LAB 1
Logic Gates, Flip Flops and Registers

In this first lab we assume that you know a little about logic gates and using them. The first experiment is an exericse to help you review combinatorial logic. Then we apply flip-flops in one of their most common uses: memory registers.

1. Logic Gates and Combinatorial Logic: The Half-Adder

NAND gates and NOR gates are commonly used to build logic circuits because of their flexibility. There are three tricks commonly used in digital design. The first is a result of DeMorgan's theorem –(AB) = (–A)+(–B). This means that a NAND gate is the same as a "bubbled" OR gate. Similarly, –(A+B) = (–A)(–B) so that NORs are also "bubbled" ANDS. This chameleon-like character is illustrated symbolically in Fig. 1.1. The second useful trick is that when a bubble on a gate's output is connected to another bubble on an input, the two bubbles cancel. This is written symbolically –(–A) = A but can be remembered easily by the saying "bubbles burst bubbles." A third trick that all digital designers have up their sleeves is how to make an inverter out of a NAND gate or a NOR gate. This is generally done by tying both inputs of the NAND or NOR together. Fig 1.1 summarizes these three principles.

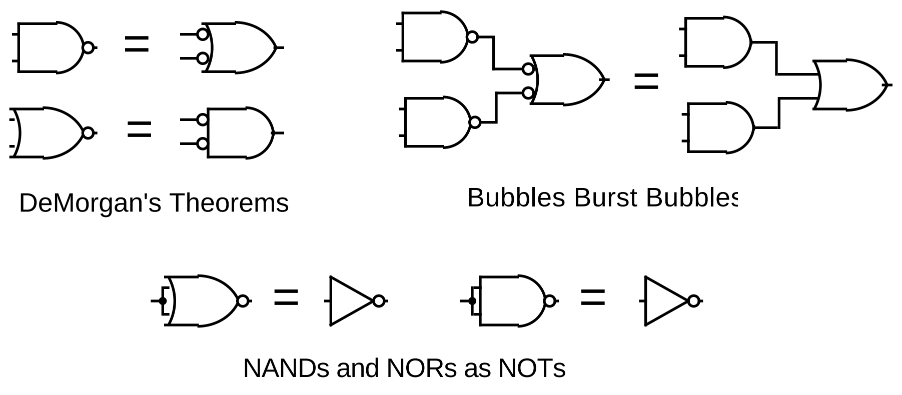


Fig. 1.1

As a first exercise, design an *exclusive or* gate, XOR, using NAND and NOR gates. The XOR output can be viewed as the lowest order bit of the sum of its inputs. Next, extend this circuit to make a *half-adder* which has two outputs. In addition to the low order sum bit which comes from an XOR, it has a carry bit which is the high order bit. The truth table below illustrates the operation of the half-adder. Modify your circuit to make a half-adder and test it. Use toggle switches for the inputs and LEDs for the two output bits. Can you measure the gate delay between a change on an input and the output change? For which output of the half-adder do you expect the largest delay? Verify by measuring the delay if you can.

Truth Table for the Half-Adder

A	B	A B	AB
		sum	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

2. Flip-flops and Registers

The simplest flip-flop is the NAND latch and its sister, the NOR latch. Each has four states. Two of the states are called "Set" and "Reset" because they cause the output to be either high or low. A third state is called the "Memory" state because this is the resting state where it retains the last value established by either the set or reset input. The fourth state has indeterminant output and is not used.

Prepackaged flip-flops like the 7474 D flip-flop have a clock input so that the set and reset inputs are active only on the rising or falling edge of the clock signal. Some chips use rising edge trigger, while others use the falling edge. Besides clocking, the D flip-flops do not have an indeterminant state which must be avoided: all possible inputs lead to a useful result.

Construct the simple memory register using two D-flip-flops as shown in Fig 1.2. The 74LS74 contains two D-type flip-flops. Most memory registers have eight or sixteen bits, but you can get the general idea by just a two-bit register. In fact, if you are pressed for time, you can build a single-bit register without loss of generality. Clock the register with a one Hz clock pulse and change the inputs with toggle switches to check out its functioning. Connect the clear inputs to +5 V.

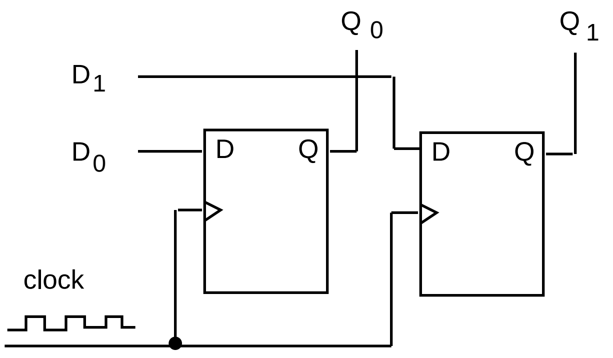


Fig 1.2

Most registers need a separate load-enable signal which tells it when to respond to the clock signal and capture the input values. The previous register could do this by ANDing the clock with an enable line so that the flip-flop would only trigger when both enable and clock became high. This isn't such a good idea because of the delay in the AND gate. The register would be activated a little later than the clock pulse edge. Normally digital circuits are designed so that all actions are synchronized precisely with the clock edges. The circuit below allows a load input while staying in synch with the clock. In this arrangement the load signal must be activated before the clock edge. The minimum time between enable and clock edge is called the setup time, t_{su}.

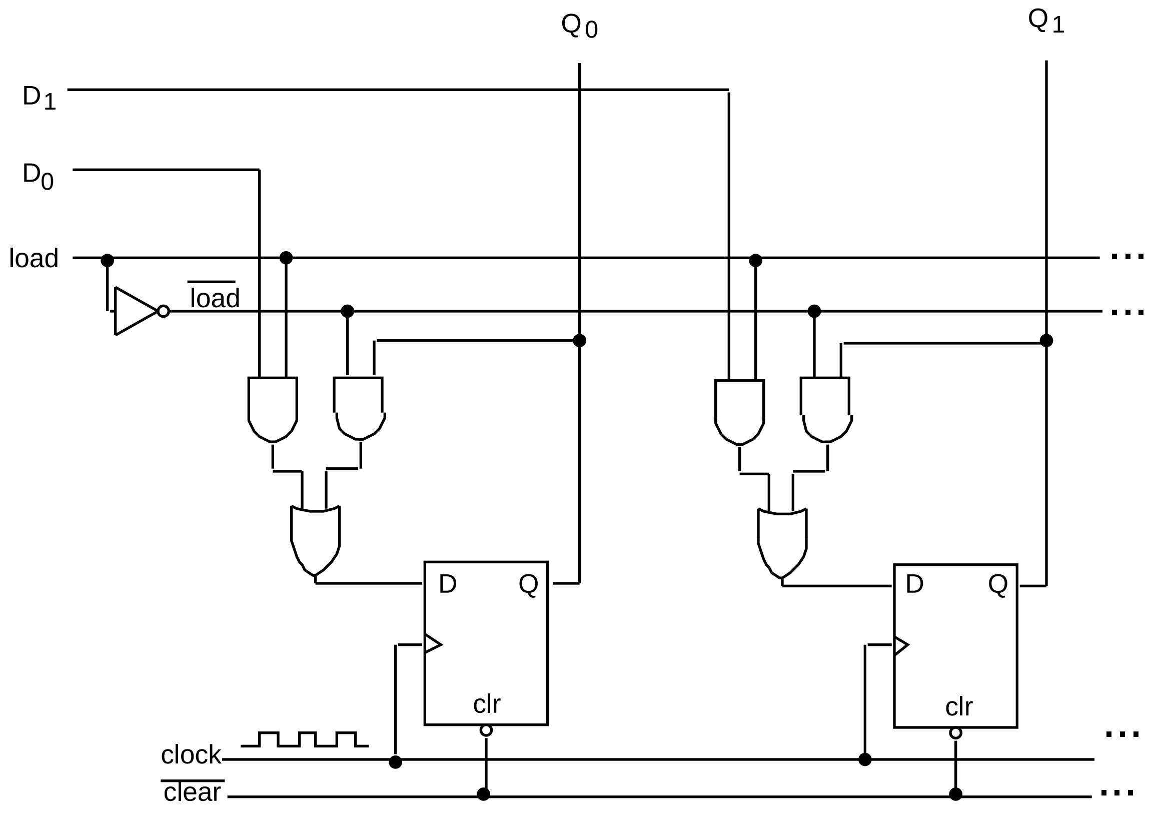


Fig 1.3

Verify that the circuit of Fig. 1.3 works as a controlled buffer register. How much setup time is required between load and the clock edge? Can you check this by measurement?

3. Open Collector Gates (7407)

The 7407 is a buffer chip with an open collector output. Open collector outputs are useful for interfacing to devices which don't obey TTL conventions. For example, the voltage level for high may not be in the 2.4 V to 5 V range, or a higher current drive may be needed. Some bus designs use open-collector outputs so that several devices may share the same bus. If all devices on the bus have a high output, then that bus signal is high. But if any device lowers its output, then the bus signal is low.

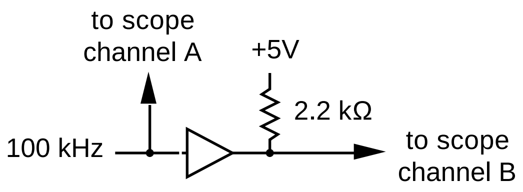


Fig. 1.4

In this experiment, connect the open collector output to +5 V through a 2.2 k resistor. Accurately record the output *vs.* input waveforms. Now, in addition add a 100 to 120 pF capacitor from the output to ground.

Explain and comment on the results.

4. Three State Outputs (74LS241)

Another way to connect outputs of several devices or registers to the same bus uses three-state buffers. The enable signal of the buffer is used to connect only one of the registers to the bus at a time. When the buffer is disabled or "three-stated," the high impedance between that buffer's input and output effectively disconnects that device from the bus. Thus only one device can be enabled at one time, the others being "three-stated."

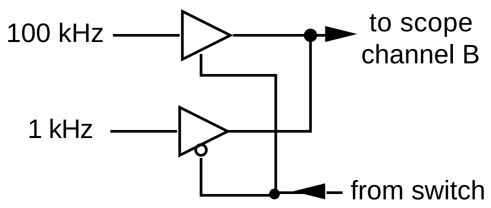


Fig. 1.5

Connect the inputs of two buffers of the 74LS241 to 100 kHz and 1 kHz clocks as shown in Fig. 1.3. The two outputs are connected together and monitored by the scope. Use a switch to alternately enable one clock signal or the other to the scope input. Accurately record the output. Now put 100-120 pF from output to ground. Compare and contrast with part 3.

LAB 2
Counters

1. The Ripple Counter

The ripple counter is the simplest counter you can build out of J-K flip-flops. All flip-flops are put in toggle mode. The first one is actuated by the clock signal so its output will change every second clock transistion. This output is then fed into the next flip-flop so its output changes every fourth clock edge. Therefore the output of the first flip-flop is the lowest bit of the binary count, the second is the second bit and so forth. The only problem with this kind of counter is the delay from the input of one flip-flop to the next one which accumulates as the number of bits increase. Still, if you only want to divide a high frequency clock signal by a lot, the ripple counter is a good and simple way to do it.

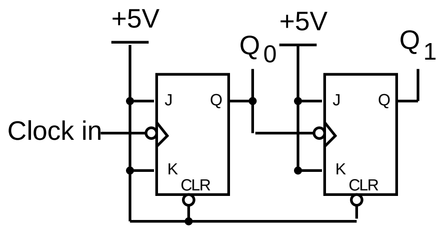


Fig. 2.1

Build this simple ripple counter and watch it work. Use LEDs to output the count and either a slow clock or a toggle switch for the clock input. Put a fast clock signal in and look at the Q_1 output frequency in relation to the input frequency.

2. The Synchronous Counter

The synchronous counter shown in Fig 2.2 is better for most counting applications because all its bits change at the same time. In order to show the construction of a synchronous counter we extend our diagram to include a third bit. See how the JK input of the higher order bits must AND all previous bits. Thus flip-flop for bit n will toggle only when all lower order bits are high.

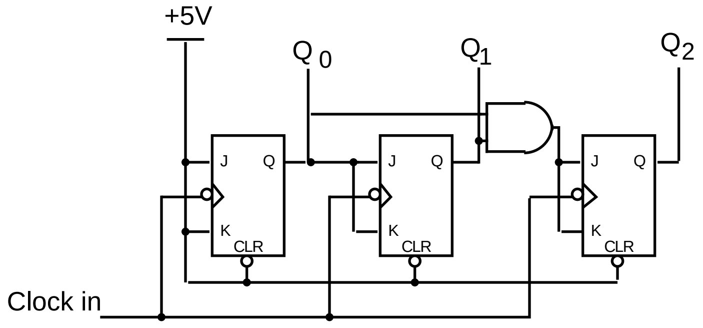


Fig 2.2

Redesign this circuit to use a NAND or NOR gate instead of the AND gate. Build it and verify that it works.

3. The Ring Counter

The ring counter's output only has one high bit. It shifts this high bit right or left with each clock pulse. When it reaches the last output it reverts back to the first position again. You should build the ring counter shown in Fig 2.3 and see how it works. These counters are useful in microprocessors where single instructions may take several clock cycles. A ring counter keeps track of which phase of the instruction it's in.

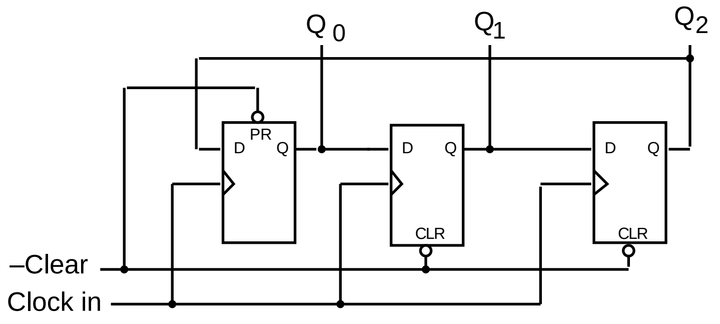


Fig. 2.3

4. 74LS161 four-bit binary counter

The 74LS161 is a prepackaged divide-by-sixteen counter that you will find useful.

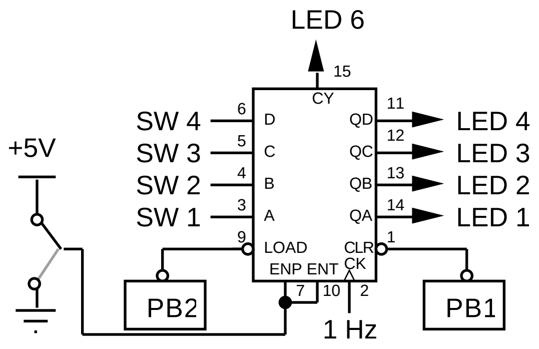


Fig. 2.4

With this setup you should be able to verify all aspects of 74LS161 operation: parallel loading, clearing, and counting. Record results and prepare a table indicating the operations of the device, including ENP = ENT = 0 or ENP = ENT = 1.

You can make a Divide-by-N counter where N is set by the parallel-load switches. Disconnect the LOAD input from the pushbutton. Route the CY output through an inverter to the

LOAD input. Now the counter will reload the parallel input bits when CY goes high. By loading in the two's complement of N you get the Divide-by-N counter.

5. The 74LS138 One-of-Eight Decoder

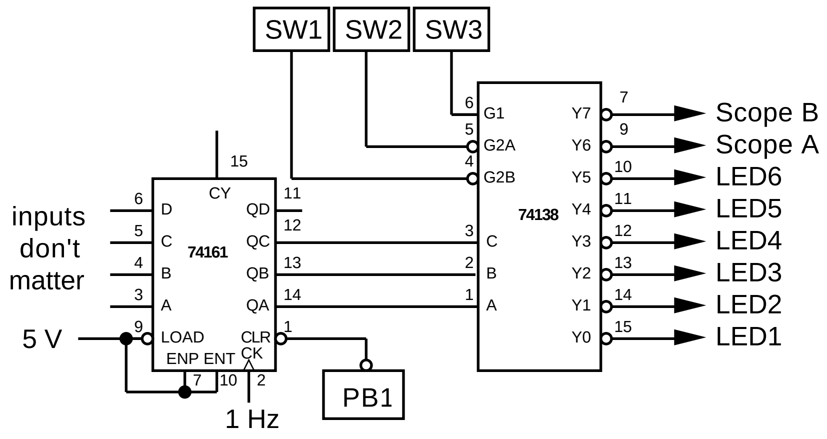


Fig. 2.5

This setup allows you to check out the entire truth table of the 74LS138. The value of ABC cycles $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \ldots \rightarrow 7 \rightarrow 0$. PB1 resets the value to 0 at any time. SW1, SW2 and SW3 control the enabling inputs. The six LEDs and the two scope channels monitor the eight outputs.

Verify the operation of the 74LS138 and prepare its complete truth table.

LAB 3

Address Decoding

Today we will begin to use the computer and to wire-wrap circuits on the prototyping board.

N.B.: Steps 1, 2 and 3 should take no more than 3/4 hour.

1. Following the instructions in section 0.2, start the computer. Start up GWBASIC. Try writing a few simple programs to write numbers to the screen, get numbers from the keyboard, *etc*.

> Time-saving hint: Press the spacebar during the boot-up memory check to skip the check and load DOS right away.

2. Plug the prototyping board into the bus slot. Using BNCs and the two jumper wires look at OSC and CLOCK, recording frequencies and amplitudes. Now, carefully use the 10X probes to look at the same signals. Comment

3. According to the handout, –DACKO should be low during refresh DMA cycles. Monitor -DACKO, noting times and amplitudes. How often do refresh cycles take place? What percentage of the bus time do they take up?

4. We will now wire up an address decoder for the I/O addresses 300H to 31FH. For reasons to be explained later, we divide this into seven addresses for which A0 and A1 don't count. For now, A0 and A1 are connected so that the decoder accepts only A0 = A1 = 0.

It is useful to establish a uniform colour code for wiring the prototype board. However, the selection of wire colours changes from year to year. Fill in the table below for this year's colours.

Wire-wrap Colour Code

Ground	_____________
+5 V	_____________
Direct to Bus	_____________
On Board	_____________
Off Board	_____________

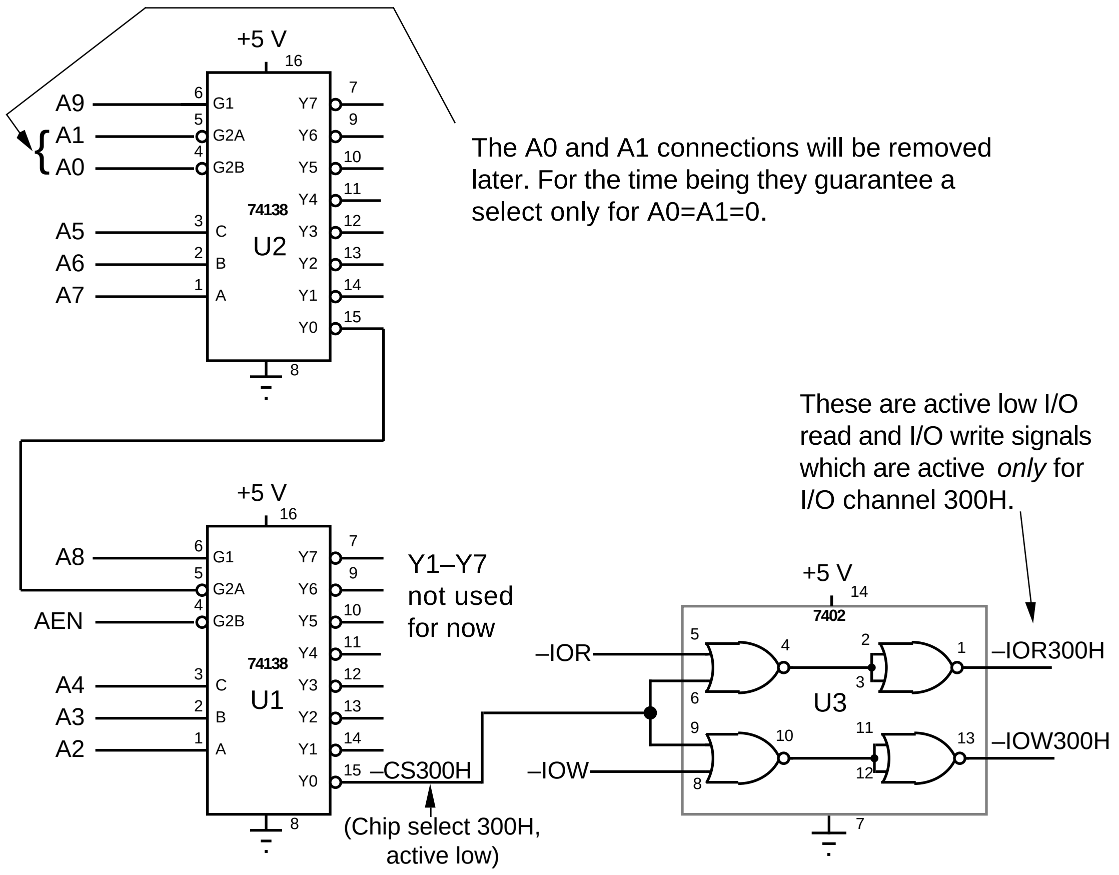

+5 V
16
A9
A1
A0
G1
G2A
G2B
Y7
Y6
Y5
Y4
Y3
Y2
Y1
Y0
74138
U2
A5
A6
A7
C
B
A
8
The A0 and A1 connections will be removed later. For the time being they guarantee a select only for A0=A1=0.
These are active low I/O read and I/O write signals which are active only for I/O channel 300H.
A8
AEN
A4
A3
A2
74138
U1
Y1–Y7 not used for now
–CS300H
(Chip select 300H, active low)
+5 V
7402
U3
–IOR
–IOW
–IOR300H
–IOW300H

Fig. 3.1

Use the scope to observe –IOR300H while doing a loop of I/O inputs from that address

```
        1 X=INP(&H300)        note: X is a dummy variable
        2 GOTO 1
```

Note that this loop will execute much more rapidly under QBASIC. Measure the frequency, if possible under both GWBASIC and QBASIC.

Similarly, monitor –IOW300H for the following loop:

```
        1 OUT &H300,0
        2 GOTO 1
```

(Optional) You can get much faster operation by using machine language. DEBUG allows programing assembly language mnemonics. DEBUG can be accessed while running Windows (in the same way as GWBASIC and QBASIC) or by typing `DEBUG` after the DOS prompt. The DEBUG prompt is just a hyphen at the left of the screen:

```
-
```

Next enter the assembly language program: (underlined characters are typed by the computer)

```
-    A 100                starts assembling the code at offset 100 hex.
xxxx:0100 MOV DX,0300     puts 300 in register pair DH,DL (16 bits), xxxx is the segment
xxxx:0103 MOV AL,00       puts zero (eight bits)  into register AL
xxxx:0105 OUT DX,AL       output contents of AL to port pointed to by DH,DL
xxxx:0106 JMP 103         go back to offset 103 (the MOV AL,00 instruction)
xxxx:0108                 type a blank line here to stop assembling
```

To make sure that memory is loaded correctly, use the `U` (unassemble) command. Just type

```
U 100
```

and 32 bytes starting from location 100 will be displayed along with their assembly mnemonics.

To run the program simply type

```
G=100                                The equal sign (=) is necessary!
```

There's no way to stop this program short of rebooting the computer.

That's a little inconvenient. A better program would loop a certain number of times, then stop:

```
    MOV CX,FFFF                      load up the loop counter with a big number
    MOV DX,0300
    MOV AL,00
    OUT DX,AL
    LOOP 106                         Decrement CX and loop back to 106 if CX is not zero
    INT 3                            Break to debugger
```

This gives you a short time to measure and then stops.

Note: Documentation for DEBUG is included in the online Help.

(Also optional) Using the instructions for UNIFORTH, program FORTH to do this and time it.

5. You can use these two lines to set and reset an external flip flop (74LS74) on the *breadboard* .

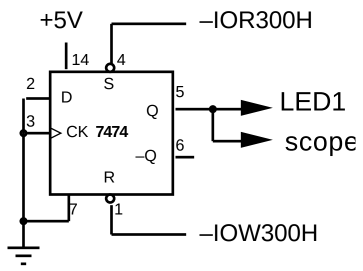


Fig. 3.2

First, set and reset the flip-flop by issuing single `X = INP(&H300)` and `OUT &H300,0` instructions then toggle it back and forth with the following loop:

```
1 OUT &H300,0
2 X = INP(&H300)
3 GOTO 1
```

What is the loop time under (1) GWBASIC , (2) QBASIC, (3 optional) assembly language and (4 optional) FORTH?

Problem to answer in your lab book:

What are the hex chip select addresses for the Y1 to Y7 outputs of the circuit of Fig. 3.1?

LAB 4
Buffers and Latches

In lab 3 you wire-wrapped an address decoder circuit which recognized up to eight chip-select addresses from 300H to 31FH, the I/O area alloted to the prototyping board. In addition, the 300H chip select was combined with –IOR and –IOW to generate I/O read and write pulses specific to device 300H. This week we will use these decoded –IOR and –IOW signals to transfer bytes of data to or from the computer.

1. Here we use the scope to verify that the correct signals are present on the data bus during I/O write operations.
First, write an IOW loop which continuously outputs a number N, supplied by you, such that 0 < N < 255 and the binary equivalent of N contains four “0s” and four “1s”

```
1 OUT &H300,N        Put in actual N value
2 GOTO 1
```

This can be run under either GWBASIC or QBASIC, but the latter is faster and, therefore, easier to watch on the scope.
First, use the scope to look at any single line of the data bus, Dn. Now, use the other scope channel to monitor –IOW300H, and trigger from that channel as well.
The Dn signal should be either always high, or always low, while –IOW300H is low, depending on whether your number N has a 0 or a 1 in bit n of its binary equivalent. (Note: D0 is the least significant bit)
Write your N as a binary number, and verify with the scope that D0…D7 during –IOW300H low correspond to the bit pattern of N.

2. Now that we know that the computer presents the byte N at the data bus at the same time as –IOW300H is active, we can latch the data so that it is permanently present, until changed.

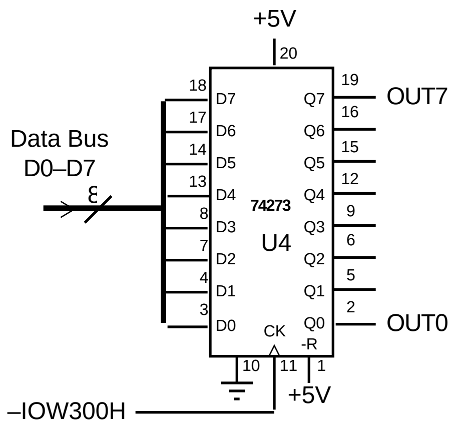


Fig. 4.1

Verify that a single execution of `OUT &H300,N` causes the bit pattern of N to appear as DC levels at OUT0 through OUT 7. This circuit is now complete for outputting one byte of data to control an arbitrary device. Verify its operation for several bytes N, and also verify that it isn't affected by `OUT`s to different addresses.

`OUT P, Q` where $P \neq 300H$

(Note: Keep P in the range 300H to 31FH or you may accidentally modify a port which is important to the computer).

3. Now we will use –IOR300H to load a byte of data into the computer. This requires an eight-bit wide chip with three-state outputs, which only drives its data onto the data bus when –IOR300H is low. The 74LS240, 74LS241 or 74LS245 could be used. Here we choose the 74LS241.

Note that the two groups of four gates in the LS241 have opposite polarities of output enable. This is no problem, since IOR300H is also available from pins 2, 3 and 4 of the 74LS02 wired up in LAB 3.

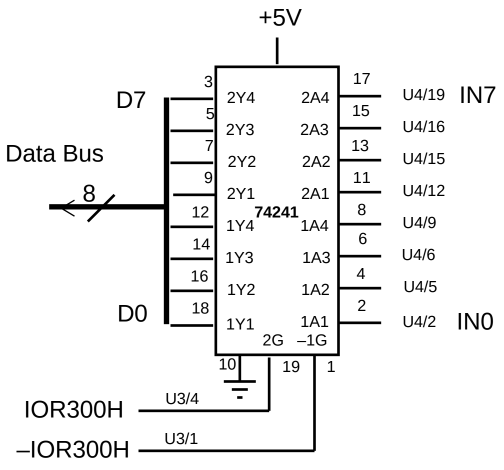


Fig. 4.2

Run a small loop to display the results of an input from 300H.

```
        10 CLS
        20 X=INP (&H300)
        30 PRINT X
        40 GOTO 10
```

If you're using the TTL 74LS241, IN0 through IN7 will float high if not grounded. Use the two jumpers to ground several combinations of IN0 thru IN7, and verify that the computer is in fact reading in the correct number each time.

4. Now we will use the computer to check that both in and out instructions function properly for all 256 combinations.

Temporarily connect eight wires between OUTn and INn (eg: OUT0 to IN0 *etc*.) Now run this program:

ALAN1.BAS

```
10 FLAG = 0
20 FOR XOUT = 0 TO 255
30     OUT &H300, XOUT
40     XIN = INP(&H300)
50     DIFF = XOUT – XIN
60     IF DIFF<>0 THEN FLAG=FLAG+1:PRINT "error for xout = ";XOUT ; " -> xin = ";XIN
70 NEXT XOUT
80 PRINT "check completed with ";FLAG;" error(s)."
```

(If you don't wish to type it in, it is available as ALAN1.BAS.)

Once the proper operation of the IN and OUT for all 256 possible outputs has been verified, *remove the eight temporary connections between OUTn and INn*.

5. IF TIME PERMITS we will use our digital output and input ports to verify operation of an IC (74LS138) and print out its truth table.

Temporarily wire up this circuit:

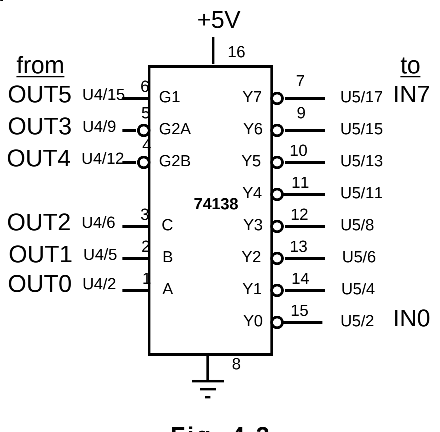


Fig. 4.3

Bits 0 through 5 of the OUTPUT determine all possible inputs to the 74LS138, while all eight outputs of the 74LS138 are simultaneously monitored by the digital INPUT.

The following program (available for copying to your work disk as ALAN2.BAS) prints out a truth table for Y0 thru Y7 for the eight combinations of A, B, C given the fixed values of G1, G2A and G2B.

Use it to verify the proper operation of the 74LS138, and print the output for G1 = 1, G2A = G2B = 0 on the printer (using SHIFT PRT SCN).

ALAN2.BAS

```
5 CLS
10 INPUT "setting for G1 (0 or 1)";G1
20 INPUT "setting for G2A (0 or 1)";G2A
30 INPUT "setting for G2B (0 ro 1)";G2B
40 OFFSET= 32*G1 + 16*G2B + 8*G2A
50 PRINT" A     B     C     O0    O1    O2    O3    O4    O5    O6    O7"
60 FOR C=0 TO 1
70 FOR B=0 TO 1
80 FOR A=0 TO 1
90 XOUT=OFFSET + 4*C + 2*B + 1*A
100 OUT &H300,XOUT
110 XIN=INP(&H300)
120 O0=(XIN AND 1)/1
130 O1=(XIN AND 2)/2
140 O2=(XIN AND 4)/4
150 O3=(XIN AND 8)/8
160 O4=(XIN AND 16)/16
170 O5=(XIN AND 32)/32
180 O6=(XIN AND 64)/64
190 O7=(XIN AND 128)/128
200 PRINT A;"   ";B;"   ";C;"   ";O0;"   ";O1;"   ";O2;"   ";O3;"   ";O4;"   ";O5;"
";O6;"   ";O7
210 NEXT A
220 NEXT B
230 NEXT C
240 PRINT:PRINT:PRINT
250 INPUT "go again with new G1, G2A, G2B (y or n)";TEST$
260 IF (TEST$="y" OR TEST$="Y") GOTO 5
```

Problem (Based upon the above circuit)

Write a BASIC program which verifies that NO output Y0 thru Y7 can be low for any combination of A, B, C except for the specific case G1 = 1, G2a = G2b = 0, and that for this specific case of G1, G2A and G2B, there will *always* be *one* output low for any A, B, C. (We don't care which one)

LAB 5

The 8255 Programmable Peripheral Interface

In lab 4 you constructed a one byte digital output port and a one byte digital input port using fairly simple TTL devices. In this lab we will investigate the use of a much more powerful (and complicated) device for digital IO, the 8255 programmable peripheral interface chip.

First, we will have to slightly rewire the address decoder of experiments 3 and 4, and for brevity, relabel some of the signals.

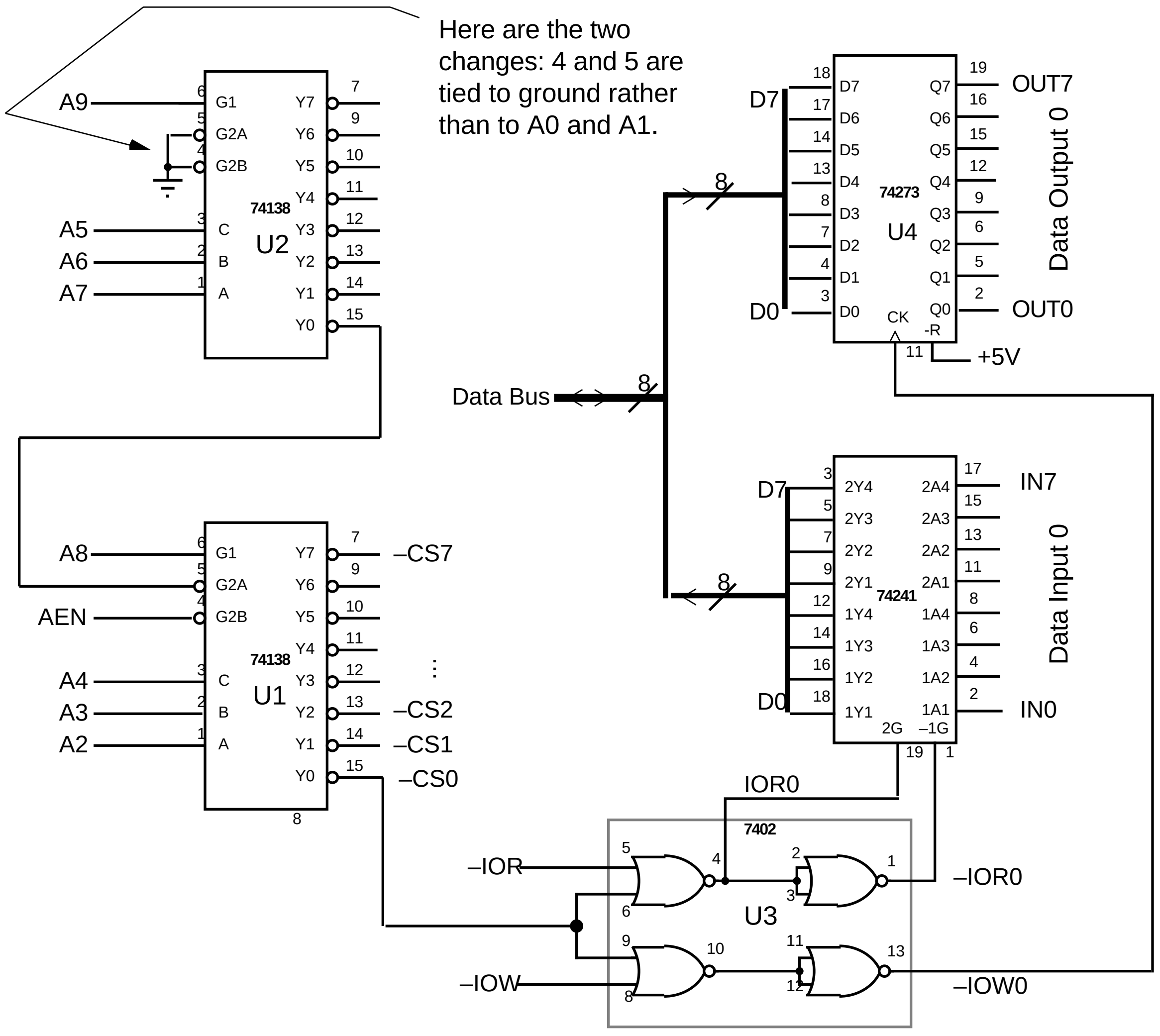


Fig. 5.1

Note that we have retained the one-byte output and input ports, but relabeled them as device 0, and their select line as chip select 0 (–CS0).

Also, note that device 0 should be activated by any of the four I/O addresses: 300H, 301H, 302H, 303H.

Verify this for both the input and output ports.

Now you can wire-wrap the bus signals to the 8255 (be careful about locating the 8255 pins on the card)

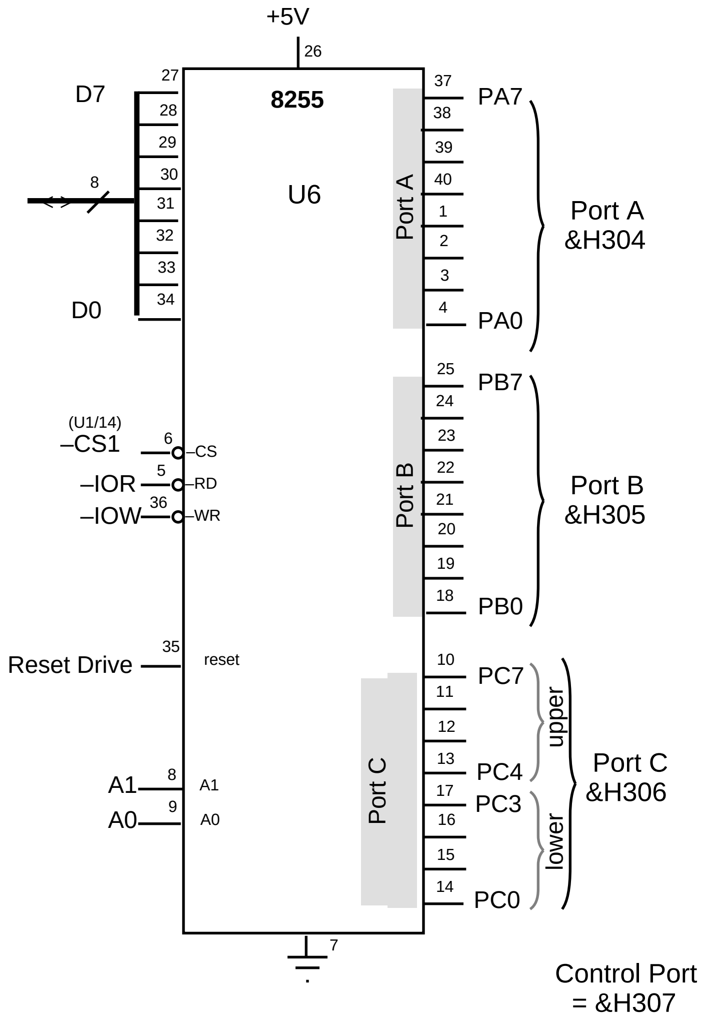


Fig. 5.2

Once this is wired, plug the board into the bus, turn on the extender box power, then turn the computer off. Wait ~15 s, and turn the computer on again. This will generate the RESET DRIVE signal, which will initialise the 8255, clearing the mode register and setting all ports to input. This means all the port pins should be in a high impedance state. Check this with the following circuit:

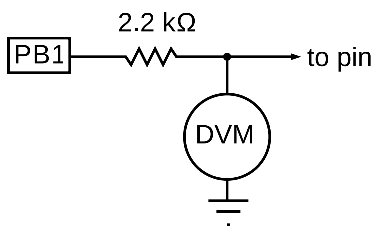


Fig. 5.3

With nothing connected, or when connected to an input (high impedance) pin, there will be little drop across the resistor, so the DVM will read ~0.2V or ~4V (when PB1 is pressed).

Verify that when this is connected to a TTL output which is either high or low, PB1 has little effect (could use IOR0 and –IOR0).

Now check all the 8255 port pins.

Next, we can load the control register. Note that the I/O addresses are:

PORT A :	304 H
PORT B:	305 H
PORT C :	306 H
CONTROL REGISTER:	307 H

The various control bytes are listed in the 8255 descriptions (Mode 0 are on 3-107 and 3-108). For example, control word = 8AH gives:

port A as output

port B as input

port C lower (PC0 - PC3) as output

port C upper (PC4 - PC7) as input

Load this control word by executing

```
OUT &H307, &H8A
```

You can now check that Port A and Port C (lower) have gone into output (low impedance) mode, while PORT B and PORT C (upper) are still in input (high impedance mode).

Having loaded this control word, you can now input and output data:

`OUT &H304, N`	0 ≤N ≤255) Port A
`X = INP(&H305)`	input byte from Port B
`OUT &H306, N`	output to PC0 thru PC3, note: 0 ≤N ≤ 15
`X = INP(&H306)`	input from PC4 thru PC7. Only the 4 high-order bits are designated for input, but the INP statement reads all 8 bits. Do some tests to determine the values input from the 4 low-order bits under various situations.

Try a few other Mode 0 configurations by loading different control bytes, and see if they operate as advertised.

If Time Permits :

Mode 0 operation is the simplest case, giving great flexibility of input or output but no extra functions. One drawback is that input is instantaneous—the state of the inputs is read whenever –IOR goes low.

In Mode 1, the general purpose nature of PORT C is lost, but input data can be strobed into an input latch by an external pulse to PC2 or PC4, where it resides until read by –IOR. This can help synchronize things. There are also other inputs and outputs from PORT C in MODE 1 which have special functions. Verify some of the operations in Mode 1.

LAB 6

The 8253 Programmable Interval Timer

In lab 5 we saw how digital I/O could be simplified by using a more powerful device — the 8255 Programmable Peripheral Interface chip. Today we will use a related device, the 8253-5 Programmable Interval Timer, which greatly simplifies I/O related to timing or counting operations.

Hooking up the 8253-5 to our existing circuitry is easy:

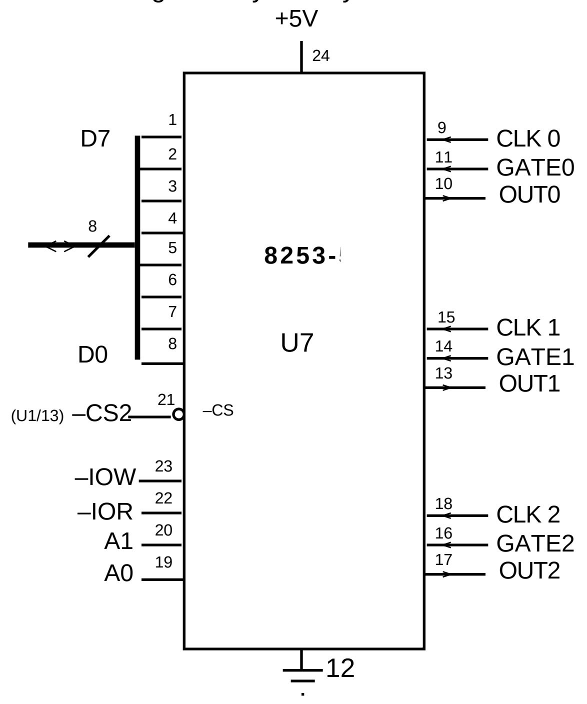


Fig. 6.1

Use the same bus hook--up as the 8255, but using –CS2 addresses: 308H, 309H, 30AH, 30BH
So the 8253 control registers are set with OUT &H30B, X

MODE 0, Interrupt on Terminal Count

Use the 16 pin jumper cable to connect signals between the prototyping board and the stimulator board.

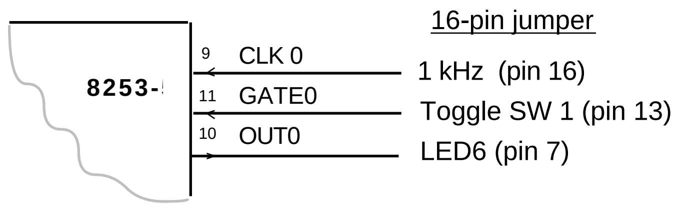


Fig. 6.2

Load the control register .

	D7	D6	D5	D4	D3	D2	D1	D0
&H30 =	0	0	1	1	0	0	0	0
	select counter 0		double byte operations		mode 0			binary counting

Fig. 6.3

So OUT &H30B, &H30 sets the mode for counter 0. The counter will count down in ms from the loaded number to 0, at which time OUT should go high a) with SW1 off (low), load a count of several seconds (ie: load a number of several thousand). Remember you have to do *two* loads to counter 0, first the least significant byte, then the most significant byte.

```
OUT &H308, 0        loads an LSB of 0
OUT &H308, &H40     loads MSB = &H40
```

so count = 2^{14} = 16384 or 16.3 seconds

After having loaded both bytes, turn SW1 ON and time the interval until LED6 lights.

Try loading again with SW1 already ON. Now the interval should be timed from the second OUT operation.

MODE 1, Programmable One Shot

The difference here is that OUTPUT will go low on the rising edge of GATE (switch 1).

```
OUT &H30B, &H32
```

sets this mode for counter 0.

Again, load a two-byte count, and verify the operation.

It should be retriggerable (every D edge of gate restarts the timing interval). So if SW1 is operated frequently enough, output will always be low.

MODE 3, Square Wave Rate Generator

First, switch clock 0 input to 1MHz (pin 2) so that we can time in µs.

Set the mode:

```
OUT &H30B, &H36
```

then output a 2 byte count.

While GATE is high, the output should be a square wave with a period in µs equal to the loaded count value.

Try this for several different values of the 16 bit count.

Verify that the 8253 works in BCD mode by setting BCD mode 3

```
        OUT &H30B, &H37
```

Now the 16 bit count you load gives a number from 0 to 9999 rather than from 0 to 65535.

In a similar manner to the foregoing, try to verify operation in modes 4 and 5 (binary counting only).

MODE 5, Hardware Triggered Strobe

For mode 5, try hooking Gate 0 to the 100 Hz signal (pin 9) and loading a count of less than 10,000 µs. Monitor OUT0 vs GATE 0 on the scope.

COUNTING

In any of these modes, the contents of the counters can be read, so we can count the number of events in a given interval. However, we must remember that these are down counters.

Connect CLOCK 0 of 8253 to PB1 (pin 8), and GATE 0 to SW1 (pin 13).

Load Mode 0, binary OUT&H30B, &H30

To start with, we must load something into the counter; all 0s seems the simplest choice.

```
      OUT &H308, 0
      OUT &H308, 0         must load both bytes
```

Now, write a looping program which looks at the contents of the counter and displays the count.

```
      LSB = INP(&H308)
      MSB = INP(&H308)                       must do both
      COUNT = 65536 – (LSB + 256 * MSB)
```

(Note: for 0 counts this incorrectly gives 65536)

Try it out, verify that GATE0 must be high to count.

The above uses non-latched reading, which can lead to errors.

To protect from the possibility of reading garbage while the counter is counting (particularly likely during 2-byte reads!) We can latch the entire 16 bit count, and then read it.

Use the same mode load two 0 bytes out. Enter a given number of counts, N, with push button . Now latch data:

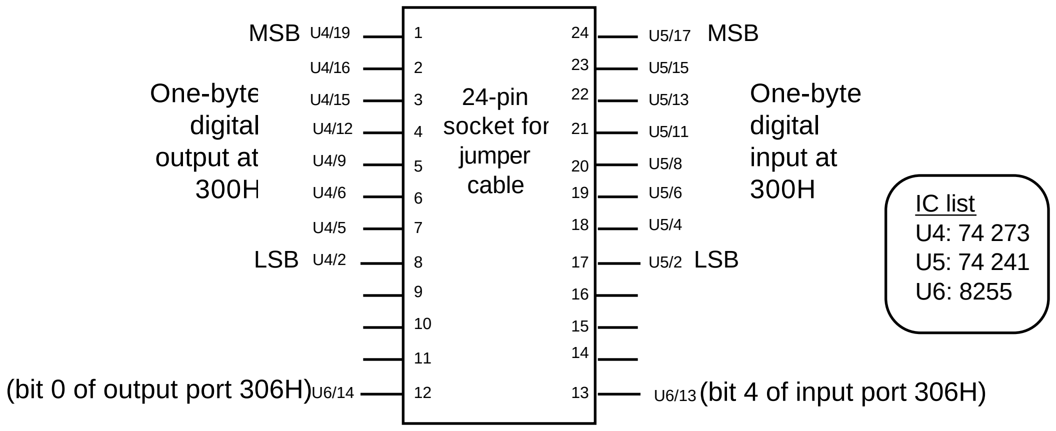
```
      OUT &H30B, 0          0 is the latching byte for counter 0
```

Enter a few more pulses (M) and read the contents as before. The number of counts should equal N, not N+M.

LAB 7
ANALOG I/O

In this lab we will connect and test an 8 bit analog-to-digital converter (A/D) and an eight-bit digital-to-analog converter (D/A). The analog circuitry will be set up on the breadboard area, and a 24-line jumper cable will be used to transmit digital data between the breadboard and the prototyping board.

First you will have to wire-wrap the signals to the 24 pin socket soldered along the top edge of your prototyping card. We will have a byte of digital output (&H300), a byte of digital input (&H300), plus port C of the 8255 split into four output bits and four input bits (&H306) (actually, only two Port C lines are needed now).



Note: When inserting the jumper cable, make sure that pin 1 on the plug matches pin 1 on the socket!

Fig. 7.1

The other end of the 24-line jumper cable can be plugged directly into the breadboard for wiring to the chips.

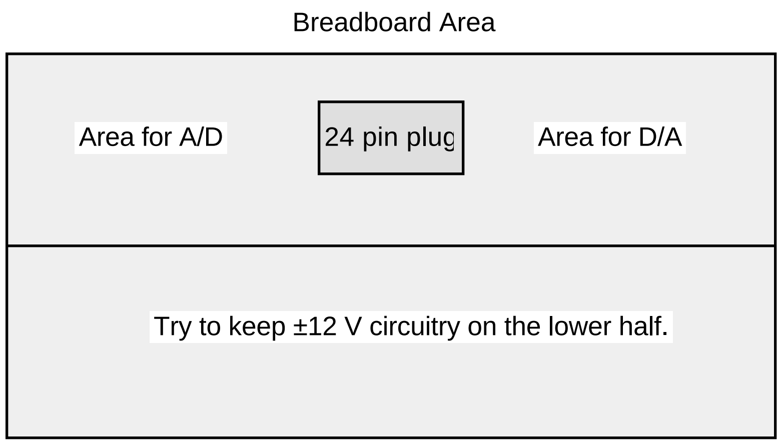


Fig. 7.2

Caution! Voltages outside of the 0 to +5 V range can be fatal for TTL devices. In these experiments, ±12V will be present on the breadboards. *Make sure* that these voltages are not applied to the wrong points!

1. Digital to Analog Conversion, Current Output

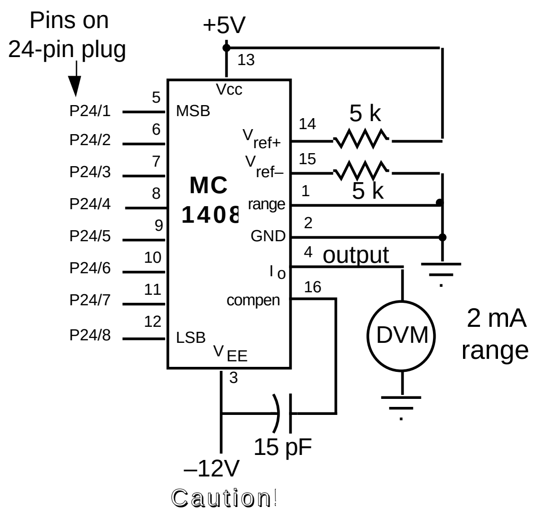


Fig. 7.3

`OUT &H300, N` loads the converter. N = 0 should give 0 current; N = 255 should give maximum current which, in this case, is almost 1 mA.

Output N = 0, N=255 and a few others; measure I and verify operation.

2. D/A Conversion, Voltage Output

Usually we want voltage as an output, not current. An op-amp is needed such as the ever-popular 741 or the higher-performance 411.

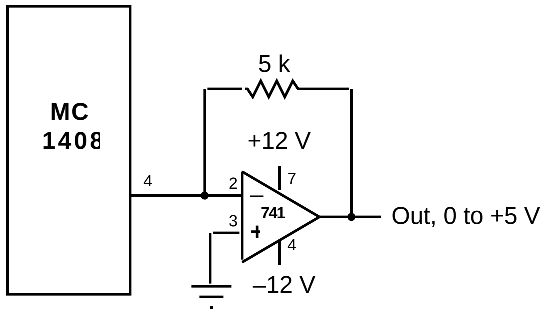


Fig. 7.4

Again verify that $V_{out} = \frac{N}{256}$ x 5 V for several N. *Watch out for ±12 V.*

3. Analog to Digital Conversion

Here we wire up a 0 to +5V analog input

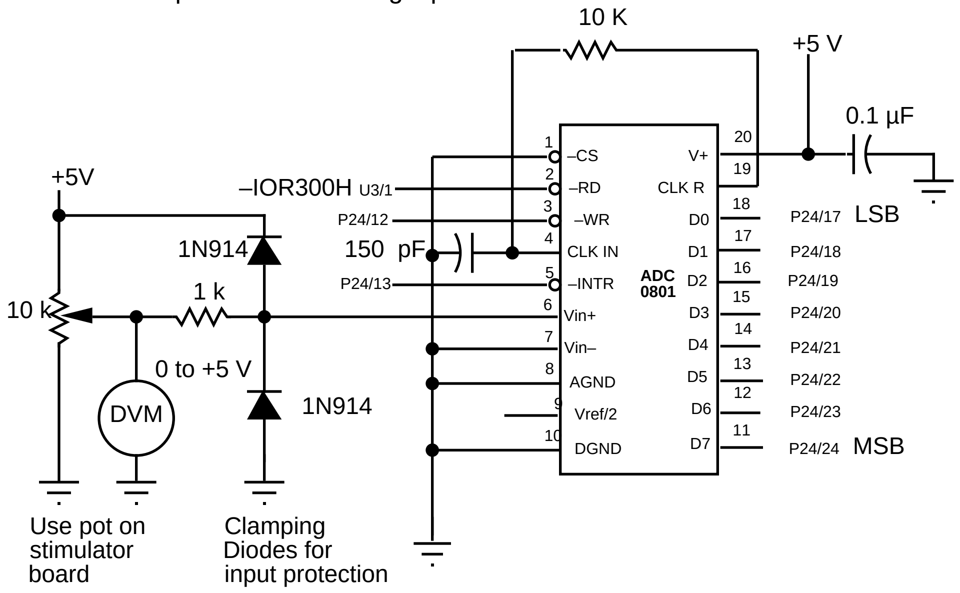


Fig. 7.5

You may substitute ADC 0804 for the 0801. It has the same pinout but less accuracy.

In order to read in the output of the A/D, we need a small program:

ALAN3.BAS

```
1 OUT &H307,&H8A
2 CLS
5 LOCATE 3,3
10 OUT &H306,0
20 OUT &H306,1
```

```
30 X=(INP(&H306) AND &H10)
40 IF X<>0 THEN GOTO 30
50 PRINT "Voltage = ";(INP(&H300)/255 * 4.95),"and Digit = ";INP(&H300)
60 GOTO 5
```

Verify proper operation of the A/D input for several input voltages, including 0V and +5V. In line 50, the input is divided by 255. Would it be better to divide by 256? Discuss.

4. Output Rise and Fall times and Conversion Delay

Execute a QB loop to alternately output ~1V and ~3V. Measure the output rise and fall times with the scope.

Measure the ADC0801 clock frequency. Execute a basic loop which keeps starting the ADC0801 and reading its output. Using the scope, measure the delay between the start of conversion (pin 3) and the end of conversion (pin 5).

How many clock cycles does a conversion take?

5. Diode Curve Tracer

We will use the analog output and input to measure the forward I-V curve of two diodes. Run the "GRAPHICS" program, so that you can do a graphic screen dump later by pressing `Shift-PrtSc`.If the HP Deskjet printer is going to be used, type `graphics deskjet` after the DOS prompt and then return to your previous application

The 1408 D/A chip and the op amp already associated with it can be used to cause a programmable 0 to 1 mA current to flow through the diode.

Since the inverting input of the op amp is virtual ground, V_0 = diode voltage drop, I = 0 to 1 mA.

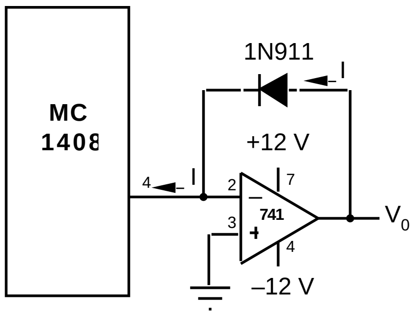


Fig. 7.6

V_0 will be less than +5V, so we can amplify it for better resolution.

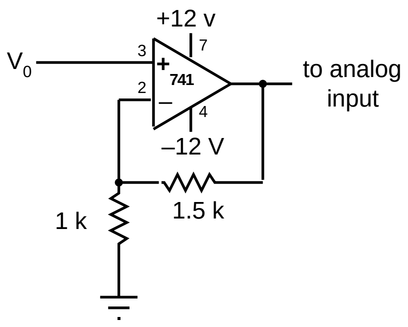


Fig. 7.7

Since this has a gain of 2.5, the 0 to 5 volt span of the ADC corresponds to a 0 to +2 volt range at V_0.

Use the ALAN7.BAS program. First check the calibration and accuracy of the D/A and A/D conversion by plotting the I-V curve of a resistor. After you've verified correct operation, plot the I-V curve of the 1N911 diode. Label the axes in mA and volts.

Now try comparing a Light Emitting Diode.

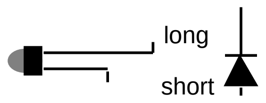


Fig. 7.8

ALAN7.BAS

```
'Diode Characteristic Program

cls
dim xin(255)
out &h307,&h8a                'configures 8255
 for xout=0 to 255
   out &h300,xout             'sends xout to get converted
   out &h306,0                'strobes wr low
   out &h306,1                'to start conversion
   xin(xout)=inp(&h300)       'reads converted value
   'print xout,xin(xout)
next xout


screen 2
locate 15,2:print "xin";                'label all the boxes
     locate 4,4:print "255";
     locate 22,5:print "0";
locate 23,30:print "xout";
     locate 23,11:print "0";
     locate 23,68:print "255";
window (-50,300)-(300,-50)
for i=0 to 255
   pset(i,0):pset(0,i):pset(i,255):pset(255,i)  'draw box
   pset(xin(i),i)                               'plot line of in vals
next i
```

LAB 8
Stepping Motors (1)

(Data on the stepping motor and controller chip can be found on the data sheets)

First, remove all wiring from the breadboard area. Since both this lab and labs 6 and 7 use +12V, we want to reduce the possibility of TTL chip destruction which increases with the complexity of the wiring.

WARNING: DO NOT rotate the motor shaft by hand while the motor is connected to any circuitry. The motor can act as a generator of damagingly high potentials. Make sure that all your wiring is correct before energizing the circuit!

1. Simple, Direct Control of Motor

We already have one byte of digital output wired to the 24 pin jumper socket (OUT &H300) and a separate single output line going to pin 12 of the 24 pin socket (port C, bit 0). These can be used to control the stepping motor.

Remember to initialize the 8255 for the required mode of operation before doing anything else: (and whenever re-energizing)

```
OUT &H307, &H8A
```

gives Port A: out, Port B: in, Port C 0 to 3: out , Port C 4 to 7: in.

Now, wire up this circuit on the breadboard

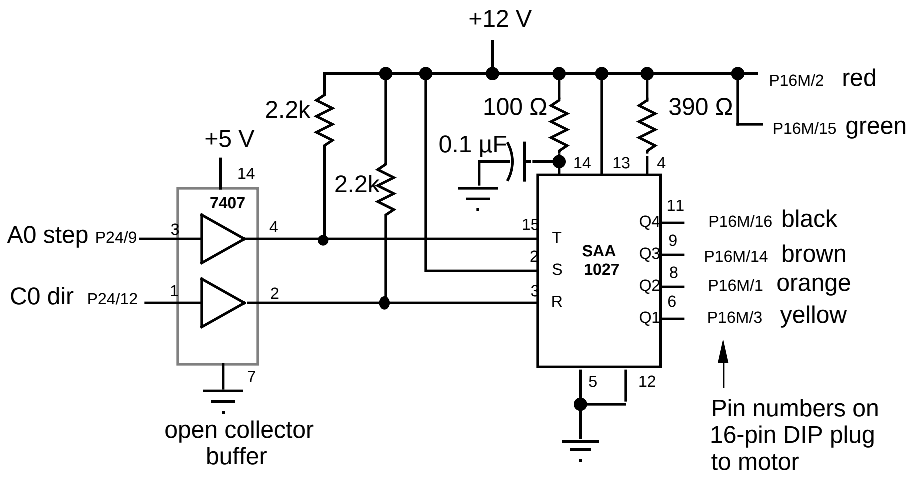


Fig. 8.1

You can now single step the motor in either direction.

```
OUT &H306, N          N=0 or 1 sets direction
OUT &H300, N          N=0 sets STEP low, N=1 sets STEP high .
```

Therefore, executing the line

```
OUT &H300, 1 : OUT &H300,0 : OUT &H300, 1
```

in GWBASIC, direct mode generates a single step pulse: high → low → high

Execute four steps in both directions to verify the operation of the SAA1027. Use the DVM to look at Q1 through Q4 each time. Remember, if Qn ≈ 0V, that winding is ON.

2. Using a Counter to Step the Motor

Sending out the signals for each step from software is one way of doing things, but it does tie up the computer. Instead, we can make a controller which can be told to generate N pulses at frequency *f*.

On the prototyping board, wire up this circuit:

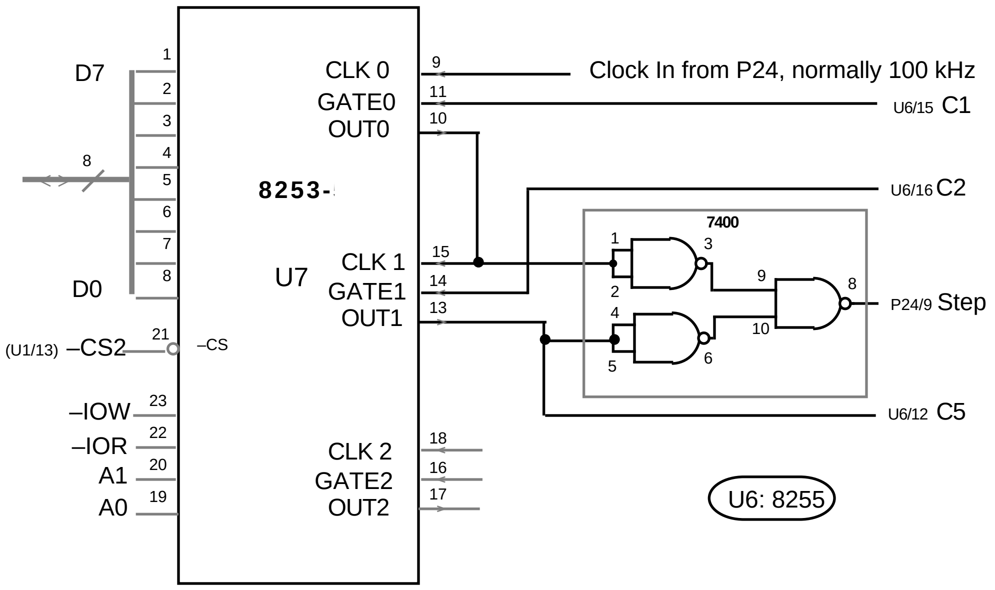


Fig. 8.2

Counter 0 sets step rate by dividing CLOCK IN. Counter 1 counts pulses to be sent out C1 and C2 control counters 1 and 2, C5 senses when counter 1 is finished. The following QBASIC program controls the previous circuit. Clock IN = 100 KHz. You must understand the program!

STEPPER.BAS

```
'Stepping Motor Driver Program

      out &h307,&h8a              'configure 8255
      out &h30B,&h34              'counter 0 to MODE 2
      out &h30B,&h72              'counter 1 to MODE 1

      portc!=&h306                  'name ports and counters for
      cnt0!=&h308
      cnt1!=&h309
```

```
top:    cls:print 360/7.5*20;" steps per turn"
       out portc!,0                     'inhibits count until rising edge

tinp:   input"Time between steps in ms (.02<T<640)";ts
       if (ts<.02 or ts>640) then
         print"ERROR - time out of allowed range"
             goto tinp
        else ts=int(ts*100):ts!=ts
                     print "ts!=";ts!,"&H";hex$(ts!)
             msb!=int(ts/256 )
                     print "msb!=";msb!,"&H";hex$(msb!)
             lsb!=ts-256*msb!
                     print "lsb!=";lsb!,"&H";hex$(lsb!)
             out cnt0!,lsb!              'loads time into counter 0
             out cnt0!,msb!
             out portc!,2                'starts clock at step freq, C1=1

       end if

sinp:   input"Number of steps (1<S<64000)";nstep!
       print "nstep!=";nstep!,hex$(nstep!)
       if (nstep!<1 or nstep!>64000) then
             print"ERROR - steps out of allowed range"
             goto sinp
        else msb!=int(nstep!/256)
                if msb!<0 then msb!=0
                print "msb!=";msb!,hex$(msb!)
             lsb!=nstep!-256*msb!
                print "lsb!=";lsb!,hex$(lsb!)
             out cnt1!,lsb!              'loads number of steps into
             out cnt1!,msb!              'counter 1
        end if

dn:     input "Direction ( 1 or 0)";c0
       if (c0<>1 and c0<>0) then print "error 1 or 0 only...":goto dn
       out portc!,2+c0

       print "Hit any key to start counting..."
aa:             a$=inkey$:if a$="" then goto aa

       out portc!,6+c0                     'starts count since C1=1 & C2=1
       print "counting...."
dd:     done=inp(portc!):if (done and 32)=0 then goto dd
       'out portc!,2+c0                      'ends count since  C2=0
       beep:print "count complete..."

       input "go again with new values";agn$
       if (agn$="y" or agn$="Y") then goto top
       end
```

Run STEPPER.BAS and verify that RATE is correct for several frequencies. Connect the new stepping controller circuit to the driver circuit on page 2 by taking the STEP input (pin 1 of the 7407) from pin 9 of the 24 pin jumper. DIR will still be set by C0 (pin 12 of the 24 pin jumper).

Run STEPPER.BAS again. Verify single step operation in both directions by looking at Q1 through Q4 with the DVM.

Verify multiple step operation by executing one (or more) full turns of the shaft (1 turn = 960 steps) in both directions at 200 steps per second.

Try doing full turns at different step rates to identify the maximum step rate at which the motor will function.

LAB 9

Stepping Motors (2)

Today we will make a few improvements to the existing controller circuit and add position feedback using an optocoupler (data sheet at end of lab). Then the computer is used to run a lengthy and repetitive experiment.

First copy STEPPER1.BAS (an improved version of STEPPER.BAS from lab 8) and STEPPER5.BAS onto your work disks and get listings for your writeup. These will only work with QBASIC.

Next, make three changes to improve the controller circuit:

1. Connect pin 2 of the SAA1027 to +12V to reduce noise pickup.

2. Take counter 0 in from counter 2 out rather than from 100KHz. Feed 1 MHz into counter 2 in and tie gate 2 to +5.

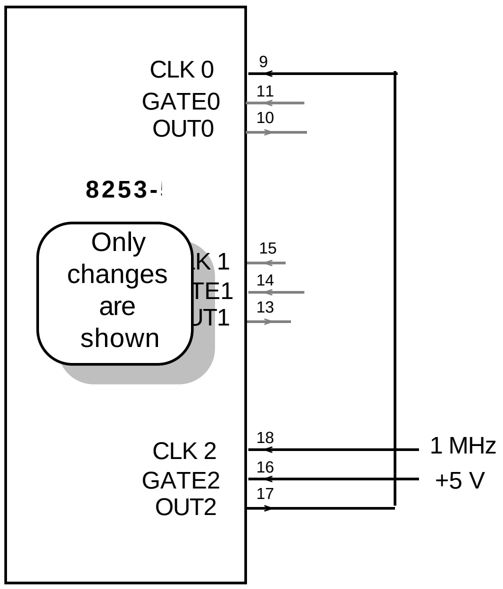


Fig. 9.1

3. Wire up new DONE circuit on the prototyping card.

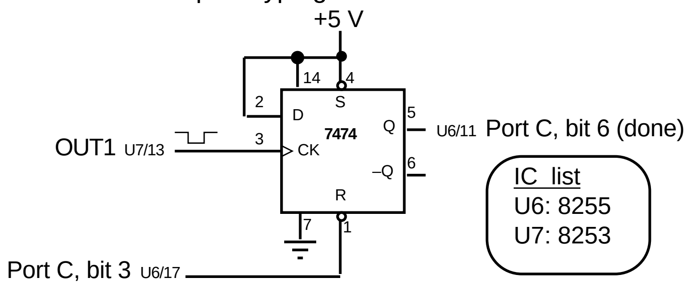


Fig. 9.2

Test this with STEPPER1.BAS and verify that the program now signals DONE properly at low step rates.

Now we can connect the optocoupler

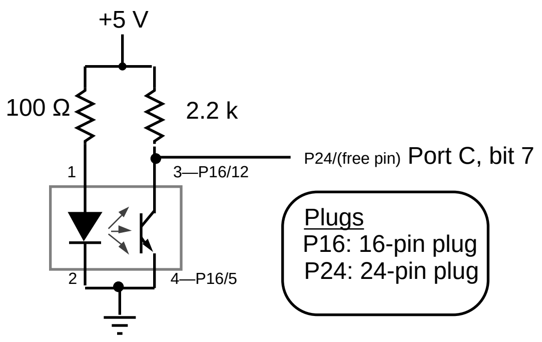


Fig. 9.3

With the slotted disk removed, measure the voltage at pin 3 with the optocoupler open and blocked. Looking at port C, bit 7 output on the scope and quickly flicking a wire through the optocoupler gap, *estimate* the response time and sketch the waveform.

Run STEPPER5.BAS, selecting single frequency operation (S) for a number of frequencies to check the operation.

Try to find frequencies at which the motor:

0. doesn't move
1. moves, but with errors
2. moves correctly

Once you are convinced that everything is working properly, set the program to loop (L). The results will be printed out for you tomorrow morning. Include these in your writeup and comment on the frequency response of the stepping motor.

Questions:

Comment on the changes between STEPPER.BAS and STEPPER1.BAS. Describe the operation of STEPPER5.BAS. Can you improve the efficiency of STEPPER5.BAS?

While STEPPER5.BAS is running you may want to use the time to plan your project.

LAB 10

I/O Device Service: Polling vs Interrupts

In this lab we wil[illegible]k at two different ways of servicing an I/O device: polling using an INP instruction, and hardware interrupts. The polling mode will be tried in both GWBASIC and QBASIC, while the interrupt driven program is available only as a compiled PASCAL program.

First, unwire the breadboard circuitry, and on the prototyping board, unwrap and remove all connections to the 8253 ports (pins 9,10,11,13,14,15,16,17,18) as well as all connections to the 74LS74 and 74LS00.

Connect the interrupt request circuit as shown in Fig 10.1

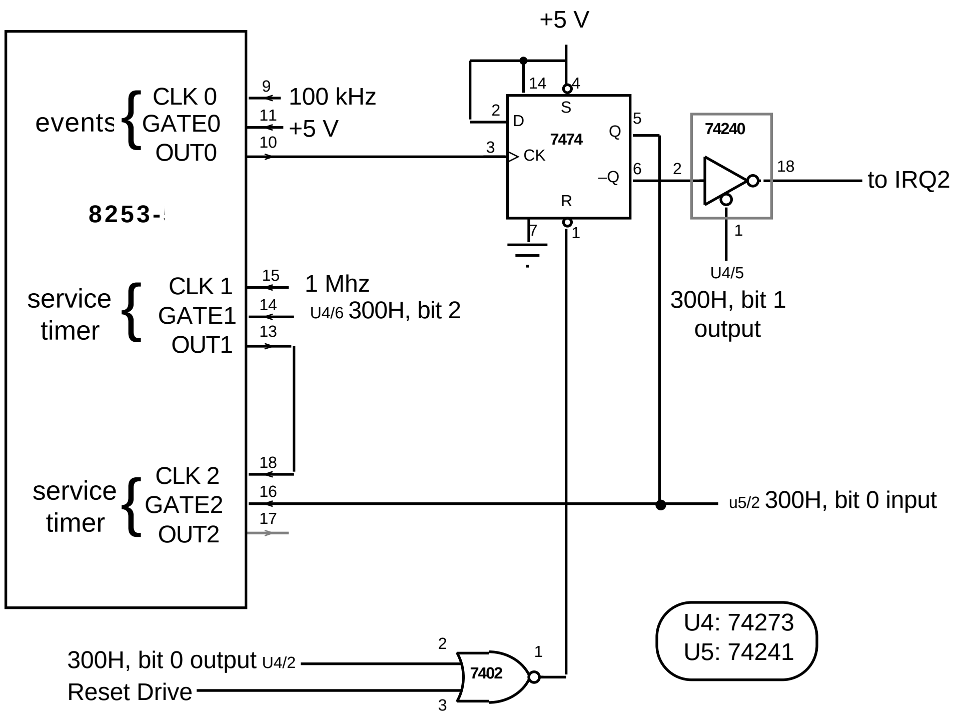


Fig 10.1

Requests for service are triggered by one of the 8253 counters dividing a 100 kHz clock. While these are repetitive requests, they are asynchronous with respect to the program execution. The time between setting requests must be longer than the time between polls in polling mode. Once a request is set, the polling line (and IRQ2 in interrupt mode) is activated, and another 8253 counter begins counting clock cycles (generated from 1 MHz by the third counter) until the service routine resets the request. The service routine also obtains the elapsed time between request and service from the second counter, and increments the corresponding element in a 600-element array. The main program loop continuously updates this array onto the screen.

NOTE: Run GRAPHICS from DOS so graphic screen dumps can be done later. If you are printing to an HP DeskJet, type `GRAPHICS DESKJET`.

POLLER.BAS

```
10 REM                 Poller Program
15 KEY OFF
20 DIM STIME%(601)
30 OUT &H307,&H8A                     :REM configure 8255
40 OUT &H30B,&H34                     :REM counter 0 to MODE 2
50 OUT &H30B,&H74                     :REM counter 1 to MODE 2
60 OUT &H30B,&HB0                     :REM counter 2 to MODE 0
70 PORT%=&H300
80 CNT0%=&H308
90 CNT1%=&H309
100 CNT2%=&H30A
105 CLS
110 INPUT "Time Increment in us (2<=T<=64000)";TINC
120 IF (TINC<2 OR TINC>64000!) THEN 110
130 MSB1=INT(TINC/256)
140 LSB1=INT(TINC-MSB1*256)
150 OUT CNT1%,LSB1
160 OUT CNT1%,MSB1
170 INPUT "Event rate in Hz (R>2)";IRATE
175 IF IRATE<2 THEN GOTO 170
180 IINC=INT(100000!/IRATE)           :REM clk1=100kHz
190 MSB0=INT(IINC/256)
200 LSB0=INT(IINC-256*MSB0)
210 OUT CNT0%,LSB0
220 OUT CNT0%,MSB0
250 BIT1%=2                   :REM Disables Interrupt Request line
260 PRINT"Hit any key to begin....";
270 A$=INKEY$:IF A$="" THEN 270
280 PRINT"initializing..."                    :REM wait to begin
290 BIT0%=1:BIT%=BIT0%+BIT1%                  :REM bit0=clear flip flop
300 OUT PORT%,BIT%                            :REM strobe reset low to clear
310 BIT0%=0:BIT%=BIT0%+BIT1%                  :REM flip flop
320 OUT PORT%,BIT%
330 BIT2%=4                                    :REM bit2=GATE for counter
340 BIT%=BIT0%+BIT1%+BIT2%
345 REM clear histogrammer
350 FOR X=0 TO 601:STIME%(X)=0:NEXT X
360 MAXPLOT=0:ENDER=0
370 OUT PORT%,BIT%                               :REM starts events
400 REM Wait and Plot results
410 SCREEN 2:CLS:WINDOW(-50,200)-(650,-10):PSET(0,0):PSET(600,0)
411 LOCATE 1,1:PRINT"working....type 'e' to end"
412 LOCATE 2,1:PRINT IRATE;"Hz";
414 LOCATE 3,1:PRINT TINC;"us";
415 FOR X=0 TO -10 STEP -1
416 PSET(0,X):PSET(100,X):PSET(200,X):PSET(300,X)
417 PSET(400,X):PSET(500,X):PSET(600,X)
418 NEXT X
420 OFFSET=0
430 FOR X=0 TO 600
450     PSET(X,STIME%(X))
480     IF STIME%(X)>MAXPLOT THEN MAXPLOT=STIME%(X)
```

```
490     A$=INKEY$:IF (A$="E" OR A$="e") THEN ENDER=1:BEEP
500     IF (ENDER<>1 AND MAXPLOT<150) THEN GOSUB 2000
510 NEXT X
520 IF (ENDER<>1 AND MAXPLOT<150) THEN GOTO 430         :REM keep looping
530 REM replot histogram as a bar graph
540 FOR X=0 TO 600
545 FOR Y=0 TO STIME%(X)
550     PSET(X,Y)
555 NEXT Y
560 NEXT X
570 BEEP
580 END
2000 REM Polling Routine...............
2010 EVENT=INP(PORT%)                    :REM Q of flip flop to in BIT0
2020 IF (EVENT AND 1)=0 THEN GOTO 2110  :REM means no event occurred
2030 BIT2%=0:BIT%=BIT0%+BIT1%+BIT2%
2040 OUT PORT%,BIT%                      :REM stops counter
2045 LSB2=INP(CNT2%):MSB2=INP(CNT2%):COUNTED=256*MSB2+LSB2
2050 XCOUNT=65536!-COUNTED
2060 IF XCOUNT>600 THEN XCOUNT=601          :REM eliminates out of range error
2070 STIME%(XCOUNT)=STIME%(XCOUNT)+1
2080 OUT CNT2%,0
2085 OUT CNT2%,0                            :REM reset counter
2090 BIT0%=1:BIT%=BIT0%+BIT1%+BIT2%:OUT PORT%,BIT%      :REM reset flip flop
2100 BIT0%=0:BIT2%=4:BIT%=BIT0%+BIT1%+BIT2%:OUT PORT%,BIT% :REM restarts events
2110 RETURN
```

INTR.COM

INTR.COM (continued)

Physics 430

Lab 10: GPIB

Part 1. Purpose: Set up a GPIB interface between a Function Generator and the computer, control operation of FG from computer and read settings of FG into the computer and print them on the screen.

•Installing the GPIB hardware and software

1. Verify that the GPIB card is installed in the computer and that a cable is attached between the computer's GPIB interface and that of the function generator. You can identify the GPIB card by the distinctive GPIB connector plug which is D-shaped and has contacts on the upper and lower inside faces like a Centronics printer connector.

2. Set the GPIB address of the FG from the front panel. When the FG is turned on it momentarily flashes the GPIB address on the screen. You may have missed or you may wish to change the address. Press SHIFT-MENU and then press > several times until the I/O menu appears. Then press v to obtain the HPIB ADDR selection. Press v again and set the address to 2. Then press ENTER to return to operational mode.

Question: Will the address change when the unit is turned off and then turned on again? Try some experiments to find out. If the address seems to survive power down does the unit have to be plugged in for the address to survive?

3. Confirm that the National Instruments GPIB device driver is loaded into the DOS operating system. From the DOS prompt type CONFIG.SYS and look for the line containing DEVICE=GPIB.COM.

4. We will be using the "Universal Language Interface" (ULI) because it is simpler to install and is purported to work with all languages. National Instruments recommends it own "standard" software which they claim has better performance. However, the ULI was Hewlett Packard's original software interface. It is commonly available and seems more straight-forward to use.

The ULI requires a "TSR" to be used. You have ULI.COM in the subdirectory AT-GPIB\ULI. You can also find it in the file server's directory PHYS430\ULI\ULI.COM. Because some sample programs are also in that server's directory you should connect to it. Make sure the networking software is loaded and log into the workgroup MASTER by typing NET LOGIN from the DOS prompt. Connect to the PHYS430 directory by typing

`NET MAP P: STN0_PC/PHYS430` (note the space after :).

Change to the ULI directory on the server.

```
P:
CD ULI
```

Load the Universal Language drivers by typing ULI.COM. This is a "Terminate and Stay Resident" (TSR) program which quits, but leaves the driver software in memory after quittig. The program should respond with a successful message and then quit, leaving the ULI driver installed.

Make C:\WORK the default directory.

```
C:
CD WORK
```

If you stay in P: the software may not work because the default drive will be write-only and some languages object to opening the pseudofile GPIB0 for output in this case.

4. Get the program HPIB1.BAS from the server. This initializes the GPIB bus and gets some status information. I suggest using GWBASIC at first because it allows interactive use of the interface. GWBASIC may be in a subdirectory of the BASIC directory. (The BASIC directory is in the file system's search path but GWBASIC may not be. To make access easier, you can either modify the system path or move GWBASIC up into the BASIC directory.)

```
GWBASIC
load "P:HPIB1.BAS"
list
```
.....(listing of program. Try to understand it.)

```
run
```

(You should get three integers corresponding to the status of the last call, the error message, if any, and the number of bytes transferred in the last call.)
Notice that the computer communicates with the interface through writes and reads to a file named GPIB0. This file doesn't really exist. The driver software simulates the existence of this file so that any language with file input/output capability can access the interface without modifying the language interpreter or compiler itself. (National Instrument's software requires subroutines be installed with each language that is to be used.)
The basic file identifier 1 is used to write to the interface: `PRINT #1`.
The file identifier 2 is used to read information from the interface: `INPUT #2`.

The string in double quotes is sent to the interface. The string "`ABORT`" sends "interface clear", IFC, to the interface to reset it. The string "`STATUS`" askes the interface to send back three status integers. These are received by the `INPUT` statement and put into three integer variables.

•Controlling the Function Generator Remotely

Try putting the function generator into remote mode by the following
command. Type it in directly without a line number so that it executes immediately. In QBASIC or VBDOS you can enter immediate commands in the immediate window after having opened the GPIB0 pseudofiles by running HPIB1.BAS.

`PRINT #1, "REMOTE 2"`   (Either upper case or lower case is acceptable.)

Then type carriage return. If it doesn't work first time, try it again.

`"REMOTE 2"` sets the device with GPIB address 2 into remote mode. (Don't confuse GPIB address 2 with the BASIC file identifier 2!)

The function generator should display a little "RMT" on its front panel indicating it went into remote mode.

> If the FG beeps and ERROR appears on the front panel, something went wrong. Review the procedures above or ask an instructor if necessary. Clear the error by pressing SHIFT <, V, ENTER on the front panel or by typing
> `PRINT #1, "OUTPUT 2;*CLS"`

`"OUTPUT 2"` indicates that the commands following in the same string will be sent on to the device with GPIB address 2. Try changing the frequency of the function generator as follows:

```
PRINT #1, "OUTPUT 2;FREQ 2.0E3"
```

FREQ is an abbreviation for FREQUENCY. Either FREQ or FREQUENCY is acceptable but not any other abbreviation; for example, FREQUE should not work. FREQ is a SCPI command for the function generators. (There is another way of setting up the HP FG with the APPLY command. This does not seem to conform to SCPI and will probably not work on any other brand of FG.)

EXERCISE: Use the Function Generator manual to find the commands to set the peak-to-peak voltage to 1 volt and the waveform to a triangle wave. Write these commands into the program by inserting line numbers in front of each print statement. Then save it to your work area using the command

```
SAVE "C:\WORK\GPIB1xx.BAS",A
```
(Replace xx with your initials)

•Reading values from the FG.

Now we are going to interrogate the device to verify its settings.

First instruct it to send a string containing the frequency back to the computer:

```
PRINT #1, "OUTPUT 2;FREQ?"
PRINT #1, "ENTER 2"
```

Notice the interrogation is done by sending a command followed by a question mark.

Get the data that it has sent with an INPUT from the GPIB interface into the string variable RD$:

```
INPUT #2, RD$
```

Now print it on your screen

```
PRINT RD$
```

A floating point number should be printed. Note that this is a string and can't be used in calculations. If you need a numeric variable use the VAL function of BASIC.

```
PRINT VAL(RD$)
```

•Reading multiple values

There are a few quirks to negotiate if you need to read several values from remote instruments. The best way to read in several values is to include all requests in a single PRINT command.

```
PRINT #1, "OUTPUT 2;FREQ?;VOLT?"
PRINT #1, "ENTER 2"
INPUT #2, RD$
```

RD$ contains two values separated by a semicolon.

If you need to request a measurement in a subsequent PRINT statement the results can be erratic. The INPUT statement may have detected and end-of-file condition on the pseudofile GPIB0 in which case the next INPUT on the same file will return an error message such as "READ PAST END". You can avoid this by checking for end-of-file on file 2 before each ENTER command.
Close and reopen the pseudofile if end-of-file has been detected.

```
IF EOF(2) THEN CLOSE(2) : OPEN "GPIB0" FOR INPUT AS #2
PRINT #1, "OUTPUT 2;FREQ?;VOLT?"
PRINT #1, "ENTER 2"
INPUT #2, RD$
```

Neither the CLOSE nor the OPEN statement will be executed if EOF(2) is false.

Part 2. Purpose: Use the oscilloscope to visualize the Function Generator output and record measurements under computer control.

1. Set up the Oscilloscope to use GPIB address 1. That is done with the UTILITY menu function. Choose CONFIG I/O with the bottom left-hand button.

2. Put the scope in remote mode by typing, in GWBASIC

```
PRINT #1, "REMOTE 1"
```

Set the horizontal time scale:

```
PRINT #1, "OUTPUT 1;HORIZ:MAIN:SCALE 1E-3"
```

Set the vertictal scale for channel 1.

```
PRINT #1, "OUTPUT 1;CH1:SCALE 1"      (1 volt/div)
PRINT #1, "OUTPUT 1;CH1:POS 0"        (Centre it on the screen)
```

The commands are documented in the Tektronix *TDS340 Programmer Manual*, Sec 2.

To measure the rms voltage on channel 1 the following commands are sent.

```
PRINT #1,"OUTPUT 1;MEASU:IMM:SOURCE CH1"
PRINT #1,"OUTPUT 1;MEASU:IMM:TYP RMS"
PRINT #1,"OUTPUT 1;MEASU:IMM:VAL?"
PRINT #1,"ENTER 1"
INPUT #2, RD$
PRINT RD$
```

Exercise: Connect FG output to CH1 and the FG synch output to CH2. Write a program to ask the user to specify a frequency and a peak-to-peak voltage then set the FG to those values and display the output and synch signal on the scope using reasonable scales. Measure V_{pp} of both signals and print on the computer screen.

If you do this correctly, type the following command:

```
PRINT #1,"OUTPUT 2;DISP:TEXT 'GOOD JOB' "
```