty cycle test I ran it was actually more like 9 watts since it was on only half the time. It's a good idea to use 100% in calculations though, since that could happen. Fortunately, the IC has a thermal shutdown circuit that helps protect it from dumb mistakes.

Here's how to calculate what the chip can take:
Since P = V * I, the maximum current at 12 volts is I = P/V = 2/12 = .167 amp.

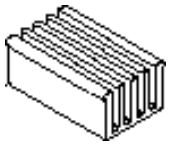
When Motor V+ was connected to +5V and one of the clip-on T0-220 heatsinks was stuck on top of the IC, it still got pretty warm, but the motor ran OK and the chip held up. The motor drew .77 amps, which means the IC would provide 3.85 watts at 100%. That's the value of a heatsink; it draws excess power off of the IC so that the IC itself never sees more than its maximum power, and never overheats. Another test was with a 20ma motor at 12 volts. The heatsink got just barely warm after more than an hour of 50% duty-cycle operation. In another test using 12 volts, a 100 ohm, 5 watt resistor was connected. Still just barely warm. Finally, a 5 volt motor that draws about 100ma was connected with an 82 ohm, 2 watt resistor in series with the h-bridge outputs using a Motor V+ of 12 volts. The IC was still just warm. All of the tests used the heatsink. The voltage divider section of How To Read A Schematic provides clues on how to figure out how to test with a motor you might have. Consider the following:



Current is the same through all elements of a series circuit. Thus, in the above example, the voltage dropped across the 82 ohm resistor with a current of 100ma is
V = I * R = .1 * 82 = 8.2 volts.
That means that 12 - 8.2 = 3.8 volts appears across the motor (providing the bridge can actually supply 12 volts, which is unlikely).

It would probably be a good idea to hold 100% power to about 2 watts and to glue on a heatsink such as the Wakefield

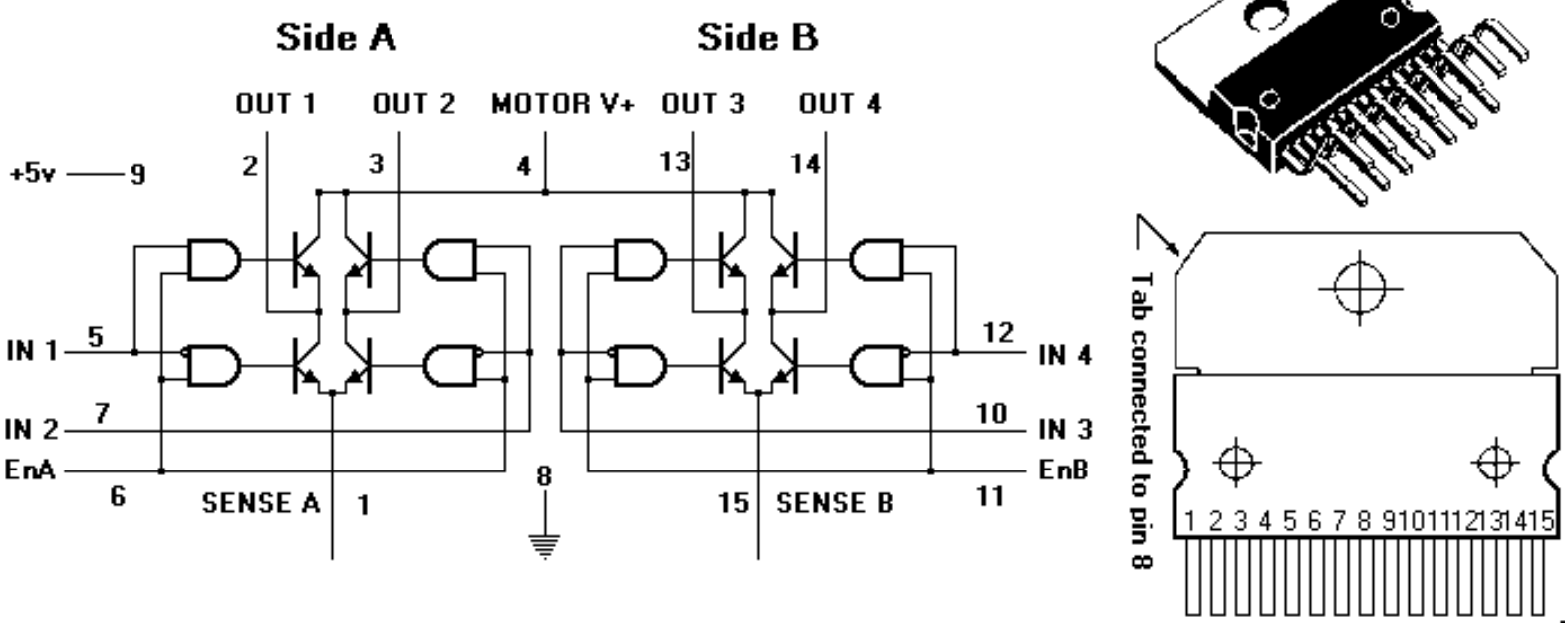
651-B:



People have even stacked 754410s for more power. Simply press one down on top of the other, solder their pins together, put a heatsink on top and go for it.

If you need more than the 754410's 2 watts but less than the 360 watts of my circuit, you might try the L298 dual h-bridge (see L298.PDF). It's good for a maximum of 25 watts.

The following is a simplified diagram, a picture and a pinout derived from the data sheet. The pins are actually in a straight line on the package and .05" apart, then bent out to accommodate standard .1" circuit board spacing. You will need to bend them left and right a little to get them to fit on a breadboard, but it can be done. The odd pins are on the front of the package and the even pins on the back:



The EnA and EnB inputs enable a side's outputs by making one side of all four of its AND gates high, which permits an IN line to turn on its output. Notice the bubble on one input of each bottom AND gate. That means that there is an inverter on the input, making it active low. A bottom gate's output will be high if one of its inputs is high and the other one is low. A top gate's output is determined in the same manner discussed in the Boolean Logic AND gate section; its output will be high if both of its inputs are high.

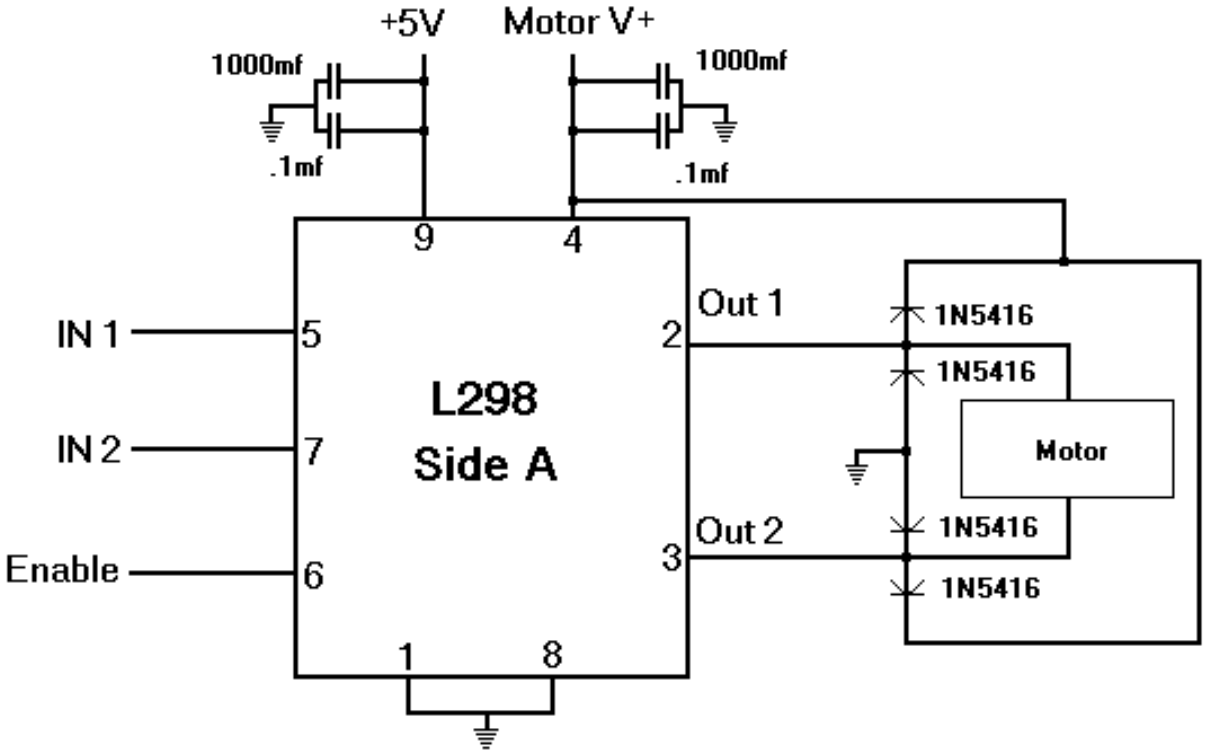
The following table shows the results of various combinations of inputs for side A. X means don't care. In other words, all switches will be off with Enable off, regardless of the status of IN 1 or IN 2:

EnA	IN 1	IN 2	OUT 1	OUT 2
0	X	X	All Switches Off - **Stop**	
1	0	0	Bottom Switches On - **Brake**	
1	0	1	Low	High
1	1	0	High	Low
1	1	1	Top Switches On - **Brake**	

Only one of the brakes is needed, so let's use the bottom transistors to short the motor out. OUT 1 high and OUT 2 low will be forward, and the reverse will be, well, reverse. The table now becomes:

EnA	IN 1	IN 2	Status
1	1	0	**Forward**
1	0	1	**Reverse**
1	0	0	**Brake**
0	X	X	**Stop**

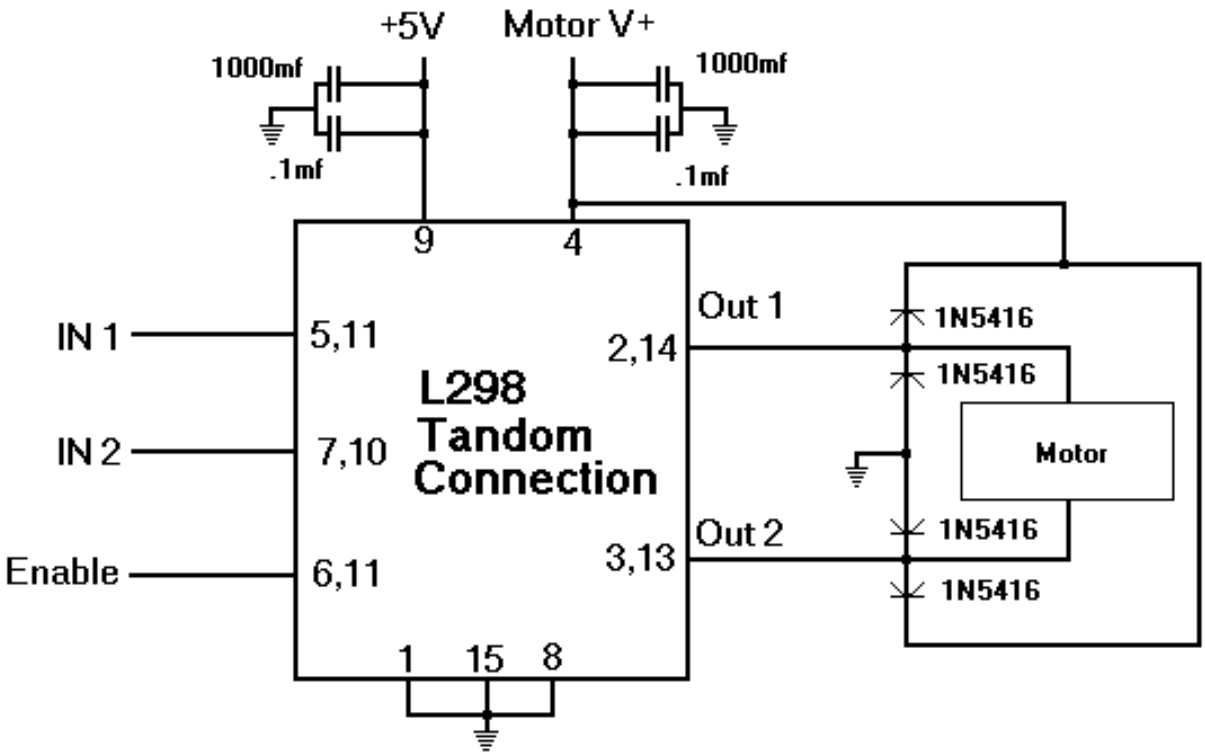
The following shows the connections for side A:



The protection rectifiers are fast recovery types such as the 1N5416. Fast recovery means they turn off very fast when the reason for them conducting (the back EMF of the motor) is removed. Recovery needs to be less than 200ns (**ns = nanoseconds = 10^{-9} second**).

This circuit was tested using 13.8 volts driving a .88 amp motor. One of the clip-on TO-220 heatsinks was fitted to the L298. It got pretty warm with this 12 watt load, but not too hot to touch. It would probably be a good idea to limit the circuit to half of the L298's 25 watt limit and stay under the 2 amp DC limit to be safe.

It's also possible to parallel both sides for more current carrying capability. This will permit going to 4 amps, although it would still be a good idea to hold down the total power:



Notice that channel 1 should be paralleled with channel 4, and channel 2 with channel 3.

The L298 can be bolted to a heatsink through its tab. No insulation is needed since the tab is ground.

The next step is to come up with some software that will make the board drive the various types of h-bridges. The first thing that must be done is to make the output control structure handle both directions of travel. To do that, some sections are added:

```
// outcont.h

struct OC
{
  int type; // type of bridge being controlled
  int direction; // 0 = off, 1 = forward, 2 = reverse

// Forward Elements

  int ForwardPortAddress; // address of the port with Forward output line

  char ForwardOnMask;     // mask that will turn Forward line on when
                          // ORed with the data value and stored back

  char ForwardOffMask;    // mask that will turn Forward line off when
                          // ANDed with the data value and stored back

  int *ForwardPortData;   // pointer to the data value for the Forward port

  long ForwardSetOn;      // on-time setting for Forward line

  long ForwardSetOff;     // off-time setting for Forward line

  long ForwardOnCount;    // counts left for on time
```

```
  long ForwardOffCount;   // counts left for off time

// Reverse Elements

  int ReversePortAddress; // address of the port with Reverse output line

  char ReverseOnMask;     // mask that will turn Reverse line on when
                          // ORed with the data value and stored back

  char ReverseOffMask;    // mask that will turn Reverse line off when
                          // ANDed with the data value and stored back

  int *ReversePortData;   // pointer to the data value for the Reverse port

  long ReverseSetOn;      // on-time setting for Reverse line

  long ReverseSetOff;     // off-time setting for Reverse line

  long ReverseOnCount;    // counts left for on time

  long ReverseOffCount;   // counts left for off time

// Brake Elements

  int BrakePortAddress; // address of the port with Brake output line

  char BrakeOnMask;     // mask that will turn Brake line on when
                         // ORed with the data value and stored back

  char BrakeOffMask;    // mask that will turn Brake line off when
                        // ANDed with the data value and stored back

  int *BrakePortData;   // pointer to the data value for the Brake port

// Direction Elements

  int DirectionPortAddress; // address of the port with Direction output line

  char DirectionOnMask;     // mask that will turn Direction line on when
                            // ORed with the data value and stored back

  char DirectionOffMask;    // mask that will turn Direction line off when
                            // ANDed with the data value and stored back

  int *DirectionPortData;   // pointer to the data value for the Direction port
};

// end outcont.h
```

Click here to download outcont.h (first save your current copy somewhere if you have made changes to it).

The new OC structure still has the basic elements from Experiment 6, but adds some. A type element and direction element

have been added to the top of the structure. The type element will hold a number indicating what type of h-bridge is being handled by this particular structure in the array. The direction element is 0 for off, 1 for forward and 2 for reverse.

Rather than having a single port address, a pair of masks and a data pointer, the structure now contains both forward and reverse sections, along with the direction and brake information needed by some bridges. Each section has its own data pointer and on and off masks.

The forward and reverse sections add on and off set value elements which serve to hold the counts to be stored in the counters which have also been added. The new configuration routine in the digital module takes changes to the structure into account:

```
// configure the array number location with the port numbers indicated
// and to the bit numbers dictated by the port numbers
// type:
// 0 = uni-directional, no brake
// 1 = uni-directional with brake
// 2 = pwm line, directional line, no brake
// 3 = pwm line, directional line, with brake
// 4 = dual pwm lines -- both high = brake
// 5 = pwm line and two direction lines as for L298
// 255 = end of points = tells isr not to look anymore & saves time
int ConfigureOutput(int arraynumber,
                    int type,
                    int ForwardPortNumber,
                    int ReversePortNumber,
                    int DirectionPortNumber,
                    int BrakePortNumber)
{
  int x;

  if(arraynumber < 0 || arraynumber > 23)
    return 0; // illegal number

  if(ForwardPortNumber < 0 || ForwardPortNumber > 23)
    return 0; // illegal number

  if(OutputControl[arraynumber] == NULL)
  {
    if((OutputControl[arraynumber] = malloc(sizeof(struct OC))) == NULL)
    {
      printf("Not enough memory for output control.\n");
      return 0;
    }
  }

  // zero out members
  memset(OutputControl[arraynumber], 0, sizeof(struct OC));

  if(type == 255)
  {
    OutputControl[arraynumber]->type = 255;
    return 1;
  }
```

```
// set up the forward masks
if(!SetPort(arraynumber, ForwardPortNumber,
            &OutputControl[arraynumber]->ForwardPortAddress,
            &OutputControl[arraynumber]->ForwardPortData,
            &OutputControl[arraynumber]->ForwardOnMask,
            &OutputControl[arraynumber]->ForwardOffMask))
  return 0;

if(!type) // uni-directional no brake has forward pwm only
  return 1;

if(type == 1 || type == 3 || type == 5) // 1,3,5 use brake line
                                        // (5 uses it for logic lines)
{
  if(!SetPort(arraynumber, BrakePortNumber,
              &OutputControl[arraynumber]->BrakePortAddress,
              &OutputControl[arraynumber]->BrakePortData,
              &OutputControl[arraynumber]->BrakeOnMask,
              &OutputControl[arraynumber]->BrakeOffMask))
    return 0;
}

if(type == 1 || type == 3 || type == 5) // 1,3,5 use direction line
                                        // (5 uses it for logic lines)
{
  if(!SetPort(arraynumber, DirectionPortNumber,
              &OutputControl[arraynumber]->DirectionPortAddress,
              &OutputControl[arraynumber]->DirectionPortData,
              &OutputControl[arraynumber]->DirectionOnMask,
              &OutputControl[arraynumber]->DirectionOffMask))
    return 0;

  OutputControl[arraynumber]->ReversePortAddress = // reverse is same as forward
    OutputControl[arraynumber]->ForwardPortAddress; // with a direction line

  OutputControl[arraynumber]->ReversePortData =
    OutputControl[arraynumber]->ForwardPortData;

  OutputControl[arraynumber]->ReverseOnMask =
    OutputControl[arraynumber]->ForwardOnMask;

  OutputControl[arraynumber]->ReverseOffMask =
    OutputControl[arraynumber]->ForwardOffMask;
}

if(type == 4) // 4 is a dual pwm so has a separate reverse line
{
  if(!SetPort(arraynumber, ReversePortNumber,
              &OutputControl[arraynumber]->ReversePortAddress,
              &OutputControl[arraynumber]->ReversePortData,
              &OutputControl[arraynumber]->ReverseOnMask,
              &OutputControl[arraynumber]->ReverseOffMask))
    return 0;
```

```
  }

  return 1;
}

// portflag settings:
// A CU B CL
// x  x x  x
// low = output, hi = input
// example:
// 0101
// A = out, CU = in, B = out, CL = in

int portflag = 0x0f;  // set by set_up_ppi() -- init to all inputs

// set up flags, etc. for an output port
int SetPort(int arraynumber,
            int PortNumber,
            int *PortAddress,
            int **PortData,
            int *OnMask,
            int *OffMask)
{
  int x;

  if(arraynumber < 0 || arraynumber > 23)
    return 0;

  if(PortNumber < 0 || PortNumber > 23)
    return ClearPort(arraynumber); // error - free port & return 0

  *OnMask = 1 << (PortNumber & 7); // shift by bits[2:0]
  *OffMask = ~*OnMask;

  x = PortNumber >> 3; // the port number is in bits 3 and 4

  switch(x) // 0 = Port A, 1 = Port B, 2 = Port C
  {
    case 0:
    if(portflag & 8) // bit 3 hi = A in, not out
      return ClearPort(arraynumber);
    *PortAddress = ppi_porta; // address for Port A
    *PortData = &porta_val;   // point to Port A data value
    break;

    case 1:
    if(portflag & 2) // bit 1 hi = B in, not out
      return ClearPort(arraynumber);
    *PortAddress = ppi_portb; // address for Port B
    *PortData = &portb_val;   // point to Port B data value
    break;

    case 2:
    if(portflag & 1 && *OnMask &0x0f) // bit 0 hi = CL in, not out
```

```
      return ClearPort(arraynumber);
    if(portflag & 4 && *OnMask &0xf0) // bit 2 hi = CU in, not out
      return ClearPort(arraynumber);
    *PortAddress = ppi_portc; // address for Port C
    *PortData = &portc_val;   // point to Port C data value
    break;
  }

  return 1;
}
```

Click here to download digital.c and here to download digital.h (first save your current copies somewhere if you have made changes to them).

After checking the range of the forward port number and array number, the routine tries to reserve memory for the position using malloc, then it uses memset to clear all members to 0. If type is 255, it means this is the last position used in the array, so it sets the structure's type member to 255 also, then leaves with a return of 1 (0 means there has been an error).

Once the routine reaches this point, the call to it is considered to be a legitimate pwm node setup. It first calls SetPort(..) to set up the forward parameters. Notice the parameters inside the call to SetPort(..) that are prefixed with an ampersand (&). Recall from Experiment 2 that variables passed by value to a routine can't be modified by the routine because a copy, rather than the actual variable, is used in the routine.

Here however, pointers are being passed of the variables that need to be set by the routine. SetPort(..) can dereference the pointers and gain access to the original variables. Notice the pointer to port data. Since it is already a pointer, SetPort(..) accepts it as a pointer to a pointer, indicated by the two asterisks for the **PortData argument. Dereferencing it allows setting the address to which it points.

Even though arguments sent to SetPort(..) are derived from the structure, they don't look like structure elements to SetPort(..). They look like normal arguments passed by value or as pointers. The result is a simplification which makes the routine a little easier to read. It also avoids using the structure as a global.

For example, the on mask is set in the following manner in Experiment 2:
OutputControl[arraynumber]->onmask = 1 << (portselect & 7);

Here, it's just:
*OnMask = 1 << (PortNumber & 7);
then the off mask is set:
*OffMask = ~*OnMask;

The portflag variable gets set to the mode variable sent to set_up_ppi(..). That allows SetPort(..) to check ports to make sure they are set up as outputs. Bits 3 and 4 of the PortNumber argument are used for the numerical reference for the switch statement. That permits checking to make sure the port is an output, then setting the port address and port data pointer.

Back to ConfigureOutput(..). The process is finished after setting up the forward elements if type is 0, which means the structure node is used for a uni-directional driver, such as one of the single T0-220 circuits. From there, decisions about what to set up are made based on the type number. For example, the brake and direction elements are set up for types 1, 3 and 5.

The new interrupt service routine (ISR) is shown below:

```
// the timer interrupt handler
// type:
// 0 = unidirctional, no brake
// 1 = unidirectional with brake
// 2 = pwm line, directional line, no brake
// 3 = pwm line, directional line, with brake
// 4 = dual pwm lines -- both high = brake
// 5 = pwm line and two direction lines as for L298
// 255 = last slot -- leave
interrupt new_timer()
{
  int x;

  disable();

  timer_counter++;

  for(x=0; x<24; x++)
  {
    if(OutputControl[x] == NULL // not set up or off
      || !OutputControl[x]->direction)
          continue;

    if(OutputControl[x]->type == 255) // last slot
          break;

    if(OutputControl[x]->direction == 1
      || OutputControl[x]->type < 2)
    {
      if(OutputControl[x]->ForwardOnCount > 0L)
      {
        OutputControl[x]->ForwardOnCount--;

        if(!OutputControl[x]->ForwardOnCount)
        {
          // keep existing bits but remove this one
          *OutputControl[x]->ForwardPortData
            &= OutputControl[x]->ForwardOffMask;

          // put the result in this node's port register
          outp(OutputControl[x]->ForwardPortAddress,
            *OutputControl[x]->ForwardPortData);

          OutputControl[x]->ForwardOffCount =
            OutputControl[x]->ForwardSetOff;
        }
      }

      // note that this will not decrement as soon as set
      // above, but will wait until the next interrupt
      else if(OutputControl[x]->ForwardOffCount > 0L)
      {
        OutputControl[x]->ForwardOffCount--;
```

```
      if(!OutputControl[x]->ForwardOffCount)
      {
        // keep existing bits and OR this one in
        *OutputControl[x]->ForwardPortData |=
          OutputControl[x]->ForwardOnMask;

        // put the result in this node's port register
        outp(OutputControl[x]->ForwardPortAddress,
          *OutputControl[x]->ForwardPortData);

        OutputControl[x]->ForwardOnCount =
          OutputControl[x]->ForwardSetOn;

      }

    }

  } // end if(OutputControl[x]->direction == 1
    // || OutputControl[x]->type < 2)

  else if(OutputControl[x]->direction == 2
      && OutputControl[x]->type > 1) // types 2,3 and 4 can have reverse
  {
    if(OutputControl[x]->ReverseOnCount > 0L)
    {
      OutputControl[x]->ReverseOnCount--;

      if(!OutputControl[x]->ReverseOnCount)
      {
        // keep existing bits but remove this one
        *OutputControl[x]->ReversePortData
          &= OutputControl[x]->ReverseOffMask;

        // put the result in this node's port register
        outp(OutputControl[x]->ReversePortAddress,
          *OutputControl[x]->ReversePortData);

        OutputControl[x]->ReverseOffCount =
          OutputControl[x]->ReverseSetOff;
      }

    } // end if(OutputControl[x]->ReverseOnCount > 0L)

    // note that this will not decrement as soon as set
    // above, but will wait until the next interrupt
    else if(OutputControl[x]->ReverseOffCount > 0L)
    {
      OutputControl[x]->ReverseOffCount--;

      if(!OutputControl[x]->ReverseOffCount)
      {
        // keep existing bits and OR this one in
        *OutputControl[x]->ReversePortData |=
```

```
            OutputControl[x]->ReverseOnMask;

          // put the result in this node's port register
          outp(OutputControl[x]->ReversePortAddress,
            *OutputControl[x]->ReversePortData);

          OutputControl[x]->ReverseOnCount =
            OutputControl[x]->ReverseSetOn;
        }

      } // end else if(OutputControl[x]->ReverseOffCount > 0L)

    } // end else if(OutputControl[x]->direction == 2
      // && OutputControl[x]->type > 1)

  } // end for(x=0; x<24; x++)

  enable();

} // end  new_timer()
```

It runs through all of the array locations and skips the ones that are not being used as indicated by their being NULL. It breaks out of the loop if a direction element is 255. It counts down the up or down counters as dictated by the direction element and turns on or off the appropriate port lines.

The pwm(..) routine is used to set up a pwm channel:

```
// Set up Pulse Width Modulation for an output
//
// arraynumber is the position in the output control array
//
// type:
// 0 = uni-directional, no brake
// 1 = uni-directional with brake
// 2 = pwm line, directional line, no brake
// 3 = pwm line, directional line, with brake
// 4 = dual pwm lines -- both high = brake
// 5 = pwm line and two direction lines as for L298
// 255 = last slot -- leave
//
// Forward and Reverse port numbers are pwm lines
//
// The Direction port number is provided for bridges that have a reverse line
// Set to anything if not used
//
// The Brake port number is provided for circuits that have a brake line
// Set to anything if not used
//
int pwm(int arraynumber, int type,
    int ForwardPortNumber, int ReversePortNumber,
    int DirectionPortNumber, int BrakePortNumber,
    double ForwardOnTime, double ForwardOffTime,
    double ReverseOnTime, double ReverseOffTime,
    int StartDirection)
```

```
{
  int x;

  if(StartDirection < 0 || StartDirection > 2)
      return 0;

  if(ForwardOnTime <= MinTime || ForwardOnTime >= MaxTime
    || ForwardOffTime <= MinTime || ForwardOffTime >= MaxTime)
      return 0;

  disable(); // no interrupts while setting up

  if(!ConfigureOutput(arraynumber, type,
                      ForwardPortNumber,
                      ReversePortNumber,
                      DirectionPortNumber,
                      BrakePortNumber))
  {
    enable();
    return 0;
  }

  OutputControl[arraynumber]->ForwardSetOn =
    (long)((frequency * ForwardOnTime) + 0.5); // round up at .5

  OutputControl[arraynumber]->ForwardSetOff
    = (long)((frequency * ForwardOffTime) + 0.5);

  OutputControl[arraynumber]->ForwardOnCount
    = OutputControl[arraynumber]->ForwardSetOn;

  OutputControl[arraynumber]->ForwardOffCount
    = OutputControl[arraynumber]->ForwardSetOff;

  OutputControl[arraynumber]->direction = StartDirection;

  OutputControl[arraynumber]->type = type;

  if(!type) // uni directional
  {
    enable();
    return 1;
  }

  if(type == 1 || type == 3) // 1 and 3 have a brake
  {
    *OutputControl[arraynumber]->BrakePortData
      &= OutputControl[arraynumber]->BrakeOffMask; // turn off the brake

    outp(OutputControl[arraynumber]->BrakePortAddress,
      *OutputControl[x]->BrakePortData);
  }
```

```
if(type > 1) // 2,3,4,5 use reverse pwm line, 2,3,5 use direction line
{
  if(ReverseOffTime <= MinTime || ReverseOffTime >= MaxTime
    || ReverseOffTime <= MinTime || ReverseOffTime >= MaxTime)
  {
    free(OutputControl[arraynumber]);
    OutputControl[arraynumber] = NULL;
    enable();
    return 0;
  }

  OutputControl[arraynumber]->ReverseSetOn =
    (long)((frequency * ReverseOnTime) + 0.5); // round up at .5

  OutputControl[arraynumber]->ReverseOnCount
    = OutputControl[arraynumber]->ReverseSetOn;

  OutputControl[arraynumber]->ReverseSetOff =
    (long)((frequency * ReverseOffTime) + 0.5);

  OutputControl[arraynumber]->ReverseOffCount
    = OutputControl[arraynumber]->ReverseSetOff;

  if(type == 2 || type == 3 || type == 5) // 2,3,5 use a direction line
  {
    if(StartDirection == 1)
    *OutputControl[arraynumber]->DirectionPortData
      != OutputControl[arraynumber]->DirectionOnMask; // set for forward

    else if(StartDirection == 2)
    *OutputControl[arraynumber]->DirectionPortData
      &= OutputControl[arraynumber]->DirectionOffMask; // clear for reverse

    outp(OutputControl[arraynumber]->DirectionPortAddress,
      *OutputControl[x]->DirectionPortData);

    if(type == 5)
    {
      if(StartDirection == 1)
        *OutputControl[arraynumber]->BrakePortData
          &= OutputControl[arraynumber]->BrakeOffMask; // turn off the brake

      else if(StartDirection == 2)
        *OutputControl[arraynumber]->BrakePortData
          |= OutputControl[arraynumber]->BrakeOnMask; // turn on the brake

      outp(OutputControl[arraynumber]->BrakePortAddress,
        *OutputControl[x]->BrakePortData);
    }
  }

} // end if(type > 1)

enable();
```

```
  return 1;

} // end int pwm(..)
```

Click here to download timer.c and here to download timer.h (first save your current copies somewhere if you have made changes to them).
These contain all of the timer setup, pwm and other timer routines on this page.

The StartDirection variable is first checked to make sure it ranges from 0 to 2 (0 = off, 1 = forward, 2 = reverse). All on and off times are checked for error against MinTime and MaxTime, which are determined when the timer is set up. MinTime is set to be equal to the period of the timer, which is 1.0/frequency. The timer can't move faster than it moves.

MaxTime is determined thus:
MaxTime = (pow(2.0, 32.0)/2.0)/frequency;
The standard compiler function pow(..) has the prototype
double pow(double base, double exponent);
It returns the base raised to the exponent. In this case, it's 2 raised to the 32nd power. Longs are 32 bits (at least with the DOS compiler I use), and binary has a base of 2. The maximum number that a long can hold is 232. The maximum allowable size is set to half that.

Interrupts are then disabled so nothing will be done that might conflict, then ConfigureOutput(..) output is called (described above). If that is successful, the forward elements are given values.

The forward and reverse sections have set on and set off elements that are longs. They hold the values to which the on and off counters will be set as appropriate. The on and off time arguments are doubles to permit as fine of resolution as possible. The set on and set off elements are calculated from the input arguments, but then cast (see Experiment 5) to longs.

For example, the forward set on value is calculated thus:

OutputControl[arraynumber]->ForwardSetOn =
(long)((frequency * ForwardOnTime) + 0.5);

The actual count will be the frequency times the on time. If the frequency is 1000Hz, for example, it takes 1000 cycles to take up one second, 1500 cycles to take up 1.5 seconds or 35 cycles to take up .035 seconds (35ms). Rounding is done at .5 counts. The long cast returns only the portion of the number to the left of the decimal.

An example: The frequency is 2386.36Hz and the on time is .025 seconds.
1. .025 * 2386.36 = 59.659
2. Add .5 and get: 59.659 + .5 = 60.159
3. Use only the portion to left of the decimal for the long set value = 60

The actual time will be 60/2386.36 = .025142895 seconds

Forward on and off counts are given their respective set values, and the structure's type and start direction values are set to the corresponding argument values. If type = 0, interrupts are enabled and a 1 is returned since the process is over.

A brake line is used if the type is 1 or 3. The routine turns the line off if that's the case.

If the type is 2, 3, 4 or 5, the reverse pwm line is used, so reverse on and off times are checked then set up if they are OK in the same manner as the forward on and off times. Notice that the memory used by the node is freed and the node set to NULL if there is an error in order to keep the ISR described above from using the node.

Types 2, 3 and 5 use a direction line. Only forward and reverse modes need be considered since the node was memset to 0. The line is turned on if StartDirection is set to 1 and turned off if it is set to 2.

Type 5 is for an L298 h-bridge. It uses the brake line in addition to the direction line for direction logic. The brake line is turned off for forward and on for reverse -- the reverse of the direction line.

Let's say we want to set up to run a simple, uni-directional pwm motor. To do that, enter the items needed and use 0s for the ones not needed:

```
pwm(MainMotor, 0, // array number, type
    PA0, 0,       // forward port, reverse port not used
    0,0,          // direction and brake not used
    0.01, 0.001,  // forward on and forward off times
    0,0,1);       // reverse on and off not used, start forward
```

MainMotor is part of an enumeration as explained in Experiment 6. PA0 is used as the pwm line, and the motor is supposed to start in the forward mode, which is all type 0 can do other than be turned off.

A shortcut can be written that takes care of all of the arguments that will be set to 0 anyway:

```
// set up a uni-directioanl pwm channel
int UniPwm(int arraynumber, int ForwardPortNumber,
    double ForwardOnTime, double ForwardOffTime)
{
  return pwm(arraynumber, 0, // arraynum, type 0
             ForwardPortNumber, 0, // forward port, no rev port
             0, 0, ForwardOnTime,  // no direction, no brake, forward on time
             ForwardOffTime, 0, 0, 1); // fwd off time, no rev on/off, fwd start
}
```

All UniPwm(..) needs is an array number, the forward port number, and the on time and off time. Notice that it returns what it gets when it calls pwm(..). An even shorter short cut only needs the duty cycle, rather than an on and off time:

```
// set up a uni-directioanl pwm channel using duty cycle
int UniPwmDuty(int arraynumber, int ForwardPortNumber,
    double ForwardDutycycle)
{
  int x;
  double ForwardOnTime,ForwardOffTime;

  if(ForwardDutycycle > 1.0)
    ForwardDutycycle/=100.0;

  if(ForwardDutycycle > 1.0)
    ForwardDutycycle = 1.0;

  if(ForwardDutycycle <= 0.5)
  {
    ForwardOnTime = minpulse;
    ForwardOffTime = ((1.0 - ForwardDutycycle)/ForwardDutycycle) * ForwardOnTime;
  }

  else
  {
    ForwardDutycycle = 1.0 - ForwardDutycycle;
    ForwardOffTime = minpulse;
    ForwardOnTime = ((1.0 - ForwardDutycycle)/ForwardDutycycle) * ForwardOffTime;
```

```
  }

  return UniPwm(arraynumber, ForwardPortNumber,
    ForwardOnTime, ForwardOffTime);
}
```

UniPwmDuty(..) allows the duty cycle to be expressed as a fraction or as a percent. The ForwardDutyCycle argument is treated as a decimal fraction if it is less than 1, or as a percent if greater. It is eventually worked with as a decimal fraction or 1 In both cases. It is divided by 100 if it is greater than 1.

If the duty cycle is then less than or equal to .5, the forward on time is set equal to the minpulse value that was set in the set_up_new_timer(..) routine. If it is greater than .5, then the forward off time is made equal to minpulse and the forward duty cycle is set to 1 - the forward duty cycle.

The purpose of all this number manipulation is to determine the caller's intent, determine whether on or off should be the smaller value and set it to the minimum pulse width size. That's about 2.5ms for the first two circuits above, or about 10ms if an L289 is included in the mix of circuits. The long pulse value is calculated using the short pulse.

For example, let's say the caller sends 30 as the duty cycle. It's first divided by 100 since it's greater than 1, which makes it .3. Since .3 is less than .5, the on time is assigned minpulse. The on time is thus equal to minpulse and is also .3 of the total time. The off time must then be .7 of the total time since the sum of the two must equal 1. Now let's say set_up_new_timer(..) sets minpulse to 10ms. Thus, if T is the total time,
.3T = .01
Divide both sides by .3:
T = .01/.3 = .033333333 seconds total time
Off time is .7 of total time, so it's
OffTime = .7 * .033333333 = .023333333 seconds
Also, OffTime = ((1 - .3)/.3) * .01
This determines how many times larger off is than on, then uses that to multiply by on to get off. There is no need to first determine the total time. This is the process used in the code above.

The fractions simply switch places if the duty cycle is greater than .5. UniPwmDuty(..) returns the return of UniPwm(..), which is given the array number, the forward port number and the calculated on and off times. Recall that UniPwm(..) returns the return of pwm(..). Cascading returns in this manner is very common.

Another handy shortcut is used to blink indicators such as LEDs:

```
// start a blinker
Blink(int arraynumber, int BlinkPort, double BlinkOnTime, double BlinkOffTime)
{
  return pwm(arraynumber, 0, // arraynum, type 0
           BlinkPort, 0, // forward port, no rev port
           0, 0, BlinkOnTime,  // no direction, no brake, forward on time
           BlinkOffTime, 0, 0, 1); // fwd off time, no rev on/off, fwd start
}
```

Just give it the array location, the port you want to use and the on and off times.

Use the following to test the above:

```
// experi7a.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>

// defines OC structure
#include "outcont.h"

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

enum
{
  MainMotor,
  LED1,
  LED2,
  LASTSLOT
};

void main(void)
{
  int x;
  double dc,oldfreq,newfreq;

  oldfreq = 1193180.0/65536.0;

  get_port(); // get the port number and establish register locations

  // make everthing an output
  set_up_ppi(Aout_CUout_Bout_CLout);

  set_up_new_timer(2000.0);

  newfreq = get_frequency();

  printf("old frequency = %f new frequency = %f Hz\n"
  ,oldfreq,newfreq);

  printf("10%%:\n");
  UniPwmDuty(MainMotor, PA0, 10.0);
  Blink(LED1, PA1, .3, .2);
  Blink(LED2, PA2, .15, .1);
  SetLast(LASTSLOT);

  showall();
  printf("\npress any key to continue\n");
  getch();

  printf("\n.01 on, .01 off:\n");
```

```
  pwm(MainMotor, 0, // array number, type
    PA0, 0,         // forward port, reverse port not used
    0,0,            // direction and brake not used
    0.01, 0.01,    // forward on and forward off times
    0,0,1);         // reverse on and off not used, start forward

  show(MainMotor);
  printf("\npress any key to continue\n");
  getch();

  while(!kbhit())
  {
    for(dc=20.0; dc<=90.0; dc+=10.0)
    {
      printf("dutycycle = %f\n",dc);

      UniPwmDuty(MainMotor, PA0, dc);

      wait(.1);

      if(kbhit())
        break;
    }

    if(dc < 90.0)
        break;

    for(dc=80.0; dc>20.0; dc-=10.0)
    {
      printf("dutycycle = %f\n",dc);

      UniPwmDuty(MainMotor, PA0, dc);

      wait(.1);

      if(kbhit())
         break;
    }

    if(dc > 20.0)
        break;
  }

  // don't forget to free memory!
  FreeOutputControl();

  portaoff();

  // be sure to restore the timer!
  restore_old_timer();
}

// end experi7a.c
```

Click here to download timer.c and here to download timer.h
Click here to download digital.c and here to download digital.h
Click here to download outcont.h
Click here to download constant.h
Click here to download extern.h
Click here to download experi7a.c
(first save your current copies somewhere if you have made changes to them).

Once you have downloaded everything, compile each of the C files then link them to produce experi7a.exe. After that's done, simply type in experi7a and press the enter key to run the program. The motor should increase then decrease speed continuously until a key is pressed.

It would be a good idea to connect LEDs to PA1 and PA2 so there will be a visual indication that the system is working. Use 390 ohm resistors for current limiting (see Experiment 3).

The following is another shortcut routine, this time for the IRF3708 circuit above that uses dual pwm lines:

```
// set up a dual pwm channel using duty cycle
int DualPwmDuty(int arraynumber, int StartDirection,
    int ForwardPortNumber, double ForwardDutycycle,
    int ReversePortNumber, double ReverseDutycycle)
{
  double ForwardOnTime,ForwardOffTime;
  double ReverseOnTime,ReverseOffTime;

  if(ForwardDutycycle > 1.0)
    ForwardDutycycle/=100.0;

  if(ForwardDutycycle > 1.0)
    ForwardDutycycle = 1.0;

  if(ForwardDutycycle <= 0.5)
  {
    ForwardOnTime = minpulse;
    ForwardOffTime = ((1.0 - ForwardDutycycle)/ForwardDutycycle) * ForwardOnTime;
  }

  else
  {
    ForwardDutycycle = 1.0 - ForwardDutycycle;
    ForwardOffTime = minpulse;
    ForwardOnTime = ((1.0 - ForwardDutycycle)/ForwardDutycycle) * ForwardOffTime;
  }

  if(ReverseDutycycle > 1.0)
    ReverseDutycycle/=100.0;

  if(ReverseDutycycle > 1.0)
    ReverseDutycycle = 1.0;

  if(ReverseDutycycle <= 0.5)
  {
    ReverseOnTime = minpulse;
```

```
    ReverseOffTime = ((1.0 - ReverseDutycycle)/ReverseDutycycle) * ReverseOnTime;
  }

  else
  {
    ReverseDutycycle = 1.0 - ReverseDutycycle;
    ReverseOffTime = minpulse;
    ReverseOnTime = ((1.0 - ReverseDutycycle)/ReverseDutycycle) * ReverseOffTime;
  }

  return pwm(arraynumber, 4, // type 4 is dual pwm
    ForwardPortNumber, ReversePortNumber,
    0,0, // no direction or brake port numbers
    ForwardOnTime, ForwardOffTime,
    ReverseOnTime, ReverseOffTime,
    StartDirection);
}
```

Most of the calculations are the same. The only real difference is the fact that DualPwmDuty(..) needs one more array location and duty-cycle variable, and that the call to pwm(..) adds the appropriate variables. Note that the motor will not immediately start if StartDirection is set to 0. This is useful in situations where it is desirable to set up the motor on initiation, but not turn it on until later.

The Forward(..) and Reverse(..) routines set the structure's direction variable as appropriate and turn on or off pwm lines as needed. Note that the L298 part of the routines (type = = 5) uses the brake line as part of its direction logic, in addition to the direction line. The Stop(..) routine sets the direction variable to 0 and turns off the pwm lines:

```
// go forward
int Forward(arraynumber)
{
  if(OutputControl[arraynumber] == NULL)
    return 0;

  if(OutputControl[arraynumber]->type < 2)
    return 0; // < 2 is unidirectional

  if(OutputControl[arraynumber]->type == 255)
    return 0; // 255 is last node

  disable();

  OutputControl[arraynumber]->direction = 1;

  // 2,3 and 5 use the direction line
  if(OutputControl[arraynumber]->type == 2
  || OutputControl[arraynumber]->type == 3
  || OutputControl[arraynumber]->type == 5)
  {
    *OutputControl[arraynumber]->DirectionPortData
      |= OutputControl[arraynumber]->DirectionOnMask; // set for forward

    outp(OutputControl[arraynumber]->DirectionPortAddress,
      *OutputControl[arraynumber]->DirectionPortData);
```

```
    if(OutputControl[arraynumber]->type == 5)
    {
      *OutputControl[arraynumber]->BrakePortData
        &= OutputControl[arraynumber]->BrakeOffMask; // turn off the brake line

      outp(OutputControl[arraynumber]->BrakePortAddress,
        *OutputControl[arraynumber]->BrakePortData);
    }
  }

  // keep existing bits but remove this one
  *OutputControl[arraynumber]->ReversePortData
  &= OutputControl[arraynumber]->ReverseOffMask;

  // put the result in this node's port register
  outp(OutputControl[arraynumber]->ReversePortAddress,
  *OutputControl[arraynumber]->ReversePortData);

  enable();

  return 1;
}

// go in reverse
int Reverse(arraynumber)
{
  if(OutputControl[arraynumber] == NULL)
    return 0;

  if(OutputControl[arraynumber]->type < 2)
    return 0; // < 2 is unidirectional

  if(OutputControl[arraynumber]->type == 255)
    return 0; // 255 is last node

  disable();

  OutputControl[arraynumber]->direction = 2;

  // 2,3 and 5 use the direction line
  if(OutputControl[arraynumber]->type == 2
  || OutputControl[arraynumber]->type == 3
  || OutputControl[arraynumber]->type == 5)
  {
    *OutputControl[arraynumber]->DirectionPortData
      &= OutputControl[arraynumber]->DirectionOffMask; // clear for reverse

    outp(OutputControl[arraynumber]->DirectionPortAddress,
      *OutputControl[arraynumber]->DirectionPortData);

    if(OutputControl[arraynumber]->type == 5)
    {
      *OutputControl[arraynumber]->BrakePortData
        &= OutputControl[arraynumber]->BrakeOnMask; // turn on the brake line
```

```
      outp(OutputControl[arraynumber]->BrakePortAddress,
        *OutputControl[arraynumber]->BrakePortData);
    }
  }

  // keep existing bits but remove this one
  *OutputControl[arraynumber]->ForwardPortData
  &= OutputControl[arraynumber]->ForwardOffMask;

  // put the result in this node's port register
  outp(OutputControl[arraynumber]->ForwardPortAddress,
  *OutputControl[arraynumber]->ForwardPortData);

  enable();

  return 1;
}

// stop a pwm channel
int Stop(int arraynumber)
{
  if(OutputControl[arraynumber] == NULL)
    return 0;

  disable();

  OutputControl[arraynumber]->direction = 0;

  // keep existing bits but remove this one for forward
  // for L289, this takes its enable line low
  *OutputControl[arraynumber]->ForwardPortData &=
    OutputControl[arraynumber]->ForwardOffMask;

  // put the result in this node's port register
  outp(OutputControl[arraynumber]->ForwardPortAddress,
       *OutputControl[arraynumber]->ForwardPortData);

  // keep existing bits but remove this one for reverse
  *OutputControl[arraynumber]->ReversePortData &=
    OutputControl[arraynumber]->ReverseOffMask;

  // put the result in this node's port register reverse
  outp(OutputControl[arraynumber]->ReversePortAddress,
       *OutputControl[arraynumber]->ReversePortData);

  enable();

  return 1;
}
```

Use the following to test the routines:

```
// experi7b.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>

// defines OC structure
#include "outcont.h"

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

enum
{
  MainMotor,
  LED1,
  LED2,
  LASTSLOT
};

void main(void)
{
  int x;
  double dc,oldfreq,newfreq;

  oldfreq = 1193180.0/65536.0;

  get_port(); // get the port number and establish register locations

  // make everthing an output
  set_up_ppi(Aout_CUout_Bout_CLout);

  set_up_new_timer(2000.0);

  newfreq = get_frequency();

  printf("old frequency = %f new frequency = %f Hz\n"
  ,oldfreq,newfreq);

  // set up a dual pwm channel using duty cycle
  DualPwmDuty(MainMotor, 0, // set up, but don't start main motor
    PA0, 75.0, // forward uses line PA0 at 75%
    PA1, 15.0); // reverse uses line PA1 at 15%
  Blink(LED1, PA2, .3, .2);
  Blink(LED2, PA3, .15, .1);
  SetLast(LASTSLOT);

  printf("MainMotor settings with 75%% forward and 15%% reverse:\n");
  show(MainMotor);
  printf("LEDs should be blinking and pwm off -- press any key to start test\n");
```

```
  getch();

  while(!kbhit())
  {
    printf("forward 75%% 2 seconds\n");
    Forward(MainMotor);
    wait(2.0); // wait 2 seconds to be kind to power supply

    printf("stop 2 seconds\n");
    Stop(MainMotor);
    wait(2.0); // wait 2 seconds to be kind to power supply

    printf("reverse 15%% 2 seconds\n");
    Reverse(MainMotor);
    wait(2.0); // wait 2 seconds to be kind to power supply

    printf("stop 2 seconds\n");
    Stop(MainMotor);
    wait(2.0); // wait 2 seconds to be kind to power supply
  }

  portaoff();
  portboff();
  portboff();

  // don't forget to free memory!
  FreeOutputControl();

  // be sure to restore the timer!
  restore_old_timer();
}

// end experi7b.c
```

Click here to download experi7b.c

This one turns on the brake for a dual pwm h-bridge by taking both pwm lines high for .1 second:

```

// brake a dual pwm channel
// taking both lines high brakes
int BrakeDualPwm(int arraynumber)
{
  if(OutputControl[arraynumber] == NULL)
    return 0;

  disable();

  OutputControl[arraynumber]->direction = 0;

  // keep existing bits but remove this one for forward
  *OutputControl[arraynumber]->ForwardPortData |=
    OutputControl[arraynumber]->ForwardOnMask;
```

```
  // put the result in this node's port register
  outp(OutputControl[arraynumber]->ForwardPortAddress,
       *OutputControl[arraynumber]->ForwardPortData);

  // keep existing bits but remove this one
  *OutputControl[arraynumber]->ReversePortData |=
    OutputControl[arraynumber]->ReverseOnMask;

  // put the result in this node's port register reverse
  outp(OutputControl[arraynumber]->ReversePortAddress,
       *OutputControl[arraynumber]->ReversePortData);

  enable();

  wait(0.1); // brake .1 sec

  Stop(arraynumber); // release lines

  return 1;
}
```

Use the following to test the brake routine:

```
// experi7c.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>

// defines OC structure
#include "outcont.h"

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

enum
{
  MainMotor,
  LED1,
  LED2,
  LASTSLOT
};

void main(void)
{
  int x;
  double dc,oldfreq,newfreq;

  oldfreq = 1193180.0/65536.0;
```

```
get_port(); // get the port number and establish register locations

// make everthing an output
set_up_ppi(Aout_CUout_Bout_CLout);

set_up_new_timer(2000.0);

newfreq = get_frequency();

printf("old frequency = %f new frequency = %f Hz\n"
,oldfreq,newfreq);

// set up a dual pwm channel using duty cycle
DualPwmDuty(MainMotor, 0, // set up, but don't start main motor
  PA0, 20.0, // forward uses line PA0 at 20%
  PA1, 20.0); // reverse uses line PA1 at 20%
Blink(LED1, PA2, .3, .2);
printf("LED1 on/off settings:\n");
show(LED1);
Blink(LED2, PA3, .15, .1);
printf("LED2 on/off settings:\n");
show(LED2);
SetLast(LASTSLOT);

printf("MainMotor on/off settings with 75%% forward and 15%% reverse:\n");
show(MainMotor);
printf("LEDs blinking and pwm off -- esc to quit, any other start test\n");

if(getch() != 27) // escape key
{
  while(!kbhit())
  {
    printf("forward 20%% 1 second\n");
    Forward(MainMotor);
    wait(1.0); // wait 1 second

    printf("brake, wait .5 seconds\n");
    BrakeDualPwm(MainMotor);
    wait(0.5); // wait 1 second

    printf("reverse 20%% 1 second\n");
    Reverse(MainMotor);
    wait(1.0); // wait 1 second

    printf("brake, wait .5 seconds\n");
    BrakeDualPwm(MainMotor);
    wait(0.5); // wait 1 second
  }
}

portaoff();
portboff();
portboff();
```

```
  // don't forget to free memory!
  FreeOutputControl();

  // be sure to restore the timer!
  restore_old_timer();
}

// end experi7c.c
```

Click here to download experi7c.c

This one sets up an L298:

```
// set up an L289
int L289PwmDuty(int arraynumber, int StartDirection,
    int PwmPortNumber, double ForwardDutycycle,
    double ReverseDutycycle, int Direction1Port, int Direction2Port)
{
  double ForwardOnTime,ForwardOffTime;
  double ReverseOnTime,ReverseOffTime;

  if(ForwardDutycycle > 1.0)
    ForwardDutycycle/=100.0;

  if(ForwardDutycycle > 1.0)
    ForwardDutycycle = 1.0;

  if(ForwardDutycycle <= 0.5)
  {
    ForwardOnTime = minpulse;
    ForwardOffTime = ((1.0 - ForwardDutycycle)/ForwardDutycycle) * ForwardOnTime;
  }

  else
  {
    ForwardDutycycle = 1.0 - ForwardDutycycle;
    ForwardOffTime = minpulse;
    ForwardOnTime = ((1.0 - ForwardDutycycle)/ForwardDutycycle) * ForwardOffTime;
  }

  if(ReverseDutycycle > 1.0)
    ReverseDutycycle/=100.0;

  if(ReverseDutycycle > 1.0)
    ReverseDutycycle = 1.0;

  if(ReverseDutycycle <= 0.5)
  {
    ReverseOnTime = minpulse;
    ReverseOffTime = ((1.0 - ReverseDutycycle)/ReverseDutycycle) * ReverseOnTime;
  }

  else
```

```
  {
    ReverseDutycycle = 1.0 - ReverseDutycycle;
    ReverseOffTime = minpulse;
    ReverseOnTime = ((1.0 - ReverseDutycycle)/ReverseDutycycle) * ReverseOffTime;
  }

  return pwm(arraynumber, 5,
    PwmPortNumber, PwmPortNumber, // frwrd & rev same = pwm/enable line
    Direction1Port, Direction2Port, // direction gets dir1, brake dir2
    ForwardOnTime, ForwardOffTime,
    ReverseOnTime, ReverseOffTime,
    StartDirection);
}
```

This one provides a brake for the L298 by taking both of the logic lines low and the pwm (enable) line high:

```

// apply brake for an L298
int BrakeL298(int arraynumber)
{
  if(OutputControl[arraynumber] == NULL)
    return 0;

  disable();

  OutputControl[arraynumber]->direction = 0;

  // make enable line high
  *OutputControl[arraynumber]->ForwardPortData
    |= OutputControl[arraynumber]->ForwardOnMask;

  // put the result in this node's port register
  outp(OutputControl[arraynumber]->ForwardPortAddress,
    *OutputControl[arraynumber]->ForwardPortData);

  // make direction and brake lines low for brake on bottom switches
  *OutputControl[arraynumber]->DirectionPortData
    &= OutputControl[arraynumber]->DirectionOffMask;

  outp(OutputControl[arraynumber]->DirectionPortAddress,
    *OutputControl[arraynumber]->DirectionPortData);

  // make brake line low
  *OutputControl[arraynumber]->BrakePortData
    &= OutputControl[arraynumber]->BrakeOnMask;

  outp(OutputControl[arraynumber]->BrakePortAddress,
   *OutputControl[arraynumber]->BrakePortData);

  enable();

  wait(0.1); // brake .1 sec

  // make enable line low
```

```
  *OutputControl[arraynumber]->ForwardPortData
    &= OutputControl[arraynumber]->ForwardOffMask;

  return 1;
}
```

Use this to test the L2289 routines:

```
// experi7e.c

#include
#include
#include

// defines OC structure
#include "outcont.h"

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

enum
{
  MainMotor,
  LED1,
  LED2,
  LASTSLOT
};

void main(void)
{
  int x;
  double dc,oldfreq,newfreq;

  oldfreq = 1193180.0/65536.0;

  get_port(); // get the port number and establish register locations

  // make everthing an output
  set_up_ppi(Aout_CUout_Bout_CLout);

  set_up_new_timer(2000.0);

  SetLast(LASTSLOT);

  newfreq = get_frequency();

  printf("old frequency = %f new frequency = %f Hz\n"
  ,oldfreq,newfreq);

  Blink(LED1, PA3, .3, .2);
```

```
  Blink(LED2, PA4, .15, .1);
  SetLast(LASTSLOT);

  // set up an L289
  L289PwmDuty(MainMotor, 0, // arraynumer, start direction
              PA0, 30.0,    // pwm/enable port, fwrd duty
              50.0,         // reverse dutycycle
              PA2,          // direction1 port
              PA1);         // direction2 port
  printf("Settings from L289 setup:\n");
  show(MainMotor);

  while(!kbhit())
  {
    Forward(MainMotor);
    printf("Forward\n");
    if(kbhit())
      break;
    wait(1.25);

    Stop(MainMotor);
    printf("Stop\n");
    if(kbhit())
      break;
    wait(1);

    Reverse(MainMotor);
    printf("Reverse\n");
    if(kbhit())
      break;
    wait(.75);

    BrakeL298(MainMotor);
    printf("Brake\n");
    if(kbhit())
      break;
    wait(1);
  }

  Stop(MainMotor);

  // don't forget to free memory!
  FreeOutputControl();

  // be sure to restore the timer!
  restore_old_timer();
}

// end experi7e.c
```

Click here to download experi7e.c

None were tested, but there are chips that use a pwm line and a direction line. This should work:

```
// set up a channel with a pwm line and a direction line -- no brake
// using duty cycle
int PwmAndDirectionDuty(int arraynumber, int StartDirection,
                        int PwmPortNumber,
                        int DirectionPortNumber,
                        double ForwardDutycycle,
                        double ReverseDutycycle)
{
  double ForwardOnTime,ForwardOffTime;
  double ReverseOnTime,ReverseOffTime;

  if(ForwardDutycycle > 1.0)
    ForwardDutycycle/=100.0;

  if(ForwardDutycycle > 1.0)
    ForwardDutycycle = 1.0;

  if(ForwardDutycycle <= 0.5)
  {
    ForwardOnTime = minpulse;
    ForwardOffTime = ((1.0 - ForwardDutycycle)/ForwardDutycycle) * ForwardOnTime;
  }

  else
  {
    ForwardDutycycle = 1.0 - ForwardDutycycle;
    ForwardOffTime = minpulse;
    ForwardOnTime = ((1.0 - ForwardDutycycle)/ForwardDutycycle) * ForwardOffTime;
  }

  if(ReverseDutycycle > 1.0)
    ReverseDutycycle/=100.0;

  if(ReverseDutycycle > 1.0)
    ReverseDutycycle = 1.0;

  if(ReverseDutycycle <= 0.5)
  {
    ReverseOnTime = minpulse;
    ReverseOffTime = ((1.0 - ReverseDutycycle)/ReverseDutycycle) * ReverseOnTime;
  }

  else
  {
    ReverseDutycycle = 1.0 - ReverseDutycycle;
    ReverseOffTime = minpulse;
    ReverseOnTime = ((1.0 - ReverseDutycycle)/ReverseDutycycle) * ReverseOffTime;
  }

  return pwm(arraynumber, 2, // type 2 = pwm, direction, no brake
    PwmPortNumber, PwmPortNumber, // forward and reverse are same
    DirectionPortNumber, 0, // no brake
    ForwardOnTime, ForwardOffTime,
```

```
    ReverseOnTime, ReverseOffTime,
    StartDirection);
}
```

Test it with this:

```
// experi7d.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>

// defines OC structure
#include "outcont.h"

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

enum
{
  MainMotor,
  LED1,
  LED2,
  LASTSLOT
};

void main(void)
{
  int x;
  double dc,oldfreq,newfreq;

  oldfreq = 1193180.0/65536.0;

  get_port(); // get the port number and establish register locations

  // make everthing an output
  set_up_ppi(Aout_CUout_Bout_CLout);

  set_up_new_timer(2000.0);

  SetLast(LASTSLOT);

  newfreq = get_frequency();

  printf("old frequency = %f new frequency = %f Hz\n"
  ,oldfreq,newfreq);

  // set up a channel with a pwm line and a direction line -- no brake
  // using duty cycle
  PwmAndDirectionDuty(MainMotor, 0, // set up, but don't start main motor
```

```
                      PA0, PA2, // pwm and direction port
                      75.0, 50.0); // forward 75%, reverse 50%
  printf("MainMotor on/off settings pwm = PA0, dir = PA2:\n");
  show(MainMotor);

  Blink(LED1, PA1, .3, .2);
  Blink(LED2, PA4, .15, .1);
  SetLast(LASTSLOT);

  while(!kbhit())
  {
    Forward(MainMotor);
    if(kbhit())
      break;
    wait(1);

    Stop(MainMotor);
    if(kbhit())
      break;
    wait(1);

    Reverse(MainMotor);
    if(kbhit())
      break;
    wait(1);

    Stop(MainMotor);
    if(kbhit())
      break;
    wait(1);
  }

  Stop(MainMotor);

  // don't forget to free memory!
  FreeOutputControl();

  // be sure to restore the timer!
  restore_old_timer();
}

// end experi7d.c
```

Click here to download timer.c and here to download timer.h
Click here to download digital.c and here to download digital.h
Click here to download outcont.h
Click here to download constant.h
Click here to download extern.h
Click here to download experi7d.c
(first save your current copies somewhere if you have made changes to them).

There are many other h-bridge configurations available. If anyone comes up with procedures for them, write them up and attach them to an e-mail. I'll be glad to publish them and even give you credit if you like.

Previous: Experiment 6 - More Precise Control Of Motors
Next: Experiment 8 - Digital To Analog Conversion
Problems, comments, ideas? Please e-mail me

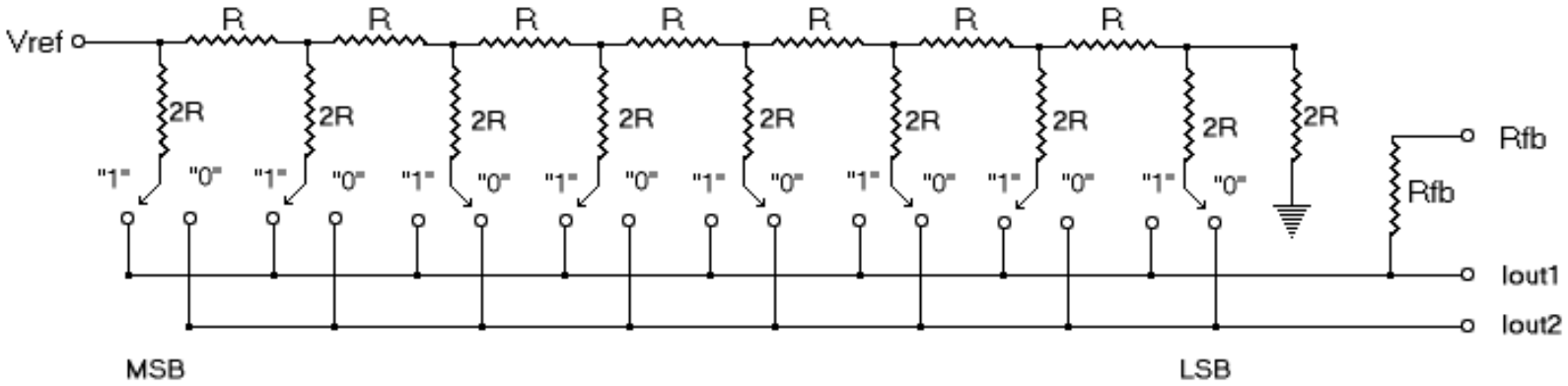
Controlling The Real World With Computers

Experiment 8 - Digital To Analog Conversion

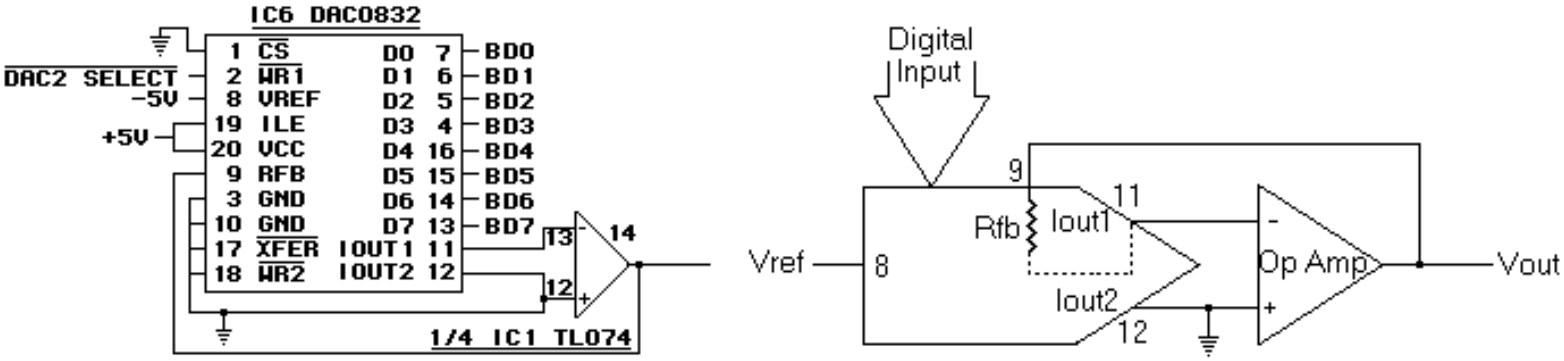
There is frequently a need to convert digital information into a voltage or current. As noted in Data lines, bits, nibbles, bytes, words, binary and HEX, digital data changes in steps. Turning on or off a bit increases or decreases a digital quantity.

The following circuit, derived from the DAC0832 **digital to analog converter** (DAC) datasheet (see DAC0832.PDF), is one approach to making the conversion from stepwise digital information to a voltage. It's called an **R 2R ladder** and is part of the circuit used in the two DAC0832s on the board. R and Rfb are about 15K ohms, which makes 2R about 30K ohms. The actual values are not as important as the fact that the resistors are very closely matched to each other.

The "1" and "0" indicate the positions of MOSFET switches in the converter. A switch will connect to the "1" side if the corresponding bit is on, and to the "0" side if the bit is off. A switch connected to the "1" position will send a portion of Vref-derived current to Iout1, whereas a switch connected to the "0" position will send a portion of the current to Iout2.

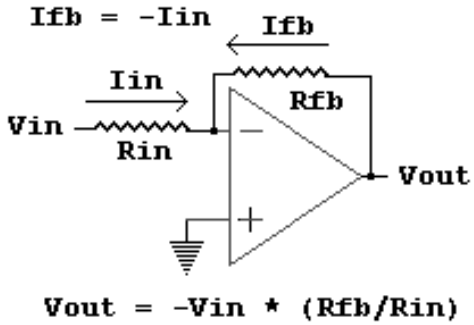


To see how the R 2R ladder fits into the scheme of things, consider the following. The left side is a copy of one of the DAC0832 sections taken from the board's schematic. On the right is a simplified block diagram of the same thing, derived from the datasheet. Rfb is drawn in the manner shown to indicate it is internal to the DAC but can be accessed outside and be connected the Op-Amp:



Iout1 of the R 2R ladder is connected to the inverting input of the Op-Amp (see How To Read A Schematic for more on Op-Amps). Iout2 of the R 2R ladder is connected to the non-inverting input and to ground. One end of the internal Rfb feedback resistor is connected to the output of the external Op-Amp. The other end is internally connected to the R 2R ladder's Iout1 as shown above. Thus, it is connected from the output of the Op-Amp to the inverting input.

Recall that the Op-Amp will attempt to cancel any current through the inverting input. To do so, it will cause the current in the feedback resistor to be equal to the current in the inverting input resistor, but with an inverted polarity:



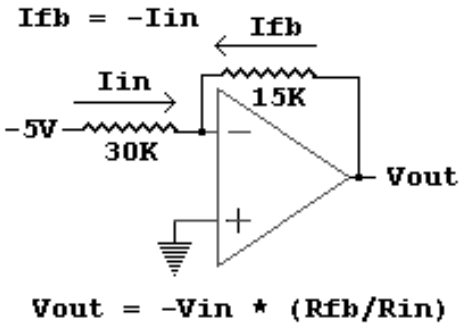
Since the currents are equal but opposite, they cancel each other out, resulting in 0 volts at the inverting input. Thus, the inverting input is at the same potential as ground. This generated ground equivalent is termed a **virtual ground**.

Here is what we know and what can be derived from it:

- Ifb = -Iin
- The voltage on the left side of Rfb is 0 due to the virtual ground, so the voltage on the right is the voltage drop across Rfb, which is Vout = Rfb * Ifb.
- Since Ifb = -Iin, that also means that Vout = Rfb * -Iin

The current being canceled at the inverting input in the block diagram above is Iout1. Thus,
Vout = Rfb * -Iout1. The datasheet says Vout = -(Iout1 * Rfb), which is the same thing.

The board uses -5V from the computer as the reference voltage. Consider the case where the only the MSB is turned on (see Data lines, bits, nibbles, bytes, words, binary and HEX for a definition of MSB). The only resistor connected to Iout1 is the 2R resistor on the left end of the ladder. Since 2R = 30K and Rfb = 15K and Vref = -5V, we now have this:



Now Vout = -(-5V) * (15K /30K) = 2.5V

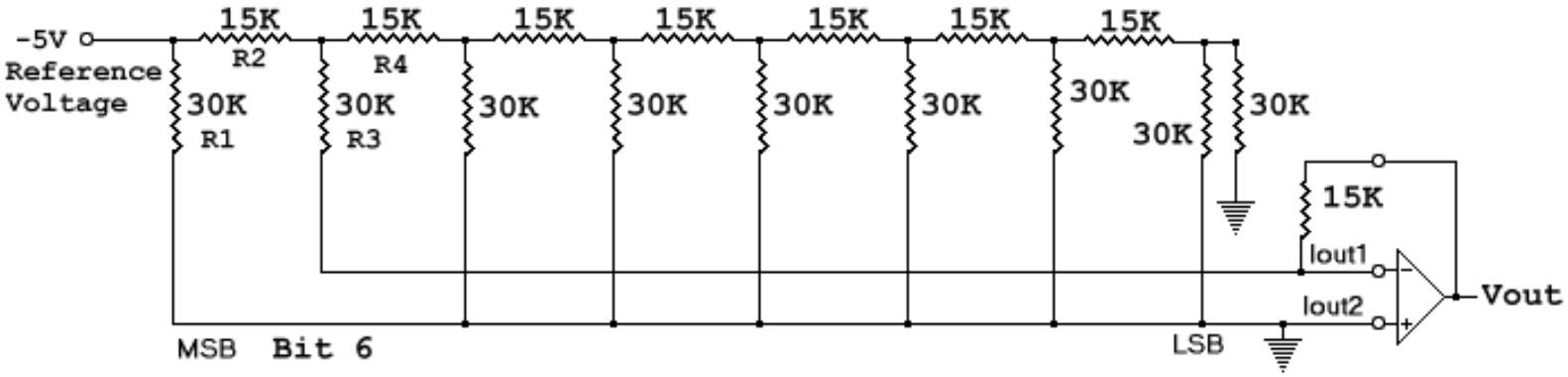
Remember that the MSB has a weight of 128. The data sheet also says that
Vout = -(Vref * (Digital Input)$_{10}$)/256 (the 10 means "base 10 number").
In the case of the MSB input,
Vout = -(-5V * 128)/256 = 2.5V, which agrees with the previous voltage calculations.

Getting rid of the double negatives and generalizing the numbers gives this:
Vout = (5 * DAvalue)/256, where DAvalue is the base 10 number output to the converter.

A value of 1 will provide the voltage per step:
Vout = (5 * 1)/256 = .01953125 volts

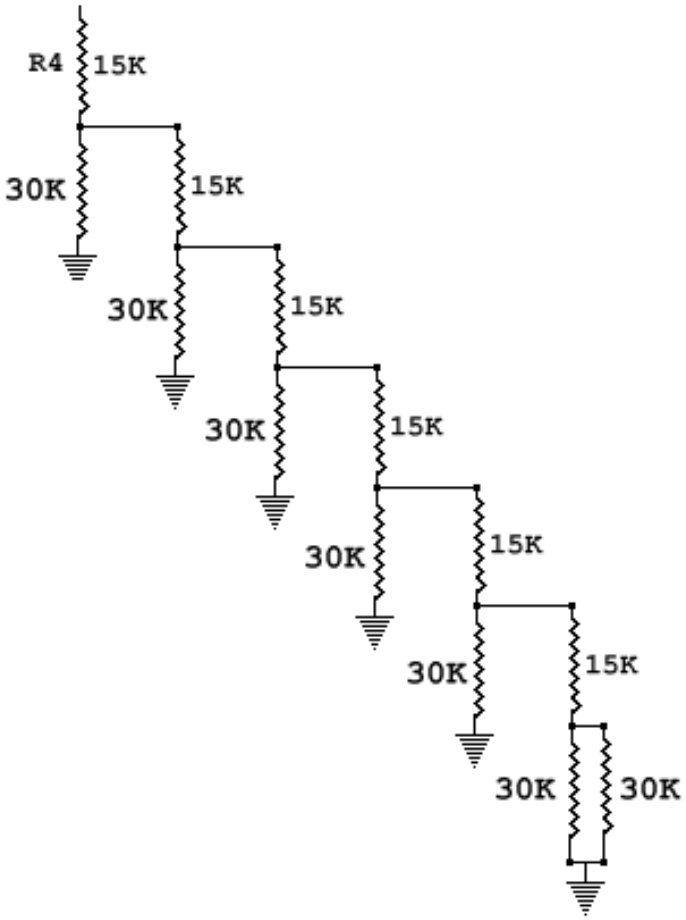
That would be true in a perfect world. There is however, no such thing as perfect in this world, despite the claims of some I have known regarding their own attributes. See the datasheet for a discussion of linearity.

The MSB case is relatively easy. Others require a bit more thought. They are also useful as exercises in a little circuit analysis. For example, consider the case where only bit 6 is high. The bit 6 MOSFET is switched to "1". All of the other bits will be connected to "0", which ends up at ground. The circuit will look like this, including the external Op-Amp and using the nominal resistor values from the datasheet:



The current through R1 does not influence the current referenced at the inverting input. Although one end is connected to the -5V reference, the other end simply goes to ground, so the current through R1 never even makes it to the inverting input.

All of the other resistors do contribute to the current at the inverting input and to the output of the amplifier. The current through R2 splits into two paths. One is through R3 which goes directly to the inverting input, and the other is through the network of R4 and all of the other resistors to ground. The method to determine their composite value is more easily seen if the R4 part of the drawing is reorganized. A careful study will show that only the drawing has changed. The circuit is the same:



First consider the two 30K resistors on the bottom. Recalling the parallel resistor equation from How To Read A Schematic, $R_{parallel}$ = 1/(1/30K + 1/30K) = 15K ohms

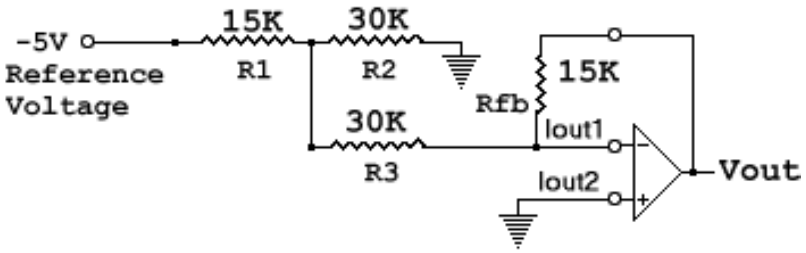
A tip: the parallel value of resistors of the same value is the value of one of the resistors divided by the number of resistors.

The 15K parallel combination adds to the 15K above it since it is in series with it. Again, from How To Read A Schematic, R_{series} = 15K + 15K = 30K ohms

This 30K combination is in parallel with the 30K to the left of it, which produces 15K again which adds to the 15K above to

produce 30K, and so on. The final value of the network is 30K. It would be a very good idea to print the picture, calculate the values and write them down as an exercise to clarify the process. There are other, more sophisticated ways to analyze the circuit, but this is straight forward and will do just fine for now.

The bit-6 circuit ends up looking like this after the resistor equivalents have been identified:



Since the inverting input is a virtual ground, the two 30Ks look like they are paralleled, which means they look like 15K ohms. It is as if R1 is in series with another 15K resistor. Recall from How To Read A Schematic that the equivalent resistance in a series circuit is the sum of the resistors. Thus, the total resistance seen by the reference is 30K ohms. The current through R1 is I = -5/30K.
Half of it goes to ground through R2 and half through R3, making
Iout1 = (-5/30K)/2 = -1/12K.

That makes Vout = -(15K * (-1/12K)) = 1.25 volts.
Also, since bit 6 has a weight of 64 and Vout = (5 * DAvalue)/256 from above,
Vout = (5 * 64)/256 = 1.25 volts.
The two agree.

I'll leave it up to the reader to perform the analysis for other numbers. The next thing to be done here will be to see how the digital to analog converters on the board can provide analog information from digital information through software.

The digital to analog converters are accessed as offsets from the main port address for the board (see Address Lines and Ports and the Hardware Description for more on port locations). The offsets are shown below. Since the analog to digital converter needs eight bytes, each device is 8 bytes from the other:

HEX Offset (add to port address)	Device On Board
0	Eight Channel Analog to Digital Converter
0x08	Digital to Analog Converter 1
0x10	Digital to Analog Converter 2
0x18	Analog to Digital Converter Ready Line And 3 Digital Inputs
0x20	Programmable Peripheral Interface
0x28	Spare Select Line
0x30	Spare Select Line
0x38	Spare Select Line

The digital to analog converters are at port + 0x08 and port + 0x10.

The first thing to be done will be to slightly modify get_port() in digital.c It currently returns nothing. It is modified to return the base port number:

```
// get the port -- this will grow into an auto-detect function in the future
unsigned get_port(void)
{
  base = 0x200; // all switches on -- change as required
  switch_port = base + 0x18;
  ppi_porta = base + 0x20;
  ppi_portb = base + 0x21;
  ppi_portc = base + 0x22;
  return base;

} // end get_port()
```

Click here to download digital.c and here to download digital.h (first save your current copies somewhere if you have made changes to them).

Now all we have to do is get the base port number, then add the offset for the required converter. Try this:

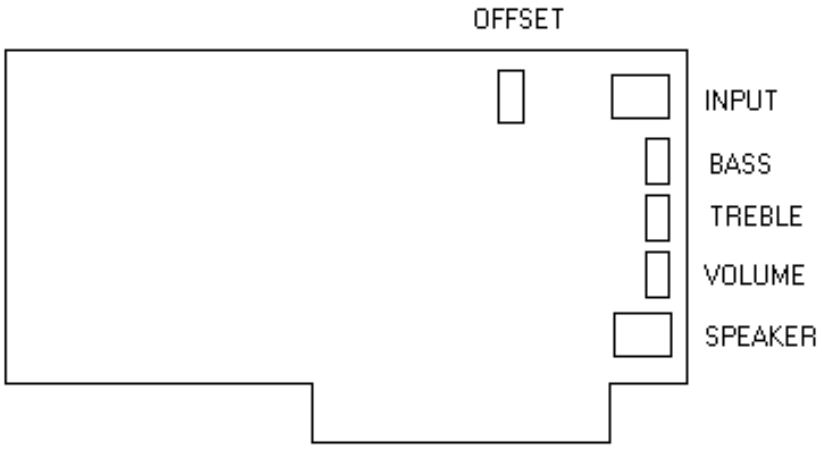
```
 //
experi8a.c

extern unsigned get_port(void);

main(void)
{
  int x;
  unsigned port,da1;

  if(!(port = get_port()))
  {
    printf("No board found\n");
    exit(0);
  }

  da1 = port + 8;

  printf("This sends 0 to 255 then 254 to 1 to DAC1\n");
  printf("as quickly as possible, then repeats.\n");
  printf("Both DACs have their grounds on pin 1 of header 1.\n");
  printf("DAC1's output is on pin 15 and DAC2's on pin 14.\n");
  printf("Press any key to begin, then press any key to end the test.\n\n");
  getch();

  disable();

  while(!kbhit())
  {
    for(x=0; x<256; x++)
    {
      outp(da1,x);
    }

    for(x=254; x>0; x--)
```

```
    {
      outp(da1,x);
    }
  }

  enable();
}

// end experi8a.c
```

Click here to download experi8a.c.

Compile experi8a.c and digital.c, then link them to produce experi8a.exe. Type experi8a <enter> to run it. The program continuously sends 0 through 255 then 254 through 1 to the first digital to analog converter. To send the same thing to the second converter in this and in the following programs, use something like da2 = port + 0x10; in the place of the one for da1 above, then outp(da2,x) rather than outp(da1,x).

The problem with the above program is that you can't see the output without an oscilloscope because it runs too fast. It can be heard on most computers, though. Just connect a speaker or headphones (you will hear one channel with phones) to the board's speaker output and adjust the volume, bass and treble for taste:



The following uses the timer module to provide a pause between each step:

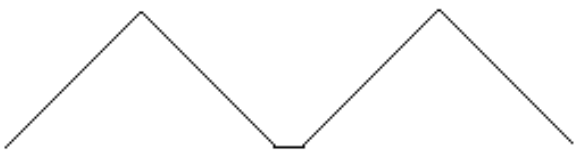
```
// experi8b.c

extern unsigned get_port(void);
extern int set_up_new_timer(double freq);
extern void wait(double seconds);
extern void restore_old_timer(void);

main(void)
{
  int x;
  unsigned port,da1;

  if(!(port = get_port()))
  {
    printf("No board found\n");
    exit(0);
  }
```

```
  da1 = port + 8;

  printf("This sends 0 to 255 then 254 to 1 to DAC1\n");
  printf("with a pause between each step then repeats.\n");
  printf("Both DACs have their grounds on pin 1 of header 1.\n");
  printf("DAC1's output is on pin 15 and DAC2's on pin 14.\n");
  printf("Press any key to begin, then press any key to end the test.\n\n");
  getch();

  set_up_new_timer(2000.0);

  while(!kbhit())
  {
    for(x=0; x<256; x++)
    {
      outp(da1,x);
      if(kbhit())
        break;
      wait(.025); // .025 second per step
    }

    for(x=254; x>0; x--)
    {
      outp(da1,x);
      if(kbhit())
        break;
      wait(.025); // .025 second per step
    }
  }

  restore_old_timer();
}

// end experi8b.c
```

Click here to download timer.c and here to download timer.h
Click here to download digital.c and here to download digital.h
Click here to download experi8b.c
(first save your current copies somewhere if you have made changes to them).

Compile experi8b, digital and timer, then link them to produce experi8b.exe. You will be able to monitor its output on header 1 with a voltmeter rather than an oscilloscope when you then run experi8b. You can pick up an inexpensive voltmeter at Radio Shack and a lot of other places.

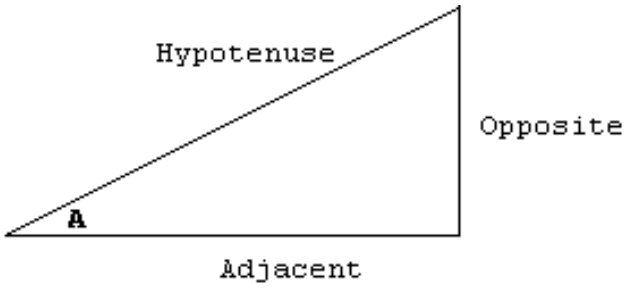
The above programs produce a somewhat distorted triangle wave:



The flat bottom can be seen on an oscilloscope while viewing the faster version. It's most likely a delay caused by the calling of the kbhit() routine. That's somewhat confirmed by the fact that it can't be seen when the slow version is viewed. A scope

with a very slow trace is needed to prove the point, though (.5 second per division).

Probably more common in analysis than the triangle wave is the sine wave. The following shows the relationship of the sides of a triangle and associated trigonometric definitions. Note that it's common to abbreviate sine, cosine and tangent. A sine wave would be a plot of the sine of a changing angle A on a graph:

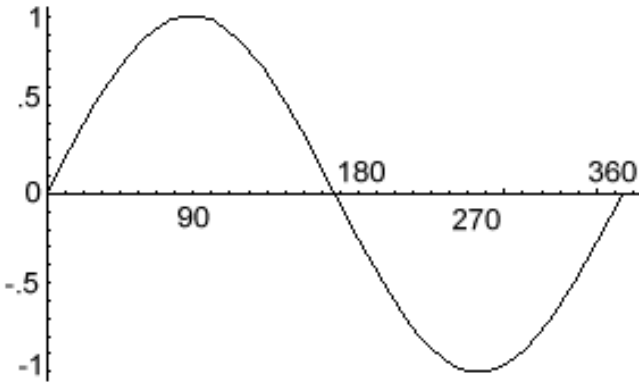



```
sin A = Opposite / Hypotenuse
cos A = Adjacent / Hypotenuse
tan A = Opposite / Adjacent
```

If you remember sine, cosine and tangent, in that order, and opposite, adjacent and hypotenuse, in that order, you can remember everything else:

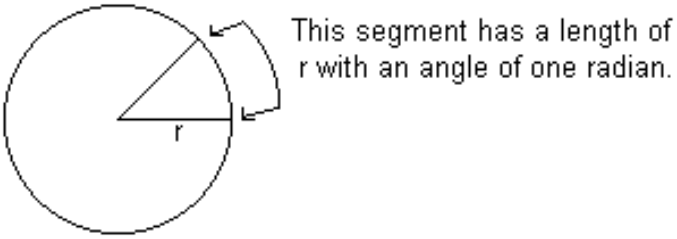
1. sin and opposite are first: sin = opposite/hypotenuse
2. cos and adjacent are next: cos = adjacent/hypotenuse
3. tan uses first and second: tan = opposite/adjacent

See Trigonometry and Right Triangles on the Zona Land site for more detailed information on trig functions. A graph of a sine wave is below. The sine of 0, 180 and 360 degrees is 0. The sine of 90 degrees is 1 and the sine of 270 degrees is -1.



Most C compilers use **radian** calculations to find trigonometric values such as sine. Most people know that the circumference of a circle is pi (about 3.141592654) times the diameter of the circle:
C = pi * D
Half of the diameter is called the radius,
D = 2r
so the circumference is also
C = pi * 2r
or, the same thing,
C = 2pi * r

Sweeping through an angle of one **radian** will trace an arc on a circle with the arc's segment length equal to the length of the circle's radius:



Since a radian moves through a segment equal to the radius, 2pi radians will trace the full circumference. There are 2pi radians in a circle. Here are a few radian to degree relationships:

Radians	Degrees	sin	cos	tan
0	0	0	1	0
.5 pi	90	1	0	undefined -- adjacent = 0
pi	180	0	-1	0
1.5 pi	270	-1	0	undefined -- adjacent = 0
2 pi	360	0	1	0

See more detailed information about radians on the Zona Land site.

The purpose of this discussion is to introduce a method of generating tables that can be used to produce sine waves. The following program will generate a header file with a table. Several procedures are new and will be briefly covered here. See your compiler manual or help system for more detail.

First consider the **FILE *out_file;** declaration. This is a structure declaration that sets up a pointer to a FILE structure (see Experiment 3 and subsequent sections for a discussion of pointers). The pointer is usually called a file pointer, and is commonly used to work with text files. A file is opened using **fopen(..)**. The prototype is:
FILE *fopen(char *filename, char *access);
The filename argument is a string holding the name of the file. The access string argument tells the routine how to open the file. Probably the most common ones are "r" for read, "w" for write and "a" for appending to a file. Since the object here is to write to the file, "w" is used below.
The flush() and close() calls make sure everything has been written to the file and closes it.

There are some functions that will provide the angle in radians if the value of the trig type is known. The call to **acos()** in the program returns the angle for an angle that has a cosine of -1. That turns out to be pi from the table above, and 2 times the value will be 2pi. It's called pi2 here for reasons outlined in Putting It All Together - Controlling The Hardware With The Software.

The puts() routine was used in Experiment 3. A call to **fputs()** does about the same thing for a file; it puts pre-defined text in the file. Formatted text is handled here with **fprintf(..)**. It works like printf(..), first described in Putting It Together, but with the addition of a file pointer.

The n variable is used in a for() loop (yup, Putting It Together) to count two cycles of a sine wave. A loop inside that one is used to run 0 through 2pi radians with a step of StepSize. In this case, StepSize is pi2/255. It can be changed for higher or lower frequencies.

Since the sine ranges from -1 to 1 and the digital to analog converter needs 0 to 255, some modification is needed. First 1 is added, which gives 0 to 2. Then the results of that are divided by that 2, giving 0 to 1. Finally, it's multiplied by 255 to give 0 to 255 (the call to sprintf with %1.0f causes a rounding of the floating point number).

The results are formated to fit on a page, then put in a file called slosin.h:

```
// makslo.c

#include <string.h>
#include <graphics.h>
#include <stdio.h>
#include <string.h>
#include <bios.h>
#include <dos.h>
#include <math.h>
#include <stdlib.h>

void main(void)
{
  int x,n;
  char temp[300];
  FILE *out_file;
  double StepSize,angle,pi2,sn,out,out2;

  pi2 = acos(-1.0) * 2.0;
  StepSize = pi2/255.0;

  out_file = fopen("slosin.h","w"); // store in slosin.h

  fputs("unsigned char sin_table[] = {",out_file);

  printf("unsigned char sin_table[] = {");

  for(n=0,x=0; n<2; n++)
  {
    for(angle=0.0; angle<=pi2; angle+=StepSize,x++)
    {
      if(x)
      {
        if(!(x % 10))
        {
          fputs("\n                              ,",out_file);
          printf("\n                               ,");
        }

        else
        {
          fputs(",",out_file);
          printf(",");
        }
      }

      sn = sin(angle);
      out = (sn + 1.0)/2.0;
      sprintf(temp,"%1.0f",255.0*out);
      out2 = atof(temp);
      fprintf(out_file,"%3d",(int)out2);
      printf("%3d",(int)out2);
```

```
    }
  }

  fprintf(out_file,"}; /* %d */\n",x);
  printf("}; /* %d */\n",x);
  fprintf(out_file,"\nint last = %d;\n",x);
  printf("last = %d\n",x);
  fprintf(out_file,"double StepSize = %f;\n",StepSize);
  printf("StepSize = %f\n",StepSize);
  printf("data send to slosin.h\n");
  fflush(out_file);
  fclose(out_file);
}

// makslo.c
```

Click here to download makslo.c.

The table thus generated can be used to generate a sine wave using the following:

```
// experi8c.c

#include "slosin.h"

extern unsigned get_port(void);

main(void)
{
  int x;
  unsigned port,da1;

  if(!(port = get_port()))
  {
    printf("No board found\n");
    exit(0);
  }

  da1 = port + 8;  // da2 = port + 0x10;

  printf("This puts a slow sinwave on DAC1.\n");
  printf("Both DACs have their grounds on pin 1 of header 1.\n");
  printf("DAC1's output is on pin 15 and DAC2's on pin 14.\n");
  printf("Press any key to begin, then press any key to end the test.\n\n");
  getch();

  disable();

  while(!kbhit())
  {
    for(x=0; x<last; x++)
    {
      if(kbhit())
```

```
        exit(0);

      outp(da1,slo_sin_table[x]);
    }
  }
  outp(da1,0);
  enable();
}

// experi8c.c
```

Click here to download experi8c.c.

While this sine wave looks good on an oscilloscope, it might not be heard through the speaker because it's too low in frequency. A faster wave is needed. To get it, use a larger step size. The makefast program gets StepSize by dividing pi2 by 16 rather than 255 and sends the data to a file called fast_sin.h:

```
 // makfast.c

#include <string.h>
#include <graphics.h>
#include <stdio.h>
#include <string.h>
#include <bios.h>
#include <dos.h>
#include <math.h>
#include <stdlib.h>

void main(void)
{
  int x,n;
  char temp[300];
  FILE *out_file;
  double StepSize,angle,pi2,sn,out,out2;

  pi2 = acos(-1.0) * 2.0;
  StepSize = pi2/16.0;

  out_file = fopen("fast_sin.h","w");

  fputs("unsigned char sin_table[] = {",out_file);

  printf("unsigned char sin_table[] = {");

  for(n=0,x=0; n<10; n++)
  {
    for(angle=0.0; angle<=pi2; angle+=StepSize,x++)
    {
      if(x)
      {
        if(!(x % 10))
        {
          fputs("\n                              ,",out_file);
```

```
          printf("\n                                ,");
        }

        else
        {
          fputs(",",out_file);
          printf(",");
        }
      }
      sn = sin(angle);
      out = (sn + 1.0)/2.0;
      sprintf(temp,"%1.0f",255.0*out);
      out2 = atof(temp);
      fprintf(out_file,"%3d",(int)out2);
      printf("%3d",(int)out2);
    }
  }

  fprintf(out_file,"}; /* %d */\n",x);
  printf("}; /* %d */\n",x);
  fprintf(out_file,"\nint last = %d;\n",x);
  printf("last = %d\n",x);
  fprintf(out_file,"double StepSize = %f;\n",StepSize);
  printf("StepSize = %f\n",StepSize);
  printf("data send to fast_sin.h\n");
  fflush(out_file);
  fclose(out_file);
}

// makfast.c
```

Click here to download makfast.c.

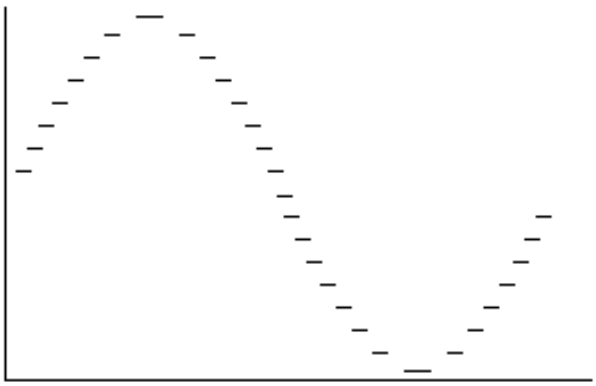
Now run it using experi8d.c below. You should be able to hear it on the speaker. It won't sound too clean because of the step size. You should be able to see the steps on an oscilloscope:

```
 // experi8d.c

#include "fast_sin.h"

extern unsigned get_port(void);

main(void)
{
  int x;
  unsigned port,da1;

  if(!(port = get_port()))
  {
    printf("No board found\n");
    exit(0);
  }
```

```
  da1 = port + 8;

  printf("This puts a faster sinwave on DAC1.\n");
  printf("Both DACs have their grounds on pin 1 of header 1.\n");
  printf("DAC1's output is on pin 15 and DAC2's on pin 14.\n");
  printf("Press any key to begin, then press any key to end the test.\n\n");
  getch();

  disable();

  while(!kbhit())
  {
    for(x=0; x<last; x++)
    {
      if(kbhit())
        exit(0);

      outp(da1,sin_table[x]);
    }
  }
  outp(da1,0);
  enable();
}

// experi8d.c
```

Click here to download experi8d.c.

While the sound can be heard, it is distorted because of the step size. This is what it looks like on a scope:



A faster means of delivering the signal is needed in order to get a reasonably smooth signal out fast enough to be audible (unless your computer is much faster than my old test computer). That will be covered in another section since it requires a discussion of assembly code.

Meanwhile, you might want to download the free software on my Super Start site. It can use the board, and the zip file includes source with the assembly code. As a bonus, Super Start is designed to help very young children get the core knowledge needed for a good beginning in education.

Previous: Experiment 7 - Bi-directional Control Of Motors And The H-Bridge
Next: Experiment 9 - Analog To Digital Conversion
Problems, comments, ideas? Please e-mail me

Controlling The Real World With Computers

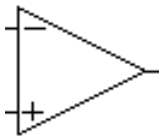
Experiment 9 - Analog To Digital Conversion

The compliment to the digital to analog converter is the **analog to digital converter**. An analog to digital converter converts analog voltages to digital information that can be used by a computer.

It is useful to note that the digital data produced by an analog to digital converter is only approximately proportional to the analog input. That's because a perfect conversion is impossible due to the fact that digital information changes in steps, whereas analog is virtually continuous, at least down to the subatomic level.

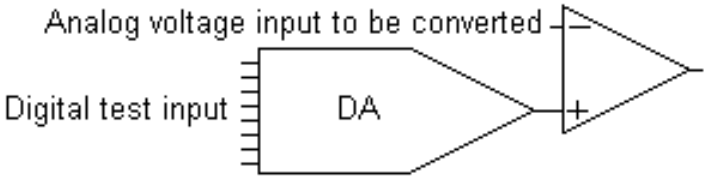
A good example of the limitation is the converter on the board. It provides 8-bit numbers. That means it can produce the numbers 0 through 255. That's 256 numbers and 255 steps. There are simply not enough steps to represent all possible analog input values. The situation improves as the number of bits increases. A 20-bit converter gets a lot closer to real-world with 1,048,575 steps. It's important to remember however, that there are infinitely more analog values possible between a single step of any analog to digital converter than can be represented over the converter's entire span of digital values. Human-made will never match real-world.

One important element in analog to digital converters such as the ADC0809 on the board (see ADC0809.PDF) is the **analog comparator**. It looks like an op-amp schematically:

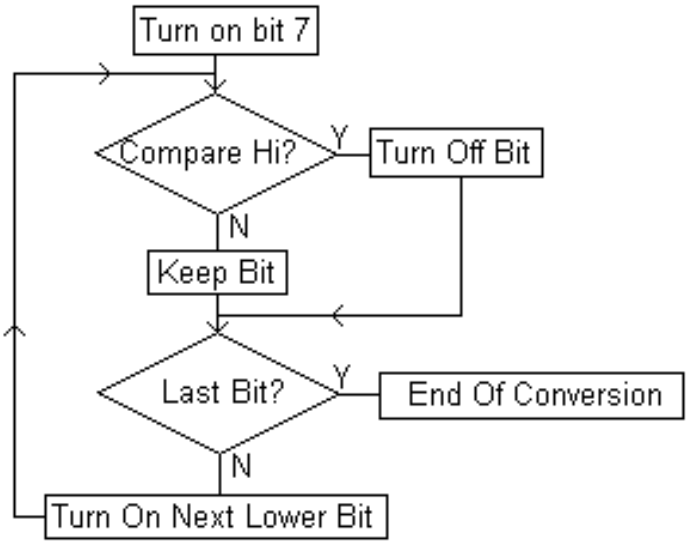


The difference is that it accepts analog inputs but produces a digital output. Its output will be high if the + analog level is greater than the - analog level, else its output will be low.

Another important component in the ADC0809 is a digital to analog converter (DA) similar to the one discussed in Experiment 8. The digital representation of the input can be determined by connecting the analog voltage input to be converted to the - input of the comparater, and the output of the DA to the + input:



The DA is initialized by storing only the high-order bit in it. That's bit 7 for the 8-bit converter on the board. If the comparator output is low, that means the test analog level from the DA is still lower than the input, so the bit is kept. If the comparator output is high, that means the test analog level from the DA is too high, so the bit is dropped. The other bits are tested in the same manner and either kept or dropped according to the status of the comparator. The process is called **successive approximation**, and is illustrated by the following **flow chart**:



Incidentally, a flow chart is a handy, easy way to describe a process before typing in the code. There are more geometric shapes in a full-blown flow chart, but diamonds for decisions and rectangles for actions can be used in most cases. One useful addition is the circle. Circles with letters in them are used to show transitions to other pages.

The flow chart above follows the written description of successive approximation. The DA is initialized by turning on bit 7. The process then moves to the decision diamond which checks the comparater. The bit being tested is turned off if the comparator output is high, else it's left on. The process then moves to another decision which asks if this is the last bit. If not, the next lower bit is turned on and control is passed to the top decision. The conversion ends after bit 0, the last bit, has been tested.

Consider the task of providing a number for an input of 3.21 volts. It will be assumed that the digital to analog converter provides a voltage and that the comparator compares voltages. An actual successive approximation converter could just as easily use current.

Recall from Experiment 8 that the output voltage of the digital to analog converter on the board is Vout = (5 * DAvalue)/256. This equation will be used to illustrate the process.

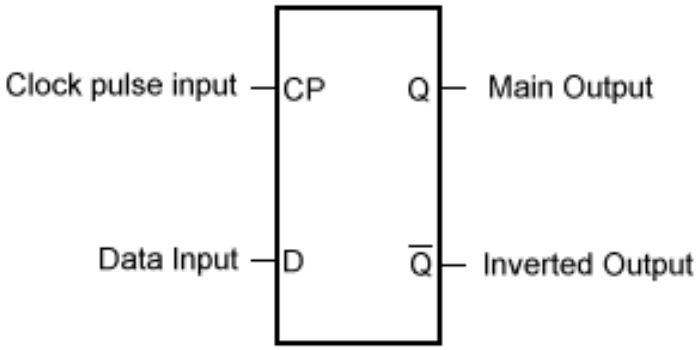
The flow chart says to start with bit 7 which has a weight of 128, then keep or drop bits according to the output of the comparator. The process used to convert 3.21 volts to a number is shown in the following table:

Test Bit	DA Binary Value	Decimal	(5 * DAvalue)/256	Comparison Result
10000000	10000000	128	2.5	low - keep bit
01000000	11000000	192	3.75	high - drop bit
00100000	10100000	160	3.125	low - keep bit
00010000	10110000	176	3.4375	high - drop bit
00001000	10101000	168	3.28125	high - drop bit
00000100	10100100	164	3.203125	low - keep bit
00000010	10100110	166	3.2421875	high - drop bit
00000001	10100101	165	3.22265625	high - drop bit

Three bits were kept, leaving 10100100 in the DA converter. The analog to digital converter will provide 164 as its numerical representation for 3.21 volts.

The ADC needs a clock to run. This is provided by dividing the 14.31818 MHz oscillator (OSC) signal on the ISA buss by 16 using a 74LS393 (see 74LS393.PDF). (Hz means Hertz, named after physicist, Gustav Ludwig Hertz. It means cycles per second, and MHz means megaHertz or millions of cycles per second.)

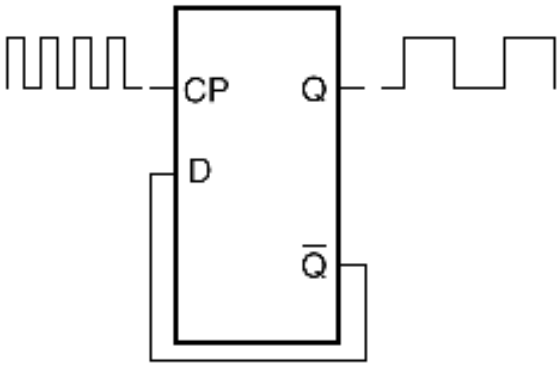
The 74LS393 is a dual, 4-bit **ripple counter**. A ripple counter uses a **flip-flop** as its basic element. The following is a simplified diagram of a single flip-flop. This one is called a D flip-flop:



The operation of a D flip-flop is very simple (see 74LS74.PDF for more detail on a typical D flip-flop).

If the data input (D) is high and a clock pulse is applied to the clock input, called **clocking** the flip-flop, the main Q output will go high and the inverted output (NOT Q) will go low. If the data input is low and the flip-flop is clocked, Q will go low and NOT Q will go high.

In other words, Q becomes the same state as the data input when the flip-flop is clocked, and NOT Q becomes the opposite state of the data input. Now notice what happens if NOT Q is tied to the data input:



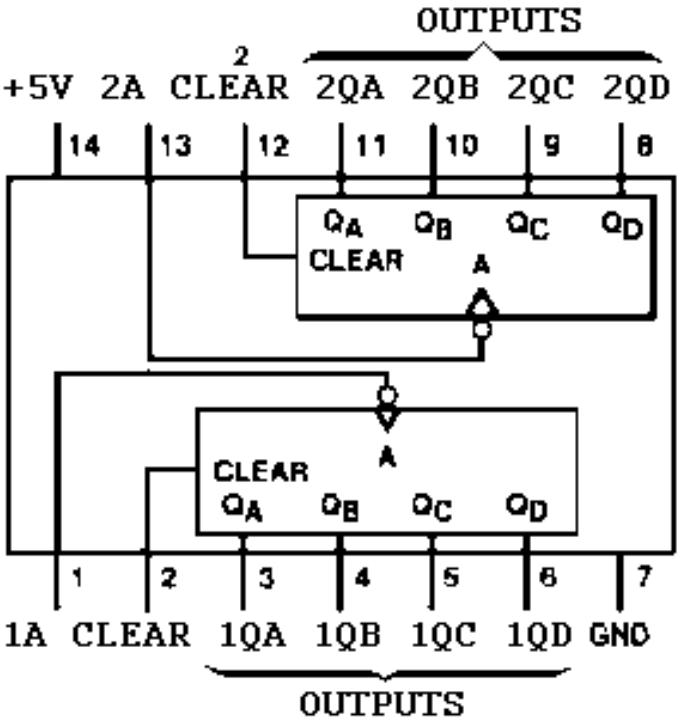
This flip-flop clocks on the falling edge of the pulse (as does the 74LS393). That means the Q and NOT Q outputs change only on the falling edge of the input. One cycle of either the input or output is from falling edge to falling edge. Consider the following sequence of events:

- Let's say NOT Q starts out high, making the data input high.
- Q is always the opposite of NOT Q, so Q will be low.
- The falling edge of the input clocks the flip-flop.
- Q will become the same state as the data input. Thus, it will go high.
- NOT Q is always the opposite of Q, so it will go low, making the data input low.
- The next falling edge of the input clocks the flip-flop again.
- Q will become the same state as the data input, thus Q will go low.
- NOT Q will go high, along with data.

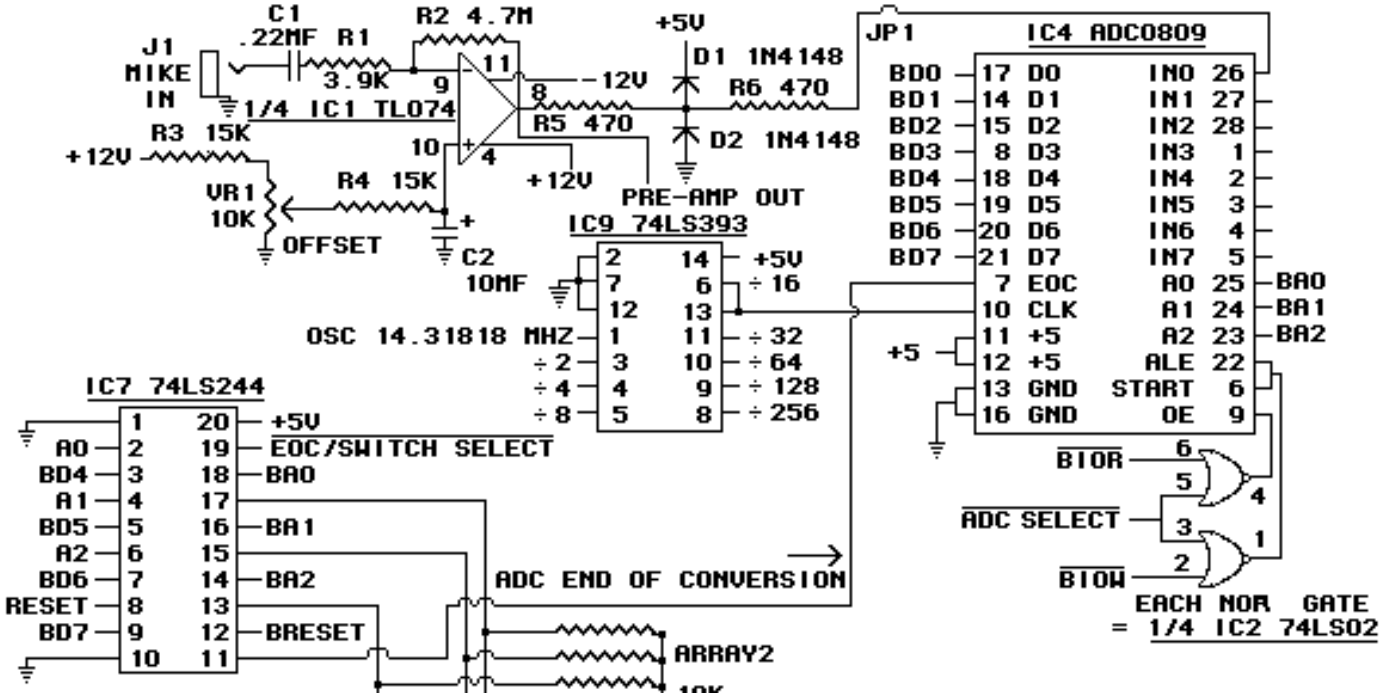
The output completes only a half cycle with each full cycle of the input. The result is a divide by two, or binary divide. Connect several of these stages together and the result is a counter that can divide an input pulse by any desired binary number (2, 4, 8, 16, etc). The flip-flop action will **ripple** through the stages.

A schematic from the data sheet of the 74LS393 is shown below. 1A and 2A are the clock inputs. They are triggered on the falling edge of the clock, as indicated by the bubble on their inputs. The clear inputs make all of the outputs

low. There are 4 Q outputs for each group of 4 flip-flops. The sections are connected to each other on the board to provide 8 outputs on Header 4 as 8 different frequencies of pulsed wave forms. The ADC0809 analog to digital converter's clock input is connected to pins 6 and 13, where the stages are connected together and where the input is divided by 16.

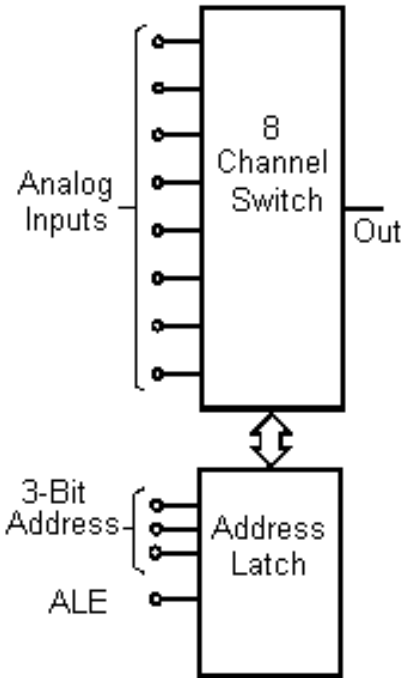


Dividing 14.31818 MHz by 16 produces a clock of 894886.25Hz which permits about 11000 samples per second from the ADC0809, which is fast enough to remove the need for an anti-aliasing filter when recording signals up to about 5kHz. (Click here to see a definition of aliasing.) The portion of the schematic showing the 74LS393 clock section connected to the converter is shown below:



Two gates of the 74LS02 quad NOR gate (see 74LS02.PDF) in the above decode the read and write select for the ADC0809 analog to digital converter. One of the converter's 8 channels is selected using buffered address lines BA0,

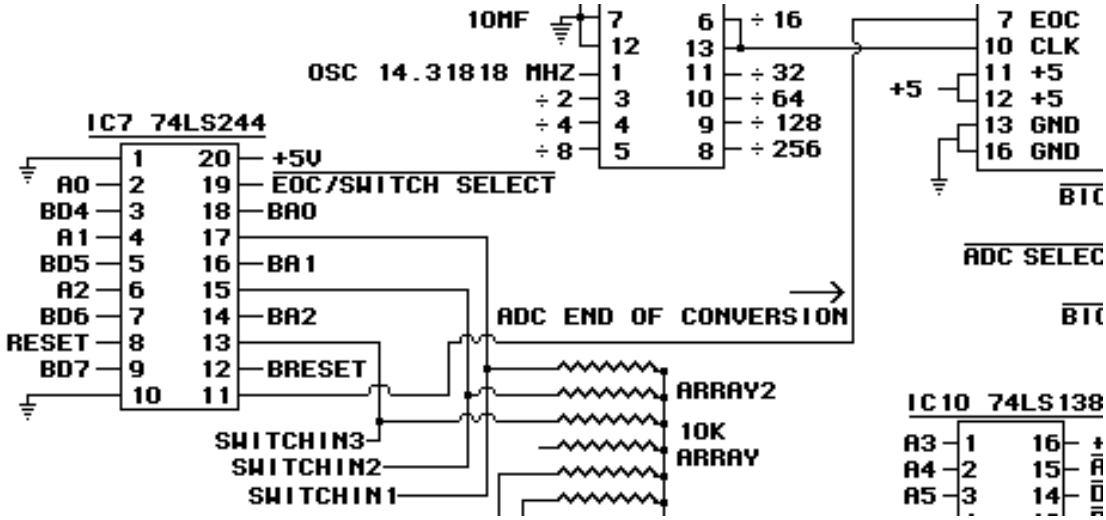
BA1 and BA2, which activates a switch array called a **multiplexer**. The Address Latch Enable (ALE) input on the ADC0809 must be high to cause the converter to lock in the selected channel for conversion:



Taking the START line high will start a conversion. ALE and START are connected together to simultaneously latch in the selected channel and start a conversion. A write port operation is used to select a channel and start a conversion. The ADC SELECT line from the 74LS138 will go low when the converter is selected. Note that when the converter is not selected, ADC SELECT will be high, forcing the outputs of both NOR gates low (see the Boolean logic section if you have forgotten the logic).

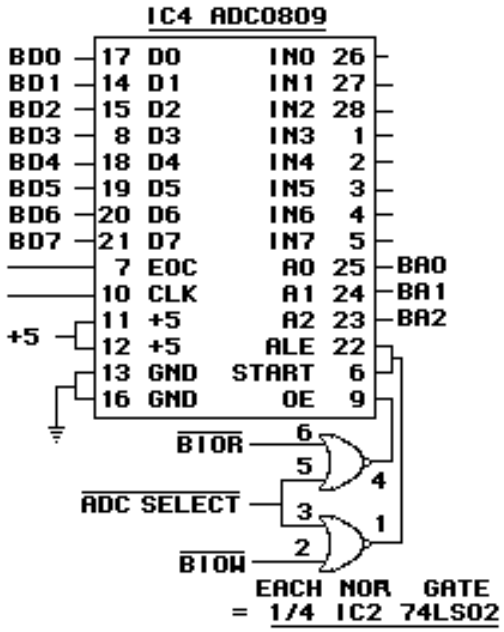
When both ADC SELECT and the BIOW (Buffered I/O Write - Active Low) lines are low, the output of the bottom NOR gate will go high. That will make the Address Latch Enable line high which will latch in the channel selected by BA0, BA1 and BA2. At the same time, the START line will go high, which will cause a conversion to start for the channel selected by the address lines.

The ADC0809 signals an End Of Conversion by taking its EOC line (pin 7) high. The EOC signal is available from the 74LS244 (see 74LS244.PDF), which is the same chip used in Experiment 1 to get the status of the three basic switch inputs:



The EOC signal can be read though the BD7 line on the 74LS244. Once the converter indicates data is available, the Output Enable (OE) line on the ADC0809 can be made high to cause the results of the conversion to be placed on the

computer's data buss:



Output Enable is made high by making both ADC SELECT and BIOR (Buffered I/O Read - Active Low) low. This will cause the output of the upper NOR gate to go high, making OE high. Data will then be place on BD0 through BD7 and can be read by a program. The data will be for the channel selected when the conversion was started as described above.

To get a conversion from a channel, write anything to the base address plus the channel number, wait for the End Of Conversion line to go high, then read the converter. The channel numbers and locations, address offsets and start instructions for the analog to digital converter channels are:

To start a conversion on this channel	Which is ADC0809 input	And Header 1 input pin (ground on 1 or 2)	Write anything to
Channel 0	26 = IN0	13 with JP1 removed, or through the preamp with JP1 in place	Base Plus 0
Channel 1	27 = IN1	12	Base Plus 1
Channel 2	28 = IN2	11	Base Plus 2
Channel 3	1 = IN3	10	Base Plus 3
Channel 4	2 = IN4	9	Base Plus 4
Channel 5	3 = IN5	8	Base Plus 5
Channel 6	4 = IN6	7	Base Plus 6
Channel 7	5 = IN7	6	Base Plus 7

What has been covered about the analog to digital converter so far can be used to automatically determine the base address being used for the board. Simply run through all of the possible base addresses, try to start a conversion for channel 0 of that base address, then look for the end of conversion signal. If the end of conversion signal is low then high, then there is a very good possibility that the correct channel number has been found. The following changes

get_port() to an auto-detect function:

```
// find hardware port if one exists
unsigned get_port(void)
{
  int x;
  static unsigned local_port;
  unsigned int not_ready_count,ready_count;

  if(local_port == 32767)
    return 0;

  if(local_port > 0)
    return local_port;

  for(x=0x200; x<0x3c0; x+=0x40)
  {
    not_ready_count = 32767;
    ready_count = 32767;

    outp(x,0); /* start conversion */

    while((inp(x+0x18) & 0x80) && --not_ready_count); /* wait for not ready */

    while(!(inp(x+0x18) & 0x80) && --ready_count); /* wait for ready */

    if(ready_count < 32767 && ready_count > 0)
    {
      local_port = base = x;
      switch_port = base + 0x18;
      ppi_porta = base + 0x20;
      ppi_portb = base + 0x21;
      ppi_portc = base + 0x22;
      return base;
    }
  }

  local_port = 32767;
  return 0;
}
```

The static variable **local_port** is used to determine if a search has already been done. A new search is conducted if local_port is 0. A value of 32767 (a little less than the maximum positive integer value [short integer for 32 bit compilers]) indicates an attempt has already been made and a board could not be found, so 0 is returned to show there is no board.

The search runs through all but the highest possible setting on the DIP switch. The highest setting is not used since that location is used for ECG displays, and many modern displays still provide compatibility registers at that location. The x variable is used in a loop to run through all of the base addresses using the loop
for(x=0x200; x<0x3c0; x+=0x40)
Recall that there are 64 bytes between the base addresses; 0x40 = 64.

The **outp(x,0)** instruction attempts to start a conversion on channel 0 at the selected base address by writing a 0 to the location. The next line looks like this:
while((inp(x+0x18) & 0x80) && --not_ready_count);
This is a single-line loop that gets a port input from the base location to be tested (**x**) plus 0x18, which is the offset of the 74LS244. This information is ANDed with 0x80, which is the weight of bit 7, the location of EOC. If EOC is low, the loop will break because the left half of the decision returns false.

The right half of the decision checks to see if **not_ready_count** has decremented to zero. If a converter is at the address and has been started, EOC will initially go low. This is contrasted with port addresses with nothing on them which will generally return 0xff, indicating all of the bits are on. Thus, **not_ready_count** is set up with a count that times out if EOC never goes low. Its purpose is to cause a break out of the loop even if EOC remains high.

Notice the "--" in front of not_ready_count. This is a **pre-decrement unary operator**. In the past, we have used only the post-decrement operator, which has the "--" after the variable name. The same two forms can be used with "++" and other operators. Here, the variable is decremented first, then tested for 0. In the case of a post-decrement operator, the variable is tested then decremented.

Looking at the operation another way:

```
 stay    the input from     ANDed with  AND  the previously decremented
here as  the port plus      the weight   |   counter is still greater
long as  0x18 which is      of bit 7     |   than zero
   |     the 74LS244        is high,     |              |
   |        |                  |        |              |
while(   (inp(x+0x18)       & 0x80)     &&     --not_ready_count);
```

The colon at the end of the line causes the loop to stay on the line until the EOC goes high or the counter counts down to zero. The counter is for protection. It's possible that the EOC line will go low then high again before the program gets to the while loop in a slow computer. After all, the converter can do its job in only about 1/11000 th second. It's also possible that a very fast computer could get to the loop line before EOC has had a chance to go low. If the loop is entered with EOC high, then the loop might run forever without the protection of the counter. It is advisable when writing bit detection loops to always have an escape. The above might not always be the best since the timeout depends on the speed of the computer, but it has worked on many computers thus far.

The next line waits for the EOC line to go high:

```
while(!(inp(x+0x18) & 0x80) && --ready_count); // wait for ready
```

The counter **ready_count** is used to judge how long EOC stays high once it turns on.

The most important difference between this line and the previous is that this line loops as long as EOC is **low**, as indicated by the exclamation mark, whereas the previous line looped while the line was **high** (neglecting the escape counters).

Both lines taken together cause the program to first wait until EOC goes low, then wait until it goes high again.

Notice what will happen to ready_count in the second line. The loop has timed out if ready_count decrements to zero. At the same time, since it takes a finite amount of time for EOC to go high, ready_count should be less than its original 32767. The ready_count variable should be less than 32767 but greater than 0 if the bit goes from low to high. Thus, the tested port number is kept and other variables set if the following condition is true:

if(ready_count < 32767 && ready_count > 0)

Click here to download digital.c.

Remember to save your copy first to protect it. The original get_port() in digital.c is renamed to oldget_port() just in case it's needed. Recompile digital.c to form an object module.

Use the following to test the new get_port(). Be sure to save whatport.c somewhere if you have downloaded it from Super Start (www.superstart.org):

```
// whatport.c

#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <time.h>
#include <dos.h>
#include <stdlib.h>
#include <string.h>

extern unsigned get_port(void);

main()
{
  unsigned port = 0;

  port = get_port();

  if(port == 0)
    printf("no hardware found\n");

  else printf("port after get_port = %X\n",port);
}

// end whatport.c
```

Click here to download whatport.c. Again, be sure to save whatport.c somewhere if you have downloaded it from Super Start (www.superstart.org):

Compile whatport.c then link it with the digital object module to form whatport.exe. Type whatport <enter> to run it. Try turning on different combinations of DIP switches then running whatport to confirm that the new settings are properly detected. It would be a good idea not to turn off all three switches so as not to interfere with video cards with ECG compatibility.

Part of the table from the hardware page is repeated below for reference. Notice the 0x40 step size:

Switch Settings	
1 2 3	Base Address
1 1 1	200
0 1 1	240
1 0 1	280

0 0 1	2C0
1 1 0	300
0 1 0	340
1 0 0	380
0 0 0	3C0 (not tested)

Getting analog data is easy once the port is known. Just write anything to the port plus the channel number, wait for EOC to go high, then read the port number (not plus the channel). The following continuously reads channel 0:

```
// experi9a.c

#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <time.h>
#include <dos.h>
#include <stdlib.h>
#include <string.h>

extern unsigned get_port(void);

main()
{
  unsigned port;
  int x;

  port = get_port();

  if(port == 0)
  {
    printf("no hardware found\n");
    return;
  }

  printf("port after get_port = %X\n",port);

  while(!kbhit())
  {
    outp(port, 0); // start channel 0 conversion
    while(!(inp(port+0x18) & 0x80)); // wait for ready
    x = inp(port);
    printf("%4d",x);
  }
}

// experi9a.c
```

Click here to download experi9a.c.

Compile experi9a.c to make an object file then link it with digital to make experi9a.exe. To run it, just type in experi9a <enter>.

The values from the analog to digital converter's channel 0 will print on the screen until a key is pressed. Try turning the multi-turn offset trimmer. You are adjusting the offset of the microphone preamp and should see the numbers change. If they don't change, make sure JP1 is in place. Adjust the trimmer so the program reads about 127 or 128. That will put the preamp offset at about 2.5 volts, or about half of the converter's 5 volt range.

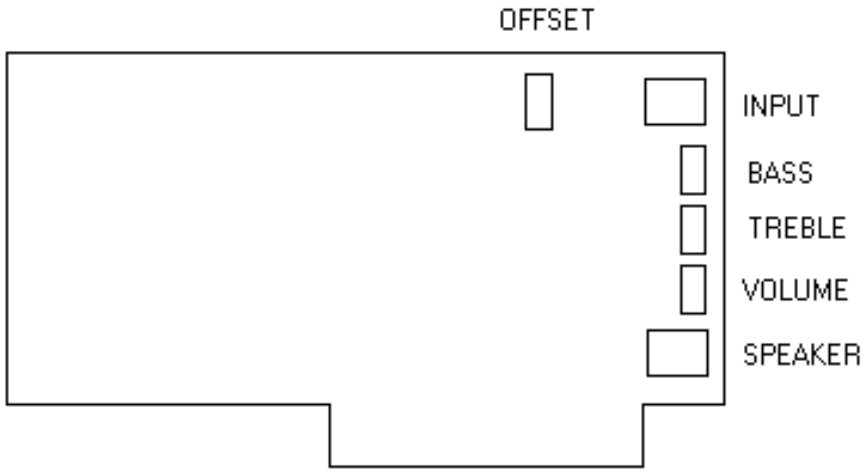


Figure 4: Controls, IN and OUT

Plug in a mike and talk into it. If you talk loud enough, you should see the numbers range from 0 to 255. The rate at which the converter is accessed is not fast enough to accurately record a voice, but some of the changing values can be seen. The rate is slowed most by the printing. It takes a long time to print - much longer than can be spared if we want to record something with frequencies as high as the human voice (about 3kHz).

The following program is capable of recording and playing back voice. The key is to remove everything that needlessly takes up time. Notice, for example, that the EOC location in the above is referenced by the inp(port+0x18) instruction. The problem is that the location is re-caluculated each iteration of the loop, even though it is a constant value. **To save processing time, pre-calculate all constant values used in a loop.** The port+0x18 term is placed into the eoc variable in the following program. It is calculated only once rather than with each iteration of a loop.

All that is done in the record operation is to record. There is no printing or playing back of data. A block of data is recorded, then the same block is played back. The data[] array is used to hold the information. It is first loaded with information from the analog to digital converter in a manner similar to experi9a. Interrupts are first disabled so they will not interfere. A conversion is started on channel 0 for each of 30000 bytes. A data byte is obtained from the converter when a high EOC is detected and placed in the appropriate array position.

Interrupts are enabled again after all 30000 bytes have been recorded, but only long enough to indicate that playback is about to begin. Playback looks a lot like record. A conversion is started on channel 0 and a high EOC is waited for. This dummy conversion is performed in order to provide playback with about the same timing as record. The appropriate data byte from the array is sent to the digital to analog converter when a high EOC is detected.

The program will first record then play back about 3 seconds of data. Pressing any key will terminate the program, although it can take up to 6 seconds since the program must first enable interrupts before the computer can handle keyboard input.

```
// experi9b.c

#include <stdio.h>
#include <conio.h>
#include <malloc.h>
#include <time.h>
#include <dos.h>
#include <stdlib.h>
#include <string.h>

extern unsigned get_port(void);

main()
{
  unsigned ADC_Chan0,dac1,eoc;
  int count;
  char data[30000];

  ADC_Chan0 = get_port();

  if(ADC_Chan0 == 0)
  {
    printf("no hardware found\n");
    return;
  }

  dac1 = ADC_Chan0 + 8;
  eoc = ADC_Chan0 + 0x18;

  printf("ADC Channel 0 after get_port = %X\n",ADC_Chan0);

  while(!kbhit())
  {
    puts("Recording");
    disable();
    for(count=0; count<30000; count++)
    {
      outp(ADC_Chan0, 0); // start channel 0 conversion
      while(!(inp(eoc) & 0x80)); // wait for ready
      data[count] = (char)inp(ADC_Chan0);
    }
    enable();

    puts("Playing");
    disable();
    for(count=0; count<30000; count++)
    {
      outp(ADC_Chan0, 0); // start channel 0 conversion
      while(!(inp(eoc) & 0x80)); // wait for ready
      outp(dac1, data[count]); // put the data in the DAC
```

```
    }
    enable();

//    for(count=0; count<30000; count++)
//      printf("%4d",data[count]);
  }
}

// experi9b.c
```

Click here to download experi9b.c.

Compile experi9b then link the object module with the digital object module. Type in experi9b <enter> to run the program. Speak into the microphone when the program says it's recording, and listen when it says it's playing back. You should hear pretty good quality as long as you don't speak too softly or too loudly.

Many industrial applications need AD sample rates that are only a fraction of the speed required for recording voice. Analog information that exhibits a maximum rate of 10 cycles per second for example, will be captured pretty well at a 100Hz sample rate.

Capturing analog information in the timer interrupt service routine provides for a known sample rate and automates the process. To accomplish this, we will start out with an array of structures and an enumeration that will be added to timer.h. There are also some prototypes that will be described shortly:

```
// DA structures, prototypes & variables
unsigned eoc;
extern unsigned base;
int DA_Enabled = 0;
int CurrentChannel = 0;

struct anachan
{
  int data;
  int status;
} AnnalogChannel[8];

enum anastat
{
  INACTIVE,
  START_CONVERSION,
  DATA_READY
};

unsigned InitializeAnalog(void);
int TurnOnAnalog(int channel);
int TurnOffAnalog(int channel);
int GetChannelValue(int channel);
```

Click here to download timer.h. Be sure to first save any copies you might have.

The structures contain a data element to hold the results of the conversion. It also has a status element to show if the channel is inactive, ready to have a conversion started for it, or has data ready to be loaded. The prototypes describe

four routines:

InitializeAnalog() checks to see if there is hardware by calling get_port() then memsets the array to 0 and sets a channel counter to 0.
TurnOnAnalog() turns on a channel by setting the status element of the channel to START_CONVERSION. It also turns on a global indicator showing the AD is active.
TurnOffAnalog() turns a channel off by setting the status element of the channel to INACTIVE. It also runs through the array and turns off the global indicator if it finds no active channels.
GetChannelValue() obtains the results of a channel's conversion

Here's what the routines look like in timer.c:

```
// ======================= DA Routines ======================

unsigned InitializeAnalog(void)
{
  base = get_port();
  eoc = base + 0x18;

  if(!base)
    return 0;

  memset(&AnnalogChannel, 0, sizeof(AnnalogChannel));

  CurrentChannel = 0;

  return base;
}

int TurnOnAnalog(int channel)
{
  if(channel < 0 || channel > 7)
    return -1;

  AnnalogChannel[channel].status = START_CONVERSION;

  DA_Enabled = 1;

  return channel;
}

int TurnOffAnalog(int channel)
{
  int x;

  if(channel < 0 || channel > 7)
    return -1;

  AnnalogChannel[channel].status = INACTIVE;

  for(x=0; x<8; x++)
```

```
  {
    if(AnnalogChannel[channel].status != INACTIVE)
      break;
  }

  if(x == 8)  // all channels are inactive
    DA_Enabled = 0;

  return channel;
}

int GetChannelValue(int channel)
{
  if(channel < 0 || channel > 7)
    return -1;

  if(AnnalogChannel[channel].status == INACTIVE)
    return -1;

  return AnnalogChannel[channel].data;

  return channel;
}
```

The timer interrupt service routine has been modified to handle AD conversions as well as pwm digital outputs:

```

// the timer interrupt handler
// pwm type:
// 0 = unidirctional, no brake
// 1 = unidirectional with brake
// 2 = pwm line, directional line, no brake
// 3 = pwm line, directional line, with brake
// 4 = dual pwm lines -- both high = brake
// 5 = pwm line and two direction lines as for L298
// 255 = last slot -- leave
interrupt new_timer()
{
  int x;

  disable();

  timer_counter++;

  if(DA_Enabled) // is DA section enabled?
  {
    // look for start conversion or data ready status
    while(!AnnalogChannel[CurrentChannel].status)
    {
      CurrentChannel++;
      if(CurrentChannel > 7)
```

```
        CurrentChannel = 0;
    }

    switch(AnnalogChannel[CurrentChannel].status)
    {
      case START_CONVERSION:
        // will be ready at next interrupt, so say so
        AnnalogChannel[CurrentChannel].status = DATA_READY;
      break;

      case DATA_READY:
        // check eoc even though it's probably already ready
        while(!(inp(eoc) & 0x80));

        // load data into structure
        AnnalogChannel[CurrentChannel].data = inp(base);

        // set up for next
        AnnalogChannel[CurrentChannel].status = START_CONVERSION;

        CurrentChannel++; // bump up to next channel
        if(CurrentChannel > 7)
          CurrentChannel = 0;

        // this will find a channel that needs
        // a start even if it's just the above
        while(AnnalogChannel[CurrentChannel].status != START_CONVERSION)
        {
          CurrentChannel++; // get away from above channel
          if(CurrentChannel > 7)
            CurrentChannel = 0;
        }

      break;

    } // end switch(AnnalogChannel[CurrentChannel].status)

    // start a conversion - either the one designated
    // or the next one found after a data load
    outp(base + CurrentChannel, 0);

  } // end if(DA_Enabled)

  if(OutputControlActive)
  {
..... rest of new_timer() continues .....
```

Click here to download timer.c. Be sure to first save any copies you might have.

You might notice that the pwm control section also has a global active indicator. The analog section first checks DA_Enabled, the global active indicator for analog, to make sure it has been set. No channels have been turned on if

it is not.

The first thing done after that is to find the first active channel. The enumeration lists INACTIVE first, so it's 0. The other two possibilities are 1 or 2, so CurrentChannel is bumped up until a non- zero is found for a status element. CurrentChannel is wrapped back around to 0 if it goes over 7.

A switch statement then looks for a status of START_CONVERSION or DATA_READY. In the first instance, the status is simply changed to DATA_READY. That can be done since it will be ready by the time another interrupt occurs. If the status is already DATA_READY, the data element for the channel is loaded with the results of the conversion. The channel is then given a status of START_CONVERSION and a search is made for the next channel that needs a conversion started. It will be the one just set if only one channel is active.

A conversion is started following the switch statement. If a START_CONVERSION was found in the switch statement, that channel will be started. If a DATA_READY was found in the switch statement, then the next channel with a status of START_CONVERSION will be started.

The following sets up channel 0 for a sampling rate of 100Hz then prints the data obtained util a key is pressed. Vary the offset trimmer and talk in the microphone to see the effect:

```
// experi9c.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>

// defines OC structure
#include "outcont.h"

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

void main(void)
{
  int x;
  double dc,oldfreq,newfreq;

  oldfreq = 1193180.0/65536.0;

  set_up_new_timer(100.0);

  newfreq = get_frequency();

  printf("old frequency = %f new frequency = %fHz\n"
  ,oldfreq,newfreq);

  x = (int)InitializeAnalog();

  printf("init ana = %X\n",x);
```

```
  x = TurnOnAnalog(0);
  printf("TurnOnAnalog(0); = %d\n",x);
  while(!kbhit())
  {
    x = GetChannelValue(0);
    printf("%4d",x);
  }

  // be sure to restore the timer!
  restore_old_timer();
}

// end experi9c.c
```

Click here to download experi9c.c.

This following will set up then print all eight channels:

```
// experi9d.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>

// defines OC structure
#include "outcont.h"

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

void main(void)
{
  int x;
  double dc,oldfreq,newfreq;

  oldfreq = 1193180.0/65536.0;

  set_up_new_timer(800.0);

  newfreq = get_frequency();

  printf("old frequency = %f new frequency = %fHz\n"
  ,oldfreq,newfreq);

  x = (int)InitializeAnalog();

  printf("init ana = %X\n",x);
  for(x=0; x<8; x++)
```

```
    printf("TurnOnAnalog(%d) = %d\n",x,TurnOnAnalog(x));
  printf("Press any key to continue. ");
  getch();
  puts(" ");

  while(!kbhit())
  {
    for(x=0; x<7; x++)
      printf("%4d",GetChannelValue(x));
    printf("%4d\n",GetChannelValue(x));
  }

  // be sure to restore the timer!
  restore_old_timer();
}

// end experi9d.c
```

Click here to download experi9d.c.

Notice that the timer is set up for 800Hz. Since a single channel is sampled with each timer interrupt, 800 interrupts per second are needed to provide 100 samples per second for eight channels. Determine how many samples per second are needed per channel then multiply that by the number of channels to be set up to get the timer frequency. It's probably a good idea not to go over about 5000Hz though. A person could get 625 samples per second for each of the eight channels with 5000Hz.

To test the program, connect pins 6, 7, 8, 9, 10, 11, 12 and 13 together on Header 1 using the 16-conductor ribbon cable supplied with Experimenter's Kit #1 (see the order page). Vary the offset control and the digital values should track each other pretty well. Speak in the microphone and all 8 channels should show a response since they are all connected to the preamp output.

Previous: Experiment 8 - Digital To Analog Converter
Problems, comments, ideas? Please e-mail me

```
// experi9d.c

#include <dos.h>
#include <stdio.h>
#include <bios.h>

// defines OC structure
#include "outcont.h"

// include header with constants
#include "constant.h"

// include header with external prototypes
#include "extern.h"

void main(void)
{
  int x;
  double dc,oldfreq,newfreq;

  oldfreq = 1193180.0/65536.0;

  set_up_new_timer(800.0);

  newfreq = get_frequency();

  printf("old frequency = %f new frequency = %f Hz\n"
  ,oldfreq,newfreq);

  x = (int)InitializeAnalog();

  printf("init ana = %X\n",x);
  for(x=0; x<8; x++)
    printf("TurnOnAnalog(%d) = %d\n",x,TurnOnAnalog(x));
  printf("Press any key to continue. ");
  getch();
  puts(" ");

  while(!kbhit())
  {
    for(x=0; x<7; x++)
      printf("%4d",GetChannelValue(x));
    printf("%4d\n",GetChannelValue(x));
  }

  // be sure to restore the timer!
  restore_old_timer();
}

// end experi9d.c
```