
PIC'n Up The Pace

PIC16/17 MICROCONTROLLER
APPLICATIONS GUIDE

FROM

SQUARE 

DAVID BENSON

VERSION 1.0

SERIAL COMMUNICATION

Since PIC16's have few pins, serial communication is, more often than not, the best way for the microcontroller to communicate with peripheral chips on the same board, or between one PIC16 and another via a short cable. Communication between a PIC16-controlled device and the outside world is typically done serially (via RS-232 for example).

If you have not been exposed to serial communication, it involves taking data which is in a parallel format, converting it to serial format for transmission down a single (data) wire and converting the data back to parallel format at the receiving end. Sending 8 bits of data in parallel requires 8 wires for data. Sending 8 bits serially requires 1 wire for data.

Serial communication involves varying numbers of wires for the various functions. Usually the count does not include ground. In this book, we will not worry about the number of wires and we won't use anyone's protocol or standard. We will just concentrate on understanding what's going on and getting the job done.

The next chapter covers shift registers. They come in two flavors--serial-in, parallel-out and parallel in, serial-out. Getting a PIC16C84 to talk to each type is a good way to get started with serial communication.

The following chapter involves interfacing a PIC16C84 and 93C46 serial EEPROM. This is another form of serial communication, the design of which is dictated by the 93C46's pin compliment and internal workings.

Next, we'll get one PIC16C84 talking to another PIC16C84. Several other examples will follow. By the time you finish, you should feel comfortable with simple serial communication.

Note that the clock signal in the examples is irregular. Timing diagrams for the serial peripheral devices used as examples show a nice symmetrical clock signal. This is not required. It also is not possible in many applications.

SHIFT REGISTERS

Shift registers are used to convert serial data to parallel or vice versa. "Talking" to shift registers is a good way to get started learning about serial communication. Shift registers are useful as parallel output and input ports which may be interfaced with a PIC16 serially.

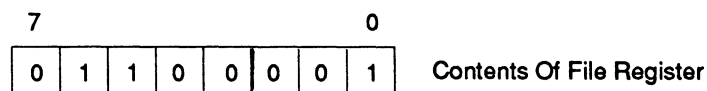
For our first example, we will use a 74HC164 which is a serial in, parallel out shift register. The "in" vs. "out" designations are with respect to the shift register. The object is to create and send 8 bits of data to the shift register serially and look at its outputs via DVM, LED's or whatever to see if the byte got there successfully.

The PIC16C84 will be used in this example.

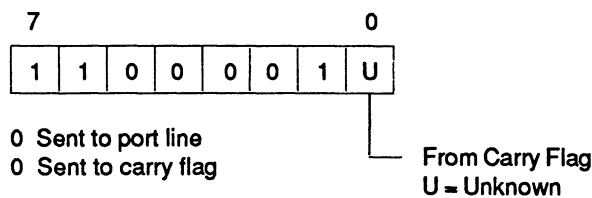
The data to be sent from the PIC16C84 is initially defined and stored in a file register as 8 bits in parallel format. In order to convert them to serial format, the 8 bits in the file register are shifted left (RLF) one at a time. Bit 7 of the file register is sent on its way via a single output port line after each shift. The most significant bit is sent first because that is what the 74HC164 expects.

Example:

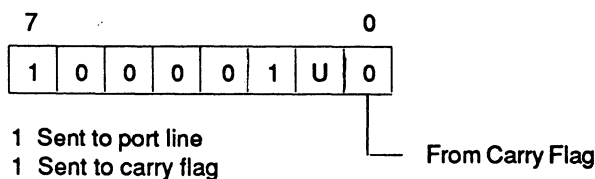
Initial 8 Bits



Shift Left (RLF) First Time



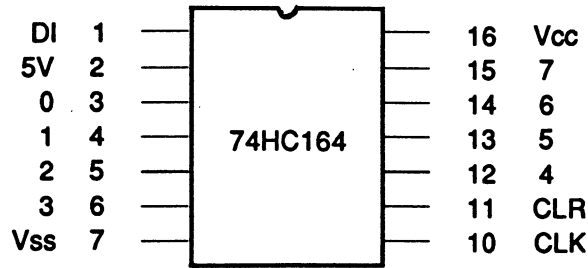
Shift Left Second Time (RLF)





The 8 bits are marched out one at a time in succession.

SERIAL IN, PARALLEL OUT SHIFT REGISTER - 74HC164

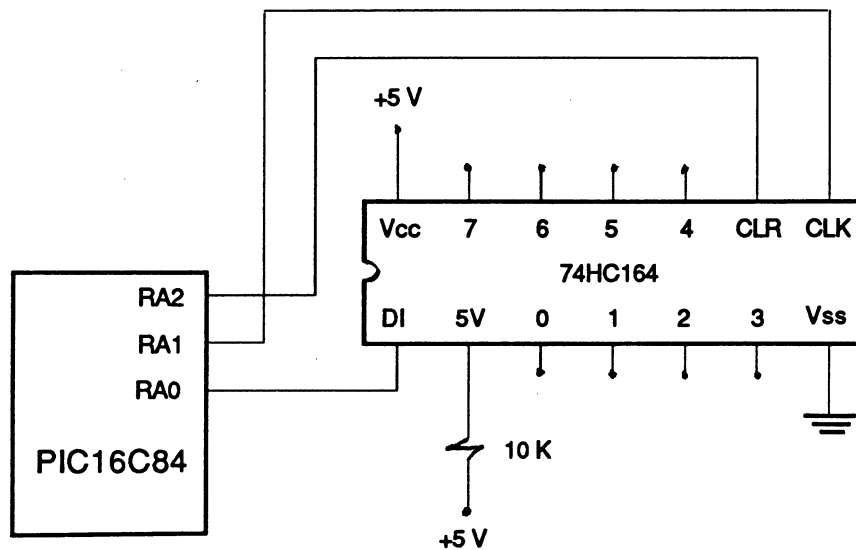
Let's talk about the hardware.



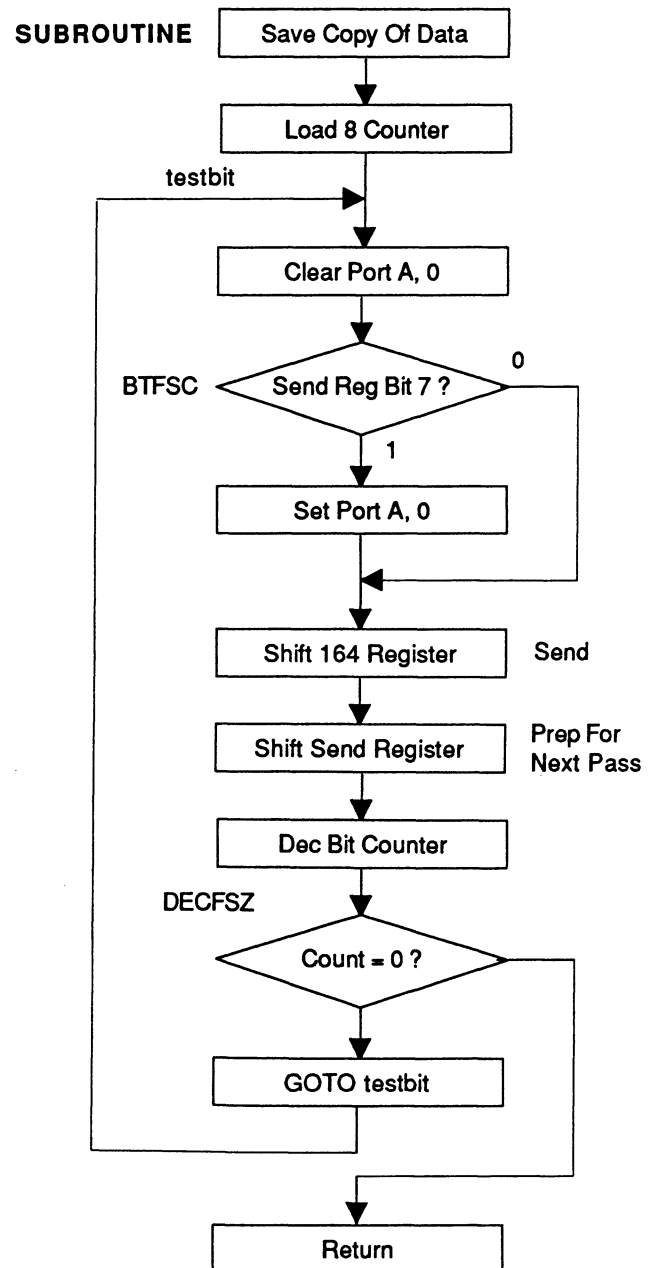
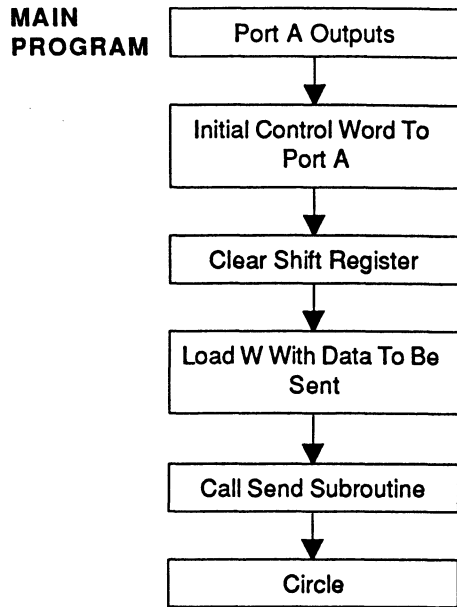
Notice the 74HC164 has three control lines.

- Serial Input
- Clear  Clears outputs to 0's (normally HI)
- Clock  Shifts data through bit 0 toward bit 7

To move data into the shift register, the first data bit is presented to the input. Then it is shifted in. The second bit is presented and shifted, and so on. Simple!



Here is how the complete process of sending one byte of data works:



The shift register outputs are cleared on initialization as part of the power-on reset housekeeping so that all outputs will be low to start with.

The assembly language program for doing all this is a subroutine (ser_out). It is a code module which may be modified, if necessary, to reflect port pin assignment, etc. and used for your own future projects.

```

;=====74HC164.ASM=====4/25/97==
        list    p=16c84
        radix   hex
;-----
;      cpu equates (memory map)
porta   equ    0x05
status  equ    0x03
sendreg equ    0x0c
count   equ    0x0d
trisa   equ    0x85
;-----
;      bit equates
rp0     equ    5
;-----
        org    0x000
;
start   bsf    status,rp0  ;switch to bank 1
        movlw  b'00000000' ;outputs
        movwf  trisa
        bcf    status,rp0  ;switch back to bank 0
        movlw  0x04        ;0000 0100
        movwf  porta      ;control word
        bcf    porta,2     ;clear shift register
        bsf    porta,2
        movlw  0x80        ;number to be sent
        call   ser_out     ;to serial out subroutine
circle  goto   circle     ;done
;-----
ser_out movwf  sendreg     ;save copy of number
        movlw  0x08        ;init 8 counter
        movwf  count
testbit bcf    porta,0     ;default
        btfsc  sendreg,7   ;test number bit 7
        bsf    porta,0     ;bit is set
shift   bsf    porta,1     ;shift register
        bcf    porta,1
rotlft  rlf    sendreg,f   ;shift number left
        decfsz count,f     ;decrement bit counter
        goto  testbit     ;next bit
        return            ;done
;-----
        end
;-----
;at blast time, select:
;      memory unprotected
;      watchdog timer disabled (default is enabled)

```

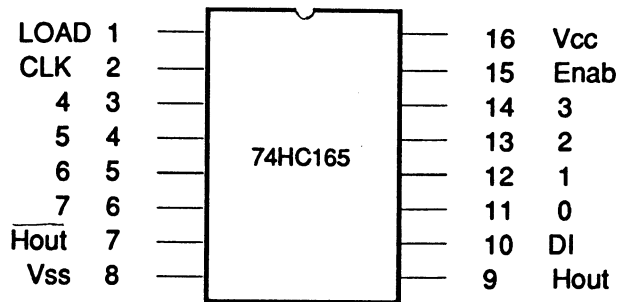
```

;      standard crystal (using 4 MHz osc for test) XT
;      power-up timer on
;=====

```

PARALLEL IN, SERIAL OUT SHIFT REGISTER - 74HC165

Bringing 8 bits of data into a PIC16 serially is done in a similar way. We will use a 74HC165 parallel in, serial out shift register.

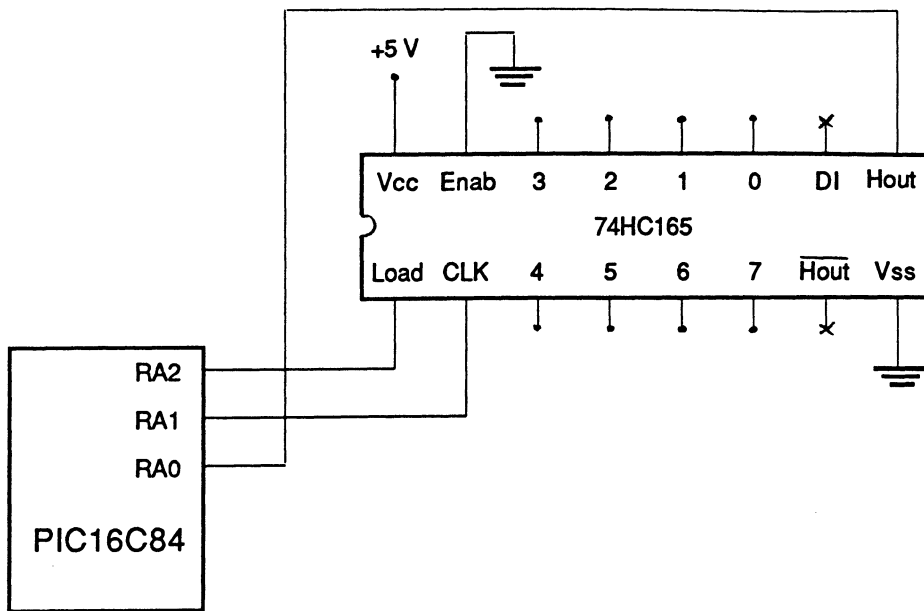


Notice the 74HC165 has three control lines.

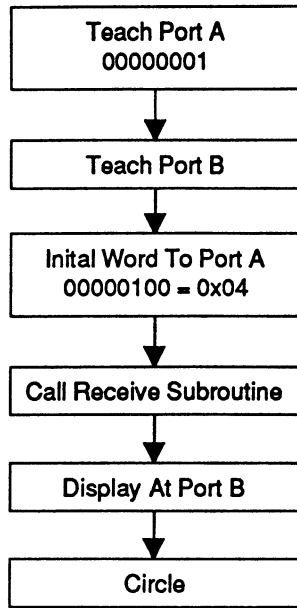
- Serial Output
- Load Loads 8 bits into shift register
- Clock Shifts data MS bit first

The 8 bits of data presented to the shift register are latched in using the load control line. This must be done so that if the input lines are changing state with time, only the data latched in at one instant in time will be transmitted to the PIC16. The 8 data bits are shifted out most significant bit first.

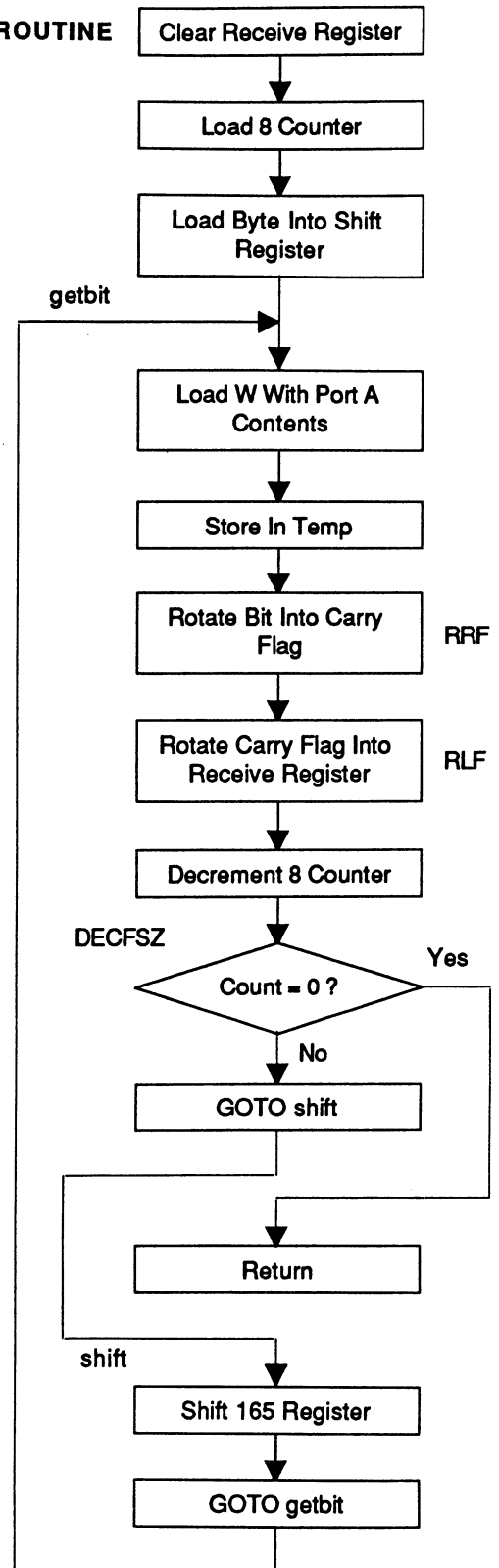
Again, one PIC16C84 port pin is used for serial in. It is convenient to use bit 0 for serial input. The program looks at the port as a whole, rotates bit "0" into the carry flag, and rotates the contents of the carry flag into the least significant bit of the file register assigned to receive the incoming data. This process is carried out for each of the 8 bits. Notice that the first bit is available at the serial output line immediately after the data is latched. Shifting 7 times (not 8) is required to access the remaining bits.



MAIN PROGRAM



SUBROUTINE



Again, this is code includes a subroutine which you may use in the future.

```
;=====74HC165.ASM=====4/25/97==
    list    p=16c84
    radix   hex
;-----
;    cpu equates (memory map)
porta    equ    0x05
portb    equ    0x06
status   equ    0x03
rcvreg   equ    0x0c
count    equ    0x0d
temp     equ    0x0e
trisa    equ    0x85
trisb    equ    0x86
;-----
;    bit equates
rp0      equ    5
;-----
    org    0x000
;
start    bsf    status,rp0    ;switch to bank 1
        movlw  b'00000001'    ;bit 0 = input
        movwf  trisa
        movlw  b'00000000'    ;outputs
        movwf  trisb
        bcf    status,rp0    ;switch back to bank 0
        movlw  0x04           ;0000 0100
        movwf  porta         ;control word
        call   ser_in        ;to serial input subroutine
        movf   rcvreg,w      ;get data
        movwf  portb        ;display data via LED's
circle   goto   circle      ;done
;-----
ser_in   clrf   rcvreg       ;clear receive register
        movlw  0x08         ;init 8 counter
        movwf  count
        bcf   porta,2      ;load shift register
        bsf   porta,2
getbit   movf   porta,w      ;read port A
        movwf  temp        ;store copy
        rrf   temp,f       ;rotate bit into carry flag
        rlf   rcvreg,f     ;rotate carry flag into rcvreg
        decfsz count,f     ;decrement counter
        goto  shift
        return             ;done
shift    bsf   porta,1      ;shift 1 bit
        bcf   porta,1
        goto  getbit       ;again
;-----
    end
;-----
;note: the 74HC165 gets shifted 7 times
```

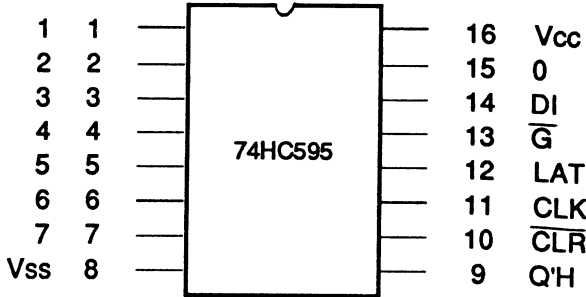
```

;-----
;at blast time, select:
;   memory unprotected
;   watchdog timer disabled (default is enabled)
;   standard crystal (using 4 MHz osc for test) XT
;   power-up timer on
;=====

```

SERIAL IN, PARALLEL OUT - 74HC595

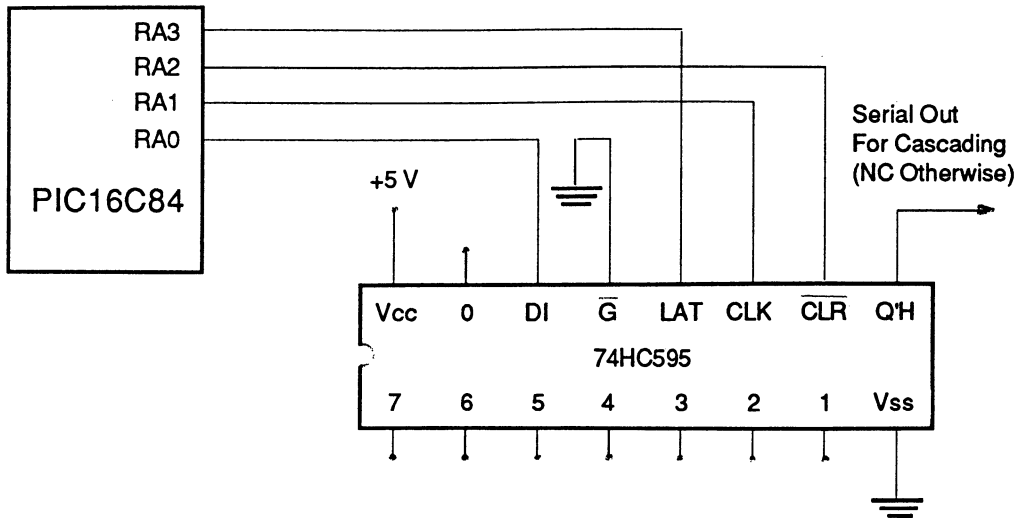
The 74HC595 is similar to the 74HC164. The 8 outputs of the 74HC164 will change state as data is shifted in. If the chip is being used as a parallel output port, this will not be a good thing. The 74HC595 has latches which hold the data presented at the output lines. New data may be shifted in while the outputs remain stable. Then the new data is latched in. This, of course, requires a 4th control line to latch data.



The 74HC595 has four control lines.

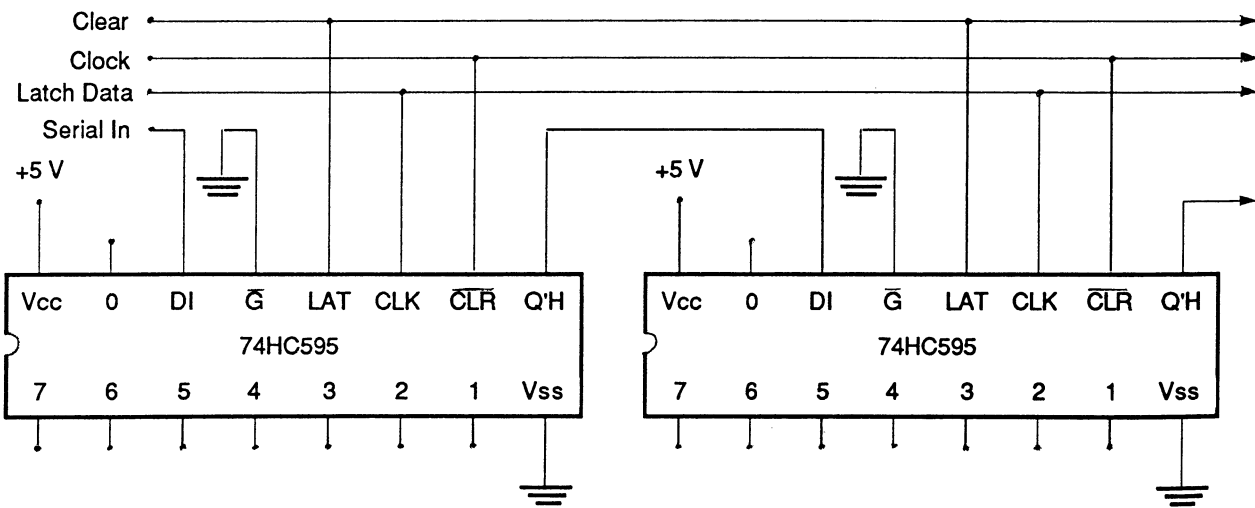
- Serial Input
- Latch Shift register contents to latches
- Clock Shifts data MS bit first
- Clear Clears shift register

Data is shifted in most significant bit first.

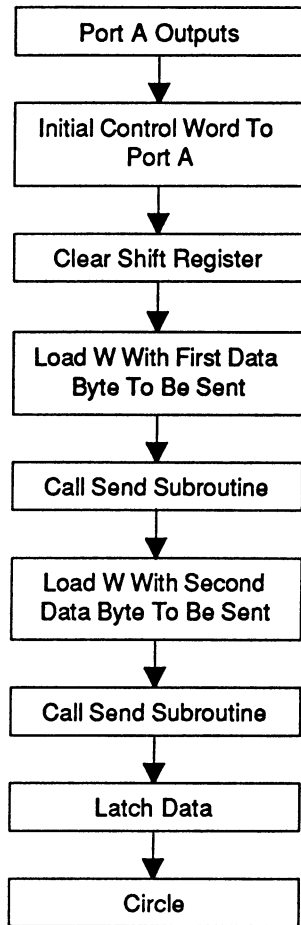


The 74HC595 has an output line designed for cascading two or more chips. 74HC595's may be cascaded by:

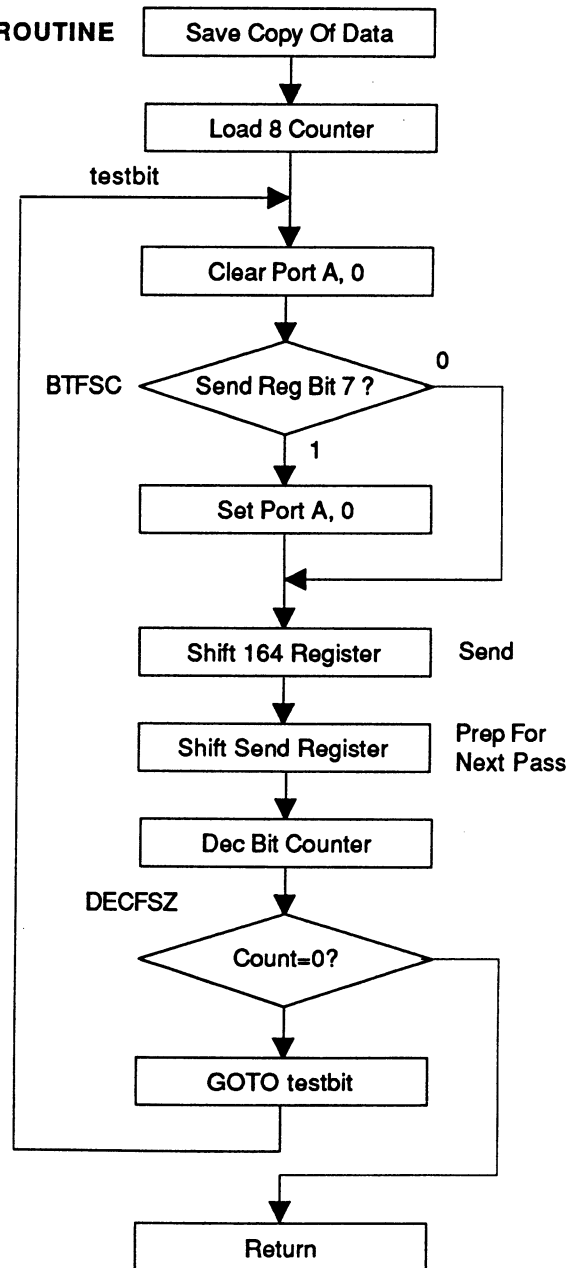
- Q'H serial output of first chip connected to serial input of second chip
- Connect shift register clear lines together
- Connect shift clock lines together
- Connect latch data lines together



MAIN PROGRAM



SUBROUTINE



```

;=====74HC595.ASM=====4/25/97==
    list    p=16c84
    radix   hex
;-----
;      cpu equates (memory map)
porta    equ    0x05
status   equ    0x03
sendreg  equ    0x0c
count    equ    0x0d
trisa    equ    0x85
;-----
;      bit equates
rp0      equ    5
;-----
    org    0x000
;
start    bsf    status,rp0    ;switch to bank 1
        movlw  b'00000000'    ;outputs
        movwf  trisa
        bcf    status,rp0    ;switch back to bank 0
        movlw  0x04          ;0000 0100
        movwf  porta        ;control word
        bcf    porta,2      ;clear shift register
        bsf    porta,2
        movlw  0x80          ;first number to be sent
        call   ser_out      ;to serial out subroutine
        movlw  0x0f          ;second number to be sent
        call   ser_out      ;to serial out subroutine
        bsf    porta,3      ;register contents to latches
        bcf    porta,3
circle   goto   circle      ;done
;-----
ser_out  movwf  sendreg      ;save copy of number
        movlw  0x08          ;init 8 counter
        movwf  count
testbit  bcf    porta,0      ;default
        btfsc  sendreg,7    ;test number bit 7
        bsf    porta,0      ;bit is set
shift    bsf    porta,1      ;shift register
        bcf    porta,1
rotlft  rlf    sendreg,f     ;shift number left
        decfsz count,f      ;decrement bit counter
        goto   testbit      ;next bit
        return              ;done
;-----
    end
;-----
;at blast time, select:
;      memory unprotected
;      watchdog timer disabled (default is enabled)
;      standard crystal (using 4 MHz osc for test) XT
;      power-up timer on
;=====

```

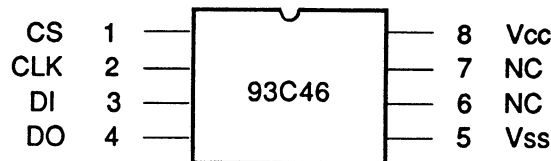
SERIAL EEPROMS

Serial EEPROMs come in three main flavors and a variety of sizes. The 93XXX devices are the easiest to interface to PIC16's (in my humble opinion). We will use the 93C46 (by Microchip and others) as an example.

The 93C46 is a small non-volatile memory peripheral chip. It is organized as 64 registers of 16 bits each. The programming voltage and write timing are developed on-chip. The self-timed write cycle takes about 10 milliseconds.

All communication with the 93C46 begins with sending 9 instruction bits. The first bit (MSB) is a logic "1" start bit. The remaining 8 bits may be an op code or an op code and address combination. If the operation is a write operation, 16 data bits follow the instruction bits, MSB first.

The 93C46 is available in an 8-pin DIP.



The control lines are:

- Serial data in
- Serial data out
- Clock
- Chip select



Some use rules are:

- 1) A register must be erased (all 1's) before it can be written to. The chip has a built-in auto erase cycle which takes place when a write is called for.
- 2) The chip select pin (CS) must be brought low for a minimum of 1 microsecond between consecutive instruction cycles to synchronize the internal logic of the device.
- 3) For read operations, a dummy "0" precedes the 16 data bits. Data is shifted out MSB first.

- 4) Completion of an erase cycle or write cycle to an individual memory location takes about 10 milliseconds. The serial data output (D₀) pin may also be used as a status pin during the self-timing phase of these operations to indicate the status of the device. On completion of erase or write, CS is brought low briefly. After that, D₀ will be low until the operation is complete. When D₀ goes high again, the device is no longer busy and is accessible for other operations.

The instructions are:

- Read a register
- Write to a register
- Erase a register
- Erase/Write enable (EWEN)
- Erase/Write disable (EWDS)
- Erase all registers (ERAL)
- Write all registers (WRAL) (with same data)

I haven't figured out why anyone would want to write the same data to all registers, but maybe you will.

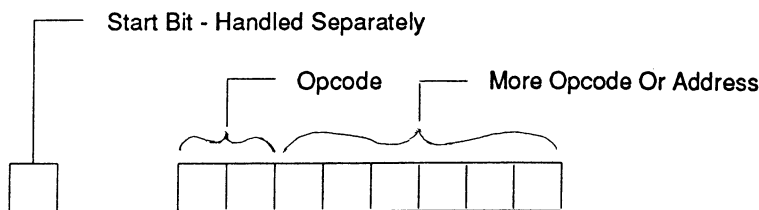
There are 6 address bits (to definite 64 register locations) contained in the instruction words that need them.

| Operation | Start Bit | Op Code | More Op Code or Address | | | | | | |
|-----------|-----------|---------|-------------------------|----|----|----|----|----|--|
| | | | A5 | A4 | A3 | A2 | A1 | A0 | |
| Read | 1 | 1 0 | A5 | A4 | A3 | A2 | A1 | A0 | |
| Write | 1 | 0 1 | A5 | A4 | A3 | A2 | A1 | A0 | |
| Erase | 1 | 1 1 | A5 | A4 | A3 | A2 | A1 | A0 | |
| EWEN | 1 | 0 0 | 1 | 1 | X | X | X | X | |
| EWDS | 1 | 0 0 | 0 | 0 | X | X | X | X | |
| ERAL | 1 | 0 0 | 1 | 0 | X | X | X | X | |
| WRAL | 1 | 0 0 | 0 | 1 | X | X | X | X | |

X = Don't care

Now we need to digest all this and figure out how to write some code to make the thing work.

One way to write a program to communicate with the 93C46 is to send the start bit as a separate operation which precedes sending the remaining 8 bits. Then the remaining 8 bits will fit into a file register. This file register is used as a working register to cook up instruction words:



The next consideration is what to do with the "X's", i.e., don't cares. Let's make them "0's". Now the instruction table looks like this:

| Operation | Hex Op Code | Op Code | | More Op Code or Address | | | | | |
|-----------|-------------|---------|-------|-------------------------|-------|-------|-------|-------|-------|
| ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- | ----- |
| Read | | 1 | 0 | A5 | A4 | A3 | A2 | A1 | A0 |
| Write | | 0 | 1 | A5 | A4 | A3 | A2 | A1 | A0 |
| Erase | | 1 | 1 | A5 | A4 | A3 | A2 | A1 | A0 |
| EWEN | 0x30 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| EWDS | 0x00 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| ERAL | 0x20 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| WRAL | 0x10 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

This method of putting "0s" in place of "X's" makes the instruction table look less intimidating. Further, there are now four hex opcodes we can use for four of the instructions to make life easier.

Next we need to deal with addresses in individual register operations. Perhaps the easiest thing to do is to dedicate a file register for holding the address prior to executing our serial routine. The routine can grab the address from there and move it to the working register (labeled "cook"). Don't worry about the upper 2 bits in the address register. For the address 00 0000 (binary), use 0x00. The range is 0x00 to 0x3F. In the "cook" register we can modify the upper 2 bits to make them an erase, read, or write op code. At that point, we have cooked up the complete instruction for the operation.

This example EEPROM serial communication program will be modular meaning a main program will call subroutines such as "read one register" or "write one register" which will, in turn, call other subroutines as needed.

To get started, we will need an erase one register subroutine, a write to one register subroutine, and a read one register subroutine. We will need to precede erase and write with an erase/write enable (EWEN) and follow with an erase/write disable (EWDS).

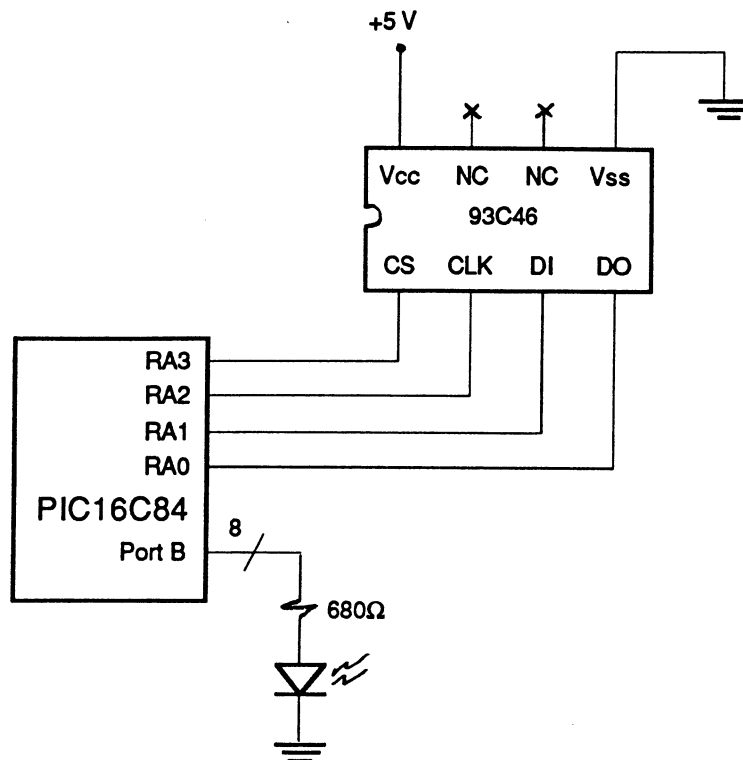
Notice that the start bit is sent as part of the code as needed (requiring 2 instructions) and that 8 bits of op code/address/data are sent at a time directly out of the "cook" register.

You can write an "erase all registers" routine on your own if you find a need for one.

These routines may be modified and used in your own programs.

DEMO CIRCUIT

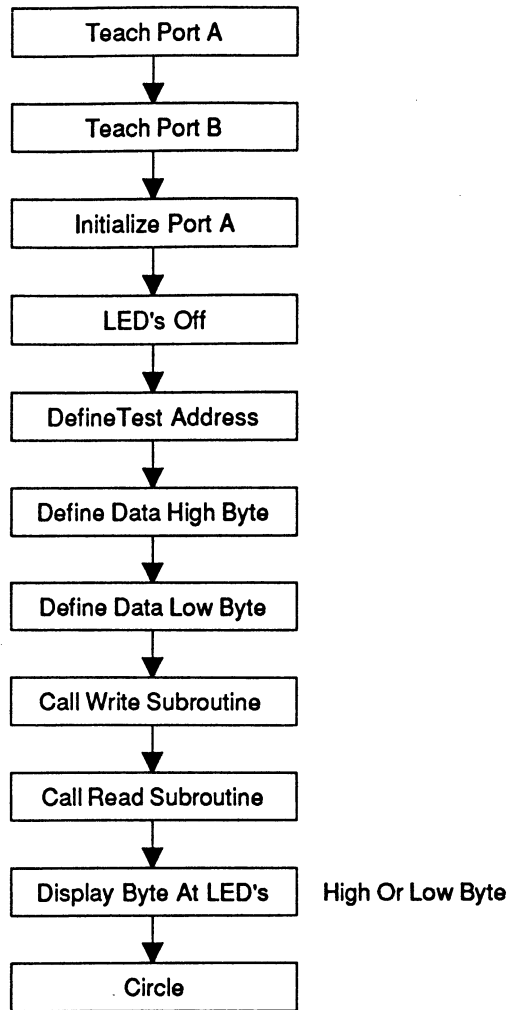
This circuit will be used to demonstrate interfacing a PIC16 to a 93C46 serial EEPROM:

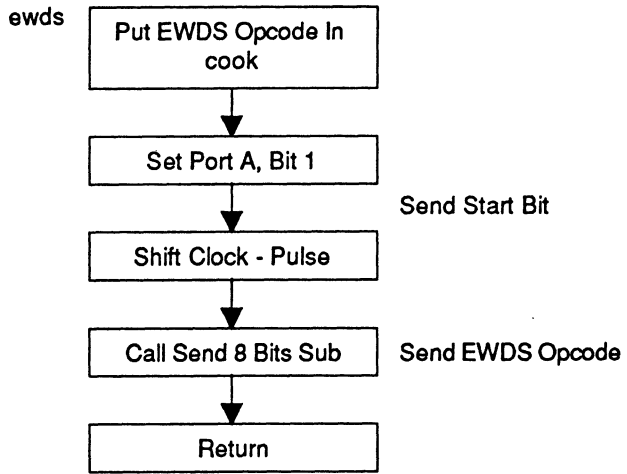
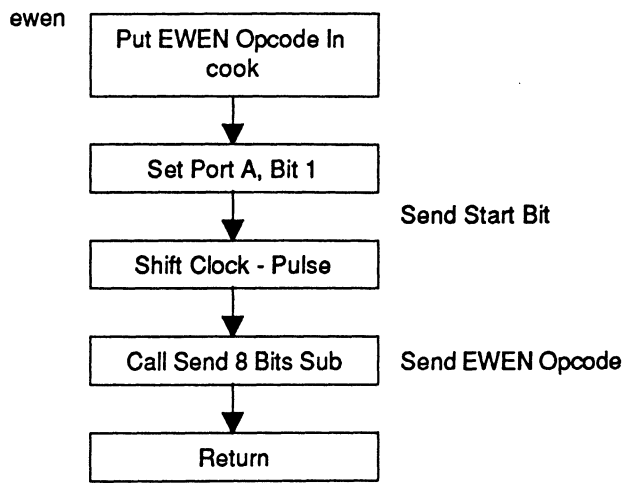


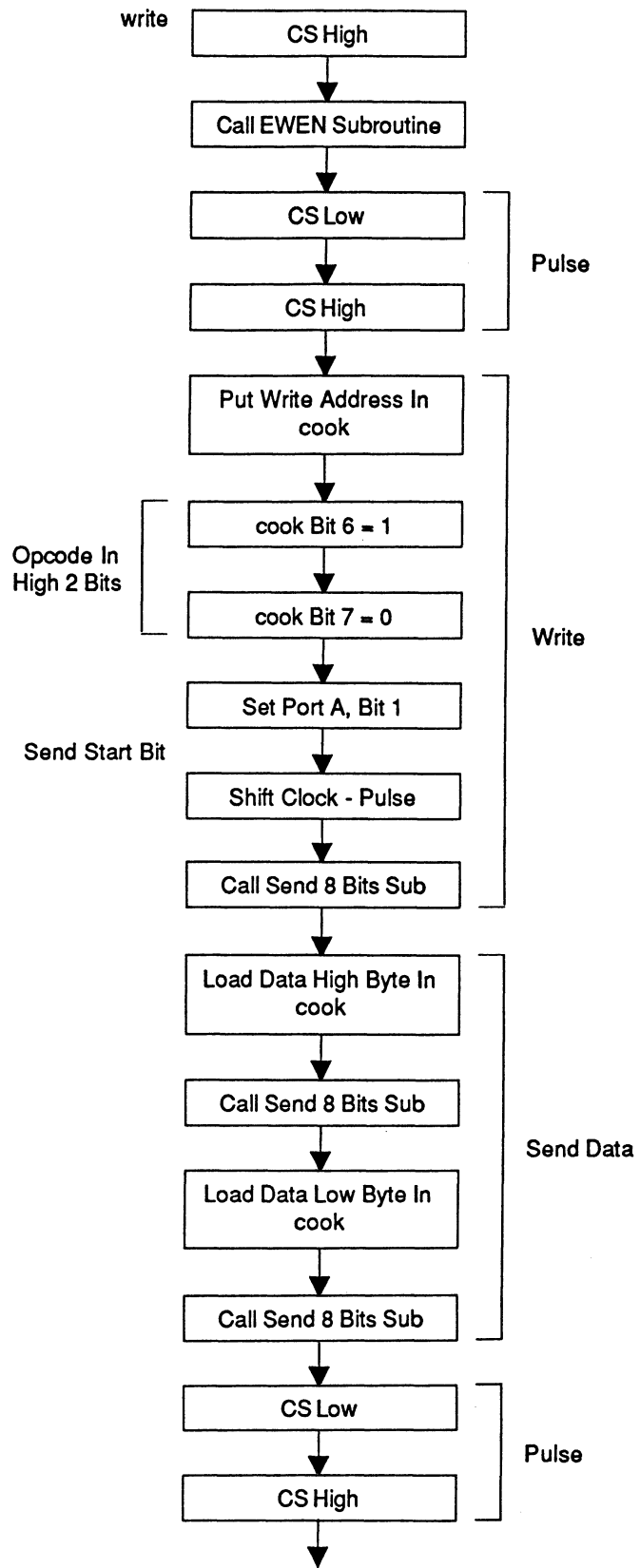
MAIN PROGRAM - INITIAL TEST

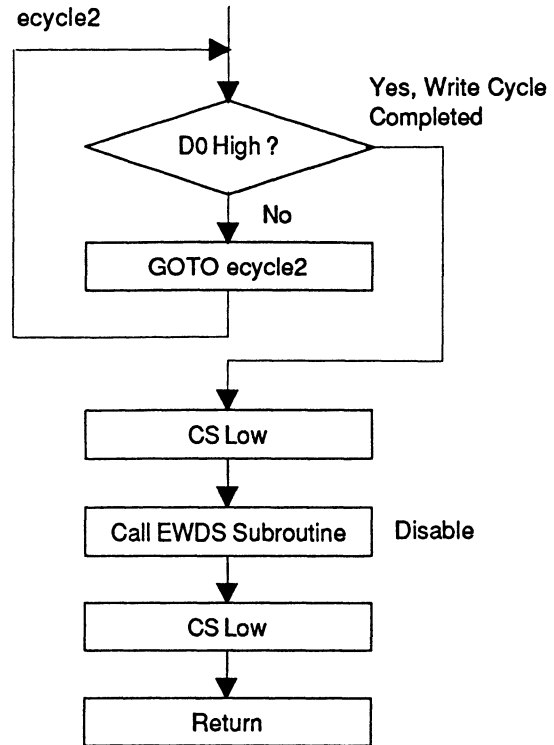
The main program will make use of subroutines. It will enable and disable operations, write to a register, and read back 16 bits of data from a register. If we can make this work, we can do anything we want with the 93C46.

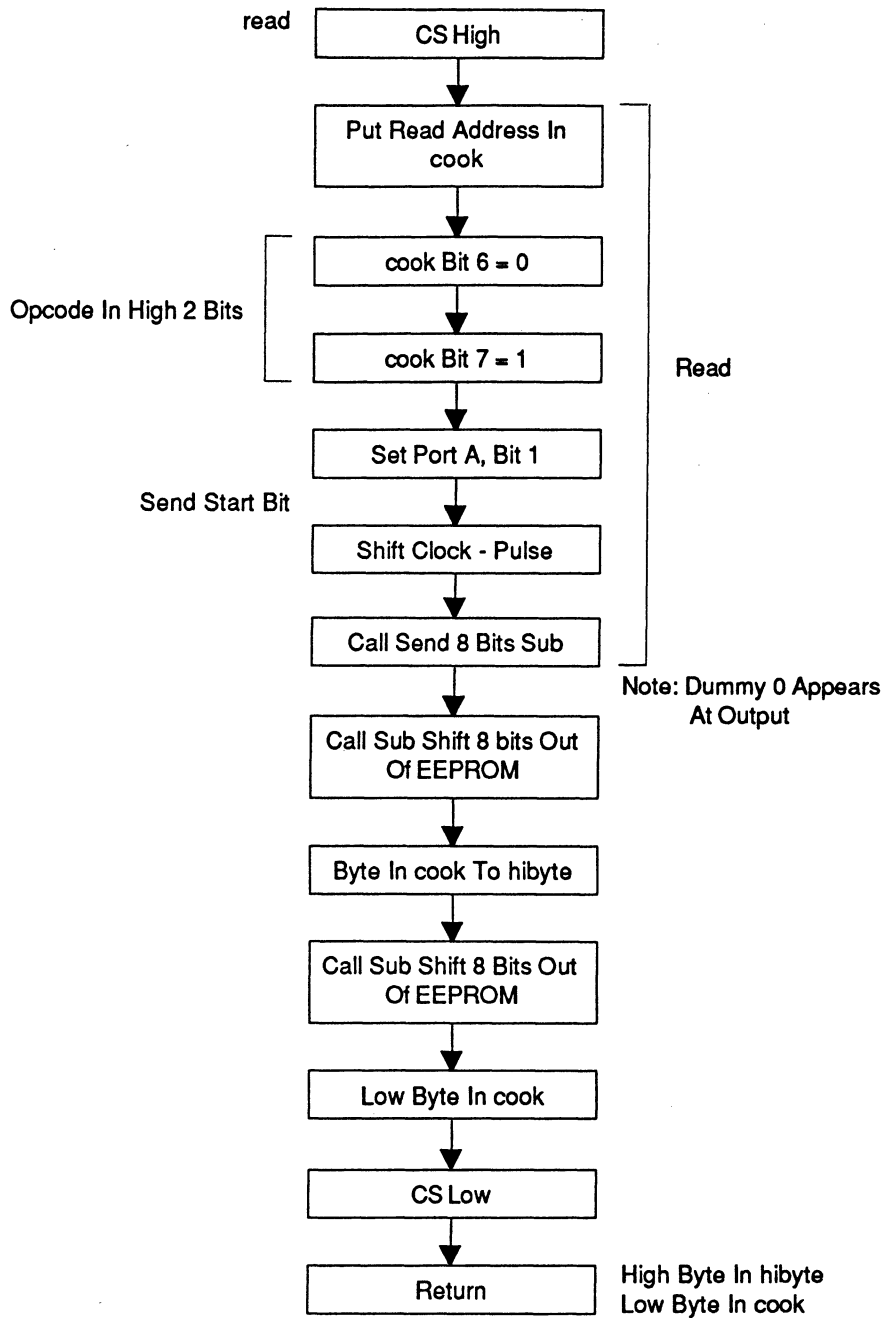
An erase the contents of a register subroutine is also shown.

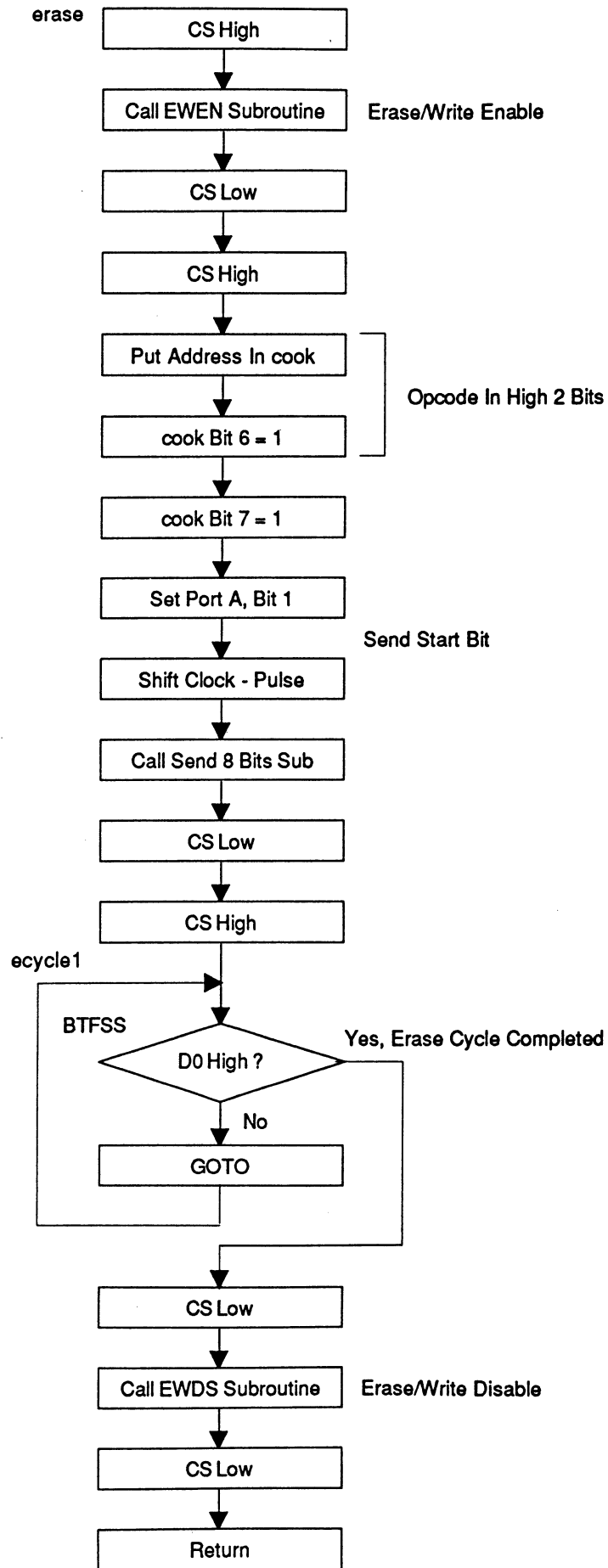


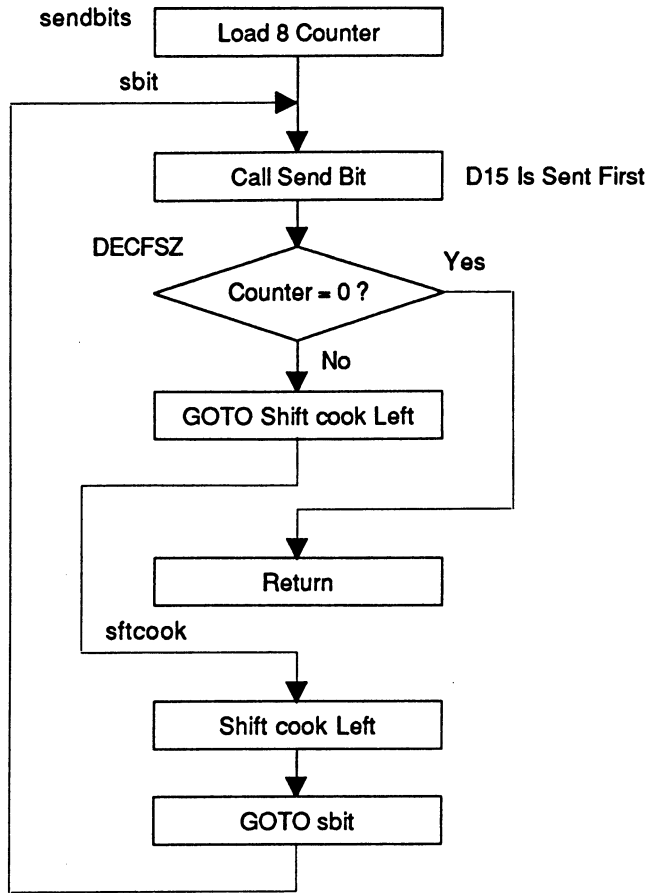


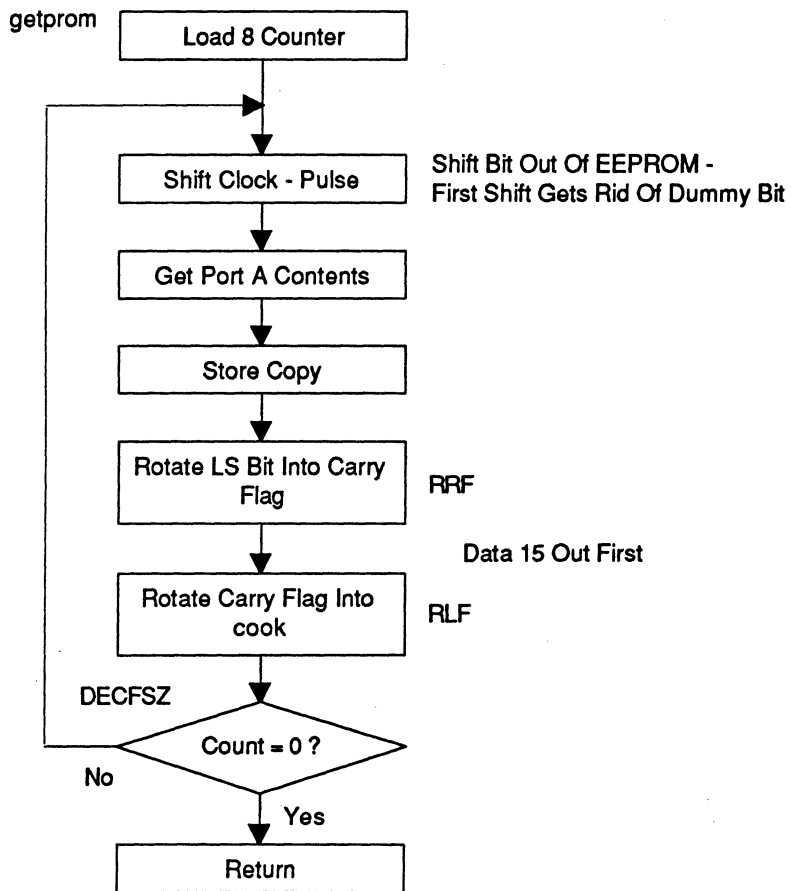
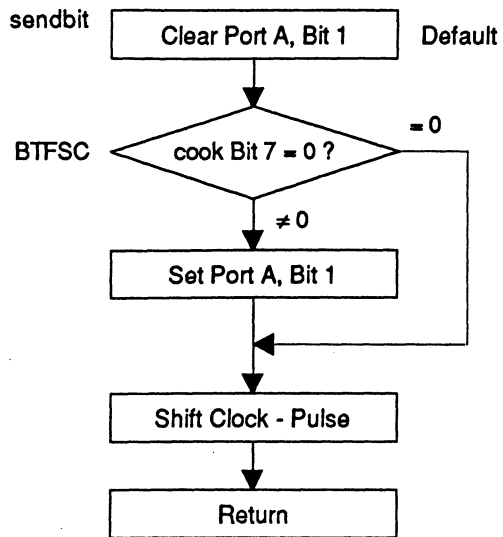












```

;=====93C46.ASM=====4/26/97==
      list    p=16c84
      radix   hex
;-----
;      cpu equates (memory map)
status equ    0x03
porta  equ    0x05
portb  equ    0x06
cook   equ    0x0c
hibyte equ    0x0d
count  equ    0x0e
address equ    0x0f
data_hi equ    0x10
data_lo equ    0x11
temp   equ    0x12
trisa  equ    0x85
trisb  equ    0x86
;-----
;      bit equates
rp0    equ    5
;-----
      org    0x000
;
start  bsf    status,rp0 ;switch to bank 1
      movlw  b'00000001' ;bit 0 = input
      movwf  trisa
      movlw  b'00000000' ;outputs
      movwf  trisb
      bcf   status,rp0 ;switch back to bank 0
      bcf   porta,1    ;initialize
      bcf   porta,2    ;initialize
      bcf   porta,3    ;initialize
      movlw 0x00        ;00000000
      movwf  portb      ;LED's off
      movlw 0x00        ;define test address
      movwf  address
      movlw 0x80        ;define test hi byte
      movwf  data_hi
      movlw 0x0f        ;define test lo byte
      movwf  data_lo
      call  write       ;write subroutine
      call  read        ;read subroutine
      movf  cook,w      ;get lo byte
      movwf portb      ;display via LED's
circle goto  circle   ;done
;-----
ewen   movlw 0x30      ;ewen op code
      movwf  cook      ;to cook
      bsf   porta,1    ;send start bit
      bsf   porta,2    ;shift
      bcf   porta,2
      call  sendbits   ;send ewen op code
      return

```

```

;-----
ewds    movlw    0x00        ;ewds op code
        movwf    cook        ;to cook
        bsf     porta,1      ;send start bit
        bsf     porta,2      ;shift
        bcf     porta,2
        call    sendbits     ;send ewds op code
        return

;-----
write   bsf     porta,3      ;cs high
        call    ewen         ;erase/write enable
        bcf     porta,3      ;cs low
        nop     ;1 microsecond min
        bsf     porta,3      ;cs high
        movf    address,w    ;get address
        movwf   cook        ;store in cook
        bcf     cook,7       ;op code
        bsf     cook,6       ;ms 2 bits
        bsf     porta,1      ;send start bit
        bsf     porta,2      ;shift
        bcf     porta,2
        call    sendbits     ;send address
        movf    data_hi,w    ;get data hi
        movwf   cook
        call    sendbits     ;send data hi
        movf    data_lo,w    ;get data lo
        movwf   cook
        call    sendbits     ;send data lo
        bcf     porta,3      ;cs low
        nop     ;1 microsecond min
        bsf     porta,3      ;cs high
ecycle2 btfss   porta,0      ;write cycle complete?
        goto    ecycle2     ;not yet
        bcf     porta,3      ;cs low
        nop     ;1 microsecond min
        bsf     porta,3      ;cs high
        call    ewds         ;yes, erase/write disable
        bcf     porta,3      ;cs low
        nop     ;1 microsecond min
        return

;-----
read    bsf     porta,3      ;cs high
        movf    address,w    ;get address
        movwf   cook
        bsf     cook,7       ;op code
        bcf     cook,6       ;ms 2 bits
        bsf     porta,1      ;send start bit
        bsf     porta,2      ;shift
        bcf     porta,2
        call    sendbits     ;send address
        call    getprom      ;shift hi 8 bits out of eeprom
        movf    cook,w       ;hi byte result in hi byte
        movwf   hi byte
        call    getprom      ;shift lo 8 bits out of eeprom

```

```

        bcf     porta,3      ;cs low
        nop
        return              ;exit sub with lo byte in cook
;-----
sendbits movlw  0x08        ;count=8
        movwf  count
sbit    call    sendbit     ;send 1 bit
        decfsz count,f      ;done?
        goto   sftcook     ;no
        return              ;yes
sftcook rlf     cook,f      ;shift cook left
        goto   sbit        ;again
;-----
sendbit bcf     porta,1     ;default
        btfsc  cook,7      ;test cook bit 7
        bsf    porta,1     ;bit is set
shift1  bsf     porta,2     ;shift
        bcf    porta,2
        return
;-----
getprom movlw  0x08        ;count=8
        movwf  count
shift2  bsf     porta,2     ;shift
        bcf    porta,2
        movf   porta,w     ;read port A
        movwf  temp        ;store copy
        rrf    temp,f      ;rotate bit into carry flag
        rlf    cook,f      ;rotate carry flag into cook
        decfsz count,f     ;decrement counter
        goto   shift2
        return              ;done
;-----
        end
;-----
;at blast time, select:
;   memory unprotected
;   watchdog timer disabled (default is enabled)
;   standard crystal (using 4 MHz osc for test) XT
;   power-up timer on
;=====
;-----
erase   bsf     porta,3     ;cs high
        call    ewen        ;erase/write enable
        bcf     porta,3     ;cs low
        nop
        bsf     porta,3     ;cs high
        movf   address,w    ;get address
        movwf  cook         ;store in cook
        bsf    cook,7       ;op code
        bsf    cook,6       ;ms 2 bits
        bsf    porta,1     ;send start bit
        bsf    porta,2     ;shift

```

```

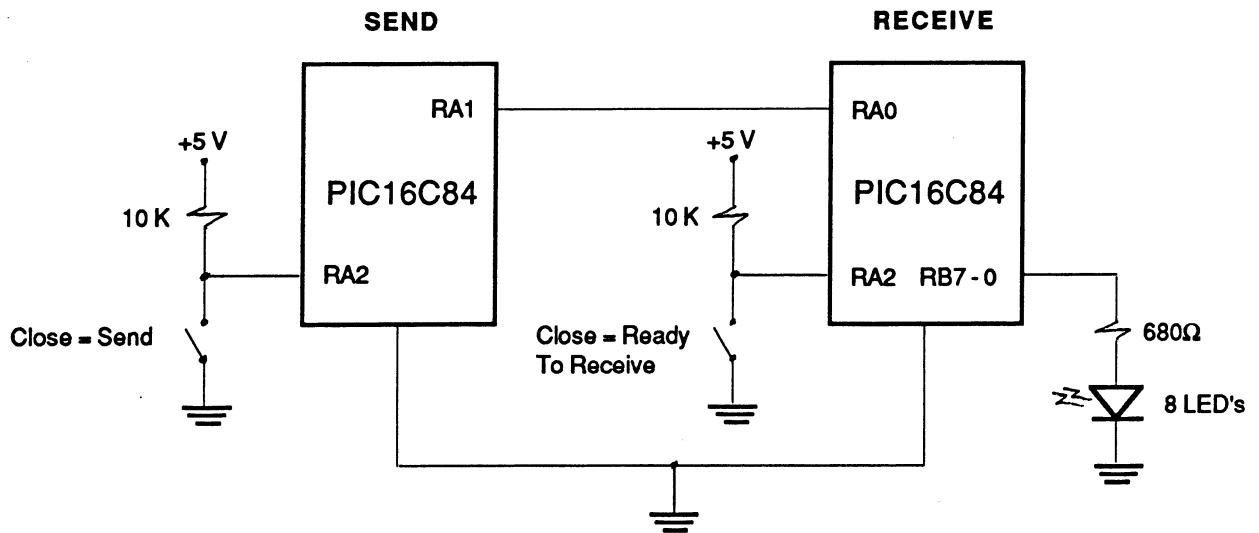
    bcf    porta,2
    call   sendbits    ;send address
    bcf    porta,3     ;cs low
    nop    ;1 microsecond min
    bsf    porta,3     ;cs high
ecycle1  btfs    porta,0 ;erase cycle complete?
    goto   ecycle1    ;not yet
    bcf    porta,3     ;cs low
    nop    ;1 microsecond min
    bsf    porta,3     ;cs high
    call   ewds       ;yes, erase/write disable
    bcf    porta,3     ;cs low
    nop    ;1 microsecond min
    return

```

NOPs are used to insure that the 93C46 chip's timing requirements are met.

PIC-TO-PIC SERIAL COMMUNICATION

In an effort to expand our serial communication capabilities, we will get a couple of PIC16's to talk to each other. Actually, we'll do part of the job by getting one PIC16 to talk while the other listens. We'll see if the listener understood what the talker said. We will set this up so you can continue on your own by sending more than one word and by interchanging the talk/listen roles (two-way communication).

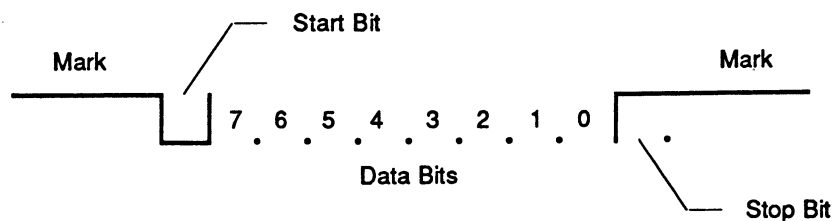


Two '84 on a board modules may be used for this experiment.

Both PIC16's are PIC16C84's with 4.0 MHz clock oscillators. For the transmitting chip, port A, bit 1 is used to transmit. The receiver uses port A, bit 0 is used to receive. We will choose the bit time interval as 256 internal clock (1 MHz) cycles. Both transmitter and receiver will use TMR0 for timing.

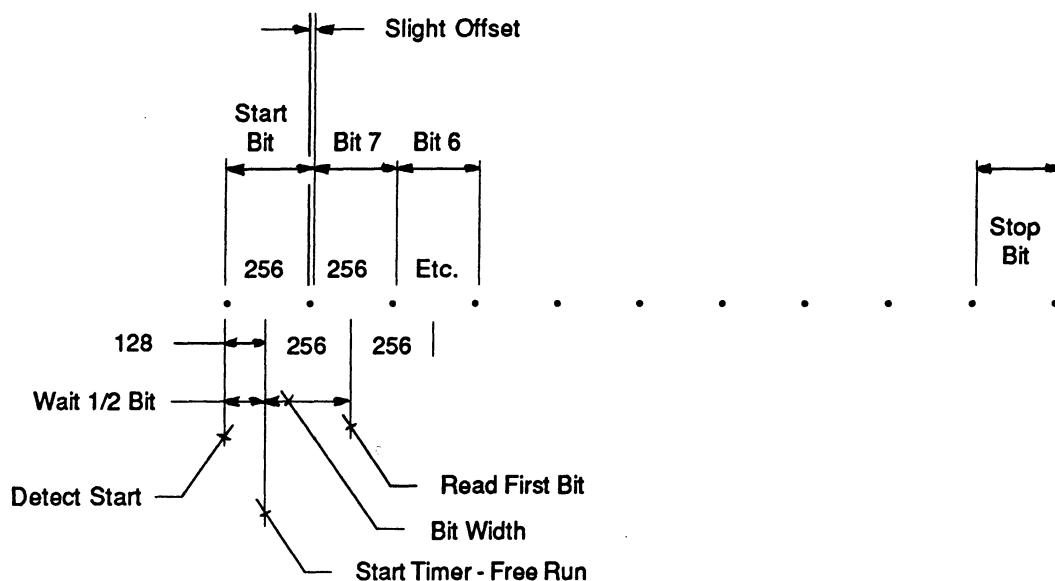
When the transmit data (TD) line is high, the condition is known as "mark". When TD is low, the condition is known as "space." The terminology comes from the old teletype days.

When one word (8 bits) is sent, the TD line output vs. time will look like this:



The TD line sits at mark = logic "1" until the word is sent. It drops to "0" first. This is the start bit which tells the receiver PIC16 that an 8-bit word is coming.

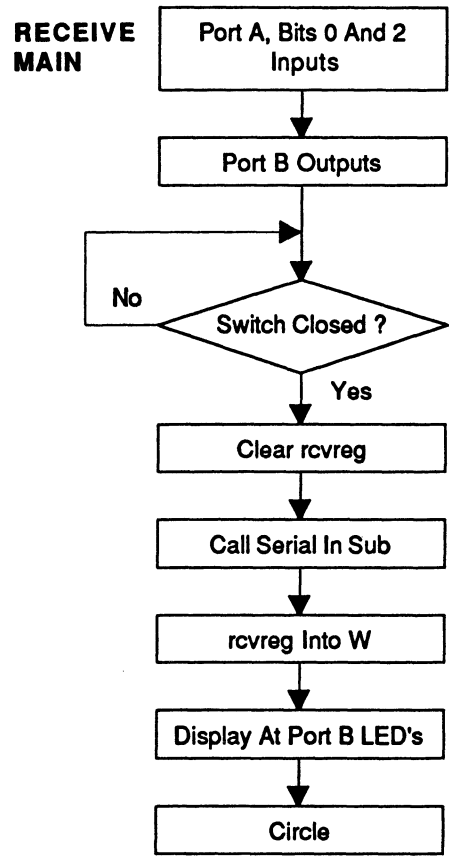
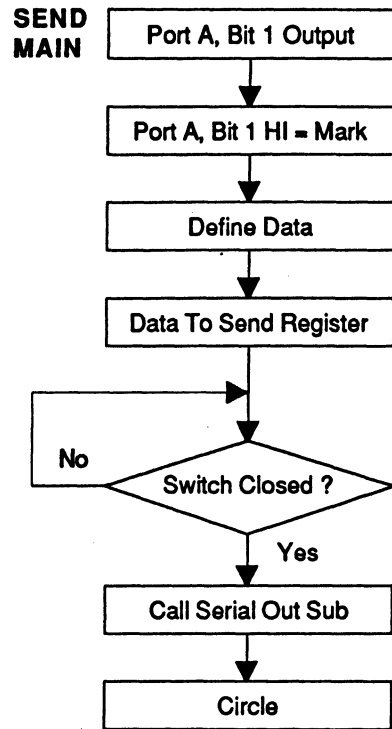
The transmitter transmits bits at some rate (bits per second = baud rate). The receiver must be set up to receive bits at the same rate. When the receiving PIC16's receive program is running, it sits in a loop looking for a start bit (high-to-low transition on the receive data (RD) line). When that transition takes place, the receiver's program waits for a time equal to half the width of the start bit. It looks at the RD line to see if it is still low. If not, a false start occurred and the program goes back to looking at the RD line. If the RD line is low, valid data follows and the program starts TMR0 (free-running mode) for a time interval equal to the width of a bit. Then it looks at the RD line to see if a "0" or a "1" is present. It grabs that bit and shifts it into a file register (shifting left, MSB received first). The program waits a bit-width (to middle of second bit, bit 6) and grabs it and stores it. This process is repeated until all 8 data bits have been received. The 8-bit word received is then displayed at the port B LED's so you can see if the correct data was received.



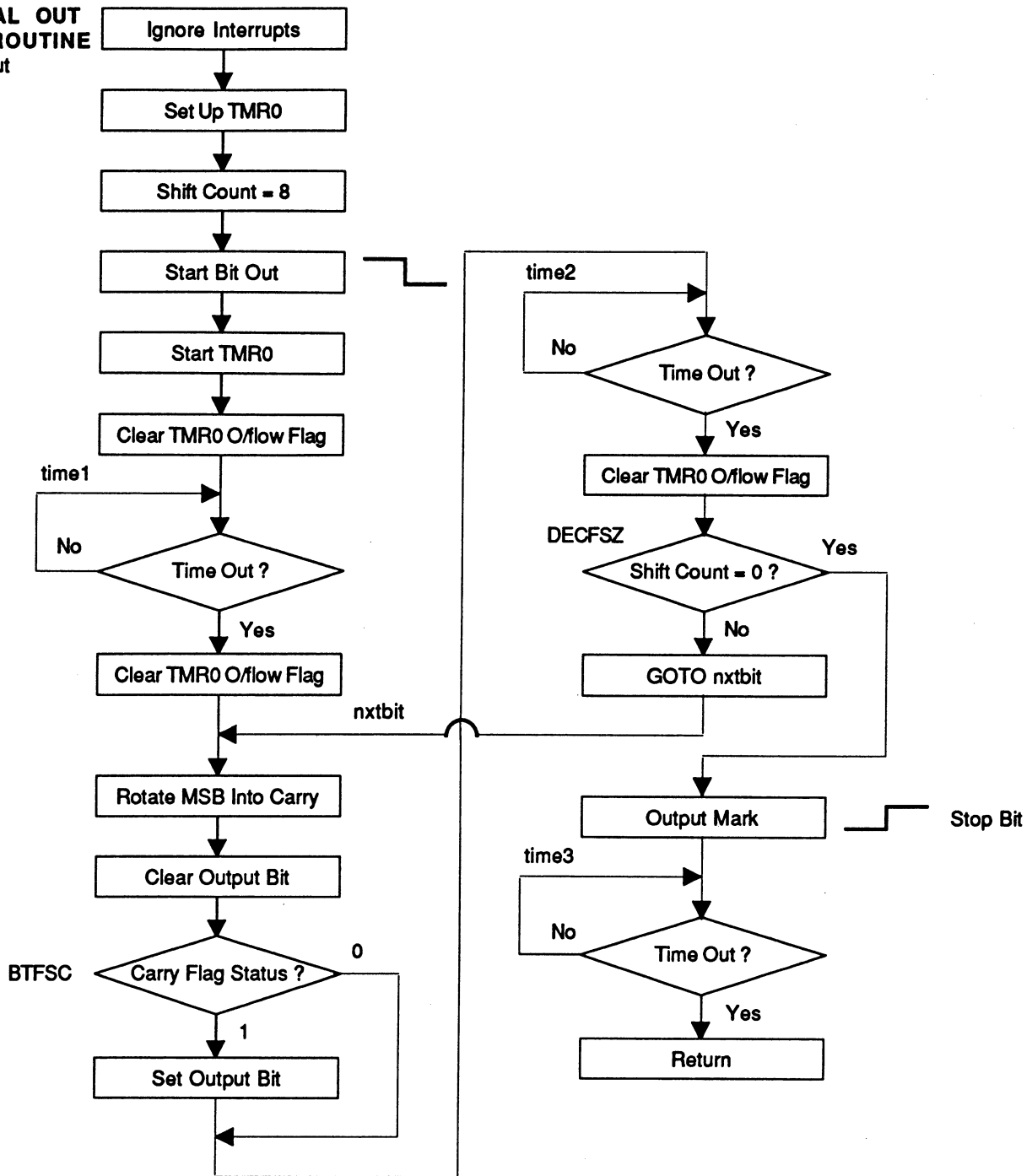
The use of TMR0 is explained in **Easy PIC'n**.

The default method for bit testing is used in the send program. The output bit may be cleared when it should be set, but it gets cleared right after that. It doesn't matter because the receiver samples the bit in the center. What goes on at the beginning or end will have no effect.

Flow charts and code for this simple example follow. The technique will be used for the serial LCD interface in the next chapter.



**SERIAL OUT
SUBROUTINE**
ser_out



;=====P2PSEND.ASM=====4/29/97==

```
list    p=16c84
radix   hex

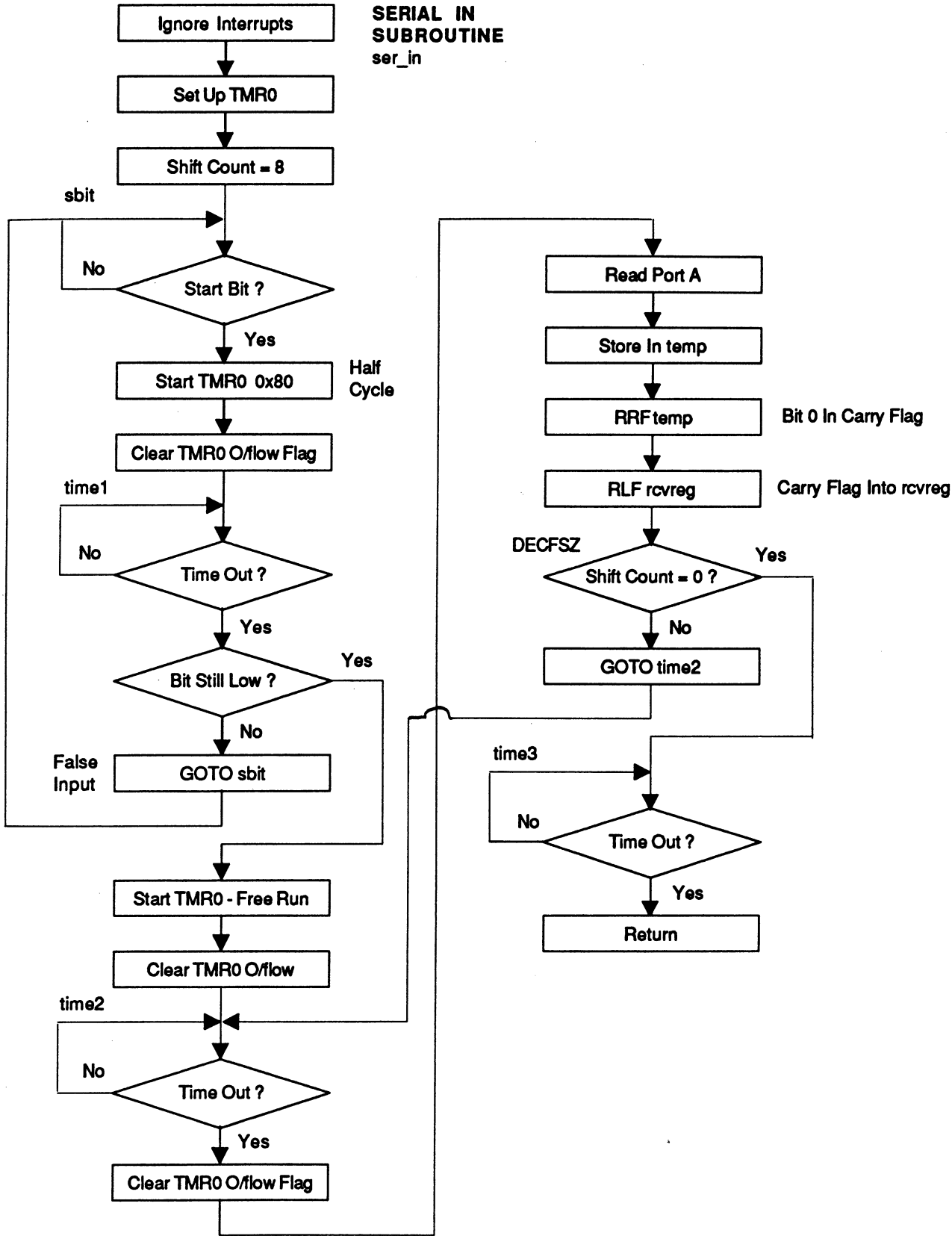
;-----
;      cpu equates (memory map)
tmr0    equ    0x01
status  equ    0x03
porta   equ    0x05
intcon  equ    0x0b
sendreg equ    0x0c
count   equ    0x0d
optreg  equ    0x81
trisa   equ    0x85
;-----
;      bit equates
c       equ    0
rp0     equ    5
;-----
org     0x000
;
start   bsf     status,rp0 ;switch to bank 1
        movlw  b'00000100' ;port A inputs/outputs
        movwf  trisa
        bcf     status,rp0 ;switch back to bank 0
        bsf     porta,1    ;output mark, bit 1
        movlw  0x80        ;number to be sent
        movwf  sendreg     ;store
switch  btfsc   porta,2    ;start send?
        goto   switch      ;not yet
        call   ser_out     ;to serial out subroutine
circle  goto   circle      ;done
;-----
ser_out bcf     intcon,5    ;disable tmr0 interrupts
        bcf     intcon,7    ;disable global interrupts
        clrf   tmr0        ;clear timer/counter
        clrwdt ;clear wdt prep prescaler assign
        bsf     status,rp0 ;to page 1
        movlw  b'11011000' ;set up timer/counter
        movwf  optreg
        bcf     status,rp0 ;back to page 0
        movlw  0x08        ;init shift counter
        movwf  count
        bcf     porta,1    ;start bit
        clrf   tmr0        ;start timer/counter
        bcf     intcon,2   ;clear tmr0 overflow flag
time1   btfss   intcon,2   ;timer overflow?
        goto   time1      ;no
        bcf     intcon,2   ;yes, clear overflow flag
nxtbit  rlf     sendreg,f   ;rotate msb into carry flag
        bcf     porta,1    ;clear port A, bit 1
        btfsc  status,c    ;test carry flag
        bsf     porta,1    ;bit is set
time2   btfss   intcon,2   ;timer overflow?
```

```

        goto    time2        ;no
        bcf     intcon,2     ;clear overflow flag
        decfsz  count,f      ;shifted 8?
        goto    nxtbit       ;no
        bsf     porta,1      ;yes, output mark
time3   btfss   intcon,2     ;timer overflow?
        goto    time3        ;no
        return                ;done
;-----
        end
;-----
;at blast time, select:
;   memory unprotected
;   watchdog timer disabled (default is enabled)
;   standard crystal (using 4 MHz osc for test) XT
;   power-up timer on
;=====

```

**SERIAL IN
SUBROUTINE
ser_in**



```

;=====P2PRCV.ASM=====4/29/97==
        list    p=16c84
        radix   hex
;-----
;      cpu equates (memory map)
tmr0    equ     0x01
status  equ     0x03
porta   equ     0x05
portb   equ     0x06
intcon  equ     0x0b
rcvreg  equ     0x0c
count   equ     0x0d
temp    equ     0x0e
optreg  equ     0x81
trisa   equ     0x85
trisb   equ     0x86
;-----
;      bit equates
rp0     equ     5
;-----
        org     0x000
;
start   bsf     status,rp0 ;switch to bank 1
        movlw   b'00000101' ;port A inputs/outputs
        movwf   trisa
        movlw   b'00000000' ;port B outputs
        movwf   trisb
        bcf     status,rp0 ;back to bank 0
        clrf   portb
        clrf   rcvreg
switch  btfsc   porta,2    ;operator ready to receive?
        goto   switch      ;no
        call   ser_in      ;yes, to serial in subroutine
        movf   rcvreg,w    ;get byte received
        movwf  portb       ;display via LED's
circle  goto   circle      ;done
;-----
ser_in  bcf     intcon,5    ;disable tmr0 interrupts
        bcf     intcon,7    ;disable global interrupts
        clrf   tmr0        ;clear timer/counter
        clrwdt ;clear wdt prep prescaler assign
        bsf     status,rp0  ;to page 1
        movlw  b'11011000' ;set up timer/counter
        movwf  optreg
        bcf     status,rp0  ;back to page 0
        movlw  0x08        ;init shift counter
        movwf  count
sbit    btfsc   porta,0    ;look for start bit
        goto   sbit        ;mark
        movlw  0x80        ;start bit received, half bit time
        movwf  tmr0        ;load and start timer/counter
        bcf     intcon,2    ;clear tmr0 overflow flag
time1   btfss   intcon,2    ;timer overflow?
        goto   time1       ;no

```

```

        btfsc   porta,0      ;start bit still low?
        goto   sbit         ;false start, go back
        clrf   tmr0         ;yes, half bit time - start timer/ctr
time2   bcf     intcon,2     ;clear tmr0 overflow flag
        btfss  intcon,2     ;timer overflow?
        goto   time2        ;no
        bcf   intcon,2     ;yes, clear tmr0 overflow flag
        movf  porta,w      ;read port A
        movwf temp         ;store
        rrf   temp,f       ;rotate bit 0 into carry flag
        rlf   rcvreg,f     ;rotate carry into rcvreg bit 0
        decfsz count,f     ;shifted 8?
        goto  time2        ;no
time3   btfss  intcon,2     ;timer overflow?
        goto  time3        ;no
        return             ;yes, byte received
;-----
        end
;-----
;at blast time, select:
;   memory unprotected
;   watchdog timer disabled (default is enabled)
;   standard crystal (using 4 MHz osc for test) XT
;   power-up timer on
;=====

```

To run the programs:

Run "send" first with switch off (RA2) - establish proper level on TD = mark.
 Run receive second with switch off (RA2) - get ready to receive.
 Stabilize, then switch on = ready.
 Send switch on.

Multiple bytes may be transmitted from the file registers by using the FSR and indirect addressing and a counter. Multiple bytes may be transmitted from a table in program memory by using relative addressing and a counter. Examples of both will be shown in the LCD Interface chapter.