Using the P82B715 I²C extender on long cables AN444

Author: Don Sherman, Sunnyvale

The P82B715 I²C Buffer was designed to extend the range of the local I²C bus out to 50 Meters. This application note describes the results of testing the buffer on several different types of cables to determine the maximum operating distances possible. The results are summarized in a table for easy reference.

The I²C bus was originally conceived as a convenient 2 wire communication method between Integrated Circuits located within a common chassis, such as inside a TV set or inside a VCR. The serial protocol contains an address, or identifying code, for each type of device and additional internal addresses, if needed within the addressed device. Each device has its own decoding circuitry to allow it to recognize its own unique address or identifying code. To communicate, a device watches the bus activity and jumps in when it sees a stop. Once a Master gets control of the bus, it sends the address of the particular device with which it wants to communicate. Communication will then transpire between the Master and the Slave device. The existence of many types of ICs which have built-in I²C interface capabilities makes system design almost as easy as drawing a block diagram. Real-time clocks, RAM, A/D converters, EEPROMs, Microcontrollers, Keyboard encoders, LCD display drivers, and many other I²C supported chips all communicate over two wires rather than needing 16 Address lines, 8 data lines and Address decoders along with handshake signals, which more conventional designs would require to be routed all over the Printed Circuit board.

Now, with the introduction of the I²C buffer chip, it is easy to branch out beyond the single chassis mode and use this convenient local area network to tie together whole systems without the need to convert from the "internal" I²C protocol to an external communication medium such as RS-232 and then RS-485. By using the new Philips I²C buffer, the external systems' components can be accessed as easily as the internal I²C connected components.

The P82B715 is an 8 pin IC which contains 2 identical amplifier sections to allow for the current amplification and buffering of both the SDA and the SCL signals on the I²C bus. Each section in the P82B715 contains a bipolar times 10 current amplifier which senses the direction of current flow through an internal 30 ohm series resistor in the I²C line. The P82B715 then boosts the current, while keeping the voltage gain at unity, and continues to maintain the voltage drop direction across the resistor. This configuration results in different waveforms as the P82B715 starts to do its job. If the driving source has a strong current sink capability, then it will start to drive the buffered I²C line immediately through the 30 ohm resistor. A microsecond later the P82B715's amplified pull down current kicks in and pulls the line down even harder. If the driving IC is only capable of the I²C specified 3 milliamp pull down current, the buffered bus will fall a little and then just wait at that voltage level for the propagation delay of the amplifier to finally turn on and bring the buffered bus down to a logic low. Thus, there will always be some form of a step in the falling edge of the buffered output waveform, see Figure 1. A weak source will have a step (plateau) up near 4 volts and a strong source, such as the Philips Semiconductors 87C751 microcontroller, will have the step occur below 2 volts. The position of the step will be determined by the current sink capability of the I²C bus driver versus the value of the pull-up resistor which is used on the buffered I²C bus, Vstep = 5V – (Isink × Rbuf). For example: Vstep = 5V – (3mA × .165 k ohms) = 5 – .495 = 4.5Volts; another example: Vstep = 5V – (20mA × .165 k ohms) = 5 – 3.3 = 1.7Volts.

Running the I²C signals over long distances poses several problems. The I²C SDA and SCL lines are monitored by all of the ICs connected on the I²C bus. These ICs each have their own circuitry to decipher the information on the bus. In normal operation, a Start occurs when there is a high to low transition on the SDA line while SCL is high. Obviously, if any external noise is coupled into the SDA line, it could be mistakenly perceived as a Start. Because of this, some form of shielding will be preferred to protect the two I²C signals from external noise sources. During the transmission of data there are signals which are active on both SDA and SCL. If these normal signals are cross-coupled, then data can be corrupted. Thus, although the standard telephone twisted pair cable is the most commonly available built in cable, it is not recommended for long I²C runs. This cable maximizes crosstalk, due to the twisted pair configuration and, since there is no shielding, is very vulnerable to adjacent wire telephone signal coupling and to any stray external electromagnetic interference. This effect can be somewhat reduced by running a signal wire and a grounded wire as adjacent pairs.

Long distance cables present capacitive loading which must be overcome with the driver chips. The limiting factor is the amount of pull-up current which is available to charge the line capacitance. With the simple resistor pull-up recommended by I²C standards, three milliamps is available for charging this line capacitance. The rise time of the signal will increase linearly with the increase in capacitive loading and the specified maximum capacitive loading is only 400 Pico Farads for guaranteed 100kHz communication rates. The P82B715 current buffer allows for 30 milliamps of pull-up current, with a resulting maximum capacitive loading of 4,000 Pico Farads (4 Nano Farads).

The I²C hardware inputs look at the I²C signals and act when those signals pass through the active linear region at about 1.2 to 1.4 volts, and are detected as digital levels. Thus, there is a delay between when an output transistor turns off and when the rising signal is detected as a logic one at the receiver. This time depends on the value of the pull-up resistor, the perceived capacitance at the transmitting end, the delay through the cable, and finally the delay through the receiver's amplifier to its output stage. The maximum allowable time is limited by the characteristic that the I²C master provides the clock signal which must travel down the cable and be received by the slave. This slave must act on the clock signal and produce data information which is sent back to the master with an additional set of delays. Upon reception the data must be put in its proper place before the master starts its next clock signal, or an error will occur.

Different types of cable were tested and the results are shown in Table 1. Keep in mind that the results are based on cable runs in a low electrical noise environment. If reliable operation is desired in a high electrical noise environment, shielded cable must be used. For "short" runs, flat cable with every other conductor grounded, seems to provide a good, low capacitance medium for I²C transmission, otherwise, the shielded audio cable seemed to provide the best price/performance. Note that for long runs, it is desirable to have a separate power supply at each end of the cable, and the shield or ground wire will provide a common reference between the two supplies. The voltage drop due to the resistance of the wire usually is the limiting factor for very long runs of cable where the power to the remote system must also come through the cable. Table 1 shows the results of testing with longer and longer cable lengths until failures were detected. The values in the table represent the maximum cable lengths which still provided error free code from a modified version of the Ping-pong program which is listed in Application Note AN430.

Using the P82B715 I²C extender on long cables AN444

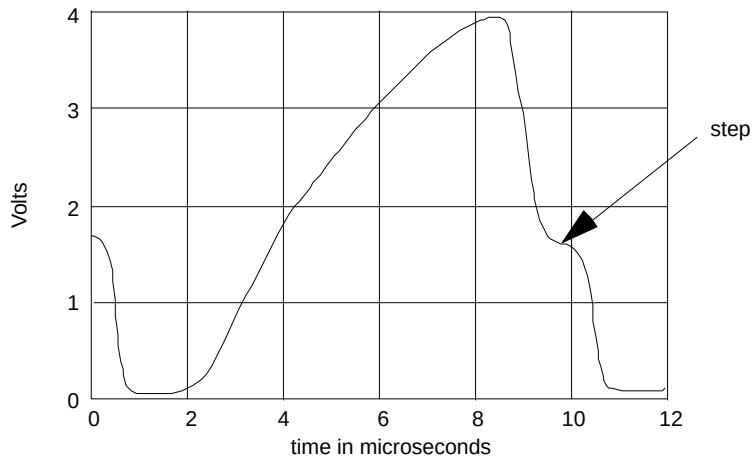

Figure 1. P82B715 Output Waveform on Long Cable

Table 1. Test Results with P82B715 Over Long Cables

CABLE TYPE	Ohms/m	pF/m	Total Length	Total Ohms	Total Cap.
Belden 8723 45 Ohm Audio 2 each 2—24AWG wire stranded Beldfoil Aluminum-polyester shielded with common drain wire SDA & ground on one pair; SCL & ground on other pair	.049	115	305M (1000')	11.5	48.2nF
Belden 8723 45 Ohm Audio using 1 shielded pair, SDA on Red, SCL on Black	.049	115	330M (1100')	12.7	53nF
RG-174/U 50 Ohm Video Cable SDA and grounded shield in one cable SCL and grounded shield in one cable	.318	101	150M (500')	47.7	15.2nF
"Telephone Cable" 22&24 AWG Solid Copper Twisted Pair, Level 3 LAN & Medium Speed Data SDA and ground in one twisted pair SCL and ground in one twisted pair	.0286	66	95M (310')	2.7	6.4nF
Flat "Ribbon" Cable, every other conductor grounded	.20	52	400M (1320')	80.5	21nF

In all of the tests, the power supply voltage was 4.5 volts. The ground for the remote test fixture was through the long cable. Since 4.5 volts is the recommended minimum voltage for both the 87C751 and the P82B715, it was not possible to operate the remote unit on power supplied through the long cable, since any ohmic drop would place the ICs out of their specified range. However, it is necessary to connect the grounds between the two units for the best noise immunity.

The P82B715 is designed to drive a 4 nF capacitive load at 100kHz. However, the actual total capacitances of the long cables which worked were substantially greater than this. The loading did effect the software driven hardware part of the 87C751. To achieve a true 100kHz data rate, it was necessary to shorten the '751 Timer values for the I²C drivers. This resulted in an asymmetrical waveform, but did achieve a 10 microsecond period (100kHz). This asymmetry in duty cycle can be easily seen in the Figure 1 waveform.

The test with the Belden 8723 Audio Cable worked if one of the shielded pair was connected to a signal and the other was connected to ground or +5volts. When both wires were connected in parallel as signal wires, the capacitance to ground doubled and the test failed. Also note that the adjacent wire mutual inductive coupling of the SDA and SCL signals did not seem to cause any problems even out to 1000 feet. This indicated that possibly the Belden 9452 45 ohm beldfoil shielded audio cable with a single set of twisted pair wires would be a good candidate to also try.

Flat ribbon cable provided a good compromise between shielding and reasonable capacitance. It is possible to increase the shielding effect by using flat cable with an etched copper foil layer on the back side of the cable. Noise can be induced into the cable by folding it back over itself for mutual induction effects, and also by operating a noise source close to the cable. A transformer type of soldering iron and florescent light transformers seemed to be good noise sources.

The P82B715 can drive multiple P82B715 remote units. The line should have some form of pull-up resistor at each driver. If only two drivers are used, as shown in Figure 2, the load should be split between the two drivers. For example, if the pull-up current is to be 30 milliamps and the voltage is 5 volts, the pull-up resistance should be: 5V/.030 amps = 165 ohms. This should be implemented by placing a 330 ohm resistor at each end of the cable so that the parallel resistance is 165 ohms and each end of the line is terminated. Remembering that the current gain can be as low as 8 and that most runs will not be to the maximum possible distance, lower values of pull-up current can

be used with the appropriate modifications to the above equations.

For larger fan-out with fixed locations, the load resistance should also be evenly divided so that the parallel combination of all of the pull-up resistors will provide the desired D.C. pull-up current.

If some of the remote units will be pluggable, it will be necessary to divide the pull-up load to accommodate all of the possible combinations of possible fanout. Figure 3 shows an example of driving up to 30 remote, pluggable peripherals. On the 3 milliamp side of the P82B715 a complete I²C system may exist. In Figure 3, a local I²C network cluster could be joined to other local network clusters through the P82B715 buffered bus so that hundreds of I²C devices could potentially be interconnected.

The ease of connecting I²C clusters into a complete LAN opens the door for many new uses of components which have an I²C bus connection. Now an electronic instrument can have access to remote keyboards and remote sensors by using the I²C bus. The instrument's output can easily be shown on multiple remote displays all connected with the I²C bus. Multiple instruments can also pass data back and forth over the I²C bus. Thus, we see that the I²C bus can become an effective and inexpensive Local Area Network by using the P82B715 I²C bus extender.

THE TEST SETUP

These tests were run on two identical test boards which each use a Philips Semiconductors 87C751 microcontroller that drives the I²C buffer which has a 330 ohm pull-up resistor. The schematic is shown in Figure 4. The software is a modified version of the "Ping-Pong" program which is described in the Philips Semiconductors Application Note, AN430, "Using the 8XC751/752 in Multimaster applications". This program sends a number down the I²C line and, when received, the receiving unit becomes a master and increments the number and sends it back to the first unit where it is checked and then the process repeats itself. The software has extensive error detection capability and monitors for corruption of data, false starts, over run of data, stuck lines and about anything else which might indicate a problem. If any errors did occur, a software counter was incremented. In this setup, the counter was stopped at Hex 07F to prevent wrap around and the contents of the counter are displayed on a bank of 8 LEDs. The MSB of the counter register was used as an indicator that the unit was working. The MSB LED flashes at about a 1 Hz rate when the unit is operating normally. When a cable length was reached which was too long, the MSB LED would stop flashing and the counter would rapidly fill up and stop with all 7 LEDs on (LED on indicates a logic "1" in this application).

THE TEST HARDWARE

A general purpose test rig was designed so that future needs of a general I²C platform could also be met. All of the port pins on the '751 were used. The inputs to the system were a toggle switch with a pull-up resistor connected to P0.2 (because this pin is Open Drain) and an octal DIP switch connected to port 1 (the internal pull ups of the port were used, so no external pull-up resistors were needed). The output is displayed through an octal buffer connected to port 3. A logical "1" on the pin will light up the LED. The I²C signals, SDA and SCL, are connected to the I²C buffer chip and the outputs of the buffer are pulled up by 330 ohm resistors. The parallel combination of the buffered transmitting end pull-up and the receiving end pull-up resistors is 330/2 ohms, which results in a pull-up load current of 30 milliamps. This current from the two pull-up resistors must be sunk by the single driving transistor of the acting sender. The effective loading seen by the '751 is the I²C buffer's load divided by 10. Thus, the '751's I²C outputs will sink 3 milliamps when driving the I²C buffer which is sinking 30 milliamps on the buffered bus.

The software monitor routine allows the user to monitor any internal '751 RAM location and display the contents on the LEDs. The monitor routine also allows the user to modify the contents of any RAM location including SFR space. The Ping-Pong program needed the first 8 locations in RAM, so the stack pointer for this application was changed from the default location of 07H to location 09H. This starts the stack at 0AH.

To read the contents of RAM, set the DIP switches to the desired RAM address. The toggle switch is set to a "1". Pressing the Reset switch causes the microprocessor to reset and then enter the monitor program where the program then waits until the toggle switch is changed. Upon closing the toggle switch (a "1" to "0" transition) the program loads the DIP switch selection into R0 of bank 1 (RAM location 08H). The program then loads the contents of the RAM location pointed to by R0 (bank 1) and copies it into port 3, where it is displayed on the 8 LEDs. Thus, the Address is seen by looking at the DIP switches and the contents pointed to are displayed on the LEDs. Note that this indirect Address latch location (R0,bank 1) would have been the normal beginning of the stack, had it not been changed.

The contents of an internal RAM location can also be modified with this program. First, set the DIP switches to the desired Address and set the toggle switch to "0". Reset the processor and then set the toggle switch to "1". This transfers the address to R0 (bank 1). Next, load the desired new data, which is to be stored in RAM, into the DIP switches, and then set the toggle switch to "0". At this time the LEDs will now show the Address of RAM and the DIP switches show what was written into the selected RAM location. To verify that the data was actually written into the RAM, follow the read RAM sequence.

Although this may seem to be a bit cumbersome, it is a workable way to see what is happening inside of the '751. Remember that it is necessary to re-enter the monitor program, or at least to duplicate the read RAM of R0 (bank 1) and output to port 3, to see the latest version of the contents of the RAM location. Since this experiment only looked at the contents of one RAM location, the above method was easy to use and the display always showed the current status of the desired RAM location because it is updated often by the software.

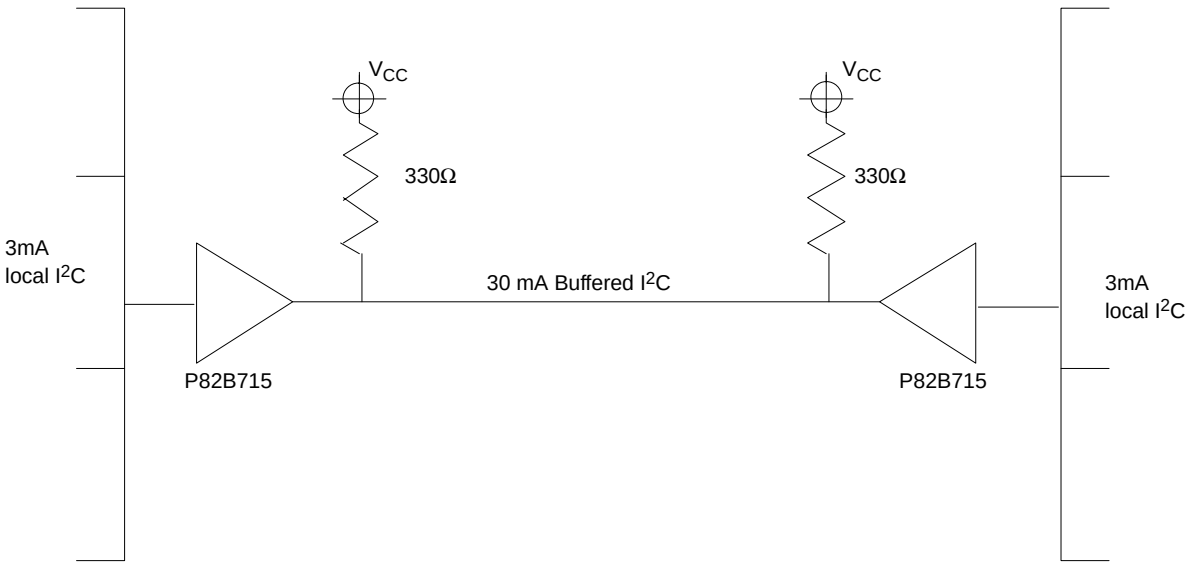

Figure 2. P82B715 Driving Long Line

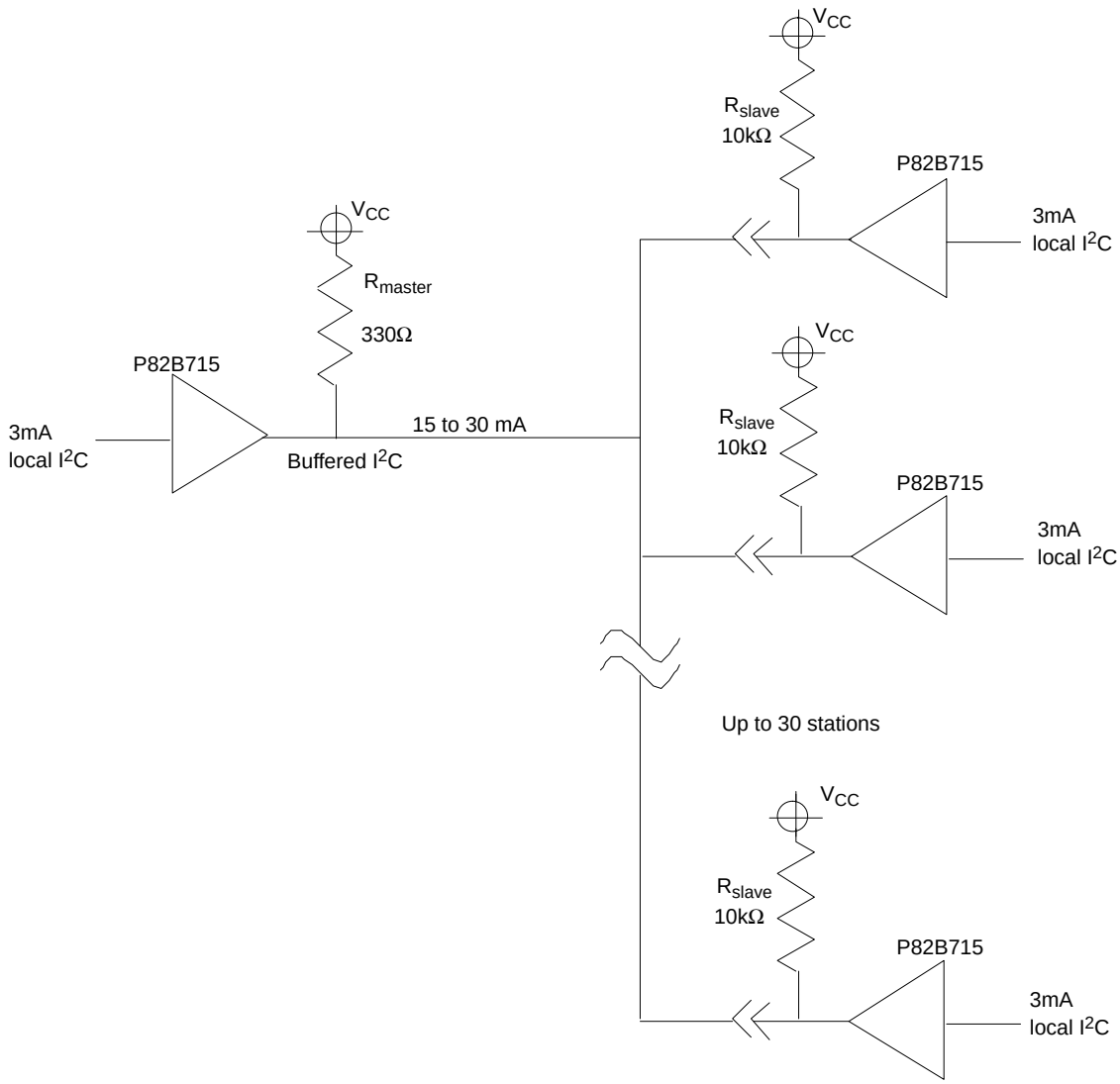

Figure 3. Large Fan-Out Configuration for P82B715

Note that V_{CC} is 5 volts for these values of load resistors. If a different voltage is desired, the calculations are as follows:

$$R_{master} = \frac{V_{CC}}{15mA} \qquad \text{example: } R_{master} = \frac{5V}{15mA} = 0.33k = 330\Omega$$

The pluggable units would be calculated as follows:

Parallel combination of $R_{slave} = R_{master}$

$R_{slave} = R_{master} \times$ Fan out

example: $R_{slave} = 330\Omega \times 30 = 9900\Omega = 10k$

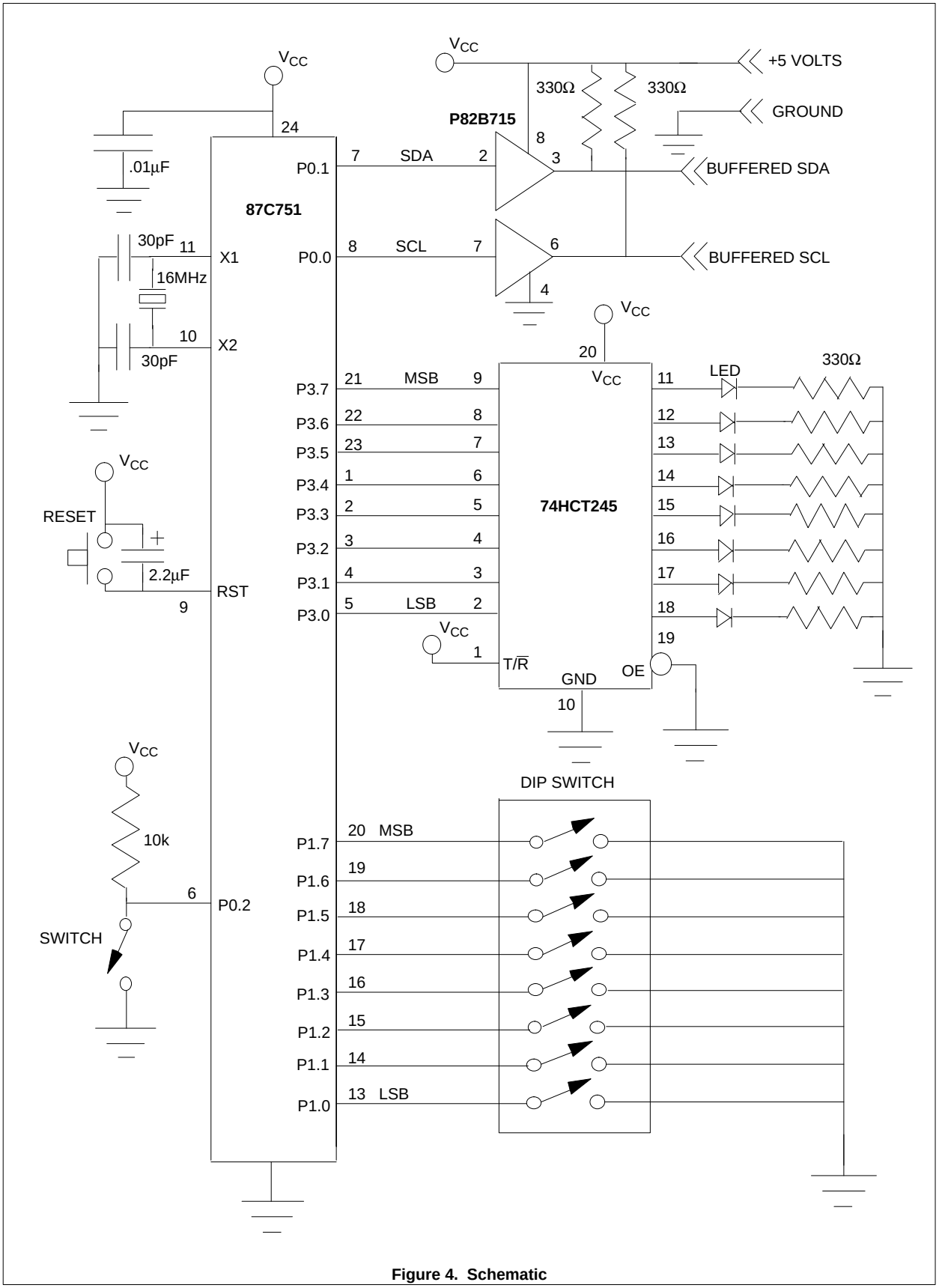

Figure 4. Schematic

```
;
;*************************************************************************
;         Multimaster Code for 83C751/83C752
;         4/14/1992    MODIFIED BY DON SHERMAN 5-21-92
;         ;; is used to show where original code was modified
;*************************************************************************
; This code was written to accompany an application note. The I2C routines
; are intended to be demonstrative and transportable into different
; application scenarios, and were NOT optimized for speed and/or memory
; utilization.
;
; Yoram Arbel

$TITLE(83C751 Multi Master I2C Routines)
$DATE(4/14/1992)
$MOD751    ;;NEED TO USE $MOD752 FOR 752 EMULATOR
;;EI2        EQU     ES         NEED ENABLE FOR EMULATOR
$DEBUG
;*************************************************************************
;       8XC751 MULTIMASTER I2C COMMUNICATIONS ROUTINES
;       Symbols and RAM definitions
;*************************************************************************

; Symbols (masks) for I2CFG bits.

BTIR       EQU     10h                ; TIRUN bit.
BMRQ       EQU     40h                ; MASTRQ bit.

; Symbols (masks) for I2CON bits.

BCXA       EQU     80h                ; CXA bit.
BIDLE      EQU     40h                ; IDLE bit.
BCDR       EQU     20h                ; CDR bit.
BCARL      EQU     10h                ; CARL bit.
BCSTR      EQU     08h                ; CSTR bit.
BCSTP      EQU     04h                ; CSTP bit.
BXSTR      EQU     02h                ; XSTR bit.
BXSTP      EQU     01h                ; XSTP bit.

; Note:
;
; Specific bits of the I2CON register are set by writing into this register a
; combination of the masks defined above using the MOV command.
; The SETB command should not be used with I2CON, as it is implemented by
; reading the contents of the register, setting the appropriate bit and
; writing  it back into the register.  As the functionality of the Read and
; Write  portions of the I2CON register is different, using SETB may cause
; unwanted results.

; Message transaction status indications in MSGSTAT:

SGO        EQU     10h         ; Started Slave message processing.
SRCVD      EQU     11h         ; as a slave, received a new message
SRLNG      EQU     12h         ; received as slave a message which is too
                               ; long for the buffer
STXED      EQU     13h         ; as slave, completed message transmission.
SRERR      EQU     14h         ; bus error detected when operating as a slave.

MGO        EQU     20h         ; Started Master message processing.
MRCVED     EQU     21h         ; As Master, received complete message from
                               ; slave.
MTXED      EQU     22h         ; As Master, completed successful message
                               ; transmission (slave acknowledged all data
                               ; bytes).
MTXNAK     EQU     23h         ; As Master, truncated message since slave did
                               ; not acknowledge a data byte.
```

```
MTXNOSLV   EQU     24h         ; AS Master, did not receive an acknowledgement
                               ; for the specified slave address.

TIMOUT     EQU     30h         ; TIMERI Timed out.
NOTSTR     EQU     32h         ; Master did not recognize Start.

; RAM locations used by I2C interrupt service routines.

MASCMD     DATA    20h
SUBADD     BIT     MASCMD.0
RPSTRT     BIT     MASCMD.1
SETMRQ     BIT     MASCMD.2

           DSEG    AT 24h

MSGSTAT:   DS      1           ; I2C communications status.
MYADDR:    DS      1           ; Address of this I2C node.
DESTADRW:  DS      1           ; Destination address + R/W  (for Master).
DESSUBAD:  DS      1           ; Destination subaddress.
MASTCNT:   DS      1           ; Number of data bytes in message (Master,
                               ; send or receive).

TITOCNT:   DS      1           ; Timer I bus watchdog timeouts counter.
StackSave: DS      1           ; SP save location (used when returning from
                               ; bus recovery routine).

MasBuf:    DS      4           ; Master receive/transmit buffer, 8 bytes.
SRcvBuf:   DS      4           ; Slave receive buffer, 8 bytes.
STxBuf:    DS      4           ; Slave transmit buffer, 8 bytes.

RBufLen    EQU     4h          ; The length of SRcvBuf
;*******************************************************************************
;          APPLICATION output pins and RAM definitions
;*******************************************************************************

; Outputs used by the application:

;;TogLED     BIT     P1.0       ; Toggling output pin, to confirm
                               ; that the ping–pong game proceeds fine.
;;ErrLED     BIT     P1.1       ; Error indication.

;;OnLED      BIT     P1.3       ;

; Application RAM
APPFLAGS     DATA    21h
  TRQFLAG   BIT     APPFLAGS.0
                      ; Flag for monitoring I2C transmission success.
  SErrFLAG  BIT     APPFLAGS.1

FAILCNT:   DS      1

TOGCNT:    DS      1          ; Toggle counter.
```

```
;*******************************************************************************
;
;                                Program Start
;
;*******************************************************************************
           CSEG
; Reset and interrupt vectors.

           AJMP    DONMON              ;;JUMP TO MONITOR
                                       ;Reset vector at address 0.

; A timer I timeout usually indicates a 'hung' bus.

           ORG     1Bh                 ; Timer I (I2C timeout) interrupt.
TimerI:    SETB    CLRTI
           AJMP    TIISR               ; Go to Interrupt Service Routine.
;*******************************************************************************
;                    I2C Interrupt Service Routine
;*******************************************************************************
;
; Notes on the interrupt mechanism:
;
; Other interrupts are enabled during this ISR upon return from XRETI.
; Limitations imposed on other ISR's:
; - Should not be long (close to 1000 clock cycles).  A long ISR will cause
;   the I2C bus to 'hang", and a TIMERI interrupt to occur.
; - Other interrupts either do not use the same mechanism for allowing
;   further interrupts, or if they do - disable TIMERI interrupt beforehand.
;
; The 751 hardware allows only one level of interrupts.  We simulate an
; additional level by software: by performing a RETI instruction (at location
; XRETI) the interrupt-in-progress flip-flop is cleared, and other interrupts
; are enabled.  The second level of interrupt is a must in our implementation,
; enabling timeout interrupts to occur during "stuck" wait loops in the I2C
; interrupt service routine.

           ORG     23h

I2CISR:    CLR     EI2         ; Disable I2C interrupt.
           ACALL   XRETI       ; Allow other interrupts to occur.
           PUSH    PSW
           PUSH    ACC
           MOV     A,R0
           PUSH    ACC
           MOV     A,R1
           PUSH    ACC
           MOV     A,R2
           PUSH    ACC

           MOV     StackSave, SP
           CLR     TIRUN
           SETB    TIRUN

           JB      STP,NoGo
           JNB     MASTER, GoSlave
           MOV     MSGSTAT,#MGO
           JB      STR,GoMaster
NoGo:      MOV     MSGSTAT,#NOTSTR
           AJMP    Dismiss             ; Not a valid Start.

XRETI:     RETI
```

```
;*************************************************************************
;                    Main Transmit and Receive Routines
;*************************************************************************

;  SLAVE CODE –
;  GET THE ADDRESS

GoSlave:   MOV     MSGSTAT,#SGO
AddrRcv:   ACALL   ClsRcv8
           JNB     DRDY, SMsgEnd       ; Must be some strange Start or Stop
                                       ; before the address byte was completed.
                                       ; Not a valid address.
STstRW:    MOV     C,ACC.0             ; Save R/W~ bit in carry.
           CLR     ACC.0               ; Clear that bit, leaving ”raw” address
           JZ      GoIdle              ; If it is a General Address
                                       ; – ignore it.

                                       ; NOTE:
                                       ; One may insert here a different
                                       ; treatment for general calls, if
                                       ; these are relevant.

           JC      SlvTx               ; It’s a Read – (requesting slave
                                       ; transmit).

; It is a Write (slave should receive the message).

;          Check if message is for us

SRcv2:     CJNE    A,MYADDR,GoIdle     ; If not my address – ignore the
                                       ; message.
           MOV     R1,#SRcvBuf         ; Set receive buffer address.
           MOV     R2,#RbufLen+1       ;
           SJMP    SRcv3
SRcvSto:   MOV     @R1,A               ; Store the byte
           Inc     R1                  ; Step address.
SRcv3:     ACALL   AckRcv8
           JNB     DRDY,SRcvEnd        ; Exit loop –end reception.
           DJNZ    R2,SRcvSto          ; Go to store byte if buffer not full.

; Too many bytes received – do not acknowledge.
           MOV     MSGSTAT,#SRLNG      ; Notify main that (as slave) we
                                       ; have received too long a message.
           ACALL   SLnRCvdR            ; Handle new data – slave event routine.
           SJMP    GoIdle

; Received a byte, but not DRDY – check if a legitimate message end.

SRcvEnd:   CJNE    R0,#7,SRcvErr       ; If bit count not 7, it was not
                                       ; a Start or a Stop.

; Received a complete message

           MOV     MSGSTAT,#SRCVD
                                       ; Calculate number of bytes received
           MOV     A,R1
           CLR     C
           SUBB    A,#SRcvBuf          ; number of bytes in ACC
           ACALL   SRCvdR              ; Handle new data – slave event routine.
           SJMP    SMsgEnd

; It is a Read message, check if for us.

SlvTx:     NOP

STx2:      CJNE    A,MYADDR,GoIdle     ; Not for us.
           MOV     I2DAT,#0            ; Acknowledge the address.
           JNB     ATN,$               ; Wait for attention flag.
```

```
            JNB     DRDY,SMsgEnd        ; Exception – unexpected Start
                                        ; or Stop before the Ack got out.
            MOV     R1,#STxBuf          ; Start address of transmit buffer.
STxlp:      MOV     A,@R1               ; Get byte from buffer
            INC     R1
            ACALL   XmByte
            JNB     DRDY,SMsgEnd        ; Byte Tx not completed.
            JNB     RDAT,STxlp          ; Byte acknowledge, proceed trans.
            MOV     I2CON,#BCDR+BIDLE   ; Master Nak'ed for msg end.
            MOV     MSGSTAT,#STXED
            ACALL   STXedR              ; Slave transmitted event routine.
            AJMP    Dismiss

SRcvErr:    MOV     MSGSTAT,#SRERR      ; Flag bus/protocol error
            ACALL   SRErrR              ; Slave error event routine.
            SJMP    SMsgEnd
StxErr:     MOV     MSGSTAT,#SRERR      ; Flag bus/protocol error
            ACALL   SRErrR
SMsgEnd:    JB      MASTER,SMsgEnd2
            JB      STR,GoSlave         ; If it was a Start, be Slave
SMsgEnd2:
            AJMP    Dismiss

; End of Slave message processing

GoIdle:
            AJMP    Dismiss

;
;

GoMaster:

; Send address & R/W~ byte

            MOV     R1,#MasBuf          ; Master buffer address
            MOV     R2,MASTCNT          ; # of bytes, to send or rcv
            MOV     A,DESTADRW          ; Destination address (including
                                        ; R/W~ byte).
            JB      SUBADD,GoMas2       ; Branch if subaddress is needed.

            ACALL   XmAddr

            JNB     DRDY,GM2
            JNB     ARL,GM3
GM2:        AJMP    AdTxArl             ; Arbitration loss while transmitting
                                        ; the address.
GM3:        JB      RDAT,Noslave        ; No Ack for address transmission.
            JB      ACC.0, MRcv         ; Check R/W~ bit
            AJMP    MTx

; Handling subaddress case:

GoMas2:     NOP                         ; Subaddress needed. Address in ACC.
            CLR     ACC.0               ; Force a Write bit with address.
            ACALL   XmAddr
            JNB     DRDY,GM4
            JNB     ARL,GM5
GM4:        AJMP    AdTxArl             ; Arbitration loss while transmitting
                                        ; the address.

GM5:        JB      RDAT,Noslave        ; No Ack for address transmission.
            MOV     A,DESSUBAD
            ACALL   XmByte              ; Transmit subaddress.
            JNB     DRDY,SMsgEnd2       ; Arbitration loss (by Start or Stop)
            JB      ARL,SMsgEnd2        ; Arbitration loss occurred.
            JB      RDAT,NoAck          ; Subaddress transmission was not ack'ed.
            MOV     A,DESTADRW          ; Reload ACC with address.
            JNB     ACC.0, MTx          ; It's a Write, so proceed
                                        ; by sending the data.
            ; Read message, needs rp. Start and add. retransmit.
```

```
            MOV     I2CON,#BCDR+BXSTR   ; Send Repeated Start.
            JNB     ATN,$
            MOV     I2CON,#BCDR         ; Clear useless DRDY while preparing
                                        ; for Repeated Start.
            JNB     ATN,$               ; expecting an STR.
            JNB     ARL,GM6
            AJMP    MArlEnd             ; oops – lost arbitration.
GM6:        ACALL   XmAddr              ; Retransmit address, this time with the
                                        ; Read bit set.
            JNB     DRDY,GM7
            JNB     ARL,GM8
GM7:        AJMP    AdTxArl             ; Arbitration loss while transmitting
                                        ; the address.
GM8:        JB      RDAT,Noslave        ; No Ack – the slave disappeared.
            SJMP    MRcv                ; Proceed receiving slave's data.

; A Write message.  Master transmits the data.

MTx:        NOP

MTxLoop:    MOV     A,@R1               ; Get byte from buffer.
            INC     R1                  ; Step the address.
            ACALL   XmByte
            JNB     DRDY,SMsgEnd2       ; Arbitration loss (by Start or Stop)
            JB      ARL,SMsgEnd2        ; Arbitration loss.
            JB      RDAT,NoAck
            DJNZ    R2,MTxLoop          ; Loop if more bytes to send.

            MOV     MSGSTAT,#MTXED      ; Report completion of buffer
                                        ; transmission.
            SJMP    MTxStop
NoSlave:    MOV     MSGSTAT,#MTXNOSLV
            SJMP    MTxStop
NoAck:      MOV     MSGSTAT,#MTXNAK
            SJMP    MTxStop

; Master receive – a Read frame

MRcv:       ACALL   ClaRcv8             ; Receive a byte.
            SJMP    MRcv2
MRcvLoop:   ACALL   AckRcv8
MRcv2:      JNB     DRDY,MArl           ; Other's Start or Stop.
            MOV     @R1,A               ; Store received byte.
            INC     R1                  ; Advance address.
            DJNZ    R2,MRcvLoop

; Received the desired number of bytes – send Nack.

            MOV     I2DAT,#80h
            JNB     ATN,$
            JNB     DRDY,MArl
            MOV     MSGSTAT,#MRCVED
            SJMP    MTxStop             ; Go to send Stop or Repeated Start.
; Conclude this Master message:
; Send Stop, or a Repeated Start

MTxStop:    JNB     RPSTRT,MTxStop2     ; Check if Repeated Start needed
                                        ; Around if not RPSTRT.
            MOV     I2CON,#BCDR+BXSTR   ; Send Repeated Start.
            SJMP    MTxStop3
MTxStop2:   MOV     C,SETMRQ            ; Set new Master Request if demanded
            MOV     MASTRQ,C            ; by SETMRQ bit of MASCMD.
            MOV     I2CON,#BCDR+BXSTP   ; Request the HW to send a Stop.

MTxStop3:   JNB     ATN,$               ; Wait for Attention
            MOV     I2CON,#BCDR         ; Clear the useless DRDY, generated
                                        ; by SCL going high in preparation
                                        ; for the Stop or Repeated Start.
            JNB     ATN,$               ; Wait for ARL, STP or STR.
            JB      ARL,MarlEnd         ; Lost arbitration trying to send
                                        ; Stop or a ReStart.
```

```
; Master is done with this message.  May proceed with new messages, if any,
; or exit.

           ACALL   MastNext           ; Master Event Routine.  May Prepare
                                      ; the pointers and data for the
                                      ; next Master message.

           JNB     MASTRQ,MMsgEnd     ; Go end service routine if MASTRQ
                                      ; does not indicate that the master
                                      ; should continue (was set according
                                      ; to SETMRQ bit, or by MastNext).

           JNB     STR,MMsgEnd        ; Return from the ISR, unless Start
                                      ; (avoid danger if we do not return:
                                      ; if there was a Stop, the watchdog
                                      ; is inactive until next Start).
           AJMP    GoMaster           ; Loop for another Master message
                                      ;
MMsgEnd:                              ;  End of Master messages,
           SJMP    Dismiss

; Terminate mastership due to an arbitration loss:

MArl:

           JNB     STR,MArl2          ; If lost arbitration due to other
                                      ; Master's Start, go be a slave.
           AJMP    GoSlave

Marl2:

           AJMP    Dismiss
; Switch from Master to Slave due to arbitration loss after completing
; transmission of a message.  The MASTRQ bit was cleared trying to write a
; Stop, and we need to set it again on order to retry transmission when the
; bus gets free again.

MArlEnd:
           SETB    MASTRQ             ; Set Master Request – which will get
                                      ; into effect when we are done as a
                                      ; slave.
           ACALL   MORERR            ;;INCREASE ERROR COUNT
           AJMP    MArl

; Handling arbitration loss while transmitting an address

AdTxArl:   JB      STR,MArl           ; Non–synchronous Start or Stop.
           JB      STP,MArl

; Switch from Master to Slave due to arbitration loss while transmitting
; an address – complete receiving the address transmitted by the new Master.

           CJNE    R0,#0,AdTxArl2
                                      ; Arl on last bit of address
                                      ; (R0 is 0 on exit from XmAddr).
           DEC     A                  ; The lsb sent, in which arl occurred
                                      ; must have been 1. By decrementing
                                      ; A we get the address that won.
           SJMP    AdAr3

AdTxArl2:
           RR      A                  ; Realign partially Tx'ed ACC
           MOV     R1,A               ; and save itin R1
           MOV     A,R0               ; Pointer for lookup table
           MOV     DPTR,#MaskTable
           MOVC    A,@A+DPTR
           ANL     A,R1               ; Set address bits to be received,
                                      ; and the bit on which we lost
                                      ; arbitration to 0
                                      ; Now we are ready to receive the rest
                                      ; of the address.
```

```
            MOV     I2CON,#BCXA+BCARL   ; Clear flags and release the clock.

            ACALL   RBit3               ; Complete the address using reception
                                        ; subroutine.
            JB      DRDY,AdAr3          ; Around if received address OK
            AJMP    SMsgEnd             ; Unexpected Start or Stop – end
                                        ; as a slave.
AdAr3:      AJMP    STstRW              ; Proceed to check the address
                                        ; as a slave.

MaskTable:  DB      0ffh,7Eh,3Eh,1Eh,0Eh,06h,02h,00h, ; 0ffh is dummy

; End I2C Interrupt Service Routine:

Dismiss:    ACALL   I2CDONE
            MOV     I2CON,#BCARL+BCSTP+BCDR+BCXA+BIDLE
            CLR     TIRUN
            POP     ACC
            MOV     R2,A
            POP     ACC
            MOV     R1,A
            POP     ACC
            MOV     R0,A
            POP     ACC
            POP     PSW
            SETB    EI2

            RET                 ; Return from I2C interrupt Service Routine
;*************************************************************************
;                Byte Transmit and Receive Subroutines
;*************************************************************************

;  XmAddr: Transmit Address and R/W~
;  XmByte: Transmit a byte

XmAddr:     MOV     I2DAT,A             ; Send first bit, clears DRDY.
            MOV     I2CON,#BCARL+BCSTR+BCSTP
                                        ; Clear status, release SCL.
            MOV     R0,#8               ; Set R0 as bit counter
            SJMP    XmBit2
XmByte:     MOV     R0,#8
XmBit:      MOV     I2DAT,A             ; Send the first bit.
XmBit2:     RL      A                   ; Get next bit.
            JNB     ATN,$               ; Wait for bit sent.
            JNB     DRDY,XmBex          ; Should be data ready.
            DJNZ    R0,XmBit            ; Repeat until all bits sent.
            MOV     I2CON,#BCDR+BCXA    ; Switch to receive mode.
            JNB     ATN,$               ; Wait for acknowledge bit.
                                        ; flag cleared.
XmBex:      RET

;
; Byte receive routines.
;
; ClsRcv8  clears the status register (from Start condition)
;          and then receives a byte.
; AckRcv8  Sends an acknowledge, and then receives a new byte.
;          If a Start or Stop is encountered immediately after the
;          ack, AckRcv8 returns with 7 in R0.
; ClaRcv8  clears the transmit active state and releases clock
;          (from the acknowledge).
;
;          A contains the received byte upon return.
;          R0 is being used as a bit counter.
;

ClsRcv8:    MOV     I2CON,#BCARL+BCSTR+BCSTP+BCXA
                            ;Clear status register.
            JNB     ATN,$
            JNB     DRDY,RCVex
            SJMP    Rcv8
```

```
AckRcv8:    MOV     I2DAT,#0            ; Send Ack (low)
            JNB     ATN,$
            JNB     DRDY,RCVerr         ; Bus exception – exit.
ClaRcv8:    MOV     I2CON,#BCDR+BCXA    ; clear status, release clock
                                            ;from writing the Ack.
            JNB     ATN,$

Rcv8:       MOV     R0,#7               ; Set bit counter for the first seven
                                        ; bits.
            CLR     A                   ; Init received byte to 0.
RBit:       ORL     A,I2DAT             ; Get bit, clear ATN.
RBit2:      RL      A                   ; Shift data.
            JNB     ATN,$               ; Wait for next bit.
            JNB     DRDY,RCVex          ; Exit if not a data bit (could be Start/
                                        ;  Stop, or bus/protocol error)
RBit3:      DJNZ    R0,RBit             ; Repeat until 7 bits are in.
            MOV     C,RDAT              ; Get last bit, don't clear ATN.
            RLC     A                   ; Form full data byte.
RCVex:      RET

RCVerr:     MOV     R0,#9               ; Return non legitimate bit count
            RET
;*************************************************************************
;           Timer I Interrupt Service Routine
;           I2C us Timeout
;*************************************************************************

; In addition to reporting the timeout in MSGSTAT, we update a failure
; counter, TITOCNT. This allows different types of timeout handling by the
; main program.


TIISR:      CLR     MASTRQ              ; "Manual" reset.
            MOV     I2CON,#BXSTP        ;
            MOV     I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP

TI1:        MOV     MSGSTAT,#TIMOUT     ; Status Flag for Main.
TI2:        ACALL   MORERR              ;;INC TITOCNT
TI4:        ACALL   RECOVER

            SETB    CLRTI               ; Clear TI interrupt flag.
            ACALL   XRETI               ; Clear interrupt pending flag (in
                                        ; order to re-enable interrupts).
            MOV     SP,StackSave        ; Realign stack pointer, re-doing
                                        ; possible stack  changes during
                                        ; the I2C interrupt service routine.
                                        ; TimerI interrupts in other ISR's
                                        ; were not allowed !
            AJMP    Dismiss             ; Go back to the I2C service routine,
                                        ; in order to return to the (main)
                                        ; program interrupted.

;*************************************************************************
;       Bus recovery attempt subroutine
;*************************************************************************

RECOVER:    CLR     EA
            CLR     MASTRQ              ; "Manual" reset.
            MOV     I2CON,#BCXA+BIDLE+BCDR+BCARL+BCSTR+BCSTP
            CLR     SLAVEN      ; Non I2C TimerI mode
            SETB    TIRUN       ; Fire up TimerI.  When it overflows, it
                                ; will cause I2C interface hardware reset.
            MOV     R1,#0ffh
DLY5:       NOP
            NOP
            NOP
            DJNZ    R1,DLY5
            CLR     TIRUN
            SETB    CLRTI

            SETB    SCL         ; Issue clocks to help release other devices.
            SETB    SDA
            MOV     R1,#08h
```

```
RC7:        CLR     SCL
            DB      0,0,0,0,0
            SETB    SCL
            DB      0,0,0,0,0
            DJNZ    R1,RC7
            CLR     SCL
            DB      0,0
            CLR     SDA
            DB      0,0
            SETB    SCL
            DB      0,0,0,0,0
            SETB    SDA
            DB      0,0,0,0,0  ; Issue a Stop.

Rex:        MOV     I2CON,#BCXA+BCDR+BCARL+BCSTR+BCSTP ; clear flags
            SETB    EA
            RET
;*************************************************************************
;
;                               Main Program
;
;*************************************************************************


; Message ping pong game.  Each message is transmitted by
; a processor that is a master on the I2C bus, and it contains one byte
; of data.  A processor that receives this data byte as a slave increments
; the data by one and transmits it back as a master.  The data received is
; confirmed to be a one increment of the data formerly sent, unless
; it is a ”reset” value, chosen to be 00h.
; The two participating processors have similar code, where the node
; address of the second processor is the destination address of this
; one, and vice versa.
; The first data byte each processor tries to send is 00h.  One of the
; processors will acquire the bus first, and the second processor that will
; receive this ”resetting” 00h will not attempt tp confirm it against an
; expected value.  It will simply increment and transmit it.  Subsequent
; receptions will be confirmed against the expected value, until 0ffh data
; bytes are sent and the game is effectively reset by the 00h resulting from
; the next increment.
; A toggling output (TogLED) tells the outer world that the ”ping pong”
; proceeds well.  If something unexpected happens we temporarily activate
; another output, ErrLED.
; The different tasks of the code are performed in a combination of main–
; line program and event routines called from the I2C interrupt service
; routine.



;  Initial set–ups:
;       Load CT1,CT0 bits of I2CFG register, according to the clock
;            crystal used.
;       Load RAM location MYADDR with the I2C address of this processor.
;    We load these values out of ROM table locations (R_CTVAL and R_MYADDR).
;    One may, instead, load with a MOV <immediate> command.

;;Reset:     MOV     SP,#07h             ;Set stack location.
RESET:      CLR     A
            MOV     DPTR,#R_CTVAL
            MOVC    A,@A+DPTR
            MOV     I2CFG,A             ; Load CT1,CT0 (I2C timing, crystal
                                        ; dependent).
            CLR     A
            MOV     DPTR,#R_MYADDR
            MOVC    A,@A+DPTR           ; Get this node’s address from ROM table
            MOV     MYADDR,A            ; into MYADDR RAM location.

;;            CLR     OnLED
```

```
;;Reset2:    CLR     ErrLED              ; Flash LED.
RESET2:    ACALL   LDELAY
;;           SETB    ErrLED
           CLR     SErrFLAG
           CLR     TRQFLAG
           MOV     FAILCNT,#50h
;;           SETB    TogLED
           MOV     TOGCNT,#050h        ; Initialize pin–toggling counter

; Enable slave operation.
; The Idle bit is set here for a restart situation – in normal
; operation this is redundant, as this bit is set upon power_up reset.
           MOV     I2CON,#BIDLE        ; Slave will idle till next Start.
           SETB    SLAVEN              ; Enable slave operation.

; Enable interrupts.
; This is necessary for both Slave and Master operations.
           SETB    ETI                 ; Enable timer I interrupts.
           SETB    EI2                 ; Enable I2C port interrupts.
           SETB    EA                  ; Enable global interrupts.

; Set up Master operation.

           MOV     MASCMD,#0h          ; ”Regular” master transmissions.
           MOV     DPTR,#PongADDR
           CLR     A
           MOVC    A,@A+DPTR
           MOV     DESTADRW,A          ; The partner address.  The LSB is
                                       ; low, for a Write transaction.
           MOV     MASTCNT,#01h        ; Message length – a single byte.

PPSTART:
           MOV     MasBuf,#00h

; ”Ping” transmission:

PP2:

           SETB    TRQFLAG
           SETB    MASTRQ
           MOV     R1,#0ffh
PP22:      JNB     TRQFLAG,PP3         ; Transmitted OK
           DJNZ    R1,PP22
MFAIL1:    DJNZ    FAILCNT,PP2
           ACALL   MORERR             ;;INCREMENT TITOCNT
           ACALL   RECOVER
           SJMP    Reset2

; ”Pong” reception:

PP3:       MOV     R0,#0ffh            ; Software timeout loop count.
PP31:      MOV     R1,#0ffh
PP32:      JB      TRQFLAG,PP2         ; Rcvd ok as slave, go transmit.
           JB      SErrFLAG,PP5
           DJNZ    R1,PP32
           DJNZ    R0,PP31
PPTO:      ACALL   RECOVER             ; Software timeout.
           AJMP    Reset2

;;PP5:       CLR     ErrLED              ; Receive error.
;;           ACALL   LDELAY
;;           SETB    ErrLED
PP5:       CLR     SErrFLAG
           AJMP    PPSTART

LDELAY:    MOV     R2,#030h            ;LONG DELAY
LDELAY1:   MOV     R1,#0ffh
           DJNZ    R1,$
           DJNZ    R2,LDELAY1
           RET
```

```
;*******************************************************************************
;  Slave and Master Event Routines.
;*******************************************************************************

;
;  Invoked upon completion of a message transaction.
;  This is the part of the application program actually dealing
;  with the data communicated on the I2C bus, by responding to
;  new data received and/or preparing the next transaction.

; Slave Event Routines
;
; These routines are invoked by the I2C interrupt service routine when a
; message transaction as a slave has been completed.  Our "application"
; reacts to a message received as a slave with the routine SRCvdR.
; The calls that indicate erroneous reception are treated the same way as
; erroneous data reception in the "ping pong" game.

;  SRcvdR
;          Invoked when a new message has been received as a Slave.

SRcvdR:    NOP
           MOV     A,SRcvBuf
           JNZ     SR2
           MOV     MasBuf,#01h          ; It was ping-pong reset value
           SJMP    SR3

SR2:       INC     MasBuf               ; The expected data.
           CJNE    A,MasBuf,ErrSR
           INC     MasBuf               ; Data for next transmission - the data
                                        ; received incremented by 1.

;          A successful two way data exchange.  Let the outside world know by
;          toggling an output pin driving a LED.  We actually toggle only
;          when a number of such exchanges is completed, in order to
;          slow down the changes for a good visual indication.

           DJNZ    TOGCNT,SR3
;;           CPL     TogLED              ; Toggle output
           XRL     TITOCNT,   #80H     ;;TOGGLE MSB LED
           MOV     TOGCNT,#050h        ;
           SETB    PSW.3               ;;RS TO 1
           MOV     LED,       @R0      ;;RAM POINTED TO BY R0
           CLR     PSW.3               ;;RS BACK TO 0
SR3:       CLR     SErrFLAG
           SETB    TRQFLAG             ; Request main to transmit
           RET

ErrSR:     SETB    SErrFLAG
           RET

;   SLnRcvdR
;          Invoked when a message received as a Slave is too long
;          for the receive buffer.

;   STXedR
;          Invoked when a Slave completed transmission of its buffer.
;          We do not expect to get here, since we do not plan to have
;          in our system a master that will request data from this node.
;

;   SRErrR
;          Slave error event subroutine.
;          In most applications it will not be used.
;

SLnRcvdR:
STXedR:
SRErrR:    JMP     ErrSR
```

```
;
;  MastNext – Master Event Routine.
;
;          Invoked when a Master transaction is completed, or terminated
;          ”willingly” due to lack of acknowledge by a slave.
;

MastNext:
           MOV     A,MSGSTAT
           CJNE    A,#MTXED,MN1
           MOV     FAILCNT,#50h
           CLR     TRQFLAG
           RET
MN1:
           RET

; I2CDONE
;          Called upon completion of the I2C interrupt service routine.
;          In this example it monitors exceptions, and invokes the bus
;          recovery routine when too many occurred.

I2CDONE:
           MOV     A,MSGSTAT
           CJNE    A,#NOTSTR,I2CD1
           ACALL   MORERR              ;;INCREMENT TITOCNT
           DJNZ    FAILCNT,I2CD1
           MOV     FAILCNT,#01h        ; Too many ”illegal” i2c interrupts
           CLR     EI2                 ; – shut off.
I2CD1:     RET


;*************************************************************************
;          I2C Communications Table:
;*************************************************************************

; We used table driven values for clarity.  One may use immediates to load
; these values and save several lines of code.

; Contents is used in the beginning of the main program to load
; RAM location MYADDR and the I2CFG register.
; The node address, in R_MYADDR, is application specific, and unique for
; each device in the I2C network.
; R_CTVAL depends on the crystal clock frequency.

R_MYADDR:   DB     4Ah          ; This node’s address
                               ;;NOTE THAT R_MYADDR AND PongADDR
                               ;;MUST BE SWITCHED ON THE OTHER
                               ;;’751
R_CTVAL:    DB     02h          ; CT1, CT0 bit values
;*************************************************************************
;          Application Code Definitions
;*************************************************************************

PongADDR:  DB      4Eh         ; The address of the ”partner” in
                               ; the ping–pong game.

;;I2CMON         THIS PROGRAM RUNS THE MONITOR ON
;;               THE SMALL TEST BOARD DESIGNED TO
;;               TEST THE I2C DRIVER CHIP.
;;               IT USES A ’751.
;
;
LED     EQU     P3
LDEL    EQU     022H
HDEL    EQU     LDEL + 1
```

```
SWITCH  EQU     P1
TOG     EQU     P0.2    ;TOGGLE SWITCH
RNAME   EQU     R0      ;R0 RAM POINTER
;
;
;
;
DONMON: MOV     SP,     #09H    ;SP=09,STARTS AT 0AH
        SETB    PSW.3           ;RS = 01
        CLR     PSW.1           ;PSW.1 FLAG=0
        JB      TOG,    ONLYAD  ;IF TOG 1, PSW1=0
        SETB    PSW.1           ;WRITE DESIRED
ONLYAD: JNB     TOG,    ONLYAD  ;WAIT FOR HI
HIWAIT: JB      TOG,    HIWAIT  ;NOW WAIT FOR LOW
        MOV     LDEL,   #0      ;DELAY TIMER
        MOV     HDEL,   #0
SDELAY: DJNZ    LDEL,   SDELAY  ;DELAY LOOP
        DJNZ    HDEL,   SDELAY  ;UPPER DELAY
        JB      TOG,    HIWAIT  ;FALSE ALARM,GO BACK
        MOV     RNAME,  SWITCH  ;VALID HI TO LO
        MOV     LED,    @RNAME  ;DISPLAY CONTENTS OF
                                ;  RAM OF RNAME
        JNB     PSW.1,  DONE    ;PSW1 FLAG, 0=DONE
STAYLO: JNB     TOG,    STAYLO  ;NOW WAIT FOR HI
HDELAY: DJNZ    LDEL,   HDELAY  ;LDEL=HDEL=0
        DJNZ    HDEL,   HDELAY
        JNB     TOG,    STAYLO  ;FALSE ALARM
        MOV     @RNAME, SWITCH  ;SUCCESSFUL LO TO HI
                                ;  SWITCH TO RAM
        MOV     LED,    RNAME   ;DISPLAY WHICH RAM
                                ;LOCATION FOR SWITCH
DONE:   CLR     PSW.3           ;RS BANK BACK TO 0
        AJMP    RESET           ;STARTS PING PONG
;
;
MORERR: PUSH    ACC
        MOV     A,      #7FH    ;;INCREMENT TITOCNT
        ANL     A,      TITOCNT
        XRL     A,      #7FH    ;;STOP AT 7F
        JZ      NOUP
        INC     TITOCNT
        SETB    PSW.3           ;;RS TO 1
        MOV     LED,    @R0     ;;DISPLAY NEW TITOCNT
        CLR     PSW.3           ;;RS BACK TO 0
NOUP:   POP     ACC
        RET
;
        END
```