Using the TMS370 SPI and SCI Modules

Kevin C. Self

Microcontroller Applications Engineering

Semiconductor Group
Texas Instruments Incorporated

Contributions by Paul Krause, Mark Palmer, and Al Lovrich

TEXAS INSTRUMENTS

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to or to discontinue any semiconductor product or service identified in this publication without notice. TI advises its customers to obtain the latest version of the relevant information to verify, before placing orders, that the information being relied upon is current.

TI warrants performance of its semiconductor products to current specifications in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Unless mandated by government requirements, specific testing of all parameters of each device is not necessarily performed.

TI assumes no liability for TI applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Contents

Illustrations

Tables

Section 1

Introduction

The TMS370 family of 8-bit microcontrollers has been designed with special features to facilitate serial communications. Both the TMS370X5X and TMS370CX10 devices incorporate the Serial Peripheral Interface (SPI) module. The TMS370X5X device also contains the Serial Communications Interface (SCI) module. These two modules greatly enhance the ability of the microcontroller to interface to other serial devices and common interfaces such as the industry standard RS-232. External hardware and software overhead are reduced by the flexibility and programmability of the interfaces.

This application report provides examples of hardware interfaces and software routines to illustrate the versatility of the SPI and SCI modules. Common applications of these modules will be discussed, which may be modified to suit the engineer's specific needs. Additional information on the Serial Interfaces may be found in the TMS370 Family Data Manual.

Section 2

Module Description: Serial Peripheral Interface (SPI)

2.1 The SPI - How It Works

The SPI module is a high-speed synchronous serial I/O port that shifts a serial bit stream of variable length and data rate between the device and other peripheral devices. The SPI is especially suited for multiprocessor and external peripheral communications, where the designer needs high-speed synchronous data transfer. The use of the SPI can greatly reduce overhead when connecting several peripherals together by transferring address or status information. The SPI can be used to communicate with other microcontrollers, serial shift registers, or display drivers. In addition, the SPI can be used to load memory (RAM or EEPROM) and allow the device to be reprogrammed in-socket.

A block diagram of the SPI is shown in Figure 2-1. In its simplest form, the SPI can be thought of as a fast, programmable shift register. Data is shifted in and out of the SPI through the SPIDAT register. Data to be transmitted is written to the SPIDAT register and received data is latched into the SPIBUF register to be read. Data transmission rates and data formatting are controlled by the SPI state logic.

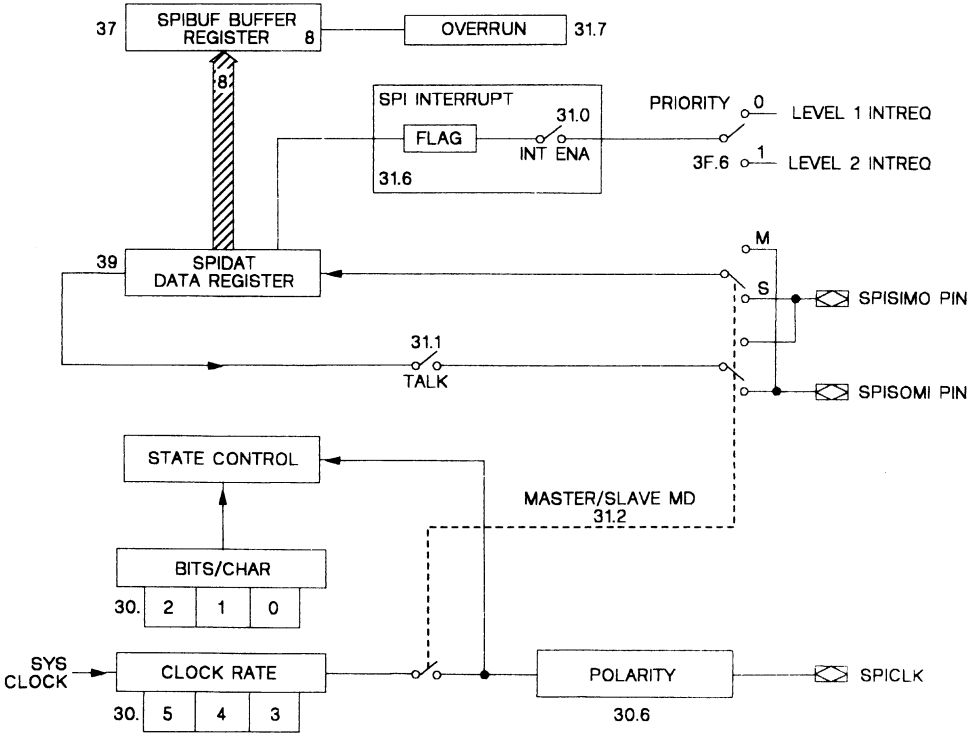


Figure 2-1. SPI Block Diagram

2.2 SPI Operating Modes

2.2.1 The Master Mode

The SPI operates in one of two modes. The Master mode is used when the SPI controls the data transfer. The Master SPI initiates and controls the data transfer by issuing the SPICLK signal. Writing data to the SPIDAT buffer starts the transfer by starting SPICLK and shifting the data out of the SPIDAT shift register onto the SPISIMO pin. New data are simultaneously gated in on the SPISOMI pin into the SPIDAT buffer.

Since the Master device controls the data transfer by issuing the SPICLK, the other devices must wait for the Master to start the transmission. Even if the Master is only interested in receiving data, it is still necessary to write "dummy" data to the SPIDAT register to initiate the transfer from the slave or external source.

Because of the way data are shifted through the SPIDAT Register, any data value in SPIDAT is always modified after a transmission, even if no new data value has been received into the register. The SPIDAT register will contain indeterminate data because no new data have been received.

2.2.2 The Slave Mode

The Slave mode is used when the SPI is controlled by another serial device. In the Slave mode, the SPI is dependent on an external clock source from a Master configured device to control the data transfer. An element of data resident in the SPIDAT buffer is shifted out upon receipt of a clock signal on the SPICLK pin, which in Slave mode becomes an input pin. Simultaneously, any data present on the SPISIMO pin are shifted into the SPIDAT register. The data transmission of a slave can be disabled by clearing the TALK bit. This allows many devices to be tied to the same serial network, but eliminates the possibility of write conflicts. Figure 2-2 illustrates two TMS370 devices in a Master/Slave connection.

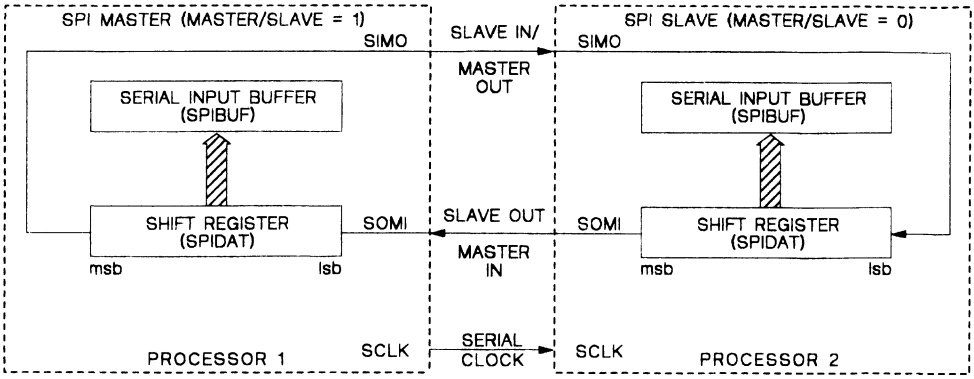


Figure 2-2. SPI Master/Slave Connection

2.3 Configuring the SPI

Data format, baud rate, interrupt generation, and operating mode are controlled by setting the SPI control registers shown in Appendix A. The SPI should be in an SPI SW RESET condition before changing any of the configuration registers. This freezes the state of the SPI while it is being configured. After setting the SPI parameters, release the reset. Before initiating a data transmission you need to initialize the parameters discussed in the following sections.

2.3.1 SPI Data Format - Transmitting and Receiving

Character length is programmable and can be set by the user from 1 to 8 bits. This is done by setting SPICCR bits 0-2 to the appropriate values shown in Table 2-1. If the character length is fewer than 8 bits, it is important to note the following:

1) Data must be written to SPIDAT left justified. Data is shifted out of the SPIDAT register MSB first, and if the character is not left justified the data will be corrupted.

2) Data is received into SPIDAT right justified. The MSB of the transmitted data is shifted into the LSB of SPIDAT and walked across. For character lengths < 8 bits, there will be extra bits containing information from previous transmissions that must be accounted for.

2.3.2 The SPICLK and Data Transfer Rate

The rate at which data are transferred out of SPIDAT is programmed by the SPI Bit Rate bits (SPICCR.3-5). The rate can be set from CLKIN/8 to CLKIN/1024 as shown in Table 2-2. The SPICLK rate is only utilized in the Master mode; in Slave mode the SPICLK rate is irrelevant because the clock signal is external. The SPICLK is output anytime a write is made to SPIDAT and the device is in the Master mode. The polarity of the clock bit can be set by the user (SPICCR.6) to latch the data on the rising or falling edge of the clock pulse. When an external clock is being used (Slave mode), the input clock frequency cannot be greater than CLKIN/32 to allow the internal clocks to synchronize.

Table 2-1. SPI Character Bit Length

Char2	Char1	Char0	Character Length
0	0	0	1
0	0	1	2
0	1	0	3
0	1	1	4
1	0	0	5
1	0	1	6
1	1	0	7
1	1	1	8

Table 2-2. SPI Clock Frequency

SPI† Bit Rate2	SPI† Bit Rate1	SPI† Bit Rate0	SPI Clock Frequency
0	0	0	CLKIN/8
0	0	1	CLKIN/16
0	1	0	CLKIN/32
0	1	1	CLKIN/64
1	0	0	CLKIN/128
1	0	1	CLKIN/256
1	1	0	CLKIN/512
1	1	1	CLKIN/1024

† If the SPI is a network slave, the module receives a clock on the SPICLK pin from the network master; and these bits have no effect on SPICLK. The frequency of the input clock should be no greater than the CLKIN frequency divided by 32.

A table showing the baud rates for common crystal frequencies vs. SPI bit rate values is shown below. See Table 2-3. The values were found using the formula

$$\text{SPI BAUD RATE} = \text{CLKIN} / (8 \times 2^{b})$$

where b = bit rate specified in the SPI Control Register (SPICCR.5-3) (range 0-7).

Table 2-3. Baud Rates for SPI Bit Rate Values

	Crystal/Oscillator Frequency (MHz)				
Divide By	**2.00**	**5.00**	**10.00**	**15.00**	**20.00**
8	250000	625000	1250000	1875000	2500000
16	125000	312500	625000	937500	1250000
32	62500	156250	312500	468750	625000
64	31250	78125	156250	234375	312500
128	15625	39062.5	78125	117187.	156250
256	7812.5	19531.2	39062.5	58593.7	78125
512	3906.25	9765.62	19531.2	29296.8	39062.5
1024	1953.12	4882.81	9765.62	14648.4	19531.2

2.4 Controlling the SPI through Interrupts and Flag Checking

The SPI interrupt logic can generate an interrupt upon receiving or transmitting a complete character as determined by the SPI character length. This provides a convenient and efficient way to handle the reception or transmission of data.

The interrupt can be enabled or disabled using the SPI INT ENA bit (SPICTL.0), and the interrupt priority set with the SPI PRIORITY bit (SPIPRI.6). Whether or not the SPI interrupt is enabled, the SPI INT flag (SPICTL.6) will be set upon the transmission or reception of a character. The SPI INT flag cannot be cleared as it is Read Only, but it is automatically cleared when SPIBUF is read, the SPI SW RESET Bit is set, or a system reset. Even if a data value is not going to be saved, it is still necessary to do a "dummy" read to clear the SPI INT flag. If the flag is not cleared and the interrupts are enabled then the interrupt routine will be called again as soon as it is completed.

Data transmission is not instantaneous in the SPI. It will be necessary to wait for the SPI to transmit or receive a character before reading from or writing to the SPIDAT register again. There are two ways to do this:

1) When the SPI has transmitted or received new data the SPI INT routine will be generated if enabled. The character is ready to be read if it was just received, or a new character can be transmitted if desired.

2) If the program cannot do anything until the new data value is received or transmitted, the SPI INT flag can be continuously polled until it goes high, at which time the character is ready to be read or a new one transmitted.

It is important to use one of the above methods to wait for the data before reading or writing again. Also, if the exact number of cycles is known, the transmission can be timed that way. When doing fast data transfers where the possibility of a data collision exists, polling the RECEIVER OVERRUN flag (SPICTL.7) will indicate if you have lost any data.

2.5 The TALK Bit and Multiprocessor Communications

If more than two processors are going to be connected to the same SPI data lines (SPISIMO/SPISOMI), then it will be necessary to limit the conversation to just two processors at a time. This is done through software using the TALK bit (SPICTL.1). When the TALK bit is 0, data transmission is disabled, but not reception. One device, usually (but not necessarily) the Master, sends out an address to the other devices in the network, who have their TALK bits set to 0. Since reception is not affected, all devices receive the transmitted address and compare it against their own address. If it matches, then that device sets its TALK bit and begins transmitting data. When it finishes, the receiving device clears its TALK bit and the network waits for another address. Another scheme for using the TALK bit is to have groups of characters (10 or so) transmitted in a block and have the address be the first character transmitted in a block. This way the address will occur at regular intervals and address checking does not need to be done constantly.

2.6 Considerations When Using the SPI

The most important thing to remember when writing SPI service routines is to keep your code short. Received data should be quickly removed from the SPIBUF register to prevent it from being overwritten. If you have to manipulate the data, wait until all the data have been received first. If your code involves long SPI routines there is a possibility that new data will be received before the previous data value has been read from the SPI buffer register. This becomes more and more important as the SPI baud rate increases.

2.7 Data Integrity and the SPI

The SPI was designed as a fast, simple interface to serial logic. As a result, it has no direct way to check for transmission errors. There are a number of software methods that can be used to check the integrity of the transmission. Parity checking is one of the most common and can be easily implemented in software for the SPI. Parity checking involves reserving one bit of the character to be used in setting the total number of 1s in a character odd or even. The Design Aids section of the TMS370 Family Data manual contains an example of a parity checking routine.

If you are going to be sending large blocks of data, there are coding methods that allow faster data transfer but still insure data integrity. Block checksums and other encoding methods can be found in most books on digital communications. These methods allow some degree of data integrity without significantly slowing the data transfer rate.

Section 3

SPI Module Software Examples

The following are examples of the various modes of operation and common software routines used in the utilization of the SPI. The SPI Control Registers are shown in Appendix A. The Register Equate table for the following examples is shown below.

Table 3-1. Common Equate Table

```
SPICCR  .equ   P030        ;SPI Configuration Control Register
SPICTL  .equ   P031        ;SPI Operation Control Register
SPIBUF  .equ   P037        ;Serial Input Buffer
SPIDAT  .equ   P039        ;Serial Data Register
SPIPC1  .equ   P03D        ;SPI Port Control Register 1
SPIPC2  .equ   P03E        ;SPI Port Control Register 2
SPIPRI  .equ   P03F        ;SPI Priority Control Register
```

3.1 Master SPI Configuration

This routine will show how to configure the SPI to operate in the Master mode. Data will be sent to a peripheral device. The value needed for the SPI Bit rate register is computed from the formula:

$$\text{SPI BAUD RATE} = \text{CLKIN} / (8 \times 2^{b})$$

where b is the bit rate from SPICCR.3-5, in the range from 0-7. This is important in applications where it is necessary to fix the real-time frequency of SPICLK, such as interfacing to slow peripheral logic.

The SPI in this example with a CLKIN of 20 MHz is connected to a shift register with a maximum operating frequency of 250 KHz. The bit rate needed is

$$b = \log_2 [\ \text{CLKIN} / (\text{SPI BAUD RATE} \times 8\ )\ ]$$
$$b = \log_2 [\ 20 \times 10^6 / (250 \times 10^3 \times 8\ )\ ] = 3.35 \text{ (approximately)}$$

Since only integers are allowed, the bit rate should be set to the next highest value, i.e., 4, which is CLKIN / 128. This gives an actual SPI BAUD Rate of 156.25 KHz, which is within the operating range of the shift register. The character size will be 8 bits.

```
SETMASTER   MOV   #0E7h,SPICCR       ;SPI Reset, clock active low, /128, 8 bits
            MOV   #006h,SPICTL       ;Master Mode, Enable TALK, Disable SPI INT
            MOV   #002h,SPIPC1       ;Set for SPICLK out
            MOV   #022h,SPIPC2       ;Enable SPISOMI, SPISIMO pins for SPI
            MOV   #040h,SPIPRI       ;SPI interrupts are low priority
            AND   #067h,SPICCR       ;Release SPI Reset
            ...                      ;Execute main program here.  When ready
                                     ; to transmit, call subroutine
            CALL  SENDDATA           ;Execute subroutine
            ...

SENDDATA    MOV   DATAOUT,SPIDAT     ;Move data to SPIDAT, initiate transmission
WAIT        BTJZ  #040h,SPICTL,WAIT  ;Loop until transmission complete
            MOV   SPIBUF,DUMMY       ;Dummy read to clear SPI INT flag
            RTS                      ;Return to main program
```

3.2 Slave SPI Configuration

This routine will show how to use the SPI interrupt to interrupt a program and load two 8-bit characters from the SPI. The program will call the SPI Interrupt upon receipt of an 8-bit character, save it, and wait for one more character. It will then save the values, and return to the main program. The characters will be saved in DATAMSB and DATALSB.

```
SETSLAVE    DINT                        ;Disable Global interrupts.
            MOV   #0E7h,SPICCR          ;SPI Reset, clock active low, /128, 8 bits.
            MOV   #001h,SPICTL          ;Slave mode, TALK disable, SPI INT Enable.

            MOV   #002h,SPIPC1          ;Set SPICLK.
            MOV   #022h,SPIPC2          ;Enable SPISOMI, SPISIMO pins for SPI.
            MOV   #040h,SPIPRI          ;SPI interrupts are low priority.
            MOV   #067h,SPICCR          ;Release SPI RESET.
            EINT                        ;Enable global interrupts.

            ...                         ;Insert main part of program here.  SPI
                                        ; INT will fetch characters when first
                                        ; is detected.

SPIINTR     MOV   SPIBUF,DATAMSB        ;Save first character already in buffer.
WAIT        BTJZ  #040h,SPICTL,WAIT     ;Wait until second character is received.
            MOV   SPIBUF,DATALSB        ;Save second character.
            RTI                         ;Return to main program.
```

3.3 Dynamic Bit Justification

On occasion it may be necessary to transmit characters of length less than 8 bits. As stated previously, the data need to be left-justified for transmitting from SPIDAT and right-justified when read from SPIBUF. If the SPI is accessing several peripherals with different character lengths, it may be more efficient to have one subroutine justify all the transmitted data.

This routine reads the value of the character length stored in SPICCR.0-2 and left-justifies the data to be transmitted as needed. If the character length is less than 5 bits, the routine swaps nibbles to save time. The value to be transmitted is stored in register DATA.

```
LJUSTIFY    MOV    SPICCR,NUMBITS       ;Save character length in temp register.
            XOR    #0FFh,NUMBITS        ;8 - numbits = number of shifts.
            AND    #007h,NUMBITS        ;clear all bits except character length.
            BTJZ   #004h,NUMBITS,ROLL   ;If < 4 shifts needed go to roll routine.
            SWAP   DATA                 ;More than 4 shifts, swap is faster.
            SUB    #004h,NUMBITS        ;Since we swapped, 4 rolls are complete.
            JZ     DONE                 ;If only 4 rolls needed we are done.
ROLL        RL     DATA                 ;Rotate one bit left.
            DJNZ   NUMBITS,ROLL         ;If not done rotating, continue.
DONE        MOV    DATA,SPIDAT          ;Data is now left justified, transmit.
```

3.4 Address Recognition by SPI

In multiprocessor systems using the SPI for communication it is necessary to keep conversations limited to two microprocessors at a time. The TALK bit is used to disable the transmit ability of a TMS370 in Slave mode until it sees its address, MYADDRESS, at which time it will transmit a byte of data. This example shows the SPI interrupt routine which is called when a character is received. If it is the correct address, the TALK bit is set, SPIDAT is loaded, and the TALK bit is cleared once again.

```
SPIINTR     MOV   SPIBUF,ADDRESS      ;Store received address
            CMP   #MYADDRESS,ADDRESS  ;Is it my address?
            JNZ   DONE                ;If not, ignore transmission.
            OR    #002h,SPICTL        ;Set TALK bit.
            MOV   DATA, SPIDAT        ;Load transmit buffer, wait for clock
                                      ; from "master".
WAIT        BTJZ  #040h,SPICTL,WAIT   ;Wait until character is sent.
            MOV   SPIBUF,DUMMMY       ;Dummy read to clear SPI INT flag
DONE        AND   #0FDh,SPICTL        ;Clear TALK bit.
            RTI                       ;Return from interrupt.
```

Section 4

SPI Module Specific Applications

4.1 Vacuum Fluorescent Display Driver

4.1.1 Use SPI to Transmit Data to Serial Shift Register

One common and very practical use of an SPI is sending serial data to a display. The use of simple software routines can simplify your design and eliminate expensive external hardware such as decoders. This example interfaces a TMS370C010 microcontroller to a vacuum fluorescent display. The only external logic necessary is one TMS0170 VF Display Driver. This device is a 33-bit shift register/display driver and is especially suited for serial display applications. The design uses only SPI and Timer 1 pins, so the designer does not need to dedicate any more I/O pins to the design. The schematic shown is for a generic serial display application, and it can be easily modified to work with an LED or LCD display.

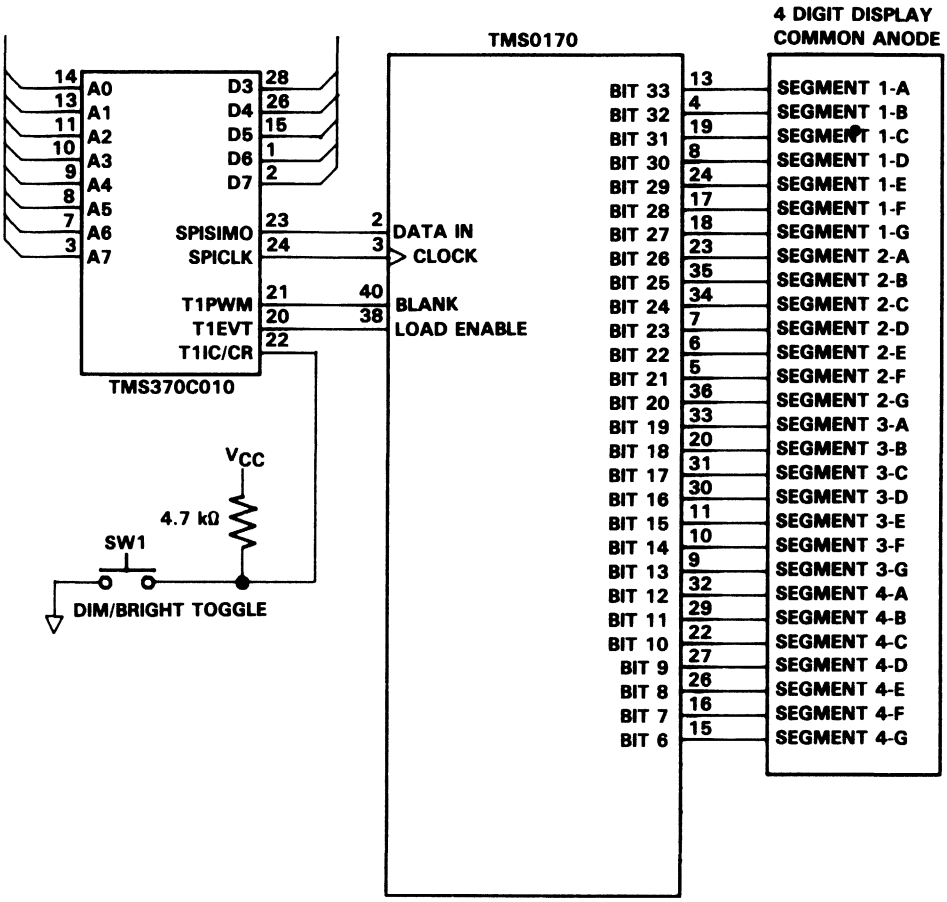


Figure 4-1. Vacuum Flourescent Interface Example

In this example, the display is pulsed periodically to adjust the intensity and update the display. In addition, the display may be put into a dim mode by toggling the T1 IC/CR pin. The Timer 1 PWM pin is used to control the brightness of the display by pulsing the blanking input of the TMS0170. The data are latched into the TMS0170 by pulsing the T1EVT pin which is configured as an output. When the new data value is to be displayed it is shifted out of the SPI.

The display update routine is controlled by Timer 1 interrupts. The Compare 1 and Compare 2 registers are set to control the refresh rate and intensity, respectively. Because the display is pulsed more frequently than new values are calculated, an interval counter is used to specify when it is time to update the display value. In this example, the following parameters are used:

Refreshes/sec	=	100 (Will eliminate flicker in display)
Display updates/sec	=	2
CLKIN Freq.	=	20 MHz
prescale divide	=	16
normal display intensity	=	90%
dim display intensity	=	40%

The Timer 1 compare register values are found from the formulas:

$$\text{Compare 1 value} = \frac{\text{CLKIN Freq.}}{4 \times \text{refreshes/sec} \times \text{prescale divide}}$$

$$\text{Compare 1 value} = \frac{20{,}000{,}000}{4 \times 100 \times 16} = 3125 \text{ or } 0C35h$$

Compare 2 value = intensity x compare 1 value
Compare 2 value (bright) = 0.9 x 3125 = 2812 or 0AFCh
Compare 2 value (dim) = 0.4 x 3125 = 1250 or 04E2h

By XORing the bright and dim values together, we get the logical "difference" between the two values. XORing the "difference" with either the bright or dim values will give the other. This is an easy and quick way to toggle the brightness.

DIFFMSB = Compare 2 value (dim) MSB XOR Compare 2 value (bright) MSB
DIFFLSB = Compare 2 value (dim) LSB XOR Compare 2 value (bright) LSB

The interval counter value is found from the following formula:

$$\text{interval} = \frac{\text{refreshes/sec}}{\text{updates/sec}}$$

interval = 100 / 2 = 50 or 32h

The source code for this application is as follows:

```
          .title    "Display Driver"

;       This routine will use the SPI and timer 1 modules to output values
;       to a serial display.  The display is updated every 0.5 seconds.
;       display intensity is changed by toggling T1IC/CR pin.

SPICCR    .equ  P030                ;SPI register assignments.
SPICTL    .equ  P031
SPIDAT    .equ  P039
SPIBUF    .equ  P037
SPIPC1    .equ  P03D
SPIPC2    .equ  P03E

T1CNTRMSB .equ  P040                ;Timer 1 register assignments.
T1CMSBLSB .equ  P041
T1CMSB    .equ  P042
T1CLSB    .equ  P043
T1CCMSB   .equ  P044
T1CCLSB   .equ  P045
T1CTL1    .equ  P049
T1CTL2    .equ  P04A
T1CTL3    .equ  P04B
T1CTL4    .equ  P04C
T1PC1     .equ  P04D
T1PC2     .equ  P04E
T1PRI     .equ  P04F

;       Allocate register space for the registers used in the application
;       routine.

DISPMSB   .equ  R5                   ;High byte of display value.
DISPLSB   .equ  R6                   ;Low byte of display value.
ICOUNT    .equ  R7                   ;Time between display refreshes.
DCOUNT    .equ  R8
DIGIT0    .equ  R10                  ;BCD values of display digits
DIGIT1    .equ  R11                  ; "
DIGIT2    .equ  R12                  ; "
DIGIT3    .equ  R13                  ; "
TEMPMSB   .equ  R14
TEMPLSB   .equ  R15
DUMMY     .equ  R16

;       Assign values for display intensity, and refresh period.
TIMER     .equ 3125                  ;100 interrupts/sec @ 20 MHz.
BRIGHT    .equ TIMER*9/10            ;Max intensity = 90 .
DIFF      .equ BRIGHT^|(TIMER*4/10);Min intensity = 40 .
INTERVAL  .equ 50

          .text 07000h

START     DINT                       ;Disable all interrupts.

;         SPI Initialization
          MOV   #0E6h,SPICCR         ;Reset SPI,data out on falling SPICLK,
                                     ; 7-bit characters.
          MOV   #006h,SPICTL         ;Master,Enable TALK, Disable SPI INT.
          MOV   #002h,SPIPC1         ;Enable SPICLK out.
          MOV   #020h,SPIPC2         ;Set SPISIMO out.

;         Set delays for brightness, and value updates
          MOV   #HI(TIMER),T1CMSB    ;Load Compare 1 register with delay.
          MOV   #HI(TIMER),T1CLSB    ; Time between refreshes (10 mS).
          MOV   #HI(BRIGHT),T1CCMSB  ;Set display to Bright intensity.
          MOV   #LO(BRIGHT),T1CCLSB  ; "
          MOV   #INTERVAL,ICOUNT     ;Temp register for interval counter.
```

```
;       Timer 1 Initialization
        MOV   #001h,T1PC1          ;Set T1EVT as general I/O.
        MOV   #062h,T1PC2          ;Set T1IC/CR to input.
        MOV   #040h,T1PRI          ;Set T1 interrupts to low priority.
        MOV   #071h,T1CTL4         ;Dual-compare,Disable interrupts.
        MOV   #005h,T1CTL1         ;System clock / 16.
        MOV   #000h,T1CTL3         ;Disable T1 interrupts, clear flags.
        MOV   #001h,T1CTL2         ;Disable Overflow interrupts,Reset T1.

;       Enable Timer1 & SPI
        MOV   #005h,T1CTL3         ;Enable T1EDGE INT, Enable T1C1 INT.
        MOV   #066h,SPICCR         ;Release SPI
        MOV   #0F0h,B              ;Move stack pointer value to B.
        LDSP                       ;Set stack Pointer.
        EINT                       ;Global interrupt enable.

MAIN                               ;Main Loop
;       Place major portion of code here.  This part of the program should
;       calculate the value to be displayed, scale it from 0 to 9999, and
;       store the result in DISPMSB and DISPLSB.  When Timer 1 counts down
;       the interrupt will be called and the program will jump to DISPLAY.

        ...

        MOV   #??,DISPMSB          ;Move sample value into memory.
        MOV   #??,DISPLSB          ;

        ...

        JMP   MAIN

;       Timer1 Interrupt Routine.
;       This routine pulls the value to be displayed from DISPMSB and
;       DISPLSB, converts it to a packed 4 nibble BCD number and shifts
;       the result out through the SPI. The routine checks to see whether
;       the routine was called by the timer or the T1C1 pin and clears
;       the appropriate flag.  DISPMSB and DISPLSB are temporary registers
;       and will not contain their original values upon completion of the
;       interrupt routine.

DISPLAY
        BTJZ  #080h,T1CTL3,TIMERINT;Was interrupt from T1IC/CR Pin?

;       T1IC/CR Pin called interrupt, toggle the intensity bright/dim.

        MOV   #003h,T1CTL1         ;Stop timer.
        MOV   #001h,T1CTL2         ;Reset timer (T1 SW RESET).
        MOV   #050h,T1PC2          ;Set PWM as general purpose I/O.
        MOV   #050h,T1PC2          ;Set T1PWM=1 (command must be repeated).
        MOV   #060h,T1PC2          ;Reenable T1PWM.
        MOV   T1CCLSB,TEMPLSB      ;Get current display intensity.
        MOV   T1CCMSB,TEMPMSB      ;
        XOR   #LO(DIFF),TEMPLSB    ;Toggle display intensity.
        XOR   #HI(DIFF),TEMPMSB    ;
        MOV   TEMPMSB,T1CCMSB      ;Update display intensity
        MOV   TEMPLSB,T1CCLSB      ;
        MOV   #005h,T1CTL1         ;Restart timer.
        AND   #07Fh,T1CTL3         ;Clear T1IC/CR interrupt flag.
        JMP   DONE                 ;End of display toggle: wait for update.

TIMERINT  DJNZ  ICOUNT,NOTNOW      ;Is it time for new value to be displayed?
                                   ; if it is not, do not calc new value.
        MOV   #INTERVAL,ICOUNT     ;Restore interval counter
```

```
;          Hex to BCD Conversion Routine.
           CLR  DIGIT2              ;Clear result registers.
           CLR  DIGIT0              ; "
           MOV  #16,R3              ;Set loop count.
LOOP       RLC  DISPLSB             ;Shift high bit out.
           RLC  DISPMSB             ;Carry contains the high bit.
           DAC  DIGIT0,DIGIT0       ;Double the number then add high bit.
           DAC  DIGIT2,DIGIT2       ; "
           DJNZ  R3,LOOP            ;Loop until multiplied 16 times.

           MOV  DIGIT0,DIGIT1       ;Save second digit
           MOV  DIGIT2,DIGIT3       ;Save third digit
           SWAP  DIGIT1             ;Swap BCD nibbles
           SWAP  DIGIT3             ;Swap BCD nibbles
           AND  #0Fh,DIGIT0         ;Clear high nibble
           AND  #0Fh,DIGIT1         ;Clear high nibble
           AND  #0Fh,DIGIT2         ;Clear high nibble
           AND  #0Fh,DIGIT3         ;Clear high nibble

;          Output Display Values.
;          This part actually outputs the BCD values to the display through the
;          SPI.  Note that in this example the display is limited to 4
;          characters, which gives a maximum value of 9999.

           MOV  #000h,DCOUNT        ;Set counter for data address.
NEXTCHAR   MOV  DCOUNT,B            ;Store DCOUNT in temp register.
           MOV  DIGIT0(B),A         ;Move BCD value of current char into A.
           XCHB A                   ;Move BCD value into B.
           MOV  TABLE(B),A          ;Look up 7seg value and store in A.
           MOV  A,SPIDAT            ;Move character byte into SPIDAT register.
WAIT1      BTJZ #040h,SPICTL,WAIT1  ;Wait for character to be sent.
           MOV  SPIBUF,DUMMY        ;Dummy read to clear SPI INT flag.
           INC  DCOUNT              ;Location of next digit register
           BTJZ #004h,DCOUNT,NEXTCHAR;If <4 characters sent, then send another.
           MOV  #005h,T1PC1         ;Toggle T1EVT to latch data.
           MOV  #001h,T1PC1         ;Pull T1EVT low again.
           OR   #001h,T1CTL4        ;Re-enable T1IC/CR interrupt here.  This
                                    ; allows delay between recognition of dim/
                                    ; bright toggles to debounce switch.
NOTNOW     AND  #0DFh,T1CTL3        ;Clear T1C1 interrupt flag.
DONE       RTI                      ;Return from interrupt.

;          Look-up table for converting BCD values to 7-segment display values.
;           Display BCD Value

TABLE      .byte  #07Eh      ; 0
           .byte  #00Ch      ; 1
           .byte  #0B6h      ; 2          The segments are decoded as follows:
           .byte  #09Eh      ; 3                 SEGMENT|gfedcba0
           .byte  #0CCh      ; 4                    BIT|76543210
           .byte  #0DAh      ; 5
           .byte  #0FAh      ; 6
           .byte  #00Eh      ; 7
           .byte  #0FEh      ; 8
           .byte  #0CEh      ; 9


;          Set up interrupt vector addresses

           .sect  "Vectors",07FF4h
           .word  DISPLAY           ;Timer 1 interrupt
           .word  START             ;All other vectors go to 'START'.
           .word  START
           .word  START
           .word  START
           .word  START
```

4.2 Bootstrap Loader

4.2.1 Reprogram Data or Program Memory through SPI Port

The SPI is very useful as a bootstrap loader for loading program or data information directly into RAM, EPROM, or EEPROM. The TMS370 family SPI and instruction set provide a fast, efficient way of moving serial data directly into memory. With the addition of a small interrupt service routine, the memory loader can become a bootstrap loader to reprogram a device in-socket, in the field. The interrupt routine must do the following:

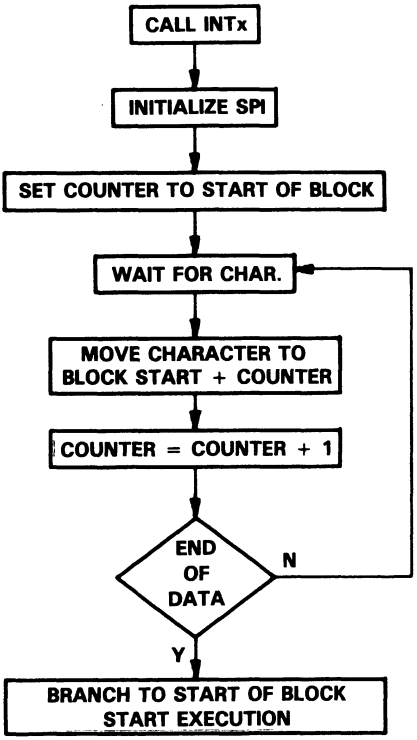


Figure 4-2. Flowchart of Bootstrap Loader Interrupt Service Routine

The interrupt routine loads the received data into program memory beginning at a specified location. After the data has been loaded in, the Program Counter is set to the beginning of the block and program execution is transferred to the new program. The new program can reconfigure the part as desired, or modify the program or data memory. An example of this is provided in the TMS370 family EEPROM Applications Report.

4.3 DSP Controller

4.3.1 Interface TMS370 SPI to TMS320C25 DSP

This example shows how the SPI can be used to communicate with other microprocessors. The exact method of communication varies from system to system, but the key parts can be shown to demonstrate how to interface the TMS320C25 and TMS370 serial ports. The TMS320C25 has a serial port similar to the TMS370, but with additional clocking and synchronization pins.

The C25's Serial Port's circuitry contains double buffering of both the transmit and receive registers. The C25 can transmit data in either 8-bit or 16-bit blocks. There are also two modes of transmission, with or without frame synchronization pulses (FSX/FSR). These serial ports (C25) are fully static, e.g., the data contained is not lost, and to transmit/receive data CLKX/CLKR must be present.

(For a complete description of the TMS320C25, see the TMS320C25 User's Guide.) An example of a typical interconnection using a TMS370C010 is shown below.

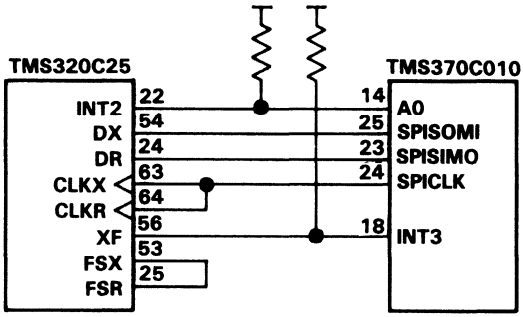


Figure 4-3. TMS370C010 - TMS320C25 Interface Example

In the setup shown below, data to and from both devices is clocked using the SPICLK. The TMS370 is configured so that receipt of an INT3 signal causes the TMS370 to load the SPIDAT register to start the SPICLK. If the TMS320C25 wants to initiate the conversation, it pulls INT3 low, waits for SPIDAT and is clocked out by the SPICLK. If the TMS370 wants to transmit, it sends out a logic 0 on A0, which is tied to INT2 on the TMS320C25. The TMS320C25 then loads the transmit buffer (DXR) to set up the synchronization circuitry(FSX/FSR). This in turn will cause the TMS320C25 to bring XF low, which activates the TMS370 INT3 routine to start the transfer. The seemingly complicated handshaking is necessary because both the TMS320C25 and the TMS370 want to be in control of the transmission. The TMS320C25 needs to generate its FSX/FSR pulse before data transmission, so it has to know when a data transfer is going to happen. By using the interrupt scheme to control the transmission, a data transfer will not start until both devices are ready. The following procedures summarize the actions required when either device wants to transmit:

- TMS320C25 wants to transmit:

C25 loads DXR	Places data to be transmitted in buffer.
C25 toggles XF low	Generates TMS370 INT3.
TMS370 executes INT3 routine	

- TMS370 wants to transmit:

TMS370 sets SPEAK370 bit	TMS370 is initiating the transmission.
TMS370 toggles A0 low	Generates TMS320 INT2.
C25 loads DXR	Places data to be transmitted in buffer.
C25 toggles XF low	Generates TMS370 INT3.
TMS370 executes INT3 routine	
C25 clears INT2 Flag	C25 did not initiate transmission.

- TMS370 INT3 routine

If first time to transmit/receive TMS370 transmits 1 character	Cause TMS320C25 to generate synchronization pulse (FSX/FSR).
TMS370 transmits 8 characters	TMS370 shifts out 8 characters to TMS320 TMS320 shifts out 8 characters to TMS370
If SPEAK370=0	TMS320C25 initiated transmission,
TMS370 clears INT3 flag	Ready for next transmission.
TMS370 clears SPEAK370 flag	Default TMS320 transmitting

Figure 4-4 shows the timing diagram of the Continuous mode of 8-bit data transmission.

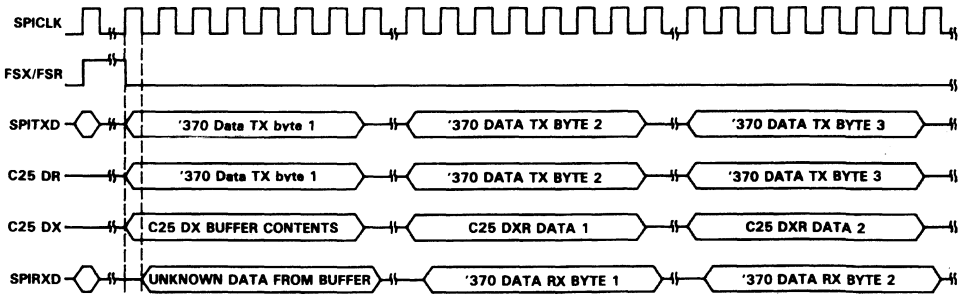


Figure 4-4. Continuous Mode No Frame Synchronization Pulse

Due to the double buffering of the transmitter, the TMS370 must also clock the C25 for one byte (word) of data to clear the buffer register, and then an-

other clocking sequence to receive the data. Therefore the C25 data is always received by the TMS370 one character after being loaded into the C25 DXR.

Different protocols can have different benefits, and which is used depends on the requirements of the system. If the system requires continual transmission of data from the C25 then the No Frame Synchronization Mode (No FSX/FSR pulse) allows greater through-put as well as less system overhead on the TMS370 processor.

If the system only has periodic data transmission of data between the two processors, and the data needs to be transmitted immediately, then the TMS370 needs to give 16 SPICLK cycles for the data from C25 to be received by the TMS370 sooner. The first byte from the C25 is dummy data. This procedure is not as efficient as the previous method, but for single bytes being transmitted between long intervals the data is transferred quicker. This is due to the TMS370 not having to wait for the C25 to load the next byte of transmit data into the buffer for transmission.

Both processors' flexible modes of transmission (such as C25's ability to transmit in either 8-bit or 16-bit mode) allows customization to the parameters of the desired system. The routines shown do not incorporate any checks if both the C25 routine and TMS370 routine try to communica te at the same time. If a situation like this occurred, both processors would think that they initiated the communication and would ignore the received data. If your system has the possibility of these asynchronous communications occurring at the same time, then a proper protocol needs to be defined.

The source code for the TMS370 in this application is as follows:

```
            .title      "TMS370 - TMS320C25 Interface Continuous Mode"

;           This is the framework of source code for an interface between a
;           TMS370 microcontroller and a TMS320C25 DSP.  The external
;           interrupts on both devices are used to synchronize the data transfer.

;           Set up equate table for Peripheral File registers used in the routine.

SP1CCR      .equ    P030                ;SPI register assignments.
SP1CTL      .equ    P031
SPIBUF      .equ    P037
SPIDAT      .equ    P039
SPIPC1      .equ    P03D
SPIPC2      .equ    P03E
SPIPRI      .equ    P03F
ADATA       .equ    P022
ADIR        .equ    P023
INT1        .equ    P017
INT2        .equ    P018
INT3        .equ    P019

;           Allocate register space for communications flags and data registers.

COM370      .equ    R4                  ;Status register for TMS320-TMS370 comm.
SPEAK370    .dbit   0,COM370            ;=1 if TMS370 is transmitting
FIRSTX      .dbit   7,COM370            ;=1 C25 in continuous mode, need to
                                        ; generate first sync pulse.
DATAIN      .equ    R5                  ;Received data.
DATAOUT     .equ    R6                  ;Data to be transmitted.
            .text   07000H

START       DINT                        ;Disable all interrupts.

            MOV     #100,B              ;SET STACK POINTER TO R100
```

```
        LDSP

        Initialize SPI, APORT, and communication status flag.

        MOV     #087h,SPICCR        ;Reset SPI,data out on rising SPICLK,
                                    ; 8-bit characters.
        MOV     #006h,SPICTL        ;Master,Enable TALK, Disable SPI INT.
        MOV     #002h,SPIPC1        ;Enable SPICLK out.
        MOV     #022h,SPIPC2        ;Set SPISIMO & SPISOMI out.
        MOV     #020h,SPIPRI        ;ENABLE EMULATOR SUSPEND
        MOV     #007h,SPICCR        ;Reset SPI,data out on rising SPICLK,
                                    ; 8-bit characters.
        MOV     #001h,ADIR          ;Set A0 as output.
        MOV     #001h,ADATA         ;Set A0 high.
        MOV     #01H,INT1           ;Initialize interrupt 1
        MOV     #01H,INT2           ;Initialize interrupt 2
        MOV     #01H,INT3           ;Initialize interrupt 3

        SBIT0   SPEAK370            ;Default is TMS370 not speaking.
        SBIT1   FIRSTX              ;Initialized as first Transmission
        MOV     #00H,DATAOUT        ;Initialize the data out register.

        EINT                        ;Enable Interrupts


;       Place main program here.  When TMS370 is ready to transmit, it will
;       call subroutine TRANSMIT.  This will cause an interrupt in the TMS320
;       which will in turn activate INT1 in the TMS370.  When the TMS320 wants
;       to initiate a transfer it will generate an INT1 interrupt, causing the
;       part to execute the INT1 service routine which will prepare it to
;       initiate a transfer.  Since both transmissions by the TMS320 and
;       TMS370 involve calling the TMS370 INT1 routine, the SPEAK370 bit is
;       set by the TMS370 when it initiates a transfer.  The data to be
;       transmitted is stored in DATAOUT and received data, if it is valid,
;       will be stored in DATAIN.
;
MAIN    ...

;
;       370 Initiates the data transfer to the C25, set appropriate Flags.
;
TRANSMIT SBIT1 SPEAK370             ;TMS370 is initiating transfer.
        AND   #0FEh,ADATA           ;Write 0 to A0, trigger INT1 in TMS320.
        OR    #001h,ADATA           ;Release TMS320 INT1.
        RTS                         ;Return from subroutine (after INT1 call)

;       Interrupt 3 service routine.  This routine is called when the
;       TMS370 is going to transmit or receive data.
;       Do frame sync once (FIRSTX).

INTR3   JBIT0 FIRSTX,DATA           ;If NOT the first Transmission goto DATA
        SBIT0 FIRSTX                ;Clear FLAG FIRSTX, this is first time
        MOV   #080h,SPICCR          ;Set Character size=1 bit
        MOV   #000h,SPICCR          ;Reset SPI,data out on rising SPICLK,
        MOV   #000h,SPIDAT          ;Transmit dummy pulse to make TMS320
                                    ;generate FSX/FSR sync pulse.
WAIT1   BTJZ  #040h,SPICTL,WAIT1    ;Wait until character has been sent.
        MOV   SPIBUF,DATAIN         ;Clear SPI Flag
        MOV   #087h,SPICCR          ;RESET SPI, Character size=8 bit
        MOV   #007h,SPICCR          ;Enable SPI, Character size=8 bit
DATA    MOV   DATAOUT,SPIDAT        ;Transmit data to TMS320.  If SPEAK370=0,
                                    ; this may be dummy data.
WAIT2   BTJZ  #040h,SPICTL,WAIT2    ;Wait until character has been sent.
        JBIT1 SPEAK370,DONE         ;If TMS370 is talking, do not save data.
        MOV   SPIBUF,DATAIN         ;Save received data, Clear SPI Flag
DONE    AND   #07Fh,INT3            ;Clear INT1 flag.
        SBIT0 SPEAK370              ;Clear TMS370 transmission flag.
        RTI                         ;End of INT3 routine.

INTR2   ...                         ;Interrupt 2 routine
        MOV   #01H,INT2             ;Clear and enable interrupt 2 Flag
```

```
          RTI

INTR1     ...                         ;Interrupt 1 routine
          MOV   #01H,INT1             ;Clear and enable interrupt 1 Flag
          RTI


          Set up interrupt vector addresses.

          .sect  "Vectors", 07FF4H
          .word  START                ;Vector goes to 'START'.
          .word  START                ;Vector goes to 'START'.
          .word  INTR3                ;INT3 vector.
          .word  INTR2                ;INT2 vector.
          .word  INTR1                ;INT1 vector.
          .word  START                ;Reset vector.
```

The source code for the TMS370C25 in this application is as follows:

```
*
*       sample program for interfacing the TMS370C10 and
*       the TMS320C25 serial ports.
*
*
DRR:    .equ     0              ; serial port receive  register
DXR:    .equ     1              ; serial port transmit register
IMR:    .equ     4              ; interrupt mask register
DATA:   .equ     96             ; general purpose register
*
        .sect  "AORG0"
        B       START           ; power-up reset
        .sect  "AORG1"
        B       INT2            ; interrupt 2 service routine
        .sect  "AORG2"
        B       RXINT           ; serial port receiver interrupt
*
        .sect  "AORG3"
START:  .equ     $
        DINT                    ; disable interrupts
        LDPK     0              ; point to page 0
        FORT     1              ; set serial port to 8-bit mode
        LALK     0ffc4h         ; enable interrupt 2
        SACL     IMR            ;
        STXM                    ; FSX is output
        RFSM                    ; continuos mode
        ZAC                     ; zero the accumulator
        SACL     DRR            ; initialize receive  register
        SACL     DXR            ; initialize transmit register
        EINT
*
*
*       Main body of program goes here. To initiate data transfer to the TMS370
*       CALL subroutine XMIT. Doing this tells the 370 to start clocking, and
*       the 320 knowns not to save the received data. When subroutine INT2 is
*       entered the 320 again tells the 370 to start clocking the serial port
*       and the 320 knows that it needs to save the data it receives. RXISR
*       lets the processor know when the when data has been receive.
*
*
XMIT    LAC      DXR            ; load data for transmission
        CALL     XMTISR         ; initiate data transfer to 370
        EINT                    ; enable interrupts
        IDLE                    ; wait for received data, do not save received
        RET                     ; data.
```

```
INT2:   .equ    $
        RPTK    40              ; give 370 enough time to detect
        NOP                     ; the XF generated interrupt. Then
        CALL    XMTISR          ; initiate data transfer to 370
        LALK    0ffd4h          ;
        SACL    IMR             ; enable int2, rxint
        EINT                    ; enable interrupts
        IDLE                    ; wait for received data
        LAC     DRR             ; load accumulator with Data Receive Register.
        ANDK    0ffh            ; save only lower 8 bits
        SACL    DATA            ; store received data
        RET
*
RXISR:  .equ    $               ; Serial receive interrupt
        EINT                    ; enable interrupts
        RET
*
XMTISR: .equ    $               ; initiate data transfer to 370 routine.
        RXF                     ; toggle XF flag low, causes 370 interrupt
        NOP                     ;
        NOP                     ;
        NOP                     ;
        SXF                     ; and then high, to clear. only want 370 INT3
        RET                     ; routine to execute once.
```

Section 5

SCI Module Description

5.1 The SCI - How It Works

The SCI module is a high-speed serial I/O port that permits Asynchronous or Isosynchronous communication between the TMS370 and other peripheral devices such as keyboards, display terminal drivers, and RS-232 interfaces. The SCI transmit and receive registers are double-buffered to prevent data collisions. In addition, the TMS370 SCI is a full duplex interface, allowing for simultaneous transmission and reception of data. Parity checking is done with on-chip hardware, eliminating the need for software overhead. The SCI is designed with the ability to do data formatting and integrity checking in hardware, further increasing execution speed.

The SCI module contains 4 major blocks as shown below: an 8-bit receiver and associated interrupt hardware, an 8-bit transmitter with its interrupt hardware, a programmable clock for setting the baud rate, and frame/format/parity error circuitry.

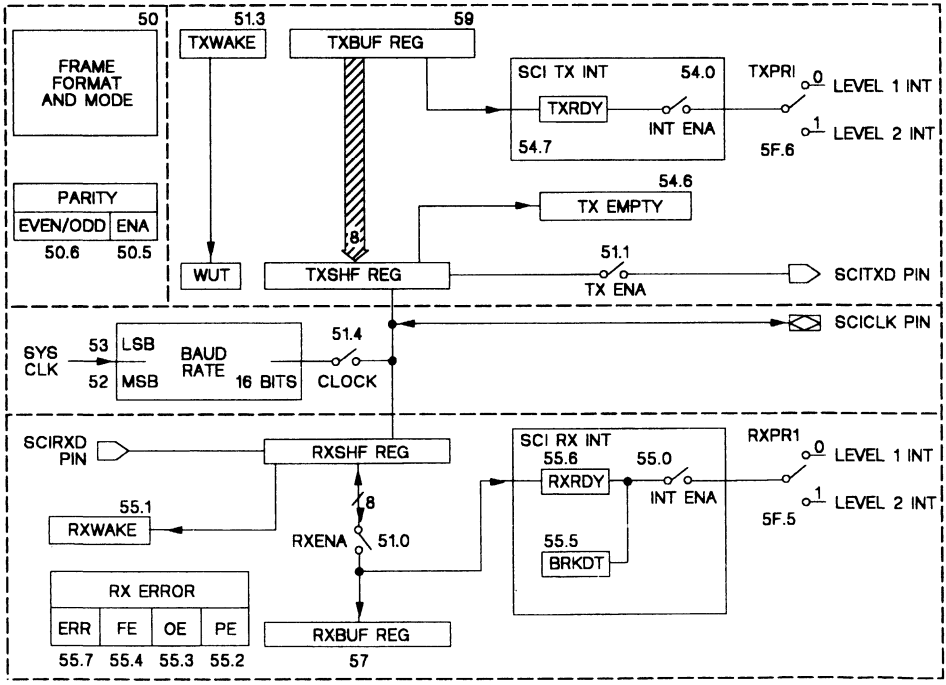


Figure 5-1. SCI Block Diagram

5.2 Choosing SCI Protocols and Formats

Data formatting is a characteristic of the SCI that sets it off from standard serial communications interfaces such as shift-registers. The basic unit of data is called a character and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, and optional parity and address bits. A character of data along with its formatting information is called a frame. Frames are organized into groups called blocks. A block of data usually begins with an address frame which specifies the destination of the data as determined by the user's protocol.

The start bit is a low bit at the beginning of each frame which marks the beginning of a frame. The SCI uses an NRZ (Non-Return-to-Zero) format, which means that in an inactive state the SCIRX and SCITX lines will be held high. Peripherals are expected to pull the SCIRX and SCITX lines to a high level when they are not receiving or transmitting on their respective lines.

The different SCI data framing formats are shown in Figure 5-2.

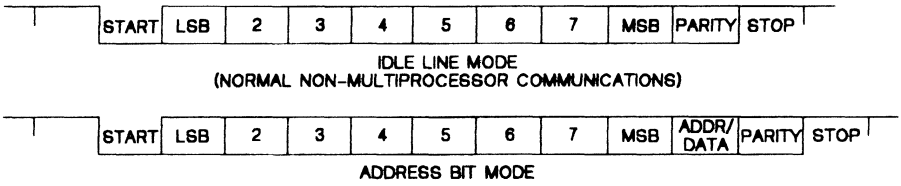


Figure 5-2. SCI Data Frame Formats

With the exception of the start bit and NRZ formatting, all the elements mentioned above are user programmable. These are controlled by the SCI Communication Control Register (SCICCR). The SCI Control registers are shown in Appendix B.

1) **Protocols:** The TMS370 SCI supports two protocols, the Idle Line and Address Bit modes. The two formats differ in how they distinguish the beginning of a block. The Address Bit mode adds an extra bit to each frame of transmitted data. Setting this bit to a logic 1 means that the current frame is an address. In the Idle line mode, an address frame is the first frame following an idle period of ten bits or more. The protocol is selected with the ADDRESS/IDLE WUP (SCICCR.3) bit.

2) **Character Length:** The length of the character to be transmitted is programmable from 1 to 8 bits. Data loaded into TXBUF is automatically right-justified (normal byte format) for transmission. When receiving data in RXBUF the data is also right-justified. Data are transmitted and received LSb first. If the character length is less than 8 bits the data value is automatically buffered by leading 0s. Character length is set by programming the SCI CHAR (SCICCR.0-2) bits to the values shown in Table 5-1.

Table 5-1. Transmitter Character Bit Length

SCI Char2	SCI Char1	SCI Char0	Character Length
0	0	0	1
0	0	1	2
0	1	0	3
0	1	1	4
1	0	0	5
1	0	1	6
1	1	0	7
1	1	1	8

3) **Parity:** Parity is a method of checking the integrity of a transmitted/received character. It sends an extra bit with the character to make sure that the sum of 1s in the character is an odd or even number. Parity checking and generation is done on-chip in hardware. It may be enabled or disabled, and if used it can be set odd or even. Bits 5 and 6 of the SCICCR register control the parity checking.

4) **Stop bits:** A stop bit is a high bit of data transmitted at the end of a frame. The number of stop bits can be one or two, depending on your application. In general, data integrity is more secure if two bits are used because the SCI is more likely to catch a framing (SCI synchronization) error. Adding the extra bit increases the number of bits transmitted per character, however, and slows the throughput of the serial port.

5.3 The SCI SW RESET Bit

The SCI SW RESET Bit (SCICTL.5) is used to reset the condition of the SCI state machine and operating flags. Writing a 0 to this bit sets the operating flags to their reset state and halts the operation of the SCI. This must be done before using the SCI for the first time or after a system RESET to guarantee the state of the SCI. Writing a 1 to the bit releases the SCI state machine and allows the SCI to resume operation.

It is good practice to reset the SCI by writing a 0 to the SCI SW RESET bit before setting up the control registers. The registers are then set to the desired value and a 1 is written to the SCI SW RESET Bit to release the SCI. This stops the operation of the SCI while it is being configured initially. The SCICTL control register values can be set in the same instruction that sets the SCI SW RESET bit to 1.

5.4 Operating Modes of the SCI

The SCI has two modes of operation. The first, Asynchronous, is the most commonly used mode and requires no synchronizing clock between the TMS370 and a peripheral device. When transmitting in the Asynchronous mode, each bit is held for 16 shift-clock cycles. This repetition insures that the data will be present long enough for the unsynchronized receiver to get valid data.

In the Isosynchronous mode, a common clock is used to increase system throughput by synchronizing the data transfer between the TMS370 and another serial port. In this mode, one bit of the frame is shifted out on every shift-clock cycle. Using the Isosynchronous mode gives a data transfer rate 16 times the corresponding Asynchronous SCICLK rate, but requires an extra line to carry the SCICLK signal. The Isosynchronous mode is superior to simpler synchronous communications such as the SPI in that you can achieve near synchronous communication speeds but still use formatting to assure data integrity. The format for Asynchronous and Isosynchronous communications is shown in Figures 5-3 and 5-4.

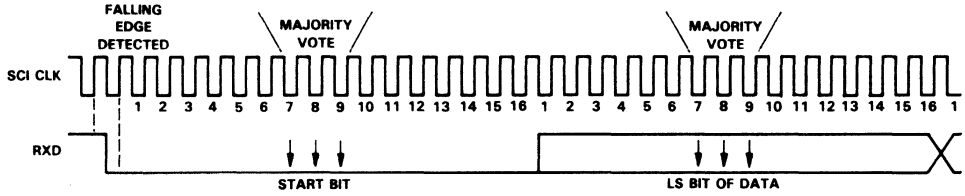


Figure 5-3. Asynchronous Communication Format

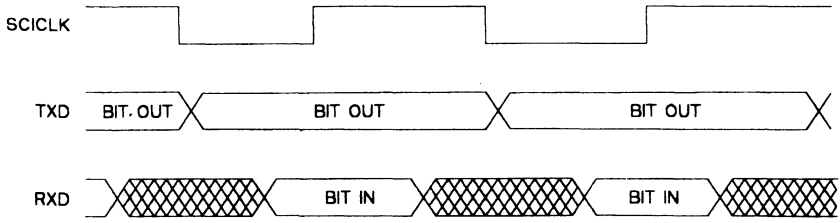


Figure 5-4. Isosynchronous Communication Format

5.5 Setting the SCICLK Pins and Baud Rate

The SCICLK is usually configured internally for Asynchronous communications, but can be external if your application requires it. For Isosynchronous communications the clock can be configured internally or externally depending on whether the TMS370 will be issuing the clock signal. If the SCICLK pin is not configured as the serial clock(SCICLK FUNCTION=0), then the pin may be used for general purpose I/O by setting SCICLK DATA DIR(SCIPC1.0) to the appropriate value and reading or writing to SCICLK DATA IN or DATA OUT. When the SCICLK is enabled (SCICLK FUNCTION=1), the contents of SCICLK DATA DIR, DATA IN, and DATA OUT are ignored.

Even though the clock is configured internally and is "independent" in the Asynchronous mode, it is necessary to have the baud rates set to exactly the same value in the transmitting and receiving devices so that the receivers can synchronize correctly on the frames. This holds whether communications are between two TMS370's or a TMS370 and a different peripheral device. The baud rate is set by writing a 16-bit value to the Baud Rate Select registers, BAUDMSB and BAUDLSB. The equations used to calculate the baud rate register values are shown below:

Asynchronous Baud Rate = CLKIN / [(BAUD RATE REG + 1) x 128]

Isosynchronous Baud Rate = CLKIN / [(BAUD RATE REG + 1) x 8]

Table 5-2 gives the Baud Rate Register values for common Asynchronous baud rates and frequencies. The values for Isosynchronous baud rates can be similarly calculated.

Table 5-2. Asynchronous Baud Rate Register Values for Common SCI Baud Rates

	Crystal Oscillator Frequency (MHz)							
	2.4576		7.3728		19.6608		20.00	
Baud Rate	BR Reg	%ERR	BR Reg	%ERR	BR Reg	%ERR	BR Reg	%ERR
75	255	0.00	767	0.00	2047	0.00	2082	0.02
300	63	0.00	191	0.00	511	0.00	520	-0.03
600	31	0.00	95	0.00	255	0.00	259	0.16
1200	15	0.00	47	0.00	127	0.00	129	0.16
2400	7	0.00	23	0.00	63	0.00	64	0.16
4800	3	0.00	11	0.00	31	0.00	32	-1.38
9600	1	0.00	5	0.00	15	0.00	15	1.73
19200	0	0.00	2	0.00	7	0.00	7	1.73
38400	-	-	-	-	3	0.00	3	1.73
156000	-	-	-	-	-	-	0	0.16

BR Reg = 16 bit Baud rate register value

Note: When using an externally generated SCICLK in Isosynchronous mode, the maximum speed at which the SCICLK can run is limited to CLKIN/40. This is necessary so that the internal clocks of the SCI have time to synchronize with the external clock. For this reason it is recommended to use the TMS370 to drive the master serial clock in a system where maximum throughput is a major concern.

5.6 SCI Receiver Operation

A flowchart showing the operation of the receiver is shown in Figure 5-5. When the SCI senses a falling edge on SCIRXD the flow described below begins. Depending on the protocol and format, the receiver checks for transmission errors and loads the data into RXSHF, the receiver shift register. When the number of bits specified by the SCI character length control bits have been read in, the contents of RXSHF are transferred to the receiver data buffer, RXBUF, and the RXRDY flag is set to show that the data value is ready to be read. An SCI receiver interrupt is generated if the SCI receiver interrupt is enabled.

If errors were detected, the RXERROR and specific error(Parity, Framing, Overrun, Break) flags are set by the hardware and operation continues. Error control is done in software. If multiprocessor communications are being used, frames received are checked to see if they are address frames and the appropriate bits are set.

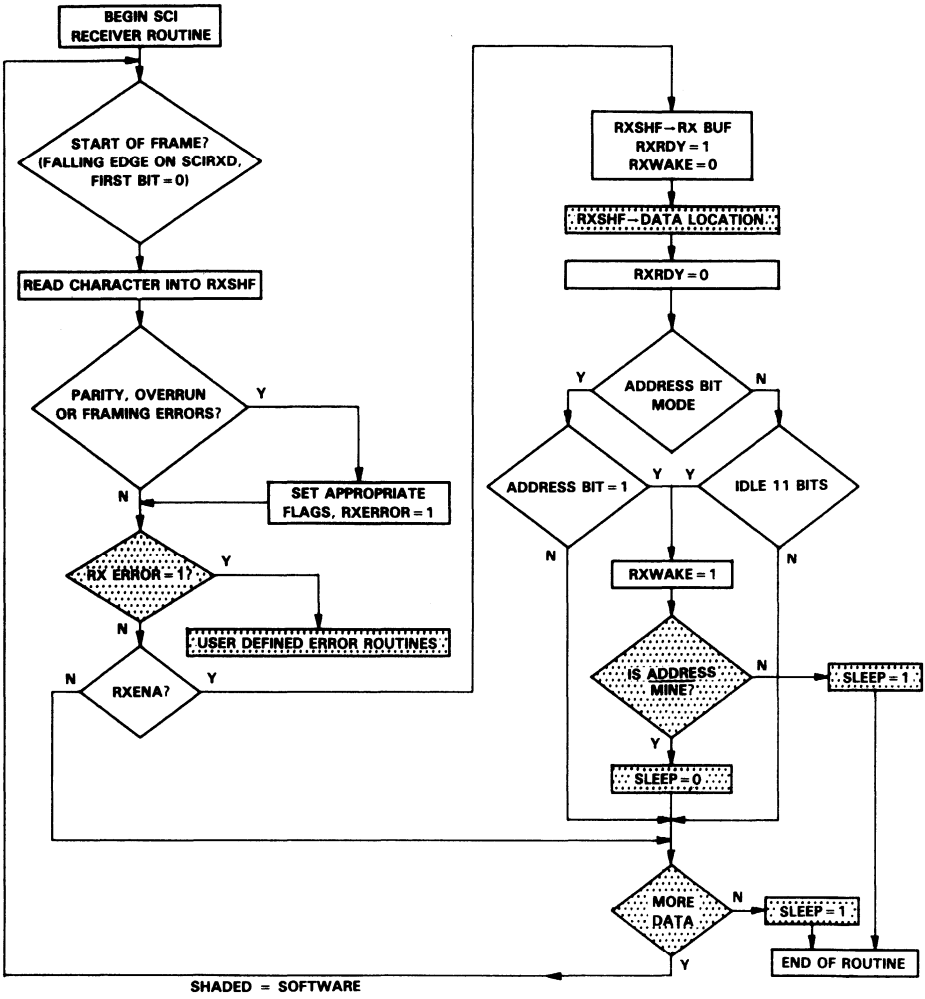


Figure 5-5. Receiver Operation Flowchart

5.7 SCI Transmitter Operation

A flowchart of the operation of the SCI transmitter is shown in Figure 5-6. The SCI transmitter is activated by loading the transmitter buffer, TXBUF, which clears the TXRDY flag. When TXSHF, the transmitter shift register, is empty, the contents of TXBUF are latched into TXSHF and the TXRDY flag is set to indicate the transmitter is ready for a new character. Depending on the protocol and format, the transmitter formats the data as needed to signal the beginning and end of frames of data.

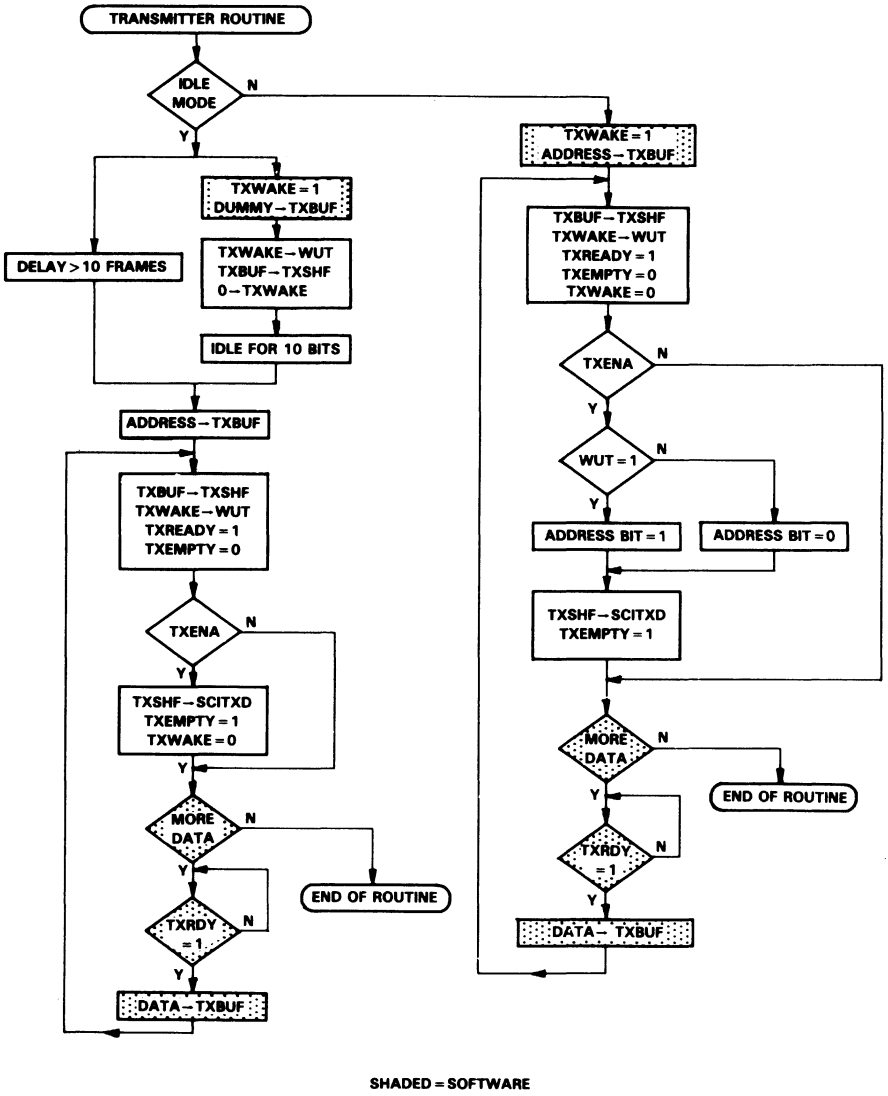


Figure 5-6. Transmitter Operation Flowchart

Data transmission is initiated by moving data into TXBUF. The status of the TXWAKE flag, set prior to writing to TXBUF, determines whether or not the current character is an address or data. The contents of TXWAKE and TXBUF are transferred to WUT(Wake Up Temporary) and TXSHF, respectively, to be shifted out as soon as the current transmission is complete. WUT and TXSHF are the actual transmission buffers and cannot be written to directly, only through TXWAKE and TXBUF. This double buffering of the transmission registers allows you to begin setting up for the transmission of a new character before the previous character has been shifted out of TXSHF, speeding up data transfer. Data is shifted out of TXSHF LSb first.

It should be noted that there are two ways to initiate a block signal when using the Idle Line protocol. The first is to write a 1 to the TXWAKE bit and then write dummy data to the TXBUF register. The transmitter will idle for 10 bits, signalling a block start. The other method is to simply wait for a period of time greater than the transmitter takes to transmit 10 bits (this is determined from the SCICLK frequency) and write the address to TXBUF.

5.8 SCI Interrupts and Flags

The SCI interrupt logic generates interrupt flags when it receives or transmits a complete character as determined by the SCI character length. This provides a convenient and efficient way of timing and controlling the operation of the SCI transmitter and receiver. The interrupt flags for the transmitter and receiver are TXRDY (TXCTL.7) and RXRDY (RXCTL.7), respectively. The TXRDY flag is set when a character is transferred to TXSHF and TXBUF is ready to receive a new character. In addition, when both the TXBUF and TXSHF registers are empty, the TX EMPTY flag (TXCTL.6) is set. The TXRDY flag signals that you can write another character to TXBUF, and the TX EMPTY flag is set when no new data value has been written to TXBUF and the SCI has finished transmitting.

When a new character has been received and shifted into RXBUF, the RXRDY flag is set. The status of data transfers can be checked by polling the flags. In this way the risk of a receiver overrun or transmitter corruption can be avoided.

The interrupts associated with the receiver and transmitter can be enabled or disabled using the SCI RX INT ENA (RXCTL.0) and SCI TX INT ENA (TXTCL.0) bits, respectively. When the interrupts are enabled and the flag is set, that particular interrupt is asserted. The priority of the SCI RX and TX interrupts can be set independently using the SCI TX and RX priority bits (SCIPRI.5-6). Note that unless the RXENA bit (SCICTL.0) is set, the received data will not be shifted into RXBUF and no interrupt will be generated. Data loaded into TXBUF will not be shifted out unless the TXENA bit is set.

5.9 Multiprocessor Communications

5.9.1 Using the SLEEP Bit

Quite often several serial ports will be tied to a common line, and a method is needed to restrict the conversation between two devices to avoid a communications conflict. The SLEEP flag can be used to disable an SCI until the start of a new block, at which time an address check can be made to see if that particular receiver is being addressed. The SLEEP bit is used in both Idle and Address Bit modes.

For the single microcontroller system SLEEP is initialized to 0. In a multiprocessor environment the SCI uses the SLEEP (SCICTL.2) flag to control when a specific receiver is addressed. In a multiprocessor system the SLEEP flag is initialized to a 1. Until a Sleeping receiver receives block start signal, the following happens:

1) SCIRX continues to load RXSHF

2) No format errors are recognized(OE=FE=PE=0), but BRKDT still is.

3) Data is shifted into RXBUF, but RXRDY is not set.

4) RXINT is disabled.

A block start signal acts like an alarm clock for the sleeping SCI receiver. A block start signal signifies that the current is an address. In the Address Bit mode, this is signalled by Address Bit=1. In the Idle mode, a block starts when a low bit is detected after an idle period of 10 bits or more. When a block start signal is received, the data received (an address) is loaded normally, including the RXWAKE flag. At this point the receiver interrupt will be called if enabled and the address byte received is checked, in software, against the "key" for that particular processor. If it matches, the software needs to clear the SLEEP bit and return to the main loop to read the rest of the block; if not, put the part back to bed (SLEEP=1), return to the program and wait until another block start is detected. Clearing the SLEEP bit informs the microcontroller that the following frames are data and not addresses.

5.9.2 Using the TXWAKE Bit

The TXWAKE bit is used by the transmitter to format the data going out as an address frame or a data frame. If a data character is being transmitted, the TXWAKE flag is left 0. If an address needs to be sent, TXWAKE is set to 1 before the address byte is loaded into TXBUF. The TXWAKE flag is automatically cleared when the byte is shifted from TXBUF to TXSHF.

Depending on which protocol you are using (Address Bit or Idle) setting the TXWAKE bit has different effects. If the Address Bit mode is being used, the Address bit will be set for that frame as it is transmitted out. If the Idle bit mode is being used, the transmitter goes idle (transmits a logic high) for a period of 10 bits when TXBUF is loaded. This is in effect a dummy write; the next data written to TXBUF will be the address and will be transmitted out as the address frame. Depending on your application, it may not be necessary to use the TXWAKE bit. If your design has only one peripheral or device tied to the SCI, then address bytes are not needed. TXWAKE can be left 0 for the duration of the transmission and no address bits will be sent.

5.9.3 Disabling the SCI Transmitter

Because the SCI uses the NRZ format, the transmitter is actually outputting a logic 1 when data is not being transmitted. If the SCITXD line is going to be tied to a bus it will be necessary to put the line in a three-state condition so that the line is not constantly being driven high. This is done by reconfiguring the SCITX pin as general I/O after transmission. Setting the SCI TXD FUNCTION bit = 0 and the SCI TXD DATA DIR = 0 will put the pin into an input configuration that will prevent bus conflicts from occurring.

5.9.4 Choosing the Right Protocol

Because no idle period is needed between blocks, the Address mode is more efficient when sending small blocks of data, typically fewer than 10 frames. When sending larger blocks, however, it is usually more efficient to use the Idle Line mode because the extra bit per frame used in Address Bit mode becomes more significant. If the receiver does not change very often, the Idle Line mode is probably the best choice because address bytes are not sent that often. For single-processor applications, the Idle Line mode is usually used. The Address Bit mode, because it is formatted to accomodate addressing easily, is frequently used for multiprocessor designs.

An important consideration to take into account when using the Idle line mode is the amount of time it takes for software overhead. If the transmitter must service a lot of code between transmissions, then there is a possibility that the transmitter will inadvertently remain idle for ten bits or more, accidentally sending a block start signal. This becomes more and more likely as the transmitter service routines become more involved and the Baud rate increases. If you are going to be using complicated transmitter routines it may be a good idea to use the Address Bit protocol, even though the extra bit may seem unnecessary in the short run.

The TMS370 SCI was designed for maximum compatibility with existing microcontroller protocols. For the purposes of interfacing to other microcon-

trollers, the Address-Bit mode is compatible with the I8051 protocol. The Idle Line mode is in accordance with the MC6801 protocol.

5.10 Timing the Flow of Data

5.10.1 Transmitting

A few items need to be taken into consideration when using the SCI transmitter. It is important not to write data to the TXBUF register before it has shifted its data to the TXSHF register. This becomes more likely as the SCI Baud rate decreases and it takes longer to shift out the data. Unlike the SCI receiver, there is no transmitter overrun flag. The programmer must guard against this in his software.

There are two ways to make sure that characters do not get overwritten in TXBUF. The first is to use transmitter interrupts to control the loading of TXBUF. By setting TXINT ENA (TXCTL.0), the TX interrupt will be called when TXRDY is set. Because TXRDY is only set (and the TX interrupt called) when TXBUF is ready to receive a new character, there is no possibility of an overwrite if the instruction is placed in the interrrupt routine. Also, in a large program that transmits from many locations in its code, interrupt-driven transmit routines are more memory efficient than other methods.

The second way to prevent transmitter overruns is to poll the TXRDY flag (TXCTL.6). If using interrupt-driven routines is not practical in your application, or the program can do nothing until the data is transmitted, it may be more practical to load TXBUF and simply loop until the TXRDY flag is set. Use the BTJZ instruction to loop on itself until the flag is set. Several of the application examples shown later use this technique.

5.10.2 Receiving

By far the most important thing to remember when receiving data is to keep your receiver routine short. If a large amount of data is being received, store it in a table and manipulate it later. As soon as the receiver interrupt is called move the data out of RXBUF and store it in another register. This will prevent new data from overwriting data that is already in RXBUF and causing a receiver overrun.

5.11 Detecting Transmission Errors

The advantage of formatting data is the ability to detect communication errors when they occur. The SCI has hardware designed features that make it easy to detect such errors. The SCI receiver has flags to detect the following errors:

1) **Parity:** The parity error bit, PE (RXCTL.2), is set when the number of 1s plus the parity bit is not odd or even, depending on whether the parity is odd or even according to the EVEN/ODD PARITY bit (SCICCR.6). Parity checking can be disabled with the PARITY ENABLE bit(SCICCR.5).

2) **Receiver Overrun:** If data is not read from RXBUF before new data is received, the overrun error bit, OE (RXCTL.3) will be set. This signifies that data received was lost before it could be read.

3) **Framing:** A framing error occurs when the receiver loses synchronization with the transmitter. The framing error bit, FE(RXCTL.4) is set when the receiver does not detect a stop bit (or bits) as expected at the end of a frame.

4) **Break Detect:** The break detect flag, BRKDT(RXCTL.6), is set when the receiver detects 11 continuous low bits after the FE flag is set. Because of the NRZ communications format, this signifies a serious error in either the transmitter or the transmission line. This will cause an interrupt if enabled.

5) **RX ERROR:** Any time any of the above flags are set, the RX ERROR flag is set. The RX ERROR flag provides an easy and quick way to see if an error has occurred without polling each bit.

All of the above flags are cleared by reading RXBUF, executing an SCI SW RESET, or executing a system reset.

Of course, if data integrity is not an issue, you can ignore checking for errors. Disabling parity decreases the number of bits sent per frame, so in effect a faster transmission rate is achieved. In most cases, however, you will want to make sure your data has been transmitted correctly and leave it enabled.

In addition to on-chip error checking, there are a number of coding methods that allow faster data transfer but still insure data integrity. Encoding the data before it is sent can speed up the transfer without sacrificing quality. Encoding methods such as Cyclic Redundancy Checking (CRC) or block encoding can be found in most good books on digital communications. The checksum method of error checking involves checking parity on a block of data as well as the individual characters.

5.12 What to Do with Transmission Errors

Once you get an error, what do you do? Unfortunately, with digital communications there is no easy way to correct bad data, and then it can only be done if complicated encoding schemes are used. The simplest way to correct the data is to have the transmitter retransmit the data. This is usually done by reserving a special NAK(Negative AcKnowledge) character in the data to signal when an error has been detected by the receiver. When the receiver detects an error, it transmits the NAK character to the other device, signaling it to retransmit the data.

Section 6

SCI Module Software Examples

The following are examples of the various modes of operation and common software routines used in the implementation of the SCI. The SCI Control Registers are shown in Appendix B. The Register Equate table for the following examples is shown below:

Table 6-1. Common Equate Table

```
SCICCR    .equ   P050    ;SCI Communication Control Register
SCICTL    .equ   P051    ;SCI Operation Control Register
BAUDMSB   .equ   P052    ;Baud Rate Select MSB Register
BAUDLSB   .equ   P053    ;Baud Rate Select LSB Register
TXCTL     .equ   P054    ;Transmitter Interrupt Control and Status Register
RXCTL     .equ   P055    ;Receiver Interrupt Control and Status Register
RXBUF     .equ   P057    ;Receiver Data Buffer Register
TXBUF     .equ   P059    ;Transmit Data Buffer Register
SCIPC1    .equ   P05D    ;SCI Port Control Register 1
SCIPC2    .equ   P05E    ;SCI Port Control Register 2
SCIPRI    .equ   P05F    ;SCI Priority Control Register
```

6.1 SLEEP Bit - Multiprocessing Control

By using the SLEEP bit (SCICTL.2), several microprocessors can be tied to common SCIRXD and SCITXD lines. This example shows a "slave" microcontroller set to listen for its own address and load its RAM with a block of data of a fixed size when it is addressed. The data is received through the use of an interrrupt routine. When the part recognizes its own address it clears the SLEEP bit and subsequent characters are loaded into memory starting at register DATA+BLOCKSIZE-1 and continuing down to register DATA. The SLEEP bit is then set and the routine waits for the next address.

```
B1200   .equ   2082

        MOV    #007h,SCICCR          ;1 stop bit, no parity, Isosynchronous,
                                     ; Idle line protocol, 8-bit characters.
        MOV    #00h,SCICTL           ;SCI SW RESET
        MOV    #HI(B1200),BAUDMSB    ;Set for 1200 BAUD @ 20 MHz.
        MOV    #LO(B1200),BAUDLSB    ;
        MOV    #001h,RXCTL           ;Enable SCIRX INT.
        MOV    #002h,SCIPC2          ;Set SCIRXD as input.
        MOV    #060h,SCIPRI          ;SCIRX/SCITX interrupts low priority.
        MOV    #033h,SCICTL          ;Release SCI, SLEEP=0,RXENA,TXENA.

        ....                         ;Main Code here.

RXINT                                ;Receiver interrupt routine.
        BTJZ   #004h,SCICTL,AWAKE    ;If SLEEP=0, do not check  address.
        XOR    #ADDRESS,RXBUF        ;Is Address mine?
        JNE    DONE                  ;If not, go back to sleep.
        MOV    #011h,SCICTL          ;Clear SLEEP bit.
        MOV    #BLOCKSIZE-1,BCOUNT   ;Get size  of Block(-1 for address).
        JMP    DONE
                                     ;Address is mine, start reading data.
AWAKE   PUSH   B                     ;Save contents of A & B registers.
        PUSH   A
        MOV    BCOUNT,B              ;Put pointer and data in temp registers.
        MOV    RXBUF,A
        MOV    A,DATA(B)             ;Store character in DATAIN table.
        POP    A                     ;Restore contents of A & B register
        POP    B
        DJNZ   BCOUNT,DONE           ;Wait for next character.
        MOV    #015h,SCICTL          ;Put part back to sleep
DONE    RTI                          ;Return from interrupt
```

6.2 System Controller Configuration

In this example the device is setup as a system controller that requests data from specific devices using the Idle Line protocol. The address of the device to be interrogated is stored in ADDROUT. The address is sent out and the controller waits for the data to be sent to it. If an error occurs the controller asks for the data to be transmitted again.

```
B1200   .equ  129

        MOV   #00h,SCICTL           ;SCI SW Reset.
        MOV   #077h,SCICCR          ;1 stop bit, even parity, Asynchronous,
                                    ; Idle line protocol, 8-bit characters.
        MOV   #HI(B1200),BAUDMSB    ;Set for 1200 BAUD.
        MOV   #LO(B1200),BAUDLSB    ;
        MOV   #001h,RXCTL           ;Enable SCIRX INT.
        MOV   #002h,SCIPC2          ;Set SCIRXD as input.
        MOV   #060h,SCIPRI          ;SCIRX/SCITX interrupts low priority.
        MOV   #032h,SCICTL          ;Internal Clock, TXENA, RXENA.

        ....                        ;Main Code here.
        CALL  XMIT                  ;Call subroutine to transmit character.
        ...                         ;More Main Code here.

XMIT    MOV   #01Ah,SCICTL          ;Set TXWAKE: address transmission.
        MOV   #000h,TXBUF           ;Dummy write to cause SCITX idle.
        MOV   ADDROUT,TXBUF         ;Send Address.

WAIT    BTJZ  #040h,RXCTL,WAIT      ;Wait for answer.
        BTJO  #080h,RXCTL,XMIT      ;If error occurred, retransmit.
        MOV   RXBUF,DATAIN          ;Save received data.
        RTS                         ;Return to main program block.
```

6.3 Nine-Bit Data Protocol

Data transfer can be made more efficient by transferring more bits per character. By using the Address bit mode, an extra bit of data can be added to each character, creating in effect a nine-bit character protocol. Extra bits, BITNINE for the transmitter and HIGHBIT for the receiver, are used to hold the ninth bits and can be assigned to any unused register. The transmit and receive routines are similar to the 8-bit character length routines with the addition of code to monitor the ninth bit. The transmitter routine, upon finding BITNINE=1 will set the TXWAKE bit. This will signal the transmitter that address character is going out and to set the address bit=1. If the TXWAKE flag is not set the address bit will remain 0. The receiver checks to see the value of the ninth bit by polling the status of the RXWAKE flag. If it is set then the received character is an address and the ninth bit is set; otherwise it is not an address and the ninth bit is 0.

```
B1200    .equ  129

         MOV   #000h,SCICTL          ;SCI SW Reset.
         MOV   #07Fh,SCICCR          ;1 stop bit, even parity, Asynchronous,
                                     ; Address bit protocol, 8-bit characters.
         MOV   #HI(B1200),BAUDMSB    ;Set for 1200 BAUD.
         MOV   #LO(B1200),BAUDLSB    ;
         MOV   #001h,RXCTL           ;Enable SCIRX INT.
         MOV   #022h,SCIPC2          ;Set SCIRXD as input.
         MOV   #060h,SCIPRI          ;SCIRX/SCITX interrupts low priority.
         MOV   #033h,SCICTL          ;Internal Clock, TXENA, RXENA.

         ....                        ;Main Code here.

XMITTER                              ;Transmitter routine
         JBIT0 BITNINE,BITLOW        ;Check to see if ninth bit=0
         MOV   #03Bh,SCICTL          ;Ninth bit is high, set TXWAKE flag.
BITLOW   MOV   DATAOUT,TXBUF         ;Load data to be transmitted.
         RTS                         ;End of subroutine. TXWAKE flag is
                                     ; cleared automatically.

RCVR                                 ;Receiver routine.
         SBIT1 HIGHBIT               ;Address bit is set, ninth bit=1
         BTJO  #002h,RXCTL,GETCHAR   ;address bit not set,
         SBIT0 HIGHBIT               ;HIGHBIT=0.
         JMP   GETCHAR
GETCHAR  MOV   RXBUF,DATAIN          ;Save other 8 bits of data.  RXWAKE is
                                     ; cleared automatically.
         RTS
```

6.4 HALT Mode Wakeup Using the SCI Receiver

In many applications, power consumption is a major concern. The TMS370 has two low-power modes, HALT and STANDBY, which stop execution of various modules in the device. This greatly reduces the power used by the part. For a complete description of the Powerdown/Idle modes see Section of the TMS370 Family Data Manual. While in a power-down mode the part ignores everything but a few select interrupts. The SCIRX interrupt is recognized while in the HALT mode and can be used to wake up the device upon receipt of a falling edge on SCIRXD. In this way the part can be put into a low-power mode and only be activated when another device wants to talk to it. The following code shows how to put a TMS370C050 into HALT mode to be awakened upon a SCIRX interrupt.

> **Note:** You must enable interrupts before executing the IDLE instruction or the part will not recover from the low-power mode (except on a system RESET).

```
B1200   .equ   129
        MOV    #00h,SCICTL              ;SCI SW Reset.
        MOV    #077h,SCICCR             ;1 stop bit, even parity, Asynchronous,
                                        ; Idle line protocol, 8-bit characters.
        MOV    #HI(B1200),BAUDMSB       ;Set for 1200 BAUD.
        MOV    #LO(B1200),BAUDLSB       ;
        MOV    #001h,RXCTL              ;Enable SCIRX INT.
        MOV    #002h,SCIPC2             ;Set SCIRXD as input.
        MOV    #060h,SCIPRI             ;SCIRX/SCITX interrupts low priority.
        MOV    #031h,SCICTL             ;Internal Clock, RXENA.
        OR     #045h,SCCR2              ;Configure for STANDBY mode.
        EINT                            ;Interrupts must be enabled to exit
                                        ; HALT mode
        IDLE                            ;Go into low-power mode.  Part will stay
                                        ; in standby mode until a valid standby
                                        ; interrupt is requested, including
                                        ; SCIRX.
```

Section 7

SCI Module Specific Applications

7.1 RS-232-C Interface

7.1.1 Interface TMS370C050 to RS-232-C Connection

The most common of the myriad of serial interfaces is the RS-232-C. Over time it has become an industry standard for digital communications, used for everything from PCs to telecommunication. This example will show the software and hardware necessary to connect a TMS370C050 to an RS-232-C interface. External hardware is needed because RS-232-C specifications call for non-TTL compatible voltage levels. This example uses the Maxim MAX232 RS-232 line driver/receiver to buffer the TTL levels to the -12 V to +12 V levels needed for RS-232 communications. The TMS370C050 will be used as the DCE (Data Communications Equipment) end of the communications link, that is, as a "slave" to another controller. For more information about the RS-232-C interface, consult a book on digital communications. Several are listed in the references section.

RS-232-C specifications are vague about the exact uses and protocols associated with the pins. This example shows a common format, using the CTS (Clear to Send) and DTR (Data Terminal Ready) lines for handshaking. The Transmitted Data and Received Data lines are used for the actual data transmission. In this example, as in most RS-232-C communications, the transmissons are asynchronous and need no synchronizing clocks. When the DTR line is pulled high, the controller is ready to receive data. Otherwise, the TMS370C050 stops data transmission until the controller pulls the line high again. The TMS370C050 can also halt data transmission from the controller by pulling the CTS line low. The SCICLK and Analog Input 7 pins are configured as general I/O pins for the CTS and DTR signals, respectively. The basic configuration for an RS-232-C connection is shown in Figure 7-1.

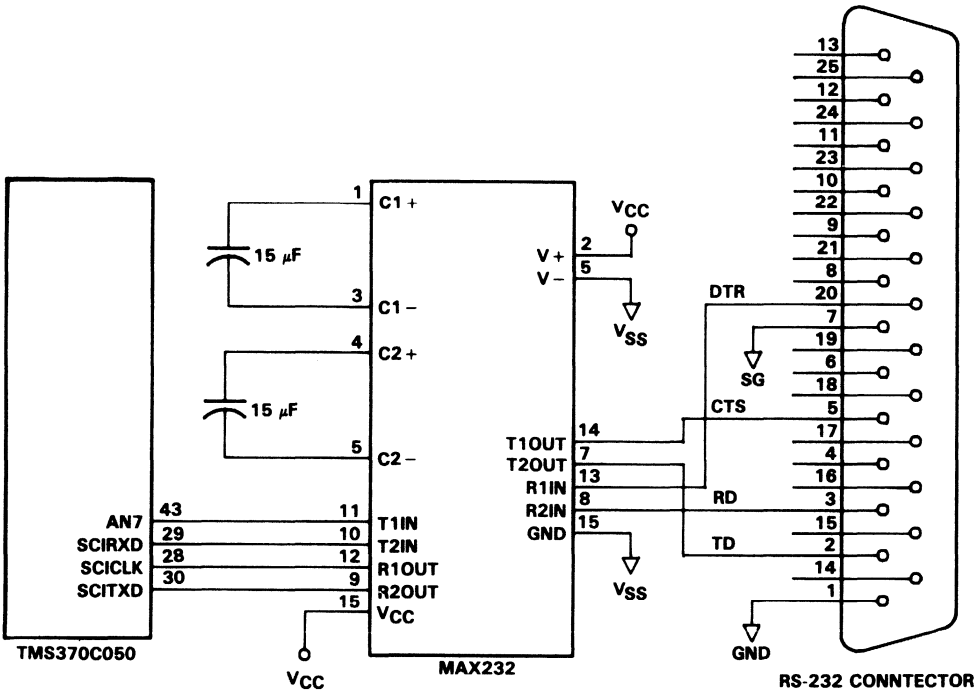


Figure 7-1. TMS370C050 - RS-232-C Interface Example

The framework of a program for controlling communications between the TMS370C050 and a DTE (Data Terminal Equipment) configured device is shown below.

```
          .title     "RS-232-C Interface"

;         This example shows the skeleton of a program for implementing an
;         RS-232-C interface in hardware and software.

;         Set up EQUATE table for Peripheral file registers used in the program.

SCICCR    .equ  P050                ;SCI Configuration Control Register
SCICTL    .equ  P051                ;SCI Operation Control Register
BAUDMSB   .equ  P052                ;Baud Rate Select MSB Register
BAUDLSB   .equ  P053                ;Baud Rate Select LSB Register
TXCTL     .equ  P054                ;Transmitter Int. Control/Status Register
RXCTL     .equ  P055                ;Receiver Int. Control/Status Register
RXBUF     .equ  P057                ;Receiver Data Buffer Register
TXBUF     .equ  P059                ;Transmit Data Buffer Register
SCIPC1    .equ  P05D                ;SCI Port Control Register 1
SCIPC2    .equ  P05E                ;SCI Port Control Register 2


;         Define registers & constants used in program

DATAIN    .equ  R2                  ;Temporary register for received data.
DATAOUT   .equ  R3                  ;Temporary register for transmitted data.
B9600     .equ  15                  ;Baud Rate Register value for 9600 BAUD.

          .text 07000h

START     DINT

;       SCI Initialization

            MOV   #000h,SCICTL      ;SCI SW RESET.
            MOV   #077h,SCICCR      ;1 stop bit, even parity, Asynchronous,
                                    ; Idle line protocol, 8-bit characters.
            MOV   #HI(B9600),BAUDMSB ;Set for 9600 BAUD (@ 19.6608MHz)
            MOV   #LO(B9600),BAUDLSB ;
            MOV   #002h,SCIPC1      ;Set SCICLK as Function Pin.
            MOV   #022h,SCIPC2      ;Set SCIRXD,SCITXD as input.
            MOV   #060h,SCIPRI      ;SCIRX interrupt low priority.
            MOV   #033h,SCICTL      ;Release SCI, Set Internal Clock,
                                    ; Sleep=0,RXENA,TXENA

            MOV    #200,B           ;Start stack pointer at R200.
            LDSP
            EINT                    ;Enable Interrupts


;         Main part of program manages and stores the data.  When the program is
;         ready to receive new data it calls subroutine RXCHAR.  When the
;         program is ready to transmit, it loads register DATAOUT and calls
;         subroutine TXCHAR.
```

```
MAIN
          ...

RECEIVE   CALL    RXCHAR              ;Get next character
          MOV     A,DATAIN
          ...

XMIT      MOV     DATAOUT,A
          CALL    TXCHAR              ;Transmit character.

          ...

          JMP     MAIN

;  SCI receiver subroutine.
;     The subroutine brings CTS high to signal that the TMS370 is ready to
;     receive data, then it waits until a character is received.  After a
;     character has been received, CTS is pulled low again to stop transmission
;     by the other device and the character is saved in register A.

RXCHAR    MOV     #005h,SCIPC1        ;Set CTS high. (TMS370 ready to receive)
RXWAIT    BTJZ    #040h,RXCTL,RXWAIT  ;Loop until character received.
          MOV     #001h,SCIPC1        ;Set CTS low to stop transmission.
          MOV     RXBUF,A             ;Save received character.
          RTS

;  SCI transmitter subroutine.
;     The subroutine waits for the other device to bring the DTR line high
;     before transmitting.  The character is then sent and the TXCTL register is
;     polled to make sure the character has been transmitted before continuing.

TXCHAR    BTJZ    #080h,ADIN,TXCHAR   ;Wait for DTR to go high.
TXWAIT    BTJZ    #080h,TXCTL,TXWAIT  ;Wait until previous characters are
                                      ; transmitted out.
          MOV     A,TXBUF             ;Send cut the character.
          RTS


;         Set up interrupt vector addresses.

          .sect   "VECTORS",07FF2h
          .word   START               ;No interrupts are used:
          .word   START               ; All vectors will jump to 'START'.
          .word   START
          .word   START
          .word   START
          .word   START
```

7.2 Dumb-Terminal Driver

7.2.1 Use TMS370C050 SCI to Interface to Dumb-Terminal

The power of the TMS370C050 microcontroller allows it to control a large number of tasks at the same time. The on-chip peripherals can operate independently of each other, releasing the CPU to do other tasks. This example shows a TMS370C050 microcontroller configured as a dumb-terminal driver. ASCII data are received from a terminal and stored in a buffer. Data to be transmitted are stored in another buffer and shifted out of the SCI when the terminal is ready to receive. An example of how the TMS370C050 and the terminal are connected is shown in Figure 7-2.

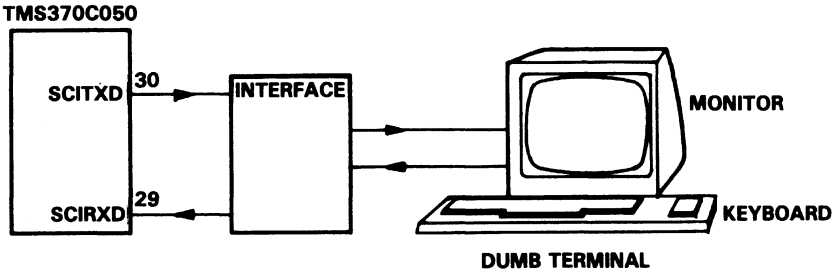


Figure 7-2. Terminal Interface Example

This example uses the X-On/X-Off method of handshaking. Only the data transmit and receive lines are needed because the handshaking is done in software. When either the terminal or TMS370 receive buffers fill up, the respective device forces an X-Off (013h) onto the transmit line to stop the other device from transmitting. When the buffer on either device empties sufficiently the respective device transmits an X-On character (011h) and the other device begins transmitting again. This simple and effective handshaking technique eliminates the need for additional signals and/or hardware to control the transmission. Because the receive and transmit routines are independent and interrupt driven they can be combined with other routines to expand the uses beyond that of a simple terminal controller.

The example shown below is the framework for a terminal controller showing the code necessary for receiving from and transmitting to the terminal. When the program receives a character it automatically branches to RXINT, the SCI receiver interrupt routine, where character is stored in the receiver buffer. If the 32 character receiver buffer contains more than 27 characters, the receiver immediately sends an X-Off signal to the terminal to stop the flow of data to the controller. The 27 character limit is set because the terminal will not recognize the X-Off immediately and may send a few more characters. When the controller is ready to process the received data it pulls the character from the receiver buffer. If the buffer contains less than 4 characters and an X-Off had been previously sent then an X-On signal is sent to the terminal to start data transmission to the controller again.

After the data is manipulated by the controller (special characters added, brightness or cursor position changed), subroutine TXCHAR is called. This subroutine loads the data into the transmitter buffer and enables the TX interrupt. The program jumps to the interrupt routine where the character is transmitted out. If the terminal has sent an X-Off, the routine waits until an X-On is received to transmit.

```
        .title  "SCI Terminal Driver"

;       Set up equate table for peripheral registers used in program.

SCCR0   .equ  P010              ;System configuration register assignments.
SCCR1   .equ  P011
SCCR2   .equ  P012

SCICCR  .equ  P050              ;SCI Configuration Control Register
SCICTL  .equ  P051              ;SCI Operation Control Register
BAUDMSB .equ  P052              ;Baud Rate Select MSB Register
BAUDLSB .equ  P053              ;Baud Rate Select LSB Register
TXCTL   .equ  P054              ;Transmitter Int. Control/Status Register
RXCTL   .equ  P055              ;Receiver Int. Control/Status Register
RXBUF   .equ  P057              ;Receiver Data Buffer Register
TXBUF   .equ  P059              ;Transmit Data Buffer Register
SCIPC1  .equ  P05D              ;SCI Port Control Register 1
SCIPC2  .equ  P05E              ;SCI Port Control Register 2
SCIPRI  .equ  P05F              ;SCI Priority Control Register

;       Allocate register space for registers used in program.  Also mark
;       beginning of spaces to be used by 32 byte data transfer buffers.

COMSTAT .equ  R2                ;Communications Status Register
LOCSTAT .dbit 0,COMSTAT         ;X-Status from local TMS370    (1=Xoff)
REMSTAT .dbit 1,COMSTAT         ;X-Status from remote terminal (1=Xoff)
RXPTR   .equ  R3                ;Location of last received data in  BUFFER.
RXPTRI  .equ  R4                ;Interrupt routine data pointer.
RXDIFF  .equ  R5                ;Number of characters in RXBUFFER.
TXPTR   .equ  R6                ;Next location to be transmitted in BUFFER.
TXPTRI  .equ  R7                ;Interrupt routine data pointer.
TXDIFF  .equ  R8                ;Number of characters in TXBUFFER.
RXBUFFER .equ  R9               ;Beginning of 32 byte receiver data buffer.
TXBUFFER .equ  R41              ;Beginning of 32 byte transmit data buffer.

;       Define constants used in program.

TXLIMIT .equ  27                ;Maximum # of characters in buffers  before
RXLIMIT .equ  27                ; XOFF or XON is sent.
RXLIMIT2 .equ  4
XON     .equ  011h              ;Control-Q character.
XOFF    .equ  013h              ;Control-S character.


        .text 07000h

START   DINT

;       Initalize SCI.

        MOV   #077h,SCICCR      ;1 stop bit, even parity, Asynchronous,
                                ; Idle line protocol, 8-bit characters.
        MOV   #000h,SCICTL      ;SCI SW Reset.
        MOV   #000h,BAUDMSB     ;Set for 9600 BAUD (@ 20MHz).
        MOV   #00Fh,BAUDLSB     ;
        MOV   #001h,RXCTL       ;Enable SCIRX INT.
        MOV   #001h,TXCTL       ;Enable SCITX INT.
        MOV   #002h,SCIPC1      ;Set SCICLK as function pin.
        MOV   #022h,SCIPC2      ;Set SCIRXD,SCITXD as input.
        MOV   #050h,SCIPRI      ;SCIRX INT - High priority.
                                ;SCITX INT - Low priority.
        MOV   #033h,SCICTL      ;Release SCI, Internal Clock,
                                ; Sleep=0,RXENA,TXENA.
```

```
;       Clear data registers.

        CLR   COMSTAT              ;Set status flags to XON.
        CLR   RXPTR                ;Clear data pointer registers.
        CLR   RXPTRI
        CLR   RXDIFF
        CLR   TXPTR
        CLR   TXPTRI
        CLR   TXDIFF

        MOV   #200,B               ;Set stack pointer below BUFFER table.
        LDSP
        EINT                       ;Global interrupt enable.
                                   ;

;       Place main block of code here.  When a character is received the SCI
;       Receiver Interrupt routine is called and the character is stored in
;       the data buffer.  When the program is ready to process a character
;       that has been received the subroutine RXCHAR is called.  When a
;       character is ready to be transmitted, the routine TXCHAR is called
;       and the character is transmitted.

MAIN
;       ...

        CMP   #00H,RXDIFF          ;Characters waiting to be processed?
        JEQ   NORCVR               ; If not, continue on.
        CALL  RXCHAR               ;Pull character from RXBUFFER.
        MOV   A,DATA
NORCVR  NOP

;       ...                        ;Massage data for terminal.
                                   ;(i.e. Formatting, Uppercase, etc)

        MOV   DATA,A
        CALL  TXCHAR               ;Place character in TXBUFFER to be
                                   ; transmitted.


        JMP   MAIN


;       SCI Receiver Subroutine.
;       This routine is called whenever the program is ready to process a
;       character in the receiver buffer.

RXCHAR
        BTJO   #0FFh,RXDIFF,CHKXON  ;Any characters in buffer?
        JMP    RXCHAR               ; if not, wait.

CHKXON  DEC    RXDIFF               ;One less character in RXBUFFER.
        JBIT0  LOCSTAT,GRABCHAR     ;XON already sent?  Don't send another.
        CMP    #RXLIMIT2,RXDIFF     ;Receiver buffer emptying?
        JGE    GRABCHAR             ; No, do not send XON.
WAIT1   BTJZ   #080h,TXCTL,WAIT1    ;Wait until present transmission complete.
        MOV    #XON,TXBUF           ;Put XON in transmitter buffer.
        SBIT0  LOCSTAT              ;I have sent an XON.

GRABCHAR PUSH  B
        MOV    RXPTR,B              ;Increment pointer.
        INC    B
        BTJZ   #020h,B,NOROLL1      ;Does RXPTR need to be rolled over?
        MOV    #0,B                 ; Yes, reset RXPTR to start of RXBUFFER.
NOROLL1 MOV    B,RXPTR              ;Save new value of RXPTR.
        MOV    RXBUFFER(B),A        ;Get new value from RXBUFFER.
        POP    B

        RTS
```

```
;       SCI Transmitter Subroutine.
;       This routine is called whenever the program is ready to transmit a
;       character to the terminal.

TXCHAR
        CMP     #TXLIMIT,TXDIFF         ;Wait until there is room in buffer.
        JGE     TXCHAR
        PUSH    B
        MOV     TXPTR,B
        INC     B                       ;Next character to be transmitted.
        BTJZ    #020h,B,NOROLL2         ;Does TXPTR need to be rolled over?
        MOV     #0,B                    ;Reset TXPTR to beginning of TXBUFFER.
NOROLL2 MOV     B,TXPTR                 ;Save new value of TXPTR.
        INC     TXDIFF                  ;Inc. # of characters to be transmitted.
        MOV     A,TXBUFFER(B)           ;Save character in transmitter buffer.
        POP     B                       ;Restore value of B.
        OR      #001h,TXCTL             ;Enable TX interrupt.
        RTS                             ;Exit.

;       SCI Transmitter Interrupt Routine.
;       This routine is called whenever the program is ready to transmit a
;       character to the terminal.

TXINT
        JBIT1   REMSTAT,TXEXIT          ;If terminal has sent XOFF, do not
                                        ; transmit.
        PUSH    A
        PUSH    B
        INC     TXPTRI                  ;Next BUFFER location.
        BTJZ    #020h,TXPTRI,NOROLL3    ;If TXPTRI past end of buffer, clear it.
        CLR     TXPTRI                  ;Set TXPTRI to beginning of buffer.
NOROLL3 DEC     TXDIFF                  ;If so, nothing to transmit.
        MOV     TXPTRI,B                ;
        MOV     TXBUFFER(B),A
TXWAIT1 BTJZ    #080h,TXCTL,TXWAIT1     ;Wait until previous characters have
                                        ; finished transmitting.
        MOV     A,TXBUF                 ;Transmit character.
        POP     B                       ;Increment TXPTR.
        POP     A                       ;
        BTJO    #0FFh,TXDIFF,TXEXIT     ;If no more characters to send,
        AND     #0FEh,TXCTL             ; disable interrupts.
TXEXIT  RTI


;  SCI Receiver Interrupt Service Routine
;
;     This interrupt routine receives characters and checks for XON and
;     XOFF characters sent by the terminal.  The received characters are
;     stored in RXBUFFER for the subroutine RXCHAR ro manipulate them.

RXINT
        PUSH    A                       ;Save A register contents.
        MOV     RXBUF,A                 ;Grab received character from buffer.
        CMP     #XON,A                  ;Was an XON received?
        JNE     TRYXOFF
        SBIT0   REMSTAT                 ;Set flag: XON received.
        JMP     RXDONE
TRYXOFF CMP     #XOFF,A                 ;Was an XOFF received?
        JNE     SAVECHAR
        SBIT1   REMSTAT                 ;Set flag: XOFF received.
        JMP     RXDONE

SAVECHAR
        PUSH    B                       ;Save B register contents.
        MOV     RXPTRI,B                ;Point to location to store new character.
        INC     B
        BTJZ    #020h,B,NOROLL4         ;Does RXPTRI need to be rolled over?
        MOV     #0,B                    ;Reset RXPTRI to beginning of BUFFER.
NOROLL4 MOV     B,RXPTRI                ;Save new value of RXPTRI.
        MOV     A,RXBUFFER(B)
        INC     RXDIFF                  ;# of stored characters + 1.
```

```
        POP     B                       ;Restore B register contents

        JBIT1   LOCSTAT,RXDONE          ;XOFF already sent?  Don't send another.
        CMP     #RXLIMIT,RXDIFF         ;Receiver buffer getting full?
        JL      RXDONE                  ; No, exit interrupt routine.
RXWAIT  BTJZ    #080h,TXCTL,RXWAIT      ;Wait until present transmission complete.
        MOV     #XOFF,TXBUF             ;Put XOFF in transmitter buffer.
        SBIT1   LOCSTAT                 ;I have sent an XOFF.

RXDONE  POP     A                       ;Restore A register contents.
        RTI                             ;End of Receiver interrupt routine.

;       Setup interrupt vectors adresses.

        .sect   "VECTORS",07FF0h
        .word   TXINT                   ;SCITX interrupt routine.
        .word   RXINT                   ;SCIRX interrupt routine.
        .word   START                   ;All other vectors will jump to 'START'.
        .word   START
        .word   START
        .word   START
        .word   START
        .word   START
```

There are a few things that should be noted about any terminal controller code. The most important is to watch the timing of the transmission of X-Off and X-On characters from the receiver routines. It is important that as soon as the receiver buffer passes its limit (in this case 27 characters) that an X-Off be transmitted to make sure that the buffer does not overflow. A problem arises in that the routine to transmit the X-Off character should be placed inside the RXINT routine so that it can be called immediately. Unfortunately, you have to wait to make sure that the current transmission is finished before starting the X-Off transmission. With all this waiting and transmitting inside the RXINT routine, it is possible at high SCI speeds that the routine will not be able to finish the current receiver interrupt and get the next character out of RXBUF before it is overwritten.

There is no simple way around this problem. One suggestion is to find the maximum time it takes for the interrupt routine with the X-Off transmission and tailor your SCI speed accordingly. If the Receiver buffer size is greatly increased, it may be possible to wait for the next transmitter interrupt to send the X-Off. You may also want to poll the receiver overrun flag and transmit a special NAK (Negative AcKnowledge) character to the terminal to have it retransmit the data. The exact solution for your particular case depends on your application.

7.3 Low-Power Remote Data Acquisition

7.3.1 Use TMS370C050 in STANDBY Mode with SCIRX Wake-Up Procedure

The low-power modes and flexible serial interface of the TMS370 family make it ideal for applications involving remote sensing. In this application example, a TMS370C050 is acting as a climate recorder in a remote location. Data from measuring instruments are collected via the on-board A/D, and stored until requested by the host controller. Power consumption is a major concern because the system is designed to be battery-operated and serviced infrequently. A basic configuration is shown below in Figure 7-3. The TMS370C050 is connected through the A/D port to a variety of analog sensing devices. The transmit and receive lines are buffered through external logic to whatever levels are necessary to communicate with the host controller. The communications link may be simple as a direct wire connection or as complicated as a modem interface.

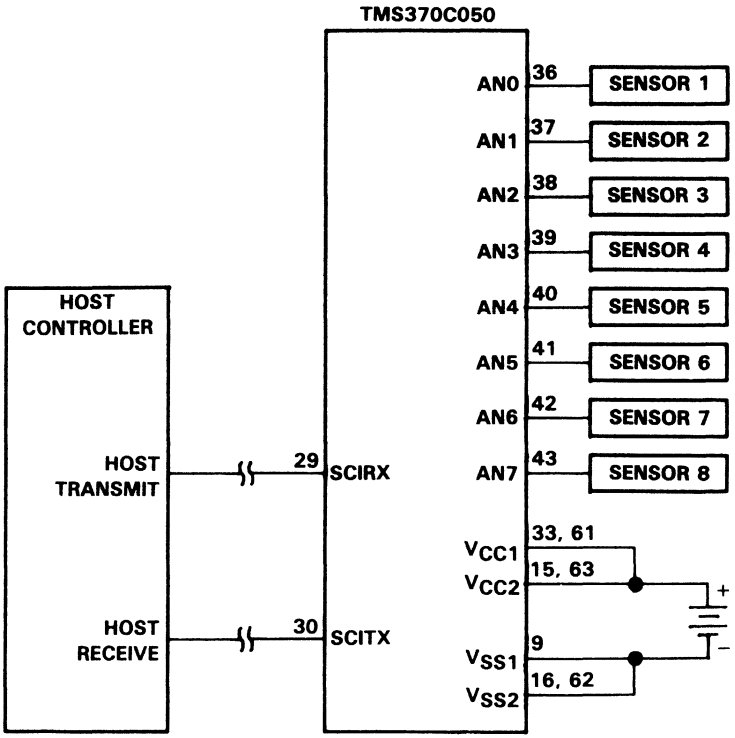


Figure 7-3. Remote Data Acquisition Example

The program uses Timer 1 to periodically read the A/D values and store them in ATABLE. Timer 1 can also bring the device out of STANDBY mode through the Timer 1 interrupt. In this way the device will draw less than one-quarter its normal operating current most of the time. The A/D conversion routine is not shown here, but examples can be found in the TMS370 Family Data Manual and related application notes. In particular the A/D routine is similar to the one shown in the Design Aids section of the TMS370 Family Data Manual. The data can be stored in RAM, or if power loss is a consideration, EEPROM memory may be used.

Because of the minimum speed of the part and the size of the timer registers, the longest timer period we can have is 33.6 seconds. For this example the time between updates is 10 minutes. To allow for the extra time a counter is included in the timer interrupt routine. If a full 10 minutes have not passed, the part goes back into STANDBY mode to wait for the next interrupt. The equation used to calculate the timer and counter values is:

$$\text{Time between updates} = \frac{4 \times \text{PRESCALE}}{\text{CLKIN}} \times \text{Timer 1 value} \times \text{Interval counter}$$

For this example:

$$10 \text{ min} = 600 \text{ sec} = \frac{4 \times 256}{2 \text{ Mhz}} \times 65104 \times 18$$

The device will periodically update ATABLE, where the data is stored. Upon receipt of information from the host (SCIRXD goes low), the remote '050 will come out of STANDBY mode. If the received data does not match the internal address, the part goes back into STANDBY mode. If the address matches, the remote will first send one byte of information with the number of bytes of data to be sent, followed by the data itself. After the device sends all the data, it will put itself back into STANDBY mode to wait for another inquiry or data acquisition.

The source code for this application is as follows:

```
        .title   "Remote Data Aquisition program"

;       This routine uses Timer 1 and SCI receiver interrupts to bring a
;       TMS0370C050 out of STANDBY mode.  The Timer 1 interrupt is used to
;       collect data from the A/D converter.

;       Set up EQUATE table for Peripheral File registers used in the program.

SCCR2   .equ    P012                ;System configuration register assignments.

SCICCR  .equ    P050                ;SCI Configuration Control Register
SCICTL  .equ    P051                ;SCI Operation Control Register
BAUDMSB .equ    P052                ;Baud Rate Select MSB Register
BAUDLSB .equ    P053                ;Baud Rate Select LSB Register
TXCTL   .equ    P054                ;Transmitter Int. Control/Status Register
RXCTL   .equ    P055                ;Receiver Int. Control/Status Register
RXBUF   .equ    P057                ;Receiver Data Buffer Register
TXBUF   .equ    P059                ;Transmit Data Buffer Register
SCIPC1  .equ    P05D                ;SCI Port Control Register 1
SCIPC2  .equ    P05E                ;SCI Port Control Register 2
SCIPRI  .equ    P05F                ;SCI Priority Control Register
```

```
T1CNTRMSB .equ  P040              ;Timer 1 register assignments.
T1CMSBLSB .equ  P041
T1CMSB    .equ  P042
T1CLSB    .equ  P043
T1CCMSB   .equ  P044
T1CCLSB   .equ  P045
T1CTL1    .equ  P049
T1CTL2    .equ  P04A
T1CTL3    .equ  P04B
T1CTL4    .equ  P04C
T1PC1     .equ  P04D
T1PC2     .equ  P04E
T1PRI     .equ  P04E

;         Allocate register space for variables and data table used in the
;         routine.

ADDRESS   .equ  R2                ;Temp register for Received value.
ICOUNT    .equ  R3                ;Counter for number of Timer 1 interrupts
                                  ; before data is sampled for table.
ATABLE    .equ  R4                ;Table where A/D data is stored before
                                  ; being transmitted.

;         Define constants used in program.

TIMEMSB   .equ  0FEh              ;Interrupt timing.
TIMELSB   .equ  050h
INTERVAL  .equ  18                ;Number of timer interrupts before data is
                                  ; stored.
MYADDRESS .equ   0FFh             ;Personal address of this device.

          .text  07000h

START     DINT                    ;Disable interrupts while initializing.

;       System Initialization

          MOV    #041h,SCCR2      ;STANDBY mode, no Priv mode, no osc fault
                                  ; reset.
;       SCI Initialization

          MOV    #000h,SCICTL     ;SCI SW Reset.
          MOV    #077h,SCICCR     ;1 stop bit, even parity, Asynchronous,
                                  ; Idle line protocol, 8-bit characters.
          MOV    #000h,BAUDMSB    ;Set for [9600 BAUD @20 MHz.
          MOV    #00Fh,BAUDLSB    ;
          MOV    #001h,RXCTL      ;Enable SCIRX INT.
          MOV    #022h,SCIPC2     ;Set SCIRXD, SCITXD function.
          MOV    #070h,SCIPRI     ;SCIRX interrupt low priority.
          MOV    #033h,SCICTL     ;Release SCI SW Reset.
                                  ; Internal Clock, TXENA, RXENA.

;       Timer 1 Initialization

          MOV    #TIMEMSB,T1CMSB  ;Set timer values
          MOV    #TIMELSB,T1CLSB
          MOV    #040h,T1PRI      ;Set T1 interrupts to low priority.
          MOV    #010h,T1CTL4     ;Dual-compare,Disable interrupts.
          MOV    #007h,T1CTL1     ;System clock / 256
          MOV    #001h,T1CTL3     ;Disable T1 interrupts, clear flags.
          MOV    #001h,T1CTL2     ;Disable Overflow interrupts,Reset T1.

          MOV    #INTERVAL,ICOUNT ;Initialize counter
          MOV    #200,B           ;Initalize the stack pointer to start at
          LDSP                    ; register 200. (away from ATABLE)
          MOV    #000h,B          ;Reset ATABLE pointer.
          EINT                    ;Interrupts must be enabled to exit
                                  ; STANDBY mode.
```

```
;       Main part of program actually does nothing but wait for interrupts.
;       The Timer 1 and SCIRX interrupt service routines actually do the work.

MAIN    IDLE                        ;Go into low-power mode
        JMP     MAIN                ;Main Loop


;       Timer1 Interrupt Routine
;       When the interrupt routine is called the part will come out of STANDBY
;       mode.  The routine will collect information from the A/D and store it
;       it in register A.  The data is then loaded into ATABLE so it can be
;       easily transmitted out.  The number of bytes of stored data is in B.
;       At the end of the routine the part will return to the main program
;       where it will go into STANDBY mode again.

TIMERINT AND    #00Fh,T1CTL3        ;Clear interrupt flags.
        DJNZ    ICOUNT,DONE         ;Time to get new A/D value? If not, skip.

        ....                        ;A/D data gathering & formatting. Value is
                                    ; stored in register A.
        INC     B                   ;Increment data counter/pointer.
        MOV     A,ATABLE-1(B)       ;Store data in ATABLE.

        MOV     #INTERVAL,ICOUNT    ;Restore counter.
DONE    RTI                         ;End of service routine

;       SCI Receiver Interrupt Routine
;       This routine is called when the part receives a low pulse on the
;       SCIRX pin.  The received datum is compared against an internal address
;       to see if the device was addressed.  If so, the routine transmits one
;       character indicating the number of bytes to be transmitted.  The
;       routine then transmits all the data stored in ATABLE, LIFO.

RXINT   MOV     RXBUF,ADDRESS       ;Read Received address.
        BTJO    #080h,RXCTL,RXDONE  ;If there was an error, wait for another
                                    ; transmission.
        CMP     #MYADDRESS,ADDRESS  ;If address not mine, ignore wake-up
                                    ; call.
        JNE     RXDONE              ;
        MOV     B,TXBUF             ;# of Characters to be transmitted.
        CMP     #00,B               ;If no data stored yet, ignore.
        JEQ     WAIT
LOOP    BTJZ    #080h,TXCTL,LOOP    ;Wait until character sent.
        MOV     ATABLE(B)-1,A       ;Transmit character.
        MOV     A,TXBUF             ;
        DJNZ    B,LOOP              ;If not done, send next character.
WAIT    BTJZ    #040h,TXCTL,WAIT    ;Wait for last character to be sent.
RXDONE  RTI                         ;Exit interrupt routine and go back into
                                    ; STANDBY mode.

;       Set up interrupt vectors.

        .sect   "VECTORS",07FF2h
        .word   RXINT               ;SCIRX interrupt routine.
        .word   TIMERINT            ;Timer 1 interrupt routine.
        .word   START               ;All other vectors will jump to 'START'.
        .word   START
        .word   START
        .word   START
        .word   START
```

Appendix A

SPI Control Registers

The SPI is controlled and accessed through registers in the Peripheral File. These registers are listed in Figure A-1 and described in the TMS370 Family Data Manual. The bits shown in shaded boxes in Figure A-1 are Privilege Mode bits, that is, they can only be written to in the Privilege Mode.

PERIPHERAL FILE FRAME 3: SERIAL PERIPHERAL INTERFACE (SPI) CONTROL REGISTERS

ADDR	PF	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	
1030h	30	SPI SW RESET	CLOCK POLARITY	SPI BIT RATE2	SPI BIT RATE1	SPI BIT RATE0	SPI CHAR2	SPI CHAR1	SPI CHAR0	SPICCR
1031h	31	RECEIVER OVERRUN	SPI INT FLAG	---	---	---	MASTER/ SLAVE	TALK	SPI INT ENA	SPICTL
1032h TO 1036h	32 TO 36	RESERVED								
1037h	37	SPI DATA BUFFER REGISTER								SPIBUF
1038h	38	RESERVED								
1039h	39	SPI SERIAL DATA REGISTER								SPIDAT
103Ah TO 103Ch	3A TO 3C	RESERVED								
103Dh	3D	---	---	---	---	SPICLK DATA IN	SPICLK DATA OUT	SPICLK FUNCTION	SPICLK DATA DIR	SPIPC1
103Eh	3E	SPISIMO DATA IN	SPISIMO DATA OUT	SPISIMO FUNCTION	SPISIMO DATA DIR	SPISOMI DATA IN	SPISOMI DATA OUT	SPISOMI FUNCTION	SPISOMI DATA DIR	SPIPC2
103Fh	3F	SPI STEST	SPI PRIORITY	SPI ESPEN	---	---	---	---	---	SPIPRI

Figure A-1. SPI Control Registers

Appendix A

Appendix B

SCI Control Registers

The SCI is controlled and accessed through registers in the Peripheral File. These registers are listed in Figure B-1 and described in the TMS370 Family Data Manual. The bits shown in shaded boxes in Figure B-1 are Privilege Mode bits, that is, they can only be written to in the Privilege Mode.

PERIPHERAL FILE FRAME 5: SERIAL COMMUNICATION INTERFACE (SCI) CONTROL REGISTERS

ADDR	PF	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	
1050h	050	STOP BITS	EVEN/ODD PARITY	PARITY ENABLE	ASYNC/ ISOSYNC	ADDRESS/ IDLE WUP	SCI CHAR2	SCI CHAR1	SCI CHAR0	SCICCR
1051h	051	---	---	SCI SW RESET	CLOCK	TXWAKE	SLEEP	TXENA	RXENA	SCICTL
1052h	052	BIT 15	BAUD RATE SELECT REGISTER MSB						BIT 8	BAUD MSB
1053h	053	BIT 7	BAUD RATE SELECT REGISTER LSB						BIT 0	BAUD LSB
1054h	054	TXRDY	TX EMPTY	---	---	---	---	---	SCI TX INT ENA	TXCTL
1055h	055	RX ERROR	RXRDY	BRKDT	FE	OE	PE	RX WAKE	SCI RX INT ENA	RXCTL
1056h	056	RESERVED								
1057h	057	RECEIVE DATA BUFFER REGISTER								RXBUF
1058h	058	RESERVED								
1059h	059	TRANSMIT DATA BUFFER REGISTER								TXBUF
105Ah 105Bh 105Ch	05A 05B 05C	RESERVED								
105Dh	05D	---	---	---	---	SCICLK DATA IN	SCICLK DATA OUT	SCICLK FUNCTION	SCICLK DATA DIR	SCIPC1
105Eh	05E	SCI TXD DATA IN	SCI TXD DATA OUT	SCI TXD FUNCTION	SCI TXD DATA DIR	SCI RXD DATA IN	SCI RXD DATA OUT	SCI RXD FUNCTION	SCI RXD DATA DIR	SCIPC2
105Fh	05F	SCI ETEST	SCI TX PRIORITY	SCI RX PRIORITY	SCI ESPEN	---	---	---	---	SCIPRI

Figure B-1. SCI Control Registers

Appendix B

Appendix C

TMS0170 Specifications

The TMS0170 Vacuum Fluorescent (VF) Display Driver is a one-chip interface between low voltage digital logic (5.0 V) and low voltage (< 18 V) VF displays.

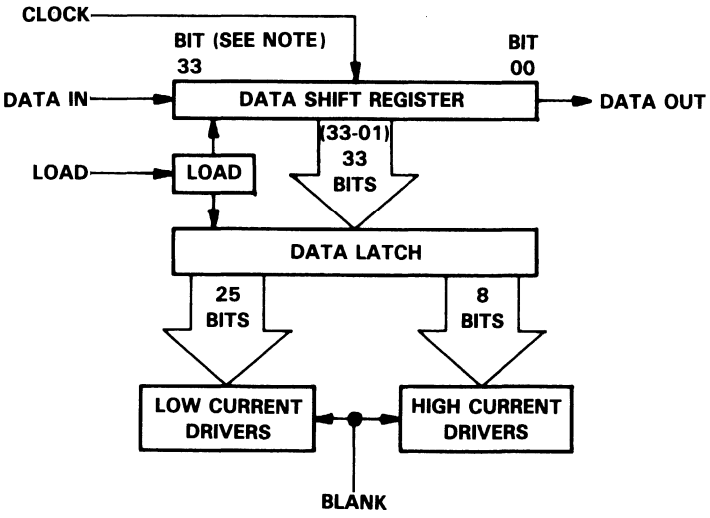
Key Features

- 33 individually controllable VF drivers. 8 high current drivers and 25 low current drivers.
- Blanking input allows duty cycling of outputs for brightness control.
- Serial interface minimizes connections between the TMS0170 and the digital system.
- Multiple TMS0170's can be cascaded using the Data Out latch.
- Self-load feature allows elimination of Load Enable line.
- Single supply, from 8 V to 18 V.
- Fabricated with high voltage PMOS technology.
- 40 pin DIP and 44 pin PLCC plastic packages available.

C.1 Functional Description

C.1.1 Architecture

The TMS0170, shown in Figure C-1 as a block diagram, consists of a 34-bit data shift register, a 33-bit data latch, and 33 VF drivers. A bit pattern is shifted into the TMS0170 using the clock input, then transferred to the data latch using the load enable input. The Blanking input can be used to turn off all of the drivers at any time. By duty cycling the Blanking input, brightness of the display can be varied.



* Note: Bit 33 is the last bit shifted into DATA IN pin.

Figure C-1. TMS0170 Block Diagram

C.1.2 Shift Register

The 34-bit shift register consists of 34 D-type flip-flops. The bits are numbered from 33 down to 00. Each data bit is clocked in on the rising edge of the Clock In pin, and enters the shift register in flip-flop #33. Upon each successive Clock In rising edge, the bit is shifted sequentially through the shift register, from flip-flop #33 to flip-flop #00. The data in the first 33 flip flops (from #33 down to #01) is transferred into the data latch on the rising edge of Load Enable. Flip flop #00 is not connected to the data latch, but instead, is connected to the Data Out output pin. This output can be used for cascading several TMS0170's together or for self loading. All of the flip flops in the shift register are cleared by the rising edge of Load Enable.

C.2 Interface

The interface between the TMS0170 and the digital logic consists of 4 lines; a Clock In line, a Data In line, and a Load Enable line, and a Blanking input.

- **Data Input:** Determines what data value is loaded into the data shift register. This data can then be latched to the output drivers upon a valid Load Enable input. A latched high level will turn the output driver on. A latched low level will turn the output driver off.

- **Clock Input:** The rising edge of the Clock Input will latch the current value of the Data Input into the data shift register and cause the shift register to shift by one.

- **Load Enable:** The rising edge of the Load Enable input transfers the data from the data shift register into the data latches and sets the data shift register to zero.

- **Blanking:** This input is used to disable all the drivers. A low level on this pin will force all driver outputs to a low level. A high level will enable the drivers to output whatever data has been loaded into their respective latches. This pin has an internal pull-up resistor.

C.3 Pin Assignment Diagram

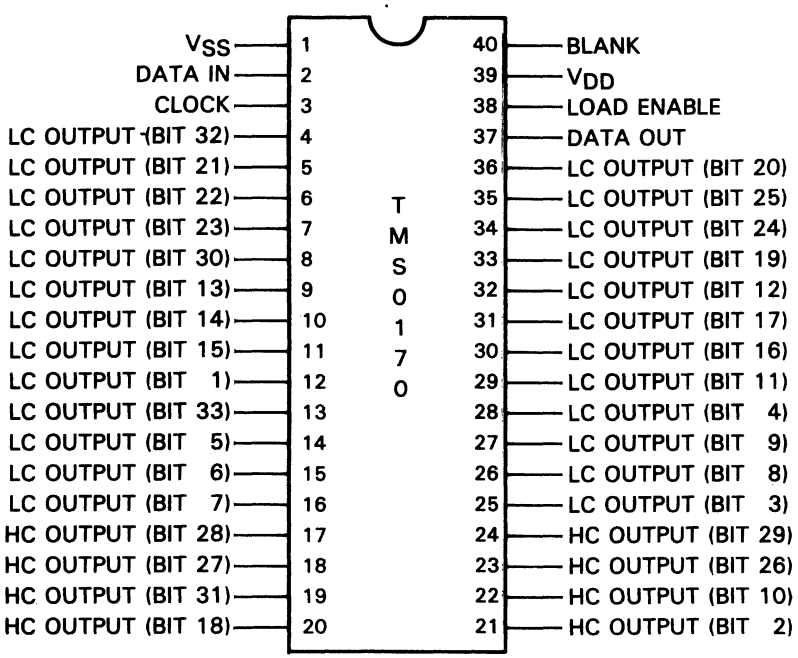


Figure C-2. TMS0170 DIP Pin Out

C.4 Electrical Specifications

C.4.1 Recommended Operating Conditions

Parameter		Min	Max	Units
V_{SS}	Supply Voltage	8	18	V
V_{IH} V_{IL}	High Level Input Voltage Low Level Input Voltage	V_{DD}+ 3.5 V_{DD}- 0.3	V_{SS}+ 0.3 V_{DD}+ 0.8	V V
T_a	Operating Free-Air Temperature	-40	85	°C

C.4.2 Electrical Characteristics over Operating Free Air Temperature Range

Parameter		Min	Max	Units
I_{SS}	Supply Current (all outputs open) V_{SS} = 8 V to 18 V		17	mA
V_{OH}	High Level Output Voltage (low current drivers) V_{SS} = 9.5 V I_{OH} = 1.5 mA	V_{SS}- 0.3		V
V_{OH}	High Level Output Voltage (high current drivers) V_{SS} = 9.5 V I_{OH} = 30.0 mA	V_{SS}- 2.5		V
V_{OH}	High Level Output Voltage (DATA OUT output) V_{SS} = 9.5 V I_{OH} = 500 µA	V_{SS}- 5.0		V
V_{OL}	Low Level Output Voltage (all drivers) V_{SS} = 9.5 V I_{OL} = 1 µA V_{SS} = 9.5 V I_{OL} = 500 µA		 V_{DD}+ 0.4 V_{DD}+ 5.0	 V V
V_{OL}	Low Level Output Voltage (DATA OUT output) V_{SS} = 9.5 V I_{OL} = 1 µA		V_{DD}+ 0.4	V
I_{IH}	High Level Input Current (CLOCK, DATA, LOAD) V_{IH} = V_{SS}		1	µA
I_{IL}	Low Level Input Current (CLOCK, DATA, LOAD) V_{IL} = V_{DD}		1	µA
I_{IH}	High Level Input Current (BLANK) V_{IH} = 3.5 V	-5	-125	µA
I_{IL}	Low Level Input Current (BLANK) V_{IL} = V_{DD}	-5	-125	µA

Appendix D

Glossary

Address Bit mode: An SCI mode of communication incorporating an extra bit into each frame to distinguish address frames from data frames. Setting the Address bit to a logic 1 signifies a frame beginning a new block.

Asynchronous mode: A communication format in which no synchronizing clocks are used. The data being transmitted is repeated several times and a majority vote is taken of selected bits to determine the transmitted value. This format is commonly used in RS-232-C and systems communications.

Block: A collection of one or more frames, the first of which is an address frame.

Baud rate: The communication rate for digital transfers, measured in line changes per sec. For serial communications, this equals 1 bit per sec.

Character: A group of bits, from 1 to 8 bits in length, that makes up one unit of data.

DCE: Data Communications Equipment. The hardware responsible for controlling digital communications.

DTE: Data Terminal Equipment. Equipment which receives or originates data transfer in a communications network.

Double-Buffered: Using a temporary storage register to hold data between register reads or writes. In the SCI, the temporary registers are TXBUF and RXBUF. They are used to hold data while transmitting or receiving and TXSHF or RXSHF are being used, speeding up data transfer and reducing the possibility of transmitter or receiver overruns.

Frame: The basic packet of serial communication. It typically contains one start bit, 1-8 bits of data, and one or two stop bits. It may also contain a parity bit and an address designator bit depending on the protocol.

Full-Duplex: A mode of communication in which transmission and reception of signals happens simultaneously.

Idle Line Mode: A serial communications protocol in which the beginning of a new block (an address frame) is identified as being the first frame after an idle period.

Idle period: A period of ten bits or longer in which no data is received.

Isosynchronous mode: A communication format in which synchonizing clocks are used. This is typically faster than Asynchronous communications because one bit of data is transmitted on each shift-clock cycle.

LSb: Least significant bit.

LSB: Least significant Byte.

Master: In its most general meaning, a mode of operation in which a microcontroller controls another microcontroller or peripheral and issues timing signals to it. It also refers to a specific mode of operation of the SPI.

MSb: Most significant bit.

MSB: Most significant Byte.

NRZ (Non-Return-To-Zero) Format: A communication format in which the inactive state is a logic one.

RS-232-C: An industry standard serial communications interface. The most commonly used serial interface for personal computers.

Parity: An error checking protocol based on the assumption that the number of 1s in a character of data is odd or even. Usually one bit is reserved in each frame to make sure that it plus the number of bits in the actual data is an odd or even number, depending on whether odd or even parity is used.

Protocol: The rules of communication and data format in a communications link between two devices.

Shift-clock cycle: One cycle of the SCI clock that gates one bit of data. For Isosynchronous communications, one shift-clock cycle gates one bit of data or format information. In the Asynchronous mode, 16 shift-clock cycles are needed per bit of information.

Slave: A mode of operation in which a microcontroller is controlled by and receives synchronizing signals from another microprocessor.

UART: Universal Asynchronous Receiver/Transmitter; An interface designed to receive and transmit asynchronous signals for a serial device.

Appendix E

References

Friend,G.E., Fike, J.L., Baker, H. C., Bellamy, J. C. Understanding Data Communications. Dallas; Texas Instruments Information Publishing Center. (1984)

Schwartz, Mischa. Information, Transmission, Modulation, and Noise. St. Louis; McGraw-Hill Book Company. (1980)

T.I. Microcontroller Applications Group. TMS370 Family Data Manual. Dallas; Texas Instruments Technical Publishing. (1988)

T.I. Digital Signal Processing Applications Group. TMS320C25 User's Guide. Dallas; Texas Instruments Technical Publishing. (1986)