Point-to-Point Serial Communications Using the SPI Module of the TMS320F240 DSP Controller

APPLICATION REPORT: SPRA451

Authors: Jeff Crankshaw
Jeff Stafford

Digital Signal Processing Solutions
May 1998

TEXAS INSTRUMENTS

IMPORTANT NOTICE

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

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale 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. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any 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.

TRADEMARKS

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Figures

Tables

Point-to-Point Serial Communications Using the SPI Module of the TMS320F240 DSP Controller

Abstract

This application report discusses serial communications using the Serial Peripheral Interface (SPI) synchronous serial port of the Texas Instruments (TI™) TMS320F240 digital signal processor (DSP) controller. A general description of SPI features is provided for necessary background information. The CPU/SPI interface description includes a discussion of the bus interface and interrupt architecture.

An overview of inter-processor communication topologies describes how the SPI module is used in several applications. A hardware implementation example of a point-to-point communication scheme uses the ’F240 SPI in slave mode connected to a Seimens ’C167 SSC configured as the SPI master.

All ’F240 code is generated using TI assembly language tools and validated on the ’F240 EVM hardware. All ’C167 code is generated using the Keil demonstration tools and validated on the Keil MCB-167 evaluation board. Source code for both processors, as well as the necessary header, build, and link files, and tools options are provided as appendices so that the reader can modify this solution to suit a specific need.

Product Support

Related Documentation

The following list specifies product names, part numbers, and literature numbers of corresponding TI documentation.

- *TMS320C24x DSP Controllers CPU, System, and Instruction Set Reference Set, Volume 1*, September 1997, Literature number SPRU160B
- *TMS320C24X DSP Controllers Peripheral Library and Specific Devices Reference Set, Volume 2*, December 1997, Literature number SPRU161B
- *XDS51x Emulator Installation Guide*, January 1996, Literature number SPNU070A

World Wide Web

Our World Wide Web site at **www.ti.com** contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Email

For technical issues or clarification on switching products, please send a detailed email to **dsph@ti.com**. Questions receive prompt attention and are usually answered within one business day.

Introduction

The TI TMS320 family of DSP controllers is designed with special features to facilitate serial communications between multiple processors. In particular, the TMS320x240[1] DSP controller incorporates an SPI module specifically for this purpose. The external hardware and software overhead for inter-processor communication is reduced by the flexibility and programmability of this module.

The SPI is a high-speed synchronous serial I/O port that allows a serial bit stream of programmed length (one to eight bits) to be shifted into and out of the device at a programmable bit transfer rate. The SPI is normally used for communications between the DSP controller and external peripherals or another controller. Typical applications include external I/O or peripheral expansion via devices such as shift registers, display drivers, and analog-to-digital (A/D) converters. Multi-processor communications are supported by the master/slave operation of the SPI. The focus of this application report is the master/slave operation of the SPI, which supports multi-processor communication.

This application report is organized into two main parts.

- The first part consists of a general description of the ’F240 SPI, its interface with the ’C2xx DSP core, and an overview of different serial communication topologies enabled by the SPI.
- The second part describes the implementation of a point-to-point interface between the ’F240 and a Seimens ’C167. This section presents a detailed description of the hardware and software used to implement the communication scheme. All relevant source code and build options are provided in the appendices.

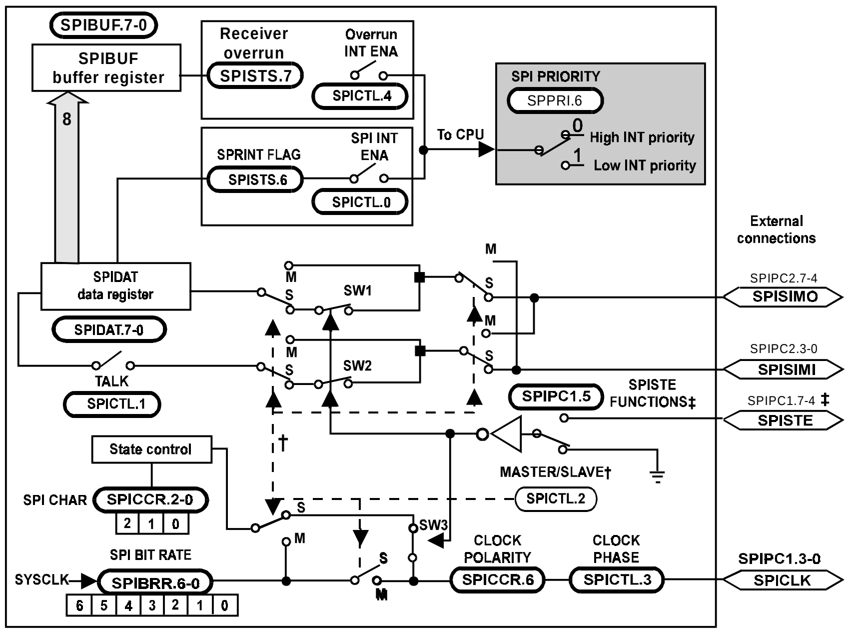
[1] The “x” indicates either ‘F” for flash EEPROM or “C” for masked ROM. For the purposes of this application report, ’F240 is used to represent both flash and ROM devices since the internal program memory type has no effect on the SPI.

SPI Overview

This brief overview of the SPI describes key features with respect to the topic application of inter-processor communications. (A complete description of the SPI module is offered in Chapter 5, of the *TMS320C24X DSP Controllers Peripheral Library and Specific Devices Reference Set, Volume 2*.)

Figure 1 shows the key features of the SPI.

Figure 1. SPI Block Diagram



†The diagram is shown in slave mode.

‡The SPISTE pin is shown as being disabled, meaning the data can be transmitted or received in this mode. Note that switches SW1, SW2, and SW3 are closed in this configuration.

The SPI has four external pins, *SPISOMI, SPISTE, SPISIMO*, and *SPICLK*, providing the interface to external devices. The SPI is a full-duplex communication port, with the simultaneous transmit and receive taking place on the *SPISOMI* and *SPISIMO* pins. The SPICLK pin provides the time base for communications. This pin is bi-directional to allow the time base to be generated in master mode and received in slave mode.

The fourth pin, *SPISTE*, has the capability to act as the slave enable. This operation is conditional, based on the status of the SPISTE FUNCTION bit. When this bit is set and the *SPISTE* input is active low, the SPI sends and receives data from the master. When the *SPISTE* pin is inactive high, the SPI serial shift register is disabled and the *SPISOMI* output pin is placed in the high impedance state. However, when the SPISTE FUNCTION bit is cleared, the SPI receives all data and transmits data based on the status of the TALK bit.

The key features of the SPI are software programmable, which make it extremely flexible and capable of communicating with many types of serial ports. Some of the key programmable features are:

- Operation mode: master or slave
- Character length: 1-8 bits
- Interrupt priority: level 1 or 5
- Serial bit transfer clock rate
- Serial clock phase and polarity

All of the SPI features are controlled by the ten control registers, which are mapped into the internal data space of the ’F240. Table 1 summarizes these ten control registers, including their symbols and addresses.

Table 1. SPI Control Register Memory Map

ADDRESS	SYMBOL	NAME
7040h	SPICCR	Configuration control register
7041h	SPICTL	Operation control register
7042h	SPISTS	Status register
7043h	—	Reserved
7044h	SPIBRR	Baud rate register
7045h–7046h	—	Reserved
7047h	SPIBUF	Serial input buffer register
7048h	—	Reserved
7049h	SPIDAT	Serial data register
704Ah–704Ch	—	Reserved
704Dh	SPIPC1	Port control register 1
704Eh	SPIPC2	Port control register 2
704Fh	SPIPRI	Priority control register

Internal Interface between the SPI Module and ’C2xx CPU

The TMS320x240 DSP controllers represent a new approach to peripheral integration for the TMS320 series. The flexible and easy-to-use peripherals found in the TMS370 family of MCUs have been integrated with the low cost ’C2xx DSP. This new approach differs from the other members in the ’C2xx family (such as the ‘C203 and ‘F206) in that the peripherals are located in data space and operate in a different clock domain based on the system clock.

To better understand the peripheral architecture of the TMS320x240, this section covers some of the basic interface characteristics that are important from a programming point of view. A solid understanding of the architecture helps the programmer to take advantage of this architecture.

Two factors concern the programmer:

- System clock speed
- Register reads/writes

The next two sections describe each factor and its relationship to application development.

System Clock

The first and most important factor to consider when beginning application development is system clock frequency selection. The SPI is clocked by the system clock, SYSCLK, which can be configured to run at either one half or one fourth the frequency of the ’C2xx CPU clock, CPUCLK. The actual SYSCLK pre-scale ratio is determined by the PLL pre-scale bit, PLLPS, in Clock Control Register 0, CKCR0, at address 0x702B in data space.

In most applications, it makes sense to configure SYSCLK to run at its highest frequency, which is one half the CPU clock frequency. This minimizes the number of clock cycles necessary to perform peripheral register accesses. A divide-by-2 ratio is accomplished by setting bit PLLPS to 1 during the initialization sequence following a power-on reset condition.

Once selected, the frequency of SYSCLK remains set to one half of CPUCLK until another power on reset condition occurs. Normal system resets do not affect the PLLPS bit. Example 1 shows a typical code sequence used to configure the clock control register.

Example 1. PLL initialization code

```
* Set Data Page pointer to page 1 of the peripheral frame
      LDP   #DP_PF1  ; Page DP_PF1 includes WET through EINT frames
* Configure PLL for 10MHz osc, 10MHz SYSCLK and 20MHz CPUCLK
      SPLK  #00B1h,CKCR1  ; CLKIN (OSC)=10MHz,CPUCLK=20MHz
      SPLK  #00C3h,CKCR0  ; CLKMD=PLL Enable, SYSCLK=CPUCLK/2,
```

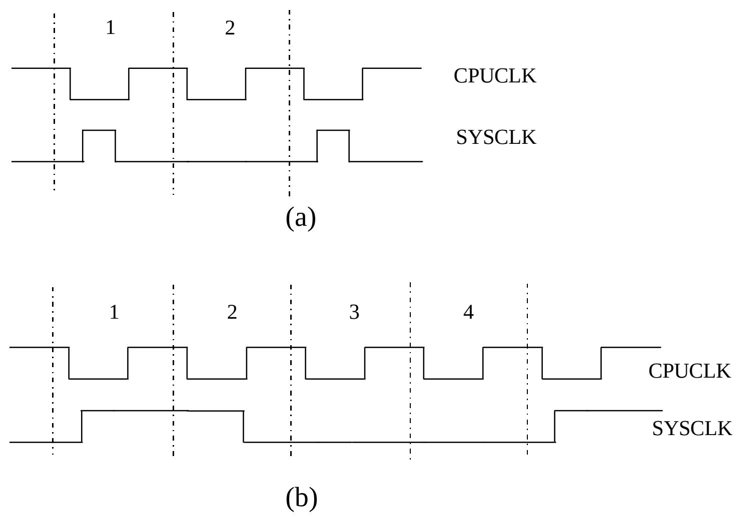
SPI Register Accesses

It is necessary to understand which CPU instructions can be used to access the peripheral registers of the TMS320x240 in order to understand peripheral register accesses. All of the peripherals, including the SPI, are located in data space. Thus, any instruction that operates on data space can be used to access the SPI control registers.

For the ’C2xx CPU, the simplest form of a read is the 16-bit load accumulator instruction, or LACL. Likewise, the simplest form of a write is the 16-bit store accumulator instruction, or SACL. Both are single-cycle instructions when executed from zero wait-state program memory (e.g. internal Flash EEPROM) and the operand is located in zero wait-state data memory (e.g. on-chip Dual Access RAM). The SPI is interfaced to the CPU through the peripheral bus. This internal bus acts as a bridge between the CPU and SPI. To maintain high performance for complex algorithms, the CPU is clocked at a higher clock frequency. The SPI, which has been designed to minimize CPU loading, does not require the faster clock frequency. The peripheral bus acts as a “bridge” between these two clock domains.

Accesses to the slower peripheral bus require the addition of wait states. There are two reasons for the wait states: first, the access has to be synchronized with the peripheral bus. At any given time, the CPUCLK could be in one of two (÷2), or one of four (÷4), phases with respect to the SYSCLK. Peripheral bus accesses will not begin until the SYSCLK is in the correct phase. This relationship is demonstrated in Figure 2.

Figure 2. Representation of Relationship between the CPUCLK and the SYSCLK for (a) Divide-by-2 Mode and (b) Divide-by-4 Mode



The second reason for wait states is the slower peripheral latency or access time. The SPI, and other eight-bit peripherals, require one SYSCLK period for reads and one and a half SYSCLK periods for writes. Thus, single writes to the peripheral bus require one more clock cycle than single reads.

Table 2. Instruction Word and Cycle Counts for Peripheral Accesses

Instruction		**# Words**	**# Cycles** ÷**2**	÷**4**
Reads				
LACL		1	3 or 4	5 - 8
BIT		1	3 or 4	5 - 8
Writes				
SACL	1st access	1	4 or 5	6 - 9
SACL	2nd or more consecutive access	1	4	8
SPLK	1st access	2	5 or 6	7 - 10
SPLK	2nd or more consecutive access	2	6	8

The net effect of the wait states is shown in Table 2. The number of cycles for instructions commonly used to access the SPI registers are shown for both divide-by-2 and divide-by-4 SYSCLK settings. Based on these facts, the following general guidelines can be followed to minimize register access times:

- Configure the SYSCLK for divide-by-2.

  The reset state of the ’F240 sets the SYSCLK divide-by-ratio to 4. The default divide-by-4 mode ensures that the system peripherals would function correctly on power up. The divide-by-ratio can be set to 2 during the initialization routines after reset to minimize the number of cycles required for register reads and writes.

- Use direct addressing with immediate operand for SPI register initialization.

  Since initialization is generally performed once and at least four registers need to be written to configure the SPI, it makes sense to modify the data page pointer and use the direct addressing mode. The store long constant (SPLK) instruction is recommended.

SPI Interrupt

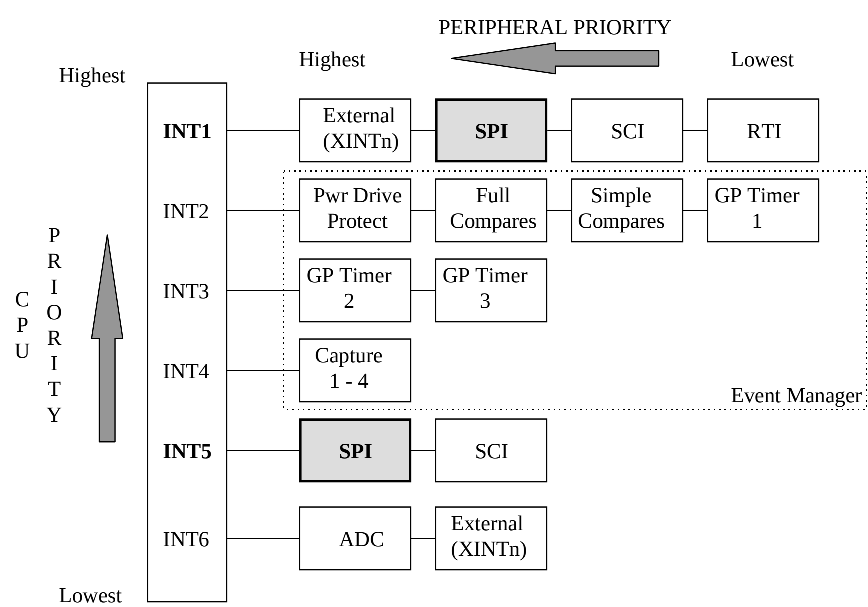
From an application point of view, one of the critical design decisions involves selection of the interrupt strategy. The ’F240 is designed to provide maximum flexibility in the use of interrupts. This section will describe the ’F240 interrupt architecture as it relates to the SPI. In addition, the programmable priority and receive-error detection capabilities of the SPI module are discussed. Finally, examples of interrupt initialization and service code are provided to illustrate these important features.

TMS320x240 Interrupt Architecture Overview

In order to select the correct SPI interrupt strategy for a given design, it is necessary to understand how the SPI interrupt fits into the overall interrupt architecture of the ’F240. An overview of the ’F240 interrupt hierarchy is presented in block diagram form in Figure 3. The DSP core provides six maskable interrupt levels, INT1 through INT6, with INT1 given highest priority and INT6 being the lowest. Since the ’F240 device has more than six maskable interrupts, each of the six interrupt levels are shared by multiple interrupt sources. These interrupt sources are generated from on-chip peripherals and external interrupt pins.

This grouping of peripheral interrupts makes it possible to isolate a particular peripheral to efficiently prioritize its use.

Figure 3. SPI Interrupt Priority within ’F240 Interrupt Hierarchy



SPI Interrupt Conditions

The SPI can initiate an interrupt after the occurrence of either of two events. The first event is defined as occurring after a character has been transmitted or received. The second event is defined as occurring after a receiver overrun condition is encountered. Both of these conditions can be independently enabled and both share a common interrupt vector. Five control bits are used to configure and determine the status of SPI interrupts. Table 3 summarizes their register locations, names, and descriptions.

Table 3. SPI Interrupt Configuration and Status Bits

Control Register	Bit Name	Bit Type	Description
SPIPRI.6	SPI PRIORITY	Control	Determines interrupt level of an SPI interrupt. When set, SPI interrupts are sent on level 5. When cleared, SPI interrupts are sent on level 1.
SPICTL.0	SPI INT ENA	Control	Enables the SPI to request an interrupt when the last bit in a transmit/receive operation has been sent/received. When set, SPI interrupts are enabled.
SPISTS.6	SPI INT FLAG	Status	Indicates the SPI has transmitted/received the last bit in a transmit/receive operation.
SPICTL.4	OVERRUN INT ENA	Control	Enables the SPI to request an interrupt when a transmit/receive operation is completed before the previous character has been read from the buffer.
SPISTS.7	RECEIVER OVERRUN	Status	Indicates the SPI has completed a transmit/receive operation before the previous character has been read from the buffer.

SPI Priority Bit

The first bit that must be configured is the SPI PRIORITY bit. This bit determines the interrupt level of the SPI interrupt request. If this bit is cleared to 0, the interrupt request is seen by the DSP core as a high priority and is received on interrupt level 1 (INT1). If this bit is set to 1, the interrupt request is seen by the DSP core as a low priority and is received on interrupt level 5 (INT5). This allows the designer to have software control over the priority of the SPI interrupt. For example, in some multi-processor applications, it may be desirable to send or receive large amounts of data at a high baud rate.

One common application is to use the SPI to receive flash programming code and data. Other on-chip resources, such as the Event Manager, would be inactive in such a case, and a high priority interrupt would be desirable. Then, after this initial burst of data has been transferred, the SPI interrupt priority can be lowered to allow other interrupts to take priority. This flexibility can be used in other situations where the SPI communications need to take high priority in particular situations, but then can be reduced to a background task with lower priority.

Transmit/Receive Interrupt

The main method to determine completion of an SPI transmit/receive operation is the SPI interrupt. Setting the SPI INT ENA bit enables this interrupt. This event is indicated by the SPI INT FLAG bit. Anytime the SPI completes sending or receiving, this bit is set and the SPI is ready to be serviced. This event causes an interrupt to be requested if the SPI interrupt is enabled.

The user must clear the SPI INT FLAG bit during the SPI interrupt service routine by reading the SPIBUF register. If this register is not read, the SPI INT FLAG remains set and no additional SPI TX/RX interrupts will be generated. This bit is also automatically cleared by writing a 1 to the SPI software reset (SPICCR.7) bit and by any device reset.

Receiver Overrun Interrupt

The SPI can be configured to detect a receiver overrun condition, which occurs anytime the previous character has not been read from the SPI buffer register before a new character is received. Setting the OVERRUN INT ENA bit to a 1 enables this error detection feature. The RECEIVER OVERRUN flag bit is set by the SPI hardware when a receive or transmit operation completes before the previous character has been read from the buffer. This flag indicates that the last received character has been overwritten and therefore lost. When the overrun interrupt is enabled, the SPI requests an interrupt each time this flag is set.

The RECEIVER OVERRUN flag bit must be cleared before another overrun interrupt will be generated by the SPI. It is up to the user to clear this bit, which is most easily done by writing a 0 to it in the SPI interrupt service routine. Other actions that clear the RECEIVER OVERRUN flag bit are an SPI software reset or a device reset, both of which require reconfiguration of the SPI control registers.

Reading the Interrupt Vector

After an interrupt request leaves the SPI peripheral, it is sent to arbitration logic, which compares the priority level of competing interrupt requests and passes the highest priority interrupt request to the DSP core. The corresponding interrupt flag is set in the DSP core's interrupt flag register (IFR). If the corresponding bit in the interrupt mask register (IMR) is set and the INTM bit is 0, the DSP core will acknowledge the interrupt and branch to the interrupt service routine (ISR). The DSP core services any remaining interrupt requests in order of priority.

As described in Figure 3, the ’C2xx CPU has six interrupt levels, but there are multiple requests per level. The ’F240 implements a vector offset scheme to distinguish between the peripheral interrupt requests within each level. Table 4 and Table 5 summarize the peripherals, their interrupt vector offsets, and priorities for both SPI interrupt levels one and five.

Table 4. Interrupt Level 1 (INT1) Interrupt Sources and Overall Priorities

Overall Priority	Interrupt Name	Vector Offset	'F240 Module	Interrupt Function
4	XINT1	0001h	System Module	High-priority external user interrupt
5	XINT2	0011h	System Module	High-priority external user interrupt
6	XINT3	001Fh	System Module	High-priority external user interrupt
7	SPIINT	0005h	SPI	High-priority SPI interrupt
8	RXINT	0006h	SCI	High-priority SCI receiver interrupt
9	TXINT	0007h	SCI	High-priority SCI transmitter interrupt
10	RTINT	0010h	WDT	Real-time interrupt

The SPI shares interrupt level one with interrupts from three external pins (XINT1-3), the SCI receive and transmit interrupts, and the WDT real time interrupt. The level five interrupt is shared between the SPI and SCI modules.

The interrupt vector offset for the requesting source is loaded into the system interrupt vector register (SYSIVR) when the CPU acknowledges its interrupt request. If more than one of the interrupt sources on the selected level is enabled, the user must read the SYSIVR to determine the source of the interrupt. For an SPI interrupt, the vector offset of 0x05 is loaded into the SYSIVR when either of the SPI interrupts (tx/rx complete or rx overrun) is acknowledged.

Table 5. Interrupt Level 5 (INT5) Interrupt Sources and Overall Priorities

Overall Priority	Interrupt Name	Vector Offset	'F240 Module	Interrupt Function
34	SPIINT	0005h	SPI	Low-priority SPI interrupt
35	RXINT	0006h	SCI	Low-priority SCI receiver interrupt
36	TXINT	0007h	SCI	Low-priority SCI transmitter interrupt

In the most general case, the user implements a two-part interrupt service routine. The first part is called the general interrupt service routine (GISR). The second part is called the specific interrupt service routine (SISR).

The CPU branches to and executes the code at GISR1 when an interrupt request on priority level INT1 is acknowledged. The GISR, after performing any necessary context saves, identifies the acknowledged interrupt by reading SYSIVR and then branches to the SISR. The SISR performs the actions specific to the triggering interrupt and then returns program control to the interrupted program sequence.

Putting It All Together

The previous section describes the process of both configuring and servicing the SPI interrupts.. This section presents a simple example of each process. These examples are used as the basis for the point-to-point programs described in the implementation section of this report. (A complete description of the ’F240 interrupt architecture is provided in the *TMS320C24x DSP Controllers CPU, System, and Instruction Set Reference Set*, *Volume 1*.)

Interrupt Initialization

The procedure to enable the SPI interrupts on level 5 involves three steps.

1) First, the CPU interrupt flag and interrupt mask registers must be initialized, as shown in Example 2.

2) Second, the SPI control and priority registers must be configured for the desired operation. The code in Example 3 shows the necessary instructions for enabling both the SPI rx/tx interrupt and rx overrun interrupt on the low priority level. In addition, the status bits are cleared to ensure the SPI is in a known state. Typically, this section of code is included within a subroutine call that configures the rest of the SPI registers.

Example 2. CPU Level 5 Interrupt Initialization Code Sequence

```
;Initialize DSP for interrupts
LDP   #0          ; set dp for CPU memory mapped registers
LACL #010h        ; load mask value for level 5 int. to ACC
SACL  IMR         ; Enable interrupt 5 only (SPI,SCI)
LACL  IFR         ; Clear IFR by reading and
SACL  IFR         ; writing contents back into itself
```

Example 3. SPI Interrupt Initialization Code Sequence

```
LDP   #DP_PF1        ; Change data page for SPI
SPLK  #0040h,SPIPRI ; Set SPI interrupt to low priority.
SPLK  #0000h,SPISTS ; Clear the SPI interrupt status bits
SPLK  #0013h,SPICTL ; Enable SPI INT, TALK, and OVERRUN
                     ; INT; set CLK ph 0, and slave mode
```

3) The third and final step is to clear the global interrupt mask bit. This bit should be cleared just prior to entering the main program loop. The code in Example 4 shows the instruction required to globally enable interrupts.

Example 4. CPU Global Interrupt Enable Code

```
CLRC  INTM       ; Enable global DSP interrupts
```

This same sequence is recommended for initialization of all interrupts. Specifically, the CPU interrupt mask and flag registers are configured and enabled first, followed by the peripheral interrupt initialization. The last step in the sequence should be to clear the global interrupt mask bit, INTM. If a peripheral interrupt is enabled and that interrupt event occurs before the CPU interrupt flag register has been cleared, that peripheral interrupt will never be acknowledged. This is because only one peripheral interrupt request is sent per event.

General Interrupt Service Routine (GISR)

The structure of the GISR varies depending on the ISR method selected. For methods 1 and 2, the GISR includes intermediate branches prior to reaching the SISR, as shown in Example 5 and Example 6. The CPU interrupt vector table contains the branch instruction and address for jumping to the GISR5 for any level 5 interrupt. Once at the GISR, the context is saved before any registers are modified. Then the SYSIVR is loaded into the accumulator with a shift left by one bit. The start location of the peripheral interrupt vector table is added to the shifted interrupt vector offset. This value is then used as the branch address to the peripheral interrupt vector table.

Example 5. CPU Interrupt Vector Table for GISR, Methods 1 and 2

```
        .sect "vectors"
RESET   B  START          ;reset branches to start
        B  PHANTOM        ;Int level 1 not used
        B  PHANTOM        ;Int level 2 not used
        B  PHANTOM        ;Int level 3 not used
        B  PHANTOM        ;Int level 4 not used
        B  GISR5          ;Int level 5, low priority SPI
```

For an SPI interrupt, the program branches to the SPI_VEC location and executes the branch to the SISR (labeled SPI_ISR).

Example 6. GISR Code Sequence, Method 1

```
GISR5   ; first, save machine context
        SST   #0, ST0_TEMP  ;Auto page-0 DP  addressing is used to
        SST   #1, ST1_TEMP  ; save status registers to B2 DARAM.
        LDP   #CONTEXT_MEM_PTR    ; change the dp to context stack
        SACL  CONTEXT_MEM_PTR     ; save lower 16-bits of ACC
        SACH  CONTEXT_MEM_PTR+1; save upper 16-bits of ACC
        SAR   AR7, CONTEXT_MEM_PTR+2
        SAR   AR6, CONTEXT_MEM_PTR+3
        ; begin GISR5 to find which source requested the interrupt.
        LDP   #DP_PF1       ; change dp for SYSIVR
        LACC  SYSIVR,1      ; read SYSIVR with shift of 1 (for x2)
        ADD   #(PERIPH_OFFSET-2)  ; add offset to peripheral
        BACC                  ; int. vector table and branch there
```

```
PERIPH_OFFSET:
XINT_VEC    B  XINT1_ISR     ; vector offset 0x1 is for XINT1
            B  PHANTOM       ; vector offset 0x2 not implemented
            B  PHANTOM       ; vector offset 0x3 not implemented
ADC_VEC     B  ADC_ISR       ; vector offset 0x4 is for ADC
SPI_VEC     B  SPI_ISR       ; vector offset 0x5 is for SPI
```

Method 3 can be used for the interrupt service routine when the SPI is the only active interrupt on level 5. This condition is met in two cases: one, in any application which does not use the SCI; and two, if the SCI is always configured for level 1 interrupts. In either case, the intermediate branches are not necessary, and the CPU interrupt vector table can be written to branch directly to the SPI interrupt service routine, as shown in Example 7.

Example 7. CPU Interrupt Vector Table for GISR, Method 3

```
        .sect "vectors"
RESET   B  START         ;reset branches to start
        B  PHANTOM       ;Int level 1 not used
        B  PHANTOM       ;Int level 2 not used
        B  PHANTOM       ;Int level 3 not used
        B  PHANTOM       ;Int level 4 not used
        B  SPI_ISR       ;Int level 5, low priority SPI
```

Specific Interrupt Service Routine (SISR)

The SISR performs actions specific to the event that caused the interrupt and then returns program control to the interrupted code sequence. The code shown in Example 8 implements the SISR for the SPI based on method 1 or 2 ISR. In the case of method 3, the context save provided at the beginning of GISR5 can be included at the beginning of the SPI_ISR.

This version of an SPI ISR includes a simple test for the rx overrun condition and, if detected, clears the condition and returns from the ISR without performing additional tasks. If the overrun condition is false, the ISR continues with the servicing of the rx/tx interrupt. Again, the specifics of this section are left blank, indicating that the user may write whatever code is deemed appropriate. Finally, the context is restored to the pre-interrupt state before returning from the ISR.

Example 8. SISR Code Sequence for SPI Interrupt Service Routine

```
SPI_ISR  ; first, save machine context
         ;Check for Overrun condition
OVER_RUN:
         LDP   #DP_PF1          ;Page DP_PF1 includes SPI
         BIT   SPISTS, 8        ;Overrun flag (SPISTS.7) set?
         BCND  CLEAR_FLAG, TC   ;If set, clear & return

  ( Insert SPI service code here )

SPI_DONE:  ;restore context as saved in GISR5
         LDP     #CONTEXT_MEM_PTR
         LAR     AR6, CONTEXT_MEM_PTR+3
         LAR     AR7, CONTEXT_MEM_PTR+2
         LACC    CONTEXT_MEM_PTR+1,16
         ADDS    CONTEXT_MEM_PTR
         LDP     #0
         LST     #1, ST1_TEMP
         LST     #0, ST0_TEMP
         CLRC    INTM
         RET

CLEAR_FLAG:
         SPLK  #0H, SPISTS
         B     SPI_DONE
```

The preceding code sequences provide a template for establishing the interrupt service routines for the SPI. These code sequences serve as the basis for the point-to-point application code presented in the implementation section. The reader is referred to the code in the appendix to see the complete implementation in context of the point-to-point application.

Interface Topologies

Three different SPI implementations are presented:

- Point-to-point
- Addressed multi-node
- Chip-enabled multi-node

Each configuration has advantages and disadvantages. Since the point-to-point configuration is the most common configuration for SPI communication, an example is provided in the *Implementation* section.

Point-to-Point

The point-to-point configuration consists of two devices and is the most basic SPI configuration. One-device controls communication by providing the synchronous clock, commonly referred to as the master. The device that receives the clock is commonly referred to as the slave. This configuration can consist of two processors or a processor communicating with a peripheral like an EPROM or an ADC.

Figure 4. High-Level Block Diagram of Point-to-Point Topology

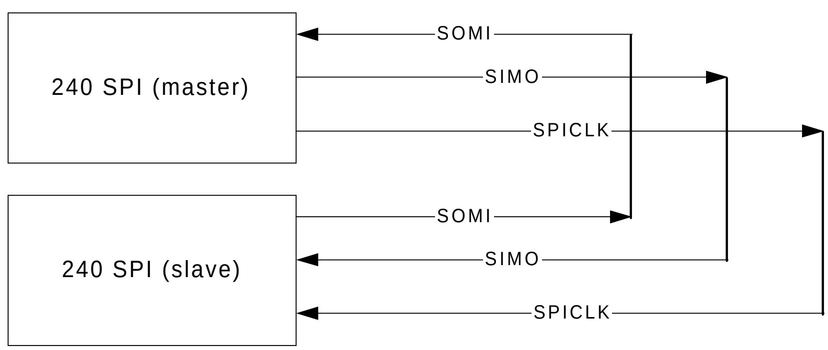
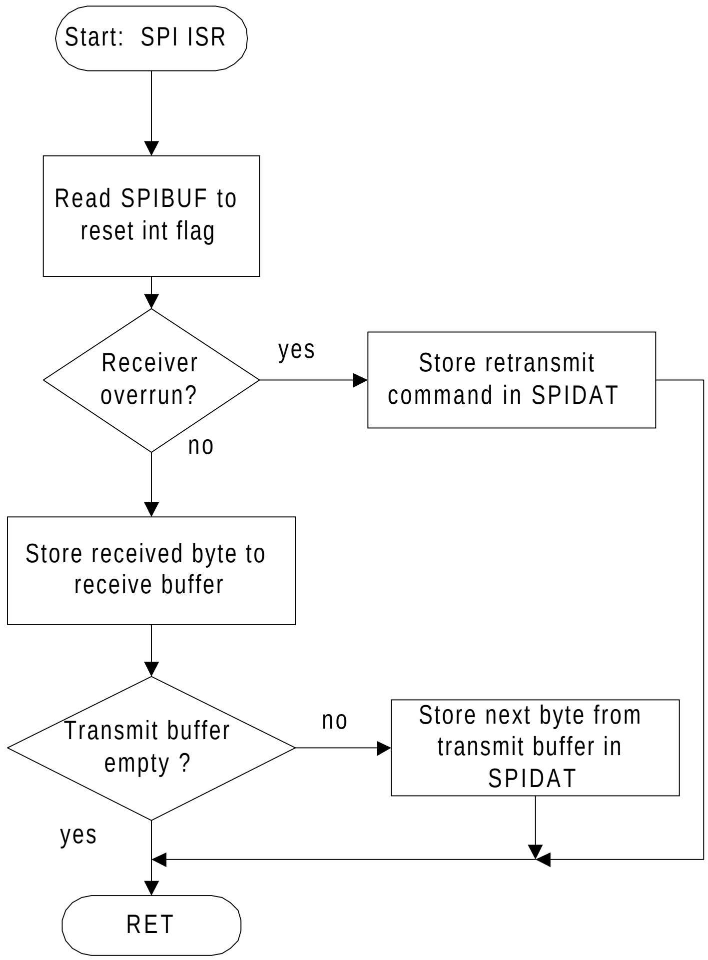


Figure 5 shows a flow chart used to process received data using the point-to-point configuration.

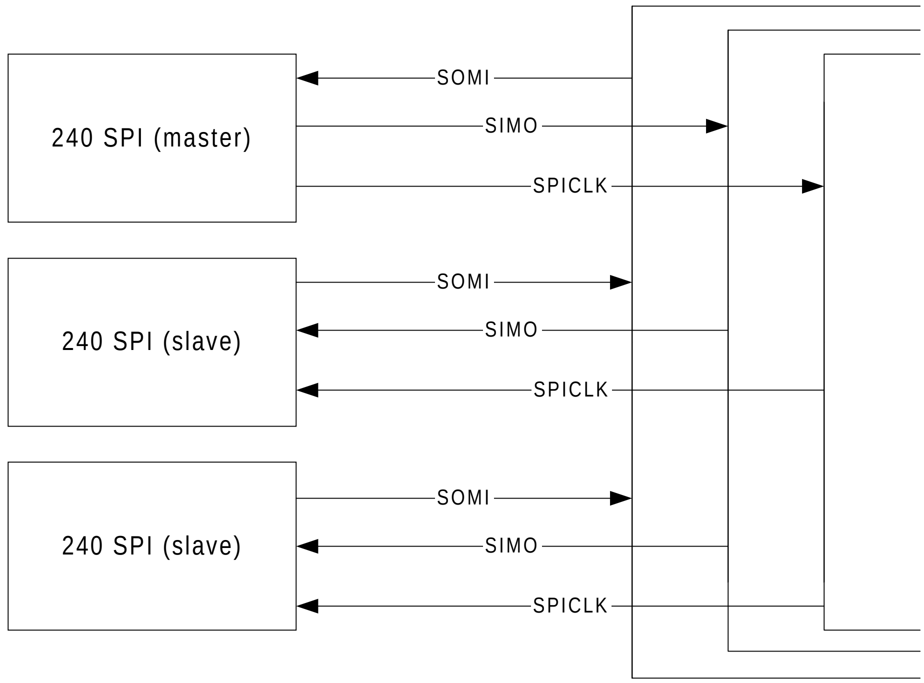
Figure 5. Flow Chart for Point-to-Point Topology



Addressed Multi-Node

The ’F240 SPI module can be used in an addressed-based, multi-node network. The SPI master is always connected to the network and supplies the SPICLK signal to the entire slave SPIs connected to the network, as shown in Figure 6.

Figure 6. High-Level Block Diagram of Addressed Multi-Node Topology

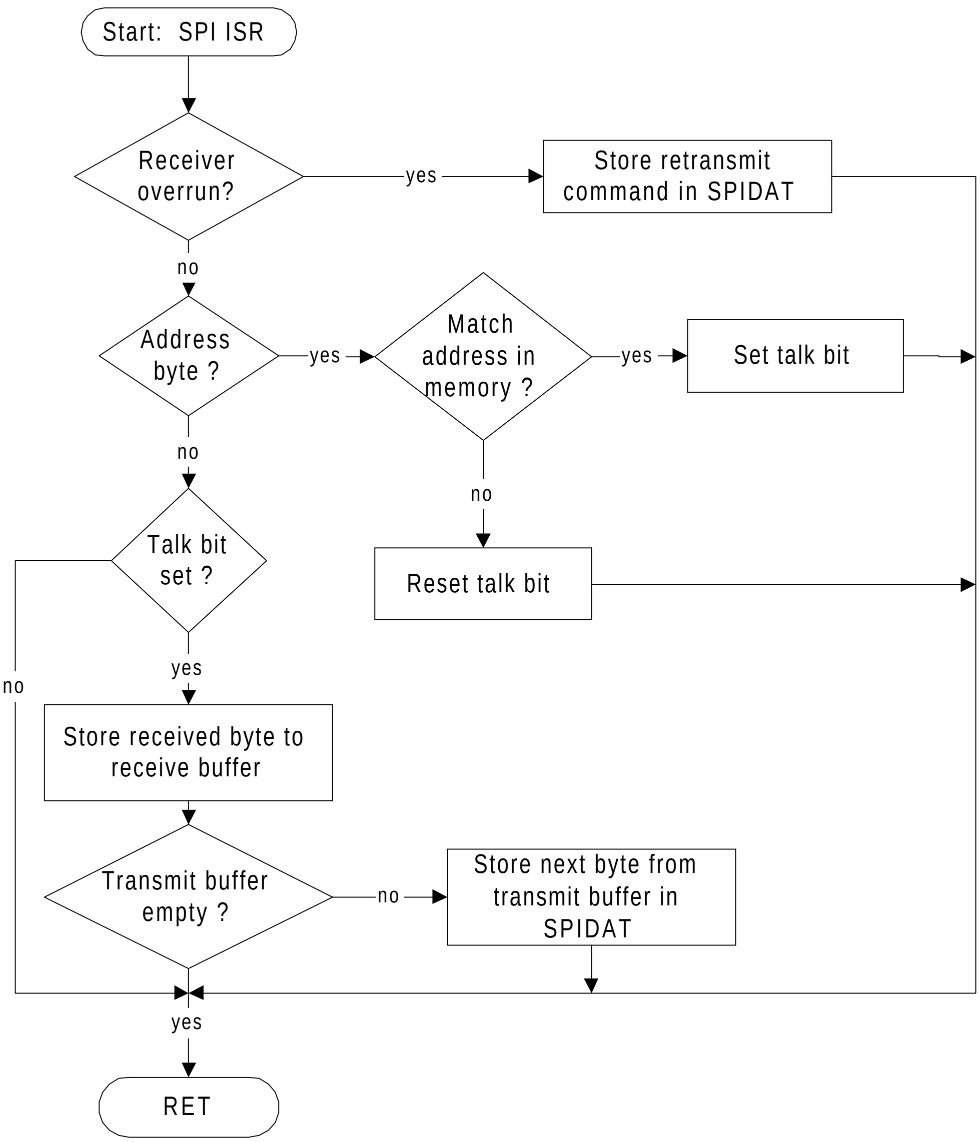


All slave nodes receive all messages sent on the network, but only the slave node addressed by the master node is able to send messages. All other slave nodes have their SOMI pins disconnected from the network via internal circuitry. When a slave node receives an address byte that matches its address stored in memory, its TALK bit is then enabled. By enabling this bit, the slave node's SOMI pin is connected to the SPI's internal circuitry. This is functionality is handled in the SPI ISR.

Once the addressed slave node is attached to the network, the master and slave establish a connection that is essentially a point-to-point type connection. In addition to the address recognition, all real-time issues discussed for the point-to-point topology are applied here.

Figure 7 shows a flow chart used to process received data using this configuration.

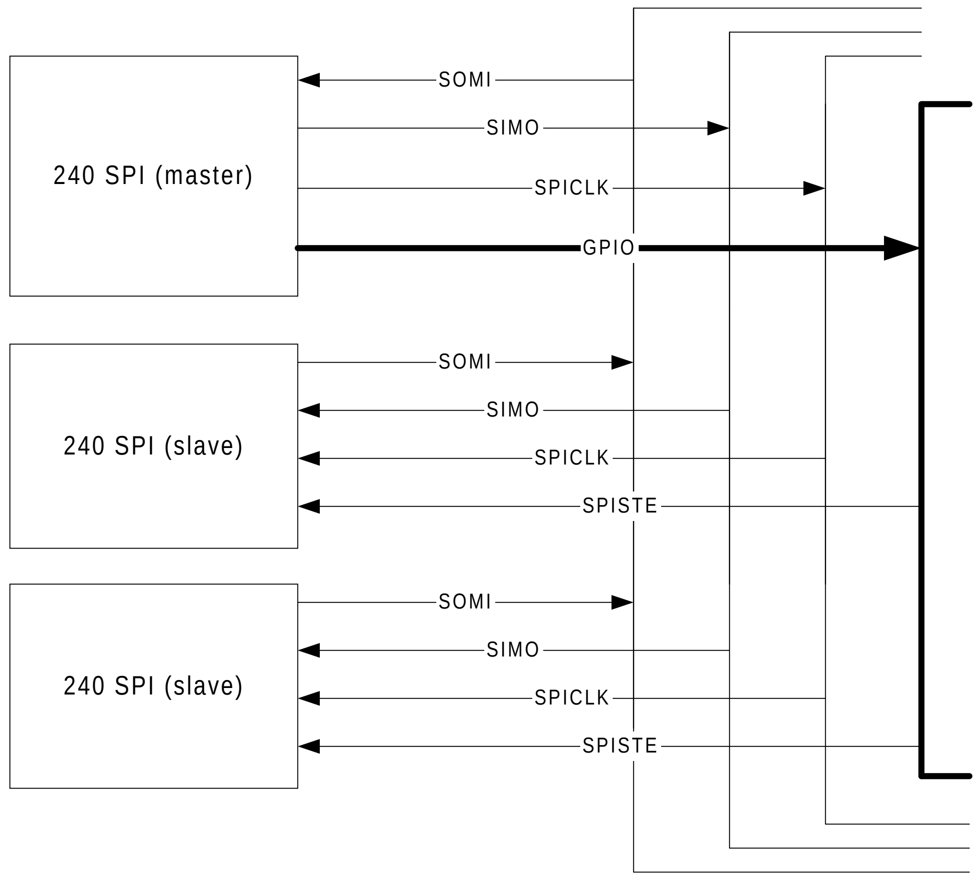
Figure 7. Flow Chart for Addressed Multi-Node Topology



Chip-Enabled Multi-Node

By using direct chip-enables for each slave node, the ’F240 SPI module can be used in multi-node networks without using addresses. The master node must provide a digital output for each slave's strobe pin (SPISTE). The master node always has its SIMO connected to the network and supplies the SPICLK signal to the entire slave SPIs connected, as shown in Figure 8.

Figure 8. High-Level Block Diagram of Chip-Enabled Multi-Node Topology



The basic function of the strobe (SPISTE) pin is to act as a transmit enable for the slave nodes. It stops the shift register so that it cannot receive data. This is an advantage over the address-based network, in which all nodes receive every byte transmitted on the network. Instead of every slave node interrupting on every byte, only the SPISTE selected slave-node will interrupt and receive the transmitted bytes from the master node.

The SPISTE pin also 3-states the SOMI pin, eliminating the need to set and reset the TALK bit. For the master node, the SPISTE pin is automatically configured as a GPIO and can be used as one of the outputs to enable selected slave nodes. The disadvantage of this approach is that the number of outputs available on the master node limits the number of nodes that can be on the network.

The procedure to process received information in this configuration is the same as the Point-to-Point configuration and is shown in Figure 5.

Implementation

The Point-to-Point communication scheme described in *Interface Topologies* is implemented using an ’F240 as the slave “point” and either a different ’F240 as the master “point” or an 80C167CR as the master “point”. The hardware used for the ’F240 platform is the ’F240 EVM available from Texas Instruments. The ’C167 hardware is the MCB-167 prototype board available from Keil Software. The hardware and software aspects of the implementations are provided in the following sections. In addition, a step-by-step procedure is provided describing how the application was tested.

Hardware Description

One of the benefits of the SPI interface is the simple three-wire interface. The slave SPI port is directly connected to the master SPI (SSC configure as SPI for the ’C167) port. The high-level block diagrams of Figure 9 show the connections for each implementation.

Figure 9. High-Level Block Diagram of Point-to-Point Interconnections for (a) ’F240 to ’F240 and (b) ’C167 to ’F240

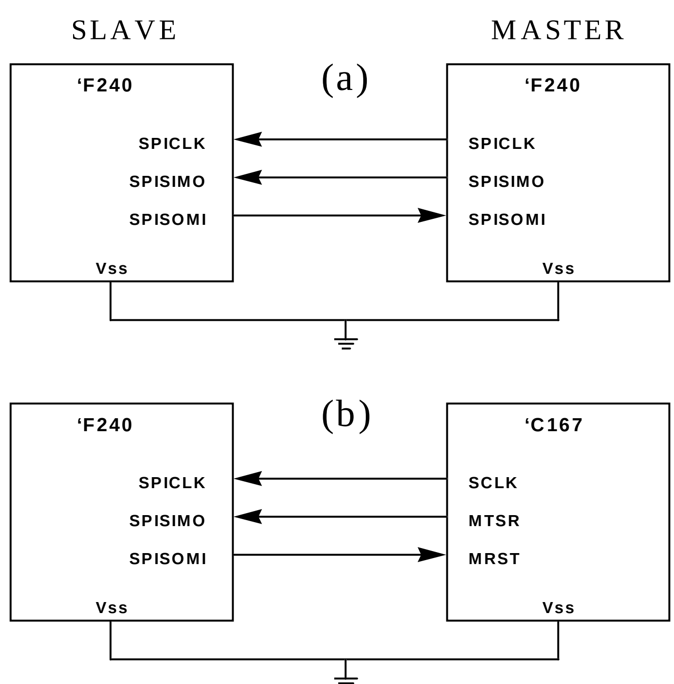


Table 6 and Table 7 list the actual pin connections for each implementation.

Table 6. ’F240 EVM to ’F240 EVM Board Connections

’F240 EVM I/O Connector (P1)			’F240 EVM I/O Connector (P1)	
SPICLK/IO	pin 31	connected to	pin 31	SPICLK/IO
SPISOMI/IO	pin 30	connected to	pin 30	SPISOMI/IO
SPISIMO/IO	pin 29	connected to	pin 29	SPISIMO/IO
GND	pin 33	connected to	pin 33	GND

Table 7. ’F240 EVM to MCB-167 Board Connections

’F240 EVM I/O Connector (P1)			MCB-167 Wire Wrap Field (P3)	
SPICLK/IO	pin 31	connected to	p3.13	SCLK
SPISOMI/IO	pin 30	connected to	p3.8	MRST
SPISIMO/IO	pin 29	connected to	p3.9	MTSR
GND	pin 33	connected to	COM,pin1	GND

Software Description

The software written for the point-to-point communication implementation is divided into three sections. The slave code written for the ’F240 EVM is described first. This is followed by two versions of the master code; one for the ’F240 and another for the ’C167. The ’F240 master code was written in assembly to debug the slave code. Once the slave code was fully functional, the ’F240 master code was rewritten in C for the ’C167. Both sets of code are included for reference in the appendices.

’F240 Slave Code

The ’F240 slave supports the three message types used in this example: (1) send new parameters and receive status update; (2) send new parameters; (3) status information request. The message formats for the three messages are the following:

(1) send new parameters and receive status update

Byte	Description
1	Message type = 3, bits 0 & 1 are set
2	message length, which is 3+n bytes long, where n is the number of parameters
3 to n+2	n data parameters
n+3	checksum of bytes 1 through n+2

(2) send new parameters

Byte	Description
1	message type = 1, bit 0 is set
2	message length, which is 3+n bytes long, where n is the number of parameters
3 to n+2	n data parameters
n+3	checksum of bytes 1 through n+2

(3) status information request

Byte	Description
1	message type = 2, bit 1 is set
2 to n	Checksum request bytes (#08) are sent by the Master until the Slave sends the last byte of the status information.

The ’F240 slave source code developed for this application report consists of the following files: SPI.ASM, SPI.CMD, and F240REGS.H. These files are available for review in Appendix D.

The file SPI.ASM contains the entire program with the following main sections:

Label	Description
START	System and SPI initialization.
MAIN	Continuous loop waiting for a byte to be received.
RD_MSG	Processes each received byte, it is called by MAIN whenever a byte is received.
SPI_ISR	Interrupt service routine for the SPI.

System Initialization

At reset the initialization code starts execution at the label START:

```
START:   CLRC  SXM
```

This is done by placing the instruction "B START" at the reset vector location. All ISRs used in this example are defined in the user named section "vectors".

```
.sect   "vectors"
B  START    ;reset vector
B  START    ;Int level 1 not used
B  START    ;Int level 2 not used
B  START    ;Int level 3 not used
B  START    ;Int level 4 not used
B  SPI_ISR  ;Int level 5, low priority SPI
```

The initialization code then disables the watchdog, defines the clock, and clears the system reset flag bits. The DARAM blocks B1 and B2 are initialized to zero since their values are not defined at reset.

```
SPLK  #06Fh, WDCR    ;  Clear WDFLAG, Disable WDT,
                     ;set WDT for 1 second overflow
SPLK  #07h, RTICR    ;  Clear RTI Flag, set RTI for
                     ;1 second overflow (max)
                     ;EVM 10MHz oscillator settings.
                     ;(XTAL2 open,OSCBYP_=GND)
SPLK  #00B1h,CKCR1   ; CLKIN(OSC)=10MHz, Mult by 2,
                     ;Div by 1.
SPLK  #00C3h,CKCR0   ; CLKMD=PLL, SYSCLK=CPUCLK/2,
; Clear reset flag bits in SYSSR
LACL  SYSSR
AND   #00FFh
SACL  SYSSR
```

After the system has been initialized, starting values are given to the variables used by the application: STATUS_INFO, NEW_BYTE_FLAG, and COPIED. The five bytes of status information are defined at memory location STATUS_INFO and used to respond to a request by the Master SPI for status information. The single-byte flag NEW_BYTE_FLAG is used by the SPI ISR to indicate to the main loop that a new byte has been received and requires service. The single-byte flag COPIED is used by RD_MSG to limit copying status information to the DATA_OUT buffer once per request. The pointers for the DATA_OUT and DATA_IN buffers are initialized to start at the top of each buffer. The default transmission byte from the slave is the ACKnowledge byte (#0FFh), and this is also stored at the top of the DATA_OUT buffer.

```
; initialize STATUS_INFO
LAR         AR1,#STATUS_INFO
MAR         *,AR1
SPLK        #01,*+   ; STAT1 = 1
SPLK        #02,*+   ; STAT2 = 2
SPLK        #03,*+   ; STAT3 = 3
SPLK        #04,*+   ; STAT4 = 4
SPLK        #05,*+   ; STAT5 = 5
```

```
;Initialize NEW_BYTE_FLAG
LDP        #NEW_BYTE_FLAG
SPLK       #01h, NEW_BYTE_FLAG
SPLK       #0, COPIED

;Initialize RX and TX buffers
LDP        #DATA_IN_PTR
SPLK       #DATA_IN, DATA_IN_PTR     ;Reset RX buffer ptr
SPLK       #DATA_OUT, DATA_OUT_PTR   ;Reset TX bfr ptr
SPLK       #0FFH, DATA_OUT           ;Init default TX byte = ACK
```

The final system initialization involves enabling core interrupt #5 that is used by the SPI interrupts and clearing the interrupt flag register.

```
;Initialize DSP for interrupts
LAR   AR6,#IMR
LAR   AR7,#IFR
MAR   *,AR6
LACL  #010h
SACL  *,AR7    ; Enable int 5 only, SPI low
               ; priority
LACL  *        ; Clear IFR by reading & writing
SACL  *,AR6    ;contents
```

SPI Initialization

The SPI is initialized with a call to INIT_SPI. It is set up to be in slave mode, to use its low priority interrupt, and to have a character length of 8 bits.

```
INIT_SPI:  ; initialize SPI in slave mode
LDP   #DP_PF1
SPLK  #00C7h,SPICCR ; Reset SPI, write 1 to SWRST
SPLK  #0000h,SPICTL ; Disable ints & TALK, normal
                    ;clock, SLAVE
SPLK  #0040h,SPIPRI ; Set SPI int to low priority.
SPLK  #0000h,SPISTS ; Clear the SPI int status bits
SPLK  #0013h,SPICTL ; Enable TALK, ena SPI int, CLK
```

```
                        ;ph 0, slave
SPLK  #0002h,SPIPC1
SPLK  #0022h,SPIPC2 ; Set SIMO & SOMI function to
                    ;serial I/O
SPLK  #0047h,SPICCR ; Release SWRST, clock polarity
                    ;1, 8 bits
RET                 ; Return to MAIN routine.
```

Main Program

Once the system and the SPI have been initialized, the final step before entering the main loop is to unmask the global interrupt mask bit. With this bit cleared, the SPI interrupts are completely enabled and a continuous loop is started at the label MAIN. This loop continues until the NEW_BYTE_FLAG is reset to zero by the SPI_ISR, indicating that a new byte has been received and requires processing.

```
;Initialize DSP interrupts globally
CLRC  INTM     ; Enable DSP interrupts
MAIN:
LDP   #NEW_BYTE_FLAG
MAR   *, AR5
LAR   AR5, NEW_BYTE_FLAG  ;Update AR5
BANZ  MAIN,*
```

Message Read Function

When a new byte has been received, the code at the label RD_MSG is executed (see Figure 10). RD_MSG begins by resetting the NEW_BYTE_FLAG. SPI interrupts are not allowed at this time to prevent the SPI ISR from attempting to reset the flag because it has already been reset from the previous SPI ISR. Therefore, the SPI ISR is disabled until the flag is set.

```
RD_MSG:
LDP   #IMR/128
LACL  IMR      ;  Mask SPI int to avoid changes
AND   #0FFEFh  ;to DATA_IN_PTR.  NEW_BYTE_FLAG is
SACL  IMR      ;a handshake with SPI_ISR & RD_MSG.
LDP   #NEW_BYTE_FLAG
SPLK  #1, NEW_BYTE_FLAG
LDP   #IMR/128
```

```
LACL  IMR      ;  Unmask SPI interrupt
OR #010h
SACL  IMR
```

The first task the RD_MSG code completes is to find out if the received byte is a CHECKSUM_RQST (#08). This byte is sent repeatedly by the master until the result of the checksum comparison is sent. Therefore, it does not require processing and RD_MSG discards it and resets the received buffer (DATA_IN).

```
CHECKSUM_RQST:
LDP     #DATA_IN           ;  Is 1st byte a Checksum Request
LACC    DATA_IN            ;msg from the Master SPI?  If so,
SUB     #CHKSM_RQST_MSG    ;then no processing is
                           ;required, reset rcv
BCND    RESET_RCV_BFR, EQ ;buffer and return.
```

If the master is requesting status information from the slave, indicated by bit #1 being set in the message type byte, then status information will be copied into the DATA_OUT buffer. The code used in this example uses 5 bytes to represent the status information. Any number of bytes can be used.

```
STATUS_INFO_RQST:   ;  Is status info requested?
BIT   RCV_MSG_TYPE, BIT1  ;Yes if bit #1 is set in
                          ;the first rcvd byte,
BCND  STATUS_RQST, NTC    ;the msg-type byte.  If
                          ;status info is requested
                          ;then copy it to
DATA_OUT.
COPY_STATUS_INFO:
BIT   COPIED, BIT0
BCND  MSG_COMPLETE, TC
LDP   #IMR/128
LACL  IMR      ;  Mask SPI interrupt until DATA_OUT
AND   #0FFEFh ;has been updated.
SACL  IMR

LDP   #DATA_OUT_PTR
MAR   *, AR6
LAR   AR6, DATA_OUT_PTR
```

```
ADRK  #01
SAR   AR6, DATA_OUT_PTR
LDP   #STATUS_INFO
RPT   #(STATUS_INFO_BYTES - 1)
BLDD  #STATUS_INFO, *+
SBRK  #1
SAR   AR6, DATA_OUT_PTR
SPLK  #01, COPIED ;  Set flag to copy only once

LDP   #IMR/128
LACL  IMR   ;  Unmask SPI interrupt
OR    #010h
SACL  IMR

STATUS_RQST:
LDP   #DATA_IN ;  Is this 1st byte a Status Request
LACC  DATA_IN  ;msg from the Master SPI?  If so,
SUB   #STATUS_RQST_MSG ;then no further processing
                       ;is required, reset rcv
BCND  RESET_RCV_BFR, EQ;buffer and return.
```

After the status information has been completed, two checks are made before further processing of the message can take place. If the message is not complete as defined by the RCV_BYTE_NO, byte #2 in the received message, the routine returns. This byte consists of the total number of bytes in the message, from byte #1 (message type) to the last byte (checksum). The purpose of this check is to verify that all of the parameter bytes and the checksum byte are received.

The second check verifies that the DATA_OUT buffer is empty. All transmission must be completed before the checksum comparison results can be saved to the DATA_OUT buffer. In addition, the COPIED flag is reset to allow new status information to be copied to the now empty DATA_OUT buffer.

```
MSG_COMPLETE:
LDP   #DATA_IN_PTR
LACC  DATA_IN_PTR   ;  Have all bytes been rcvd?
SUB   #DATA_IN      ;If not, then do not process.
SUB   #04h          ; Min bytes in msg = 4,
BCND  MAIN, LT      ; excluding ChksumRqst.
LACC  DATA_IN_PTR
SUB   #DATA_IN
LAR   AR6, #RCV_BYTE_NO
MAR   *, AR6
SUB   *
BCND  MAIN, LT

DATA_OUT_EMPTY:
LACC  DATA_OUT_PTR  ;  DATA_OUT buffer empty?
SUB   #DATA_OUT     ;If not, then do not process.
BCND  MAIN, NEQ
SPLK  #0, COPIED    ;  Reset flag for next status
                    ;rqst msg
```

Now that all of the bytes have been received, the checksum is calculated and compared to the received checksum. The results of this comparison are stored in the DATA_OUT buffer in one byte. Bit #3 of this byte is used to indicate whether the result was valid or invalid: VALID_CHKSUM (#00) or INVALID_CHKSUM (#04).

```
VERIFY_CHECKSUM:
LACL  *-    ;  Calc checksum & compare w/ rcvd
SUB   #02H  ;value.  Chksum is calc'd by adding all
SACL  VAR1  ;rcvd bytes except the rcvd checksum.
LACC  #0    ;Only the LSB of the result is used.
RPT   VAR1
ADD   *+
AND   #0FFh
SACL  VAR1
SUB   *
BCND  STORE_CHECKSUM, EQ
LACL  #INVALID_CHKSUM
```

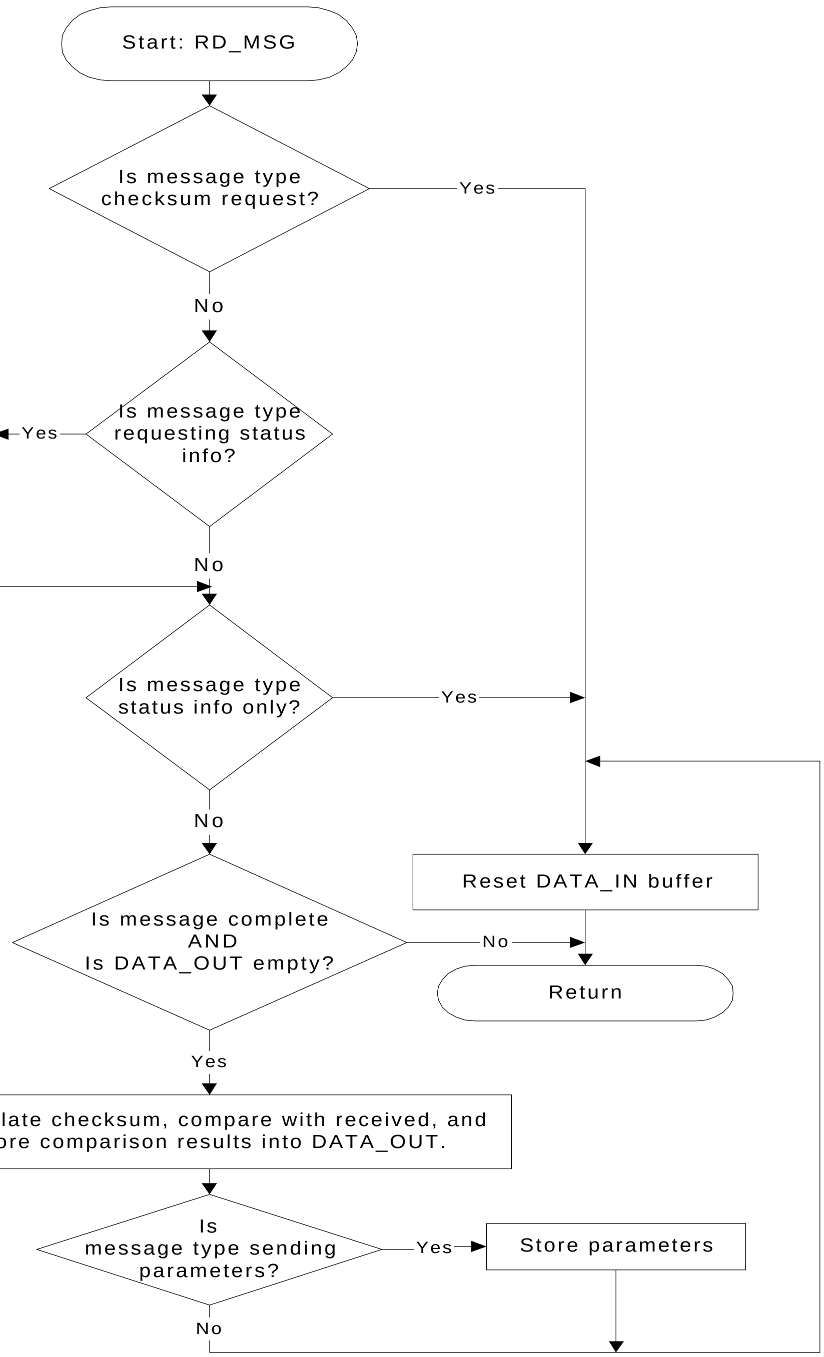
```
STORE_CHECKSUM:        ;  Store chksum comparison
LAR   AR6, DATA_OUT_PTR;into DATA_OUT buffer.
ADRK  #01h
SAR   AR6, DATA_OUT_PTR
SACL  *+
```

If the message type indicates that the message contains parameters being sent from the master, they are stored into memory. The received parameters have been received without errors since the checksum was verified in the earlier step.

```
PARAMS_RQST:     ;  Were new parameters downloaded?
BIT   RCV_MSG_TYPE, BIT0  ;If so, transfer from
BCND  RESET_RCV_BFR, NTC  ; DATA_OUT buffer.

STORE_PARAMS:
LACL  RCV_BYTE_NO ;Get # of data bytes
SUB   #04H
SACL  VAR1
LAR   AR6, #RCV_DATA_START  ;Point to 1st data byte
RPT   VAR1       ;Copy data bytes to mem PARAMS
BLDD  *+, #PARAMS
```

Figure 10. Flow Chart for Read Message Function



The SPI ISR is designed to complete the required tasks quickly to enable the use of the highest baud rate possible. After the context save, the SPIDAT is loaded with the next transmit byte, which is the one DATA_OUT_PTR points to. This is the first task since the SPI is essentially a shift register. Once the SPIDAT is updated with the next transmit byte, the slave's SPI is ready to receive the first bit of the next byte from the master.

```
SPI_ISR:
SST   #0, ST0_TEMP   ;  Context save, auto page-0 DP
SST   #1, ST1_TEMP
LDP   #CONTEXT_MEM_PTR
SACL  CONTEXT_MEM_PTR
SACH  CONTEXT_MEM_PTR+1
SAR   AR7, CONTEXT_MEM_PTR+2
SAR   AR6, CONTEXT_MEM_PTR+3
LDP   #IMR/128
LACL  IMR
LDP   #CONTEXT_MEM_PTR
SACL  CONTEXT_MEM_PTR+4

OVER_RUN:            ;  Overrun?
LDP   #DP_PF1        ;  Page DP_PF1 includes SPI
BIT   SPISTS, 8      ;Overrun flag (SPISTS.7) set?
BCND  CLEAR_FLAG, TC ;If set, clear & return

NEXT_TX:      ;  Send next TX byte
LDP   #DATA_OUT_PTR
LAR   AR6, DATA_OUT_PTR
MAR   *,AR6
LACL  *-      ;  ACC = next byte out
LDP   #DP_PF1
SACL  SPIDAT  ;  Send next byte out, ACK is default

LDP   #DATA_OUT_PTR
LACC  #DATA_OUT      ;  If the TX pointer is not at
SUB   DATA_OUT_PTR   ;the start pos, then point
BCND  READ_SPI, EQ   ;it at next byte to be sent.
```

```
SAR   AR6, DATA_OUT_PTR;Save TX pointer back to
                        ;memory
```

Once the SPI is ready to receive the next byte, the received byte is then read from SPIBUF and stored in the DATA_IN buffer. The NEW_BYTE_FLAG is reset to zero to trigger the RD_MSG routine to process this new byte. The last task for the SPI ISR is to do a context restore.

```
READ_SPI:
LDP   #DP_PF1          ;  Page DP_PF1 includes SPI
LACC  SPIBUF

LDP   #DATA_IN_PTR
LAR   AR7, DATA_IN_PTR ;  Update RX ptr
MAR   *, AR7
SACL  *+
SAR   AR7,DATA_IN_PTR  ;  Set RX ptr to next entry

LDP   #NEW_BYTE_FLAG   ;  Trigger RD_MSG
SPLK  #0, NEW_BYTE_FLAG

SPI_DONE:
LDP   #CONTEXT_MEM_PTR ;  Context restore
LACL  CONTEXT_MEM_PTR+4
LDP   #IMR/128
SACL  IMR
LDP   #CONTEXT_MEM_PTR
LAR   AR6, CONTEXT_MEM_PTR+3
LAR   AR7, CONTEXT_MEM_PTR+2
LACC  CONTEXT_MEM_PTR+1,16
ADDS  CONTEXT_MEM_PTR
LDP   #0
LST   #1, ST1_TEMP
LST   #0, ST0_TEMP
CLRC  INTM
RET
```

Assemble and Link Options

dspcl -v2xx -i%x -gk -alsx spi.asm -z spi.cmd

%x = project directory

’F240 Master Test Code

The ’F240 master test code is based on the slave code. External interrupt-1 (XINT1) is added to manually trigger the master to send the next byte in the TEST_MSG to the slave. At the beginning of the code TEST_MSG is initialized with the message that is being tested. This message can be changed directly from the debugger interface with a Memory window located at memory location TEST_MSG.

```
; initialize TEST_MSG
LAR   AR1,#TEST_MSG ; AR1 <= TEST_MSG start address
MAR   *,AR1
SPLK  #03h,*+
SPLK  #06h,*+
SPLK  #02h,*+
SPLK  #03h,*+
SPLK  #01h,*+
SPLK  #0Fh,*+
MAR   *-
LDP   #TEST_MSG_END
SAR   AR1, TEST_MSG_END
LAR   AR1, #TEST_MSG_PTR
SPLK  #TEST_MSG, *
LDP   #NEW_BYTE_FLAG
SPLK  #1, NEW_BYTE_FLAG
```

SPI Initialization

The SPI is initialized with call to INIT_SPI. It is setup to be in master mode, to use its low priority interrupt, and to have a character length of 8 bits.

```
INIT_SPI:   ; initialize SPI in slave mode
LDP   #DP_PF1
SPLK  #00C7h,SPICCR ; Reset SPI, write 1 to SWRST
SPLK  #0004h,SPICTL ; Disable ints & TALK, normal
                    ;clock, SLAVE
SPLK  #0040h,SPIPRI ; Set SPI int to low priority.
SPLK  #0000h,SPISTS ; Clear the SPI int status bits
SPLK  #007Fh,SPIBRR ; Slow baud rate to test code
SPLK  #0017h,SPICTL ; Enable TALK, ena SPI int, CLK
                    ;ph 0, master
SPLK  #0002h,SPIPC1
SPLK  #0022h,SPIPC2 ; Set SIMO & SOMI function to
                    ;serial I/O
SPLK  #0047h,SPICCR ; Release SWRST, clock polarity
                    ;1, 8 bits
RET                 ; Return to MAIN routine.
```

The order of initializing the SPI control registers can be important in master mode. Transitions on the SPICLK pin of the master SPI can be interpreted by the slave SPI as a valid clock cycle. Unwanted transitions on the master SPICLK pin can be avoided by initializing the SPI registers in the order shown in Table 8.

Table 8. Master Mode SPI Register Initialization Sequence for Falling Edge

Action	SPICLK Direction	SPICLKOutput State
1. Device reset	input	x
2. Configure SPICLK for inactive state (0xC7 => SPICCR)	input	Inactive High
3. Configure SPICLK as an serial clock (0x02 => SPIPC1)	output	Inactive High
4. Maintain SPICLK inactive state when SPI reset is released (0x47 => SPICCR)	output	Inactive High

The first write to the SPI configuration control register (SPICCR) must set the CLOCK POLARITY bit so that the SPICLK is in the appropriate inactive state for the clocking scheme used. The first write is required to initiate SPI software reset. Once the appropriate state has been set the SPICLK function can be selected.

In master mode, selecting the SPICLK function immediately configures the SPICLK pin as an output. The CLOCK POLARITY bit determines the state. The second write to SPICCR releases the SPI software reset condition and maintains the appropriate inactive state on SPICLK.

XINT1 External Interrupt Service Routine (TX begin)

After the context save of the XINT1 ISR loads SPIDAT with the byte pointed to by TEST_MSG_PTR. The XINT1 input is then debounced to eliminate multiple XINT1 interrupts.

```
SEND_TEST_MSG:
   LDP   #TEST_MSG
   LAR   AR0, TEST_MSG_END
   LAR   AR6, TEST_MSG_PTR
   MAR   *, AR6
   LACL  *+
   SAR   AR6, TEST_MSG_PTR
   LDP   #DP_PF1
   SACL  SPIDAT

   CMPR  2  ;Compare AR6 and AR0
   BCND  DEBOUNCE_SW, NTC ;PTR WITHIN RANGE?
   LDP   #TEST_MSG
   LAR   AR6, #TEST_MSG;RESET IT IF NOT
   SAR   AR6, TEST_MSG_PTR

DEBOUNCE_SW
   LAR   AR0, #0100h
DELAY2
   LAR   AR6, #03FFFh
   MAR   *, AR6
DELAY1
```

```
    BANZ  DELAY1
    MAR   *, AR0
    BANZ  DELAY2

    LDP   #DP_PF1
WAIT
    BIT   XINT1CR, BIT6
    BCND  HIGH, TC
    B  WAIT  ;Wait for XINT1 pin to goto a logic 0
HIGH
    LDP   #0 ;Debounce switch, clear pending ints
    LACL  #01
    SACL  IFR
    LDP   #DP_PF1
    SPLK  #001h, XINT1CR;Clear transition detect
```

’C167 Master Test Code

The ’C167 C-source code developed for this application report implements a simple test routine to verify the three message types supported by the ’F240 slave implementation. The function of each software module is described in the following sections. The entire program is contained within three files:

- spi.c

  This file contains the declarations for the entire program and contains the following functions:

Function	Description
main	Initializes variables, calls system initialization, and performs 3 message transmit/receive sequences.
tx_isr	Services the SSC transmit interrupts by loading the next byte to be transmitted into the SSC transmit buffer register, SSCTB
start_tx	Begins message transmit/receive sequence by loading the first byte into the SSC transmit buffer register, SSCTB
rx_isr	Services the SSC receive interrupts by reading the received byte from the SSC receive buffer register, SSCRB, into the status buffer, STATUS_BUF.

- spi.h

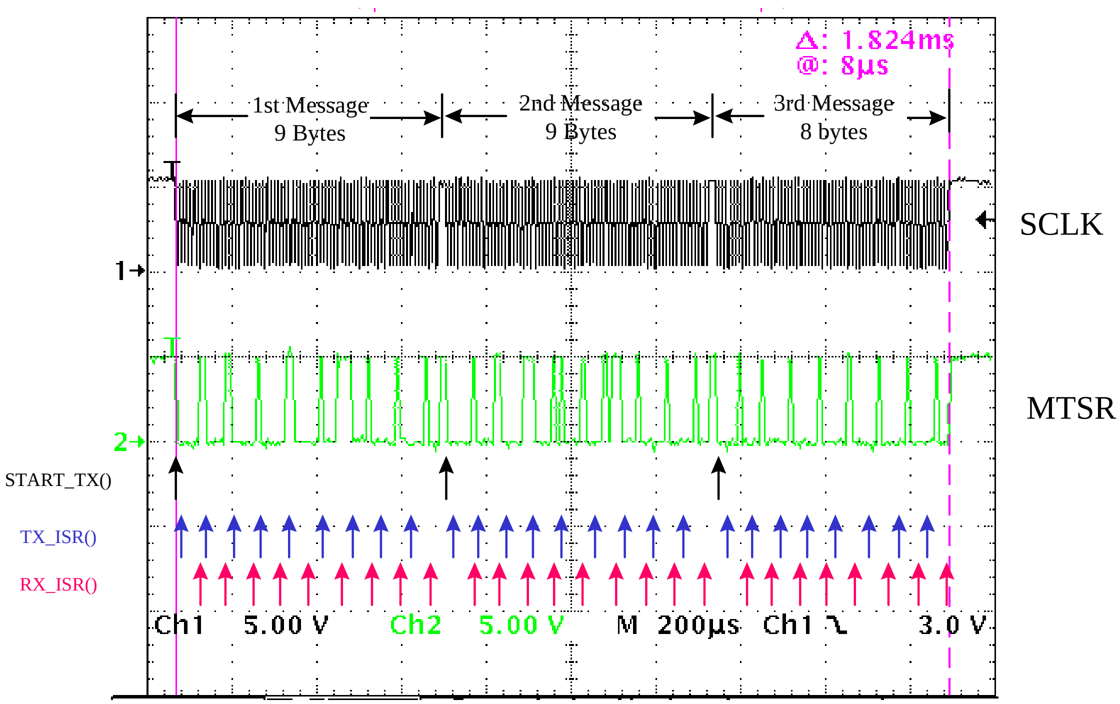
  Defines constants used by the program.

- sys_init.c

  The file contains the code that initializes the SSC module of the ’C167 for operation as the SPI master for the ’F240. The receive and transmit interrupts are enabled for normal interrupt processing. The port 3 pins P3.8 (MRST), P3.9 (MTSR) and P3.13 (SCLK) are configured for their alternate functions as SSC pins, with SCLK and MTSR set as outputs, and MRST set as an input. The operating mode of the SSC is set for 8-bits in master mode with transmissions beginning on the falling edge of SCLK with no delay, MSB first.

Figure 11 shows the actions of the master, including the clock and data signals generated by the master code.

Figure 11. ’C167 Master Event Time Line for Three Message Sequence



The main task of the program is to initiate message sequences for each of the three message types: (1) send new parameters and receive status update; (2) send new parameters; (3) receive status update. Within each sequence, the following actions are performed.

1) Start transmission sequence

2) Service transmit interrupt, refilling the transmit buffer register (performed once per byte).

3) Service receive interrupt, storing data from the receive buffer register (performed once per byte).

Figure 11 shows these three actions at their approximate execution time.

When the message from the Master is shorter than the Slave's message, send "08" (Checksum Request) until all bytes have been sent by the Slave. The Master knows that all bytes have been sent when it receives "FF" (ACK).

When messages from Slave are shorter than messages from the Master, the Slave will send "FF" until the Master message has been received completely. The Slave then verifies the checksum of the received message and sends “00” (Checksum Match) if the checksum matches, or “04” (Invalid Checksum) if the checksum does not match. The Master sends "08" three times to get this result.

Each byte in a parameter update message sent by the master is defined as follows:

Byte	Description
1	Message type
2	Message length, which is 3+*n* bytes long, where *n* is the number of parameters
3 to *n*+2	*n* data parameters
n+3	Checksum of bytes 1 through *n* + 2
n+4	Checksum request
n+5	Checksum request
n+6	Checksum request

Each byte in a status update message received by the master is defined as follows:

Byte	Description
1 and 2	Dummy byte, value is ignored
3 to *m*+2	*m* status bytes
m+3	Checksum status byte, 0x00, indicates slave received all master data bytes correctly
m+4	Slave acknowledge, 0xff, indicates slave has sent all status bytes

For a status only data transfer from the slave to the master, each byte in the message sent by the master is defined as follows:

Byte	Description
1	Message type
2 to *m*+3	checksum request byte, 0x08 (*m* is the # of status bytes)

In the case of parameter update only is transferred, each byte in the message received by the master is defined as follows:

Byte	Description
1 to *n*+4	dummy bytes, value is ignored (*n* is # of parameters)
n+5	checksum status byte, 0x00, indicates slave received all master data bytes correctly
n+6	slave acknowledge, 0xff, indicates slave has sent all status bytes

Compile, Assemble and Link Options

The C166 compiler options are as follows:

- ❑ Command Line Options String:

  SB CD DB M167 WL(3) DF(MCB167)

- ❑ Listings:

  Warnings - Level 3.

- ❑ Object:
  - Include debug information
  - Optimization: level 6
  - Emphasis: favor fast code
- ❑ Memory:
  - Small memory model
  - Initialize variables
  - Save DPP on interrupt entry
  - Alias checking on pointer access
  - User stack accessed with DPP2
  - Far threshold: 6
  - Default location: near

The A166 assembler options are as follows:

- ❑ Enable macro processor
- ❑ Define 80C166 SFR's
- ❑ Set SMALL

The L166 linker options are as follows:

- Linking:
  - Generate Interrupt Vector Table
  - Warnings: level 2
  - Command Line Options String:

    IX(NOLIBRARIES) CL(NCODE (0H - 0EFFFH) /* no on-chip RAM here! */,NDATA (10000H - 13FFFH),NCONST (0 - 03FFFH),ICODE (0H - 0EFFFH) /* no on-chip RAM here! */,NDATA0 (0F600H - 0F700H) /* on-chip here */) &

    SE(?C_INITSEC (200H) /* must be somewhere in ROM */) &

    RE(8H-0BH, 88H - 8BH /* for MCB167 */) &
  - Object Debug Information:

    Include comments, line numbers, public/local symbols, and type information.
- Sections:

  Section 1: ?C_INITSEC (200H)
- Location:

  Reserve 1: 08H-0BH, 88H-8BH
- Classes:
  - Class 1: NCODE (0H - 0EFFFH)
  - Class 1: NDATA (10000H - 13FFFH)
  - Class 1: NCONST (0H - 03FFFH)
  - Class 1: ICODE (0H - 0EFFFH)
  - Class 5: NDATA0 (0F600H - 0F700H)

The dScope debugger options are as follows:

- dScope command file: MCB167.ini

Running the Programs

Each program was downloaded to the target processor from the host PC using the standard debugger/emulation tool included with the respective development boards.

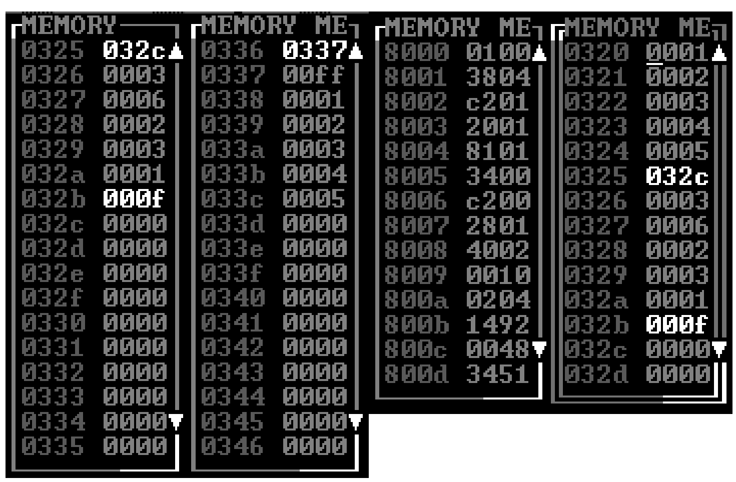
Implementing Point-to-Point Communication between Two ’F240 EVMs

The following procedure describes how to implement point-to-point communication between two ’F240 EVMs:

1) While both boards are powered down, make connections as described in Table 6.
2) Apply power to both boards.
3) Start the slave ’F240 debugger and load object file/configure debugger by executing a “take” command on file spi1.tak.
4) Start the master ’F240 debugger and load object file/configure debugger by executing a “take” command on file spi1.tak.
5) Run slave code by executing a “run” command in the slave ’F240 debugger.
6) Run master code by executing a “run” command in the master ’F240 debugger.
7) Wait several seconds for the program to execute.
8) Stop both programs in their respective debuggers.

The program runs correctly when the contents of the ’F240 master DARAM block 1 and external memory at 0x8000 are as shown in Figure 12. From left to right, the memory windows display DATA_IN (0x325), DATA_OUT (0x336), STATUS_INFO (0x320), and PARAMS (0x326).

Figure 12. Memory Image of ’F240 Master after Successful Program Run



Implementing Point-to-Point Communication between the ’F240 and ’C167

The following procedure describes how to implement point-to-point communication between the ’F240 and ’C167:

1) While both boards are powered down, make connections as described in Table 7.
2) Apply power to both boards.
3) Start the ’F240 debugger and load object file/configure debugger by executing a “take” command on file spi1.tak.
4) Start the ’C167 debugger.
5) Run slave code by executing a “run” command in the ’F240 debugger.
6) Run master code by executing a “go” command in the ’C167 debugger.
7) Wait several seconds for the program to execute.
8) Stop both programs in their respective debuggers.

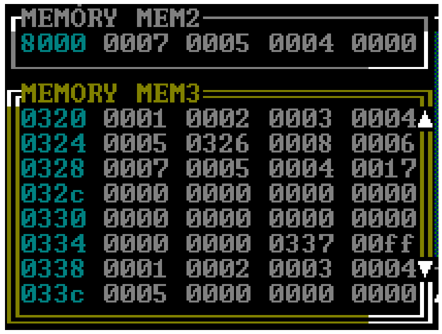
The program runs correctly when the ’C167 variable, *ERROR_TYPE,* equals 0x00, and the status buffer, *STATUS_BUF*, contains the values shown in Table 9.

Table 9. Expected Results for All Three Message Types

Send PARAMs and Receive STATUS		**Send PARAMs**		**Receive STATUS**	
PARAM_BUF	**STATUS_BUF**	**PARAM_BUF**	**STATUS_BUF**	**PARAM_BUF**	**STATUS_BUF**
0x03	0x00	0x01	0xff	0x2	0xff
0x06	0xff	0x06	0xff	0x08	0xff
0x02	0x05	0x07	0xff	0x08	0x05
0x03	0x04	0x05	0xff	0x08	0x04
0x01	0x03	0x04	0xff	0x08	0x03
0x0f	0x02	0x17	0xff	0x08	0x02
0x08	0x01	0x08	0xff	0x08	0x01
0x08	0x00	0x08	0x00	0x08	0xff
0x08	0xff	0x08	0xff		

On the slave side, the contents of DARAM block 1 and external memory at 0x8000 will be as shown in Figure 13.

Figure 13. Memory Image of ’F240 Slave after Successful Program Run



Appendix A. FAQ Summary of SPI Features

This appendix offers a summary of SPI features in FAQ (frequently asked questions) format. The summary reinforces important topics from the previous sections describing the SPI, the interface between the CPU, and the interrupt structure.

The FAQ consists of the following questions (the answers to each are found on the following page):

1) Name the four (4) signal pins used by the SPI.
2) The SPI can transmit data characters that are __ to __ bits in length.
3) Which bit is transmitted first, LSB or MSB?
4) Who originates the transmission?
5) What is the function of the SPISTE pin when in slave mode? In master mode?
6) Which feature of the SPI would be used only during debug using an emulator?

Name the Four (4) Signal Pins Used by the SPI.

1) SPISIMO (SPI slave in, master out)
2) SPISOMI (SPI slave out, master in)
3) SPICLK (SPI clock)
4) SPISTE (SPI slave transmit enable)

The SPI can transmit data characters that are __ to __ bits in length.

One to eight

Which bit is transmitted first, LSB or MSB? Who originates the transmission?

The Master and Slave's MSBs are both transmitted first.

The Master originates the transmission by writing to the SPIDAT register. If the Slave has data to transmit, it must already be loaded into its SPIDAT register before the SPICLK is received from the Master.

What is the function of the SPISTE pin when in slave mode? In master mode?

When in slave mode, the SPISTE pin can operate as either a general purpose I/O pin or as the slave transmit enable. The mode is determined by the value of the SPISTE FUNCTION bit (SPIPC1.5). When the SPISTE pin operates as the slave transmit enable, an active low signal on the SPISTE pin enables the SPI to transmit. Conversely, an active high signal on the SPISTE pin disables transmission.

When in master mode, the SPISTE pin always operates as a general purpose I/O pin. The value of the SPISTE FUNCTION bit (SPIPC1.5) is ignored when the SPI is in master mode.

Which feature of the SPI is used only during debug using an emulator?

The emulator suspend enable bit provides the option of either completing the current SPI transmit/receive sequence or freezing the state of the SPI at the point when the emulator suspends device operation.

Appendix B. ’C167 Master Mode Program Files

SPI.C

```
/*********************************************************************
* File Name:  spi.c  (File revision 0.1)                              *
* Project:    SPI Point-to-Point Application Report                   *
* Originator: J.Crankshaw    (Texas Instruments)                      *
*                                                                     *
* Target Sys: Keil MCB167                                             *
*                                                                     *
* Description: main program for SPI master.                           *
*                                                                     *
* Status:  runs ok                                                    *
*                                                                     *
* Last Update:  23 Jan 98                                             *
* ____________________________________________________________________*
* Date of Mod |         DESCRIPTION                                   *
* ------------|-------------------------------------------------------*
*       |                                                             *
*       |                                                             *
*       |                                                             *
*********************************************************************/

#include <reg167.h>               /* special function register 80C167  */
#include <intrins.h>
#include "spi.h"       /* definitions */


extern void SYS_INIT (void);  /* use external routine for system intialization
*/
void START_TX (void);     /* prototype def */

unsigned int idata ERROR_TYPE;   /* MSG_TYPE that caused an eror condition */
unsigned int idata MSG_QUEUED;   /* status of msg in progress */
unsigned int idata TX_TOTAL;  /* the number of bytes to transfer */
unsigned int idata TX_BYTE_NO;   /* holds the number of bytes transferred */
unsigned int idata RX_BYTE_NO;   /* holds the number of bytes transferred */
unsigned int idata STATUS_BUF [4 + MSG_LENGTH];   /* buffer for status*/
unsigned int idata PARAM_BUF [4 + MSG_LENGTH]; /* buffer for new parameters */
```

```
void main (void)
{
      int i,j,MSG_TYPE;
                                 /* initialize the serial interface        */
      #ifndef MCB167          /* do not initialize if you use Monitor-166    */
        P3  |= 0x0400;        /* SET PORT 3.10 OUTPUT LATCH (TXD)            */
        DP3 |= 0x0400;        /* SET PORT 3.10 DIRECTION CONTROL (TXD OUTPUT)*/
        DP3 &= 0xF7FF;        /* RESET PORT 3.11 DIRECTION CONTROL(RXD INPUT)*/
        S0TIC = 0x80;         /* SET TRANSMIT INTERRUPT FLAG                 */
        S0RIC = 0x00;         /* DELETE RECEIVE INTERRUPT FLAG               */
        S0BG  = 0x40;         /* SET BAUDRATE TO 9600 BAUD                   */
        S0CON = 0x8011;       /* SET SERIAL MODE                             */
      #endif

      j = 0;

      for ( i = 1; i <=(MSG_LENGTH + 2); i++ )
      {
         STATUS_BUF[i] = 0;
      }

      TX_TOTAL = 0x00; /* reset no. of bytes to transmit to zero */
      TX_BYTE_NO = 0x00;  /* reset no. of bytes transferred to zero */
      RX_BYTE_NO = 0x00;  /* reset no. of bytes received to zero */

      SYS_INIT ();     /* initialize all devices needed for the demo program */

      for (MSG_TYPE = 0; MSG_TYPE <= 2; MSG_TYPE++ )    /* do 3 times */
      {
         if (MSG_TYPE == 0)
         {
            PARAM_BUF[0] = 0x03;
            PARAM_BUF[1] = 0x06;
            PARAM_BUF[2] = 0x02;
            PARAM_BUF[3] = 0x03;
            PARAM_BUF[4] = 0x01;
            PARAM_BUF[5] = 0x0F;
            PARAM_BUF[6] = 0x08;
```

```
        PARAM_BUF[7] = 0x08;
        PARAM_BUF[8] = 0x08;
        TX_TOTAL = 9;
    }
    else if (MSG_TYPE == 1)
    {
        PARAM_BUF[0] = 0x01;
        PARAM_BUF[1] = 0x06;
        PARAM_BUF[2] = 0x07;
        PARAM_BUF[3] = 0x05;
        PARAM_BUF[4] = 0x04;
        PARAM_BUF[5] = 0x17;
        PARAM_BUF[6] = 0x08;
        PARAM_BUF[7] = 0x08;
        PARAM_BUF[8] = 0x08;
        TX_TOTAL = 9;
    }
    else if (MSG_TYPE == 2)
    {
        PARAM_BUF[0] = 0x02;
        PARAM_BUF[1] = 0x08;
        PARAM_BUF[2] = 0x08;
        PARAM_BUF[3] = 0x08;
        PARAM_BUF[4] = 0x08;
        PARAM_BUF[5] = 0x08;
        PARAM_BUF[6] = 0x08;
        PARAM_BUF[7] = 0x08;
        TX_TOTAL = 8;
    }

    MSG_QUEUED = 1;

    START_TX (); /* start SSC msg transmit/receive sequence */

    while ( MSG_QUEUED )   /* loop until msg has been sent/received */
    {
        j = 0;

        for ( i = 1; i <=10; i++ )
```

```
                j = j +1;
            }

            if ((STATUS_BUF[TX_TOTAL-2] != 0) & (MSG_TYPE < 2))
            {
                ERROR_TYPE = MSG_TYPE;
                MSG_TYPE = 3;
            }
        }
        while ( 1 )  /* testing done, loop forever */
        {
            j = 0;

            for ( i = 1; i <=10; i++ )
                j = j +1;
        }
}

void TX_ISR (void) interrupt txintno
{
        if ((TX_BYTE_NO >= 0) & (TX_BYTE_NO < TX_TOTAL ))
        {
            _bfld_ ( P3, 0x0001, 0x0000 );   /* switch Slave Transmit Ena low */

            SSCTB = PARAM_BUF[TX_BYTE_NO];  /* Begin TX */
            TX_BYTE_NO = TX_BYTE_NO + 1;  /* increment TX byte counter */

            _bfld_ ( P3, 0x0001, 0x0001 );   /* switch Slave Transmit Ena high */
        }
        else if (TX_BYTE_NO == TX_TOTAL)
        {
            TX_BYTE_NO = 0;  /* reset TX byte counter */
        }
}

void START_TX ( void )
{
        SSCTB = PARAM_BUF[TX_BYTE_NO];  /* Begin TX */
        TX_BYTE_NO = TX_BYTE_NO + 1;  /* increment TX byte counter */
```

```
}

void RX_ISR (void) interrupt rxintno
{

       if (RX_BYTE_NO < TX_TOTAL - 1)
       {
          STATUS_BUF[RX_BYTE_NO] = SSCRB;
          RX_BYTE_NO = RX_BYTE_NO + 1;  /* increment RX byte counter */
       }
       else if (RX_BYTE_NO == TX_TOTAL -1)
       {
          STATUS_BUF[RX_BYTE_NO] = SSCRB;
          MSG_QUEUED = 0;
          RX_BYTE_NO = 0;
       }

}
```

SPI.H

```
/************************************************************************
* File Name:  spi.h  (File revision 0.1)                                *
* Project:    SPI Point-to-Point Application Report                     *
* Originator: J.Crankshaw   (Texas Instruments)                         *
*                                                                       *
* Target Sys: Keil MCB167                                               *
*                                                                       *
* Description: header file for SPI master program.                      *
*                                                                       *
* Status:  runs ok                                                      *
*                                                                       *
* Last Update:  23 Jan 98                                               *
* ______________________________________________________________________*
* Date of Mod |     DESCRIPTION                                         *
* ------------|---------------------------------------------------------*
*       |                                                               *
*       |                                                               *
*       |                                                               *
```

```
*************************************************************************/

#define rxintno 0x2E   /* hardware interrupt # of SSC receive */
#define txintno 0x2D   /* hardware interrupt # of SSC transmit */
#define SSC_TXINT 0x40 + 4 * 0x08 + 0x01
#define SSC_RXINT 0x40 + 4 * 0x07 + 0x02
#define MSG_LENGTH 6   /* # of bytes transferred in base master message */
#define ENABLE_SSC 0x8000  /* set SSC enable bit, SSCCON.15 */
#define MASTER 0x4000      /* set SSC Master mode bit, SSCCON.14 */
#define BIT8_FNDLY_MSB 0x0057    /* set SSC MSB first, falling edge no delay, 8
bits data ea. TX/RX */
#define fCPU 20000000      /* system clock frequency */
#define BAUDssc 115200     /* SSC baud rate, or ~8.5us bit clock */
#define RELOAD_VALUE (fCPU/(2*BAUDssc)) - 1  /* 0x55 is the reload value */
```

SYS_INIT.C

```
/*************************************************************************
* File Name:  sys_init.c  (File revision 0.1)                            *
* Project:    SPI Point-to-Point Application Report                      *
* Originator: J.Crankshaw    (Texas Instruments)                         *
*                                                                        *
* Target Sys: Keil MCB167                                                *
*                                                                        *
* Description:'C167 CPU initialization function.                         *
*                                                                        *
* Status:  runs ok                                                       *
*                                                                        *
* Last Update:  23 Jan 98                                                *
* _______________________________________________________________________*
* Date of Mod |     DESCRIPTION                                          *
* ------------|----------------------------------------------------------*
*       |                                                                *
*       |                                                                *
*       |                                                                *
*************************************************************************/

#include <reg167.h>       /* register definitions */
#include <intrins.h>
```

```
#include "spi.h"       /* int # definitions */

void SYS_INIT (void)
{
      _bfld_ ( SSCRIC, 0xFF, SSC_RXINT ); /* set SSC rx interrupt priority */
                                           /* & group level */
      _bfld_ ( SSCTIC, 0xFF, SSC_TXINT ); /* set SSC tx interrupt priority */
                                           /* & group level */

      SSCCON = 0;                   /* reset SSC */
      SSCBR = RELOAD_VALUE;         /* constant defined in spi.h */

      _bfld_ ( P3, 0x2300, 0x2300 );   /* set P3.8 (MRST), P3.9 (MTSR) */
                                       /* and P3.13 (SCLK) */
      _bfld_ ( DP3, 0x2301, 0x2201 );  /* switch P3.0 to output mode */
                    /* switch MRST to input; MTSR and SCLK to output mode */
      SSCCON = ENABLE_SSC | MASTER | BIT8_FNDLY_MSB;
}
```

Appendix C. ’F240 Master Mode Program Files

Example ’C2xx Debugger Command Line Options

The following example command line options invoke the ’C2xx debugger with XDS510 and XDS511:

Emu2xxwm.exe -n cpu_a -f board.dat -t’F240evm.cmd -p 240

Where

-n cpu_a provides the name of the processor

-f board.dat describes the devices on the scan path, in this case, just the ’F240.

-t ’F240evm.cmd is a user-customized version of the emuinit.cmd file included with the debugger.

-p 240 specifies the port address of the XDS510 controller card. This option is determined by your specific hardware setup. (See the *XDS51x Emulator Installation Guide* for more details.)

Emulator Initialization Command File – ’F240evm.cmd

```
echo’F240evm.CMD for’F240 EVM

;Reset Memory Map
mr

;DATA MEMORY
ma 0x00000,1,0x0060,ram   ;MMRs
ma 0x00060,1,0x0020,ram   ;On-Chip RAM B2
ma 0x00200,1,0x0100,ram   ;On-Chip RAM B0 if CNF=0
ma 0x00300,1,0x0100,ram   ;On-Chip RAM B1
ma 0x07010,1,0x0010,ioport   ;Peripheral - System Config & Control
ma 0x07020,1,0x0010,ioport   ;Peripheral - WDT / RTI
ma 0x07030,1,0x0010,ioport   ;Peripheral - ADC
ma 0x07040,1,0x0010,ioport   ;Peripheral - SPI
ma 0x07050,1,0x0010,ioport   ;Peripheral - SCI
ma 0x07070,1,0x0010,ioport   ;Peripheral - Ext Ints
ma 0x07090,1,0x0010,ioport   ;Peripheral - Digital I/O
ma 0x07400,1,0x000D,ioport   ;Peripheral - Event Mgr GPT
ma 0x07411,1,0x000C,ioport   ;Peripheral - Event Mgr CMP,PWM
ma 0x07420,1,0x0007,ioport   ;Peripheral - Event Mgr CAP,QEP
```

```
ma 0x0742C,1,0x0009,ioport   ;Peripheral - Event Mgr Int Cntl

; for EVM, external RAM is in memory map
ma 0x08000,1,0x08000,ram  ;Ext SRAM

;PROGRAM MEMORY
ma 0x00000,0,0x04000,RAM   ;Internal Program memory - FLASH
ma 0x04000,0,0x0BE00,RAM   ;External Program memory - SRAM
ma 0x0FE00,0,0x00100,RAM   ;Available if CNF=1 i.e. B0

;I/O MEMORY
;~~~~~~~~~~
ma 0x0000,2,0x0008,WOM     ;I/O Memory Mapped DAC Registers
ma 0x0008,2,0x0004,ROM     ;I/O Memory Mapped DIP Switches
ma 0x000C,2,0x0004,WOM     ;I/O Memory Mapped LEDs

mem  0x0200
mem1 0x0300
mem2 0x0060

wa (ST0&(0x03ff))>>9,INTM,d   ;INTM Int Mode Bit
wa (ST0&0x0f),DP,x

; SPI application note programs
;----------------------------------------------------
take c:\dspcode\x240\f240evm\SPI_app\master\spi1.tak  ; final version - working

echo  ‘F240evm.CMD HAS BEEN LOADED
```

SPI.ASM

```
 ;*******************************************************************
 ;*
 ;*    DESCRIPTION:  SPI Code Example for Point-to-Point Communication
 ;*
 ;*    AUTHOR:  Jeff Stafford
 ;*
;*******************************************************************
;---------------------------------------------------------------------
; Debug directives
;---------------------------------------------------------------------
             .def   STATUS_INFO            ;General purpose registers.
             .def   DATA_IN
             .def   DATA_OUT

             .include "..\..\f240regs.h"

; 1st Data Page of peripheral registers (7000h/80h)
DP_PF1                 .set 224

BAD_CHKSUM             .set   004h
DATA_IN_LEN            .set   010h
DATA_OUT_LEN           .set   010h
STATUS_INFO_BYTES      .set   005h
CONTEXT_MEM_PTR_BYTES  .set   020h
SLAVE_SEND             .set   0FEh

                    .bss   CONTEXT_MEM_PTR, CONTEXT_MEM_PTR_BYTES
                    .bss   STATUS_INFO, STATUS_INFO_BYTES
                    .bss   DATA_IN_PTR,1
                    .bss   DATA_IN, DATA_IN_LEN
                    .bss   DATA_OUT_PTR,1
                    .bss   DATA_OUT, DATA_OUT_LEN

TEST_MSG_BYTES      .set   20h
                    .bss   TEST_MSG_PTR,1
                    .bss   TEST_MSG, TEST_MSG_BYTES
                    .bss   TEST_MSG_END,1
```

```
                    .bss   NEW_BYTE_FLAG, 1

RCV_MSG_TYPE        .set   DATA_IN
RCV_BYTE_NO         .set   DATA_IN + 1
RCV_DATA_START      .set   DATA_IN + 2
PARAMS              .usect "VARS", 020h

                    .bss   VAR1, 1      ;Scratchpad

                    .data
ST0_TEMP            .word 0
ST1_TEMP            .word 0


KICK_DOG      .macro
          LDP    #00E0H
          SPLK   #055H, WDKEY
          SPLK   #0AAH, WDKEY
          LDP    #DATA_IN_PTR
                 .endm

          .text

START:    CLRC   SXM        ; Clear Sign Extension Mode
          CLRC   OVM        ; Reset Overflow Mode

      ; Set Data Page pointer to  page 1 of the peripheral frame
          LDP #DP_PF1         ; Page DP_PF1 includes WET through EINT frames

      ; initialize WDT registers
          SPLK #06Fh, WDCR    ; clear WDFLAG, Disable WDT, set WDT for 1 second
                              ; overflow (max)
          SPLK #07h, RTICR    ; clear RTI Flag, set RTI for 1 second
                              ; overflow (max)

      ; EVM 10MHz oscillator settings.  (XTAL2 open, OSCBYP_=GND)
          SPLK #00B1h,CKCR1   ; CLKIN(OSC)=10MHz, Mult by 2, Div by 1.
          SPLK   #00C3h,CKCR0 ; CLKMD=PLL Enable,SYSCLK=CPUCLK/2,
```

```
; Clear reset flag bits in SYSSR (PORRST, PLLRST, ILLRST, SWRST, WDRST)
    LACL   SYSSR     ; ACCL <= SYSSR
    AND #00FFh       ; Clear upper 8 bits of SYSSR
    SACL   SYSSR     ; Load new value into SYSSR

; initialize B1 RAM to zero's.
    LAR  AR1,#B1_SADDR  ; AR1 <= B1 start address
    MAR  *,AR1          ; use B1 start address for next indirect
    ZAC                 ; ACC <= 0
RPT #(CONTEXT_MEM_PTR_BYTES+STATUS_INFO_BYTES+DATA_OUT_LEN+DATA_IN_LEN+2)
; set repeat counter for sizeof(.bss)-1 loops
    SACL *+             ; write zeros to B1 RAM

; initialize B2 RAM to zero's.
    LAR  AR1,#B2_SADDR  ; AR1 <= B2 start address
    MAR  *,AR1          ; use B2 start address for next indirect
    ZAC                 ; ACC <= 0
    RPT  #1fh           ; set repeat counter for 1fh+1=20h or 32 loops
    SACL *+             ; write zeros to B2 RAM

; initialize STATUS_INFO
    LAR    AR1,#STATUS_INFO    ; AR1 <= STATUS_INFO start address
    MAR    *,AR1               ;
    SPLK   #01,*+          ; STAT1 = 1
    SPLK   #02,*+          ; STAT2 = 2
    SPLK   #03,*+          ; STAT3 = 3
    SPLK   #04,*+          ; STAT4 = 4
    SPLK   #05,*+          ; STAT5 = 5

; initialize TEST_MSG
    LAR    AR1,#TEST_MSG   ; AR1 <= TEST_MSG start address
    MAR    *,AR1
    SPLK   #03h,*+
    SPLK   #06h,*+
    SPLK   #02h,*+
    SPLK   #03h,*+
    SPLK   #01h,*+
    SPLK   #0Fh,*+
    MAR    *-
```

```
        LDP     #TEST_MSG_END
        SAR     AR1, TEST_MSG_END
        LAR     AR1, #TEST_MSG_PTR
        SPLK    #TEST_MSG, *
        LDP     #NEW_BYTE_FLAG
        SPLK    #1, NEW_BYTE_FLAG

    ;Initialize DSP for interrupts
        LAR     AR6,#IMR       ;
        LAR     AR7,#IFR       ;
        MAR     *,AR6
        LACL    #011h           ;Enable INT1 also! XINT1 high pri
        SACL    *,AR7          ; Enable interrupt 5 only (SPI low priority)
        LACL    *          ; Clear IFR by reading and
        SACL    *,AR6      ; writing contents back into itself

    ; call SPI initialization routine
        CALL    INIT_SPI

        LDP     #DATA_IN_PTR

    ;Initialize RX and TX buffers
        SPLK    #DATA_IN, DATA_IN_PTR  ;Reset RX buffer pointer
        SPLK    #DATA_OUT, DATA_OUT_PTR;Reset TX buffer pointer
        SPLK    #0FFH, DATA_OUT     ;Initialize default TX byte = ACKnowledge

    ;Initialize and enable XINT1
        LDP     #DP_PF1
        SPLK    #01h, XINT1CR;Enable XINT1, high pri, falling edge

    ;Enable DSP interrupts
        CLRC    INTM

MAIN:
        LDP     #NEW_BYTE_FLAG
        MAR     *, AR5
        LAR     AR5, NEW_BYTE_FLAG  ;UPDATE AR5
        BANZ    MAIN,*
```

```
RD_MSG:   ;Check if "send status info" bit is set in 1st rcvd byte (msg type).
          LDP    #NEW_BYTE_FLAG
          SPLK   #1, NEW_BYTE_FLAG   ;UPDATE FLAG, NEW BYTE HAS BEEN READ

RESET_RCV_BFR:
          B      MAIN

 ;*********************************************************************
 ;*
 ;** INIT_SPI
 ;*
 ;*  REGISTER USAGE:   assumes DP = 224 (addresses 0x7000 - 0x707f
 ;*
 ;*  DESCRIPTION:   SPI initialization subroutine.  This SR initializes
 ;*              the SPI for data stream transfer to a master SPI.
 ;*              The '240 SPI is configured for 8-bit transfers as a slave.
 ;*
 ;*********************************************************************
INIT_SPI:
       ; initialize SPI in slave mode
          LDP    #DP_PF1
          SPLK   #008Fh,SPICCR; Reset SPI by writing 1 to SWRST
          SPLK   #0004h,SPICTL; Disable ints & TALK, normal clock, MASTER mode

          SPLK   #007Fh, SPIBRR   ;Slowest baud rate for testing code

          SPLK   #0002h,SPIPC1
          SPLK   #0022h,SPIPC2; Set SIMO & SOMI functions to serial I/O
          SPLK   #0040h,SPIPRI; Set SPI interrupt to low priority.
                              ; For emulation purposes, allow the SPI
                              ; to continue after an XDS suspension.
                              ; HAS NO EFFECT ON THE ACTUAL DEVICE.
          SPLK   #0000h,SPISTS; Clear the SPI interrupt status bits

       ; falling edge with No delay
          SPLK   #0047h,SPICCR; Release SWRST, clock polarity 1, 8 bits
          SPLK   #0017h,SPICTL; Enable TALK, ena SPI int, CLK ph 0, MASTER mode

          RET                 ; Return to MAIN routine.
```

```
;********************************************************************
;*
;** XINT1_ISR
;*
;*  REGISTER USAGE:
;*
;*  DESCRIPTION:   XINT1 interrupt service routine.
;*
;*    History:
;*       12/15/97  created
;********************************************************************

XINT1_ISR:
          SST    #0, ST0_TEMP ;Auto page-0 DP
          SST    #1, ST1_TEMP
          LDP    #CONTEXT_MEM_PTR
          SACL   CONTEXT_MEM_PTR
          SACH   CONTEXT_MEM_PTR+1
          SAR    AR0, CONTEXT_MEM_PTR+2
          SAR    AR6, CONTEXT_MEM_PTR+3
          LDP    #0
          LACL   IMR
          LDP    #CONTEXT_MEM_PTR
          SACL   CONTEXT_MEM_PTR+4

SEND_TEST_MSG
       ;Send data
          LDP    #TEST_MSG
          LAR    AR0, TEST_MSG_END   ;END OF MEM
          LAR    AR6, TEST_MSG_PTR
          MAR    *, AR6
          LACL   *+
          SAR    AR6, TEST_MSG_PTR
          LDP    #DP_PF1
          SACL   SPIDAT

          CMPR   2                ;Compare AR6 and AR0
          BCND   DEBOUNCE_SW, NTC       ;PTR WITHIN RANGE?
```

```
          LDP    #TEST_MSG
          LAR    AR6, #TEST_MSG      ;RESET IT IF NOT
          SAR    AR6, TEST_MSG_PTR

DEBOUNCE_SW
          LAR    AR0, #0100h
DELAY2    LAR    AR6, #03FFFh
          MAR    *, AR6
DELAY1    BANZ   DELAY1
          MAR    *, AR0
          BANZ   DELAY2

          LDP    #DP_PF1
WAIT      BIT    XINT1CR, BIT6
          BCND   HIGH, TC
          B      WAIT         ;Wait for XINT1 pin to goto a logic 0
HIGH      LDP    #0           ;Debounce switch by clearing pending ints
          LACL   #01
          SACL   IFR
          LDP    #DP_PF1
          SPLK   #001h, XINT1CR   ;Clear transition detect

          LDP    #CONTEXT_MEM_PTR
          LACL   CONTEXT_MEM_PTR+4
          LDP    #0
          SACL   IMR
          LDP    #CONTEXT_MEM_PTR
          LAR    AR6, CONTEXT_MEM_PTR+3
          LAR    AR0, CONTEXT_MEM_PTR+2
          LACC   CONTEXT_MEM_PTR+1,16
          ADDS   CONTEXT_MEM_PTR
          LDP    #0
          LST    #1, ST1_TEMP
          LST    #0, ST0_TEMP
          CLRC   INTM

          RET

;*******************************************************************
```

```
 ;*
 ;** SPI_ISR
 ;*
 ;*  REGISTER USAGE:   ST0, ST1, ACC, AR6, AR7
 ;*
 ;*  DESCRIPTION:   SPI interrupt service routine.
 ;*
;*******************************************************************

SPI_ISR:
          SST    #0, ST0_TEMP ;Auto page-0 DP
          SST    #1, ST1_TEMP
          LDP    #CONTEXT_MEM_PTR
          SACL   CONTEXT_MEM_PTR
          SACH   CONTEXT_MEM_PTR+1
          SAR    AR7, CONTEXT_MEM_PTR+2
          SAR    AR6, CONTEXT_MEM_PTR+3
          LDP    #0
          LACL   IMR
          LDP    #CONTEXT_MEM_PTR
          SACL   CONTEXT_MEM_PTR+4
OVER_RUN:
          LDP #DP_PF1            ; Page DP_PF1 includes SPI
          BIT    SPISTS, 8       ;Overrun flag (SPISTS.7) set?
          BCND   CLEAR_FLAG, TC  ;If set, clear & return

READ_SPI:
          LDP    #DP_PF1         ; Page DP_PF1 includes SPI
          LACC   SPIBUF          ;Load rcvd byte into ACC

          LDP    #DATA_IN_PTR    ; Set data page pointer to B1 page
          LAR    AR7, DATA_IN_PTR ; update RX ptr
          MAR    *, AR7
          SACL   *+
          SAR    AR7, DATA_IN_PTR ;Set RX pointer to next entry

;****** DEBUG CODE ******
          LDP    #NEW_BYTE_FLAG
          SPLK   #0, NEW_BYTE_FLAG
```

```
;***********************

SPI_DONE:
          LDP     #CONTEXT_MEM_PTR
          LACL    CONTEXT_MEM_PTR+4
          LDP     #0
          SACL    IMR
          LDP     #CONTEXT_MEM_PTR
          LAR     AR6, CONTEXT_MEM_PTR+3
          LAR     AR7, CONTEXT_MEM_PTR+2
          LACC    CONTEXT_MEM_PTR+1,16
          ADDS    CONTEXT_MEM_PTR
          LDP     #0
          LST     #1, ST1_TEMP
          LST     #0, ST0_TEMP
          CLRC    INTM

          RET

CLEAR_FLAG:
          SPLK    #0H, SPISTS
          B       SPI_DONE



 ;*******************************************************************
 ;*
 ;** Interrupt Vectors
 ;*
 ;*  DESCRIPTION:   Used by linker to place this section at the reset vector
location.
 ;*
 ;*******************************************************************

          .sect   "vectors"
          B   START        ;reset
          B   XINT1_ISR    ;Int level 1, high priority ext int
          B   START        ;Int level 2 not used
          B   START        ;Int level 3 not used
```

```
          B   START          ;Int level 4 not used
          B   SPI_ISR        ;Int level 5, low priority SPI
```

SPI.CMD

```
MEMORY
{
      PAGE 0:    /* Program Memory Map for'F240EVM in MP mode */
        VECS:           org=0h,       len=40h        /* external */
        EXT_PROG:       org=40h,      len=0FDC0h     /* external */

      PAGE 1:    /* Data Memory Map for'F240EVM */
        B2:             org=60h ,     len=20h        /* internal DARAM */
        B0:             org=0200h,    len=100h       /* internal DARAM */
        B1:             org=0300h,    len=100h       /* internal DARAM */
        EXT_SRAM:       org=08000h,   len=08000h     /* external SRAM */
}

SECTIONS
{
        .text:  >       EXT_PROG      PAGE 0
        .data:  >       B2            PAGE 1
        .bss:   >       B1            PAGE 1
        vectors >       VECS          PAGE 0
        VARS    >       EXT_SRAM      PAGE 1
}
```

SPI1.TAK

```
cd c:\dspcode\x240\f240evm\spi_app

load spi.out

;Serial Peripheral Interface (SPI) Registers
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
wa *0x07040, SPICCR       ;SPI Config Control Reg
wa *0x07041, SPICTL       ;SPI Operation Control Reg
wa *0x07042, SPISTS       ;SPI Status Reg
wa *0x07044, SPIBRR       ;SPI Baud rate control reg
```

```
wa *0x07046, SPIEMU          ;SPI Emulation buffer reg
wa *0x07047, SPIBUF          ;SPI Serial Input buffer reg
wa *0x07049, SPIDAT          ;SPI Serial Data reg
wa *0x0704D, SPIPC1           ;SPI Port control reg1
wa *0x0704E, SPIPC2           ;SPI Port control reg2
wa *0x0704F, SPIPRI        ;SPI Priority control reg

wa *0x07070, XINT1CR       ;XINT1 Control Register
wa *0x0701A, SYSSR         ;System Status Register
wa (ST0&(0x03ff))>>9,INTM,x   ;INTM Int Mode Bit
wa (ST0&0x0f),DP,x
wa (ST0&0x0f)*128,Base,x
wa (ST0>>13),ARP,x

WA TEST_MSG,,x
WA *TEST_MSG_PTR,,x
WA *TEST_MSG_END,,x

WA DATA_IN,,x
WA DATA_IN_PTR,,x
WA *DATA_IN_PTR,,x

WA DATA_OUT,,x
WA DATA_OUT_PTR,,x
WA *DATA_OUT_PTR,,x

WA STATUS_INFO,,x
WA *STATUS_INFO,,x

WA *VAR1,,x
WA *NEW_BYTE_FLAG,,x

;ba XINT1_ISR
;ba SPI_ISR
ba RD_MSG
;BA VERIFY_CHECKSUM

MEM DATA_IN_PTR
MEM1 DATA_OUT_PTR
```

```
MEM2 TEST_MSG
MEM3 STATUS_INFO
bl
```

F240REGS.H

```
;***********************************************************************
; File Name:  f240regs.h
; Originator: Texas Instruments
;
; Description:   F240 Header file containing all peripheral register
;            declarations as well as other useful definitions.
;
; Last Updated:   27 May 1997
;
;***********************************************************************



;-----------------------------------------------------------------------
; On Chip Periperal Register Definitions (All registers mapped into data
; space unless otherwise noted)
;-----------------------------------------------------------------------

;C2xx Core Registers
;~~~~~~~~~~~~~~~~~~~~
IMR       .set   0004h        ;Interrupt Mask Register
GREG      .set   0005h        ;Global memory allocation Register
IFR       .set   0006h        ;Interrupt Flag Register

;System Module Registers
;~~~~~~~~~~~~~~~~~~~~~~~
SYSCR     .set   07018h    ;System Module Control Register
SYSSR     .set   0701Ah    ;System Module Status Register
SYSIVR .set   0701Eh    ;System Interrupt Vector Register

;Watch-Dog(WD) / Real Time Int(RTI) / Phase Lock Loop(PLL) Registers
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
RTICNTR   .set   07021h    ;RTI Counter Register
WDCNTR    .set   07023h    ;WD Counter Register
```

```
WDKEY      .set   07025h    ;WD Key Register
RTICR      .set   07027h    ;RTI Control Register
WDCR       .set   07029h    ;WD Control Register
CKCR0      .set   0702Bh    ;Clock Control Register 0
CKCR1      .set   0702Dh    ;Clock Control Register 1

;Analog-to-Digital Converter(ADC) registers
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
ADCTRL1    .set   07032h    ;ADC Control Register 1
ADCTRL2    .set   07034h    ;ADC Control Register 2
ADCFIFO1   .set   07036h    ;ADC Data Register FIFO1
ADCFIFO2   .set   07038h    ;ADC Data Register FIFO2

;Serial Peripheral Interface (SPI) Registers
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
SPICCR     .set   07040h    ;SPI Configuration Control Register
SPICTL     .set   07041h    ;SPI Operation Control Register
SPISTS     .set   07042h    ;SPI Status Register
SPIBRR     .set   07044h    ;SPI Baud Rate Register
SPIEMU     .set   07046h    ;SPI Emulation buffer Register
SPIBUF     .set   07047h    ;SPI Serial Input Buffer Register
SPIDAT     .set   07049h    ;SPI Serial Data Register
SPIPC1     .set   0704Dh    ;SPI Port Control Register 1
SPIPC2     .set   0704Eh    ;SPI Port Control Register 2
SPIPRI     .set   0704Fh    ;SPI Priority control Register

;Serial Communications Interface (SCI) Registers
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
SCICCR     .set   07050h    ;SCI Communication Control Register
SCICTL1    .set   07051h    ;SCI Control Register 1
SCIHBAUD   .set   07052h    ;SCI Baud Select register, high bits
SCILBAUD   .set   07053h    ;SCI Baud Select register, high bits
SCICTL2    .set   07054h    ;SCI Control Register 2
SCIRXST    .set   07055h    ;SCI Receive Status Register
SCIRXEMU   .set   07056h    ;SCI Emulation data buffer Register
SCIRXBUF   .set   07057h    ;SCI Receiver data buffer Register
SCITXBUF   .set   07059h    ;SCI Transmit data buffer Register
SCIPC2     .set   0705Eh    ;SCI Port Control Register 2
SCIPRI     .set   0705Fh    ;SCI Priority Control Register
```

```
;External Interrupt Registers
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~
XINT1CR   .set   07070h    ;Interrupt 1 Control Register
NMICR     .set   07072h    ;Non-maskable Interrupt Control Register
XINT2CR   .set   07078h    ;Interrupt 2 Control Register
XINT3CR   .set   0707Ah    ;Interrupt 3 Control Register

;Digital I/O
;~~~~~~~~~~~
OCRA      .set   07090h    ;Output Control Reg A
OCRB      .set   07092h    ;Output Control Reg B
PADATDIR  .set   07098h    ;I/O port A Data & Direction reg.
PBDATDIR  .set   0709Ah    ;I/O port B Data & Direction reg.
PCDATDIR  .set   0709Ch    ;I/O port C Data & Direction reg.

;General Purpose Timer Registers - Event Manager (EV)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
GPTCON    .set   7400h        ;General Purpose Timer Control Register
T1CNT     .set   7401h        ;GP Timer 1 Counter Register
T1CMPR    .set   7402h        ;GP Timer 1 Compare Register
T1PR      .set   7403h        ;GP Timer 1 Period Register
T1CON     .set   7404h        ;GP Timer 1 Control Register
T2CNT     .set   7405h        ;GP Timer 2 Counter Register
T2CMPR    .set   7406h        ;GP Timer 2 Compare Register
T2PR      .set   7407h        ;GP Timer 2 Period Register
T2CON     .set   7408h        ;GP Timer 2 Control Register
T3CNT     .set   7409h        ;GP Timer 3 Counter Register
T3CMPR    .set   740Ah        ;GP Timer 3 Compare Register
T3PR      .set   740Bh        ;GP Timer 3 Period Register
T3CON     .set   740Ch        ;GP Timer 3 Control Register

;Full & Simple Compare Unit Registers - Event Manager (EV)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
COMCON    .set   7411h        ;Compare Control Register
ACTR      .set   7413h        ;Full Compare Action Control Register
SACTR     .set   7414h        ;Simple Compare Action Control Register
DBTCON    .set   7415h        ;Dead-band Timer Control Register
CMPR1     .set   7417h        ;Full Compare Unit 1 Compare Register
```

```
CMPR2     .set   7418h        ;Full Compare Unit 2 Compare Register
CMPR3     .set   7419h        ;Full Compare Unit 3 Compare Register
SCMPR1    .set   741Ah        ;Simple Compare Unit 1 Compare Register
SCMPR2    .set   741Bh        ;Simple Compare Unit 2 Compare Register
SCMPR3    .set   741Ch        ;Simple Compare Unit 3 Compare Register

;Capture & QEP Registers - Event Manager (EV)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
CAPCON    .set   7420h        ;Capture Control Register
CAPFIFO   .set   7422h        ;Capture FIFO Status Register
CAP1FIFO  .set   7423h        ;Capture 1 Two-level deep FIFO Register
CAP2FIFO  .set   7424h        ;Capture 2 Two-level deep FIFO Register
CAP3FIFO  .set   7425h        ;Capture 3 Two-level deep FIFO Register
CAP4FIFO  .set   7426h        ;Capture 4 Two-level deep FIFO Register

;Interrupt Registers - Event Manager (EV)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
EVIMRA    .set   742Ch        ;EV Interrupt Mask Register A
EVIMRB    .set   742Dh        ;EV Interrupt Mask Register B
EVIMRC    .set   742Eh        ;EV Interrupt Mask Register C
EVIFRA    .set   742Fh        ;EV Interrupt Flag Register A
EVIFRB    .set   7430h        ;EV Interrupt Flag Register B
EVIFRC    .set   7431h        ;EV Interrupt Flag Register C
EVIVRA    .set   7432h        ;EV Interrupt Vector Register A
EVIVRB    .set   7433h        ;EV Interrupt Vector Register B
EVIVRC    .set   7434h        ;EV Interrupt Vector Register C

;Wait State Generator Registers (mapped into I/O space)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
WSGR      .set   0FFFFh    ;Wait State Generator Register

;-----------------------------------------------------------------------
; Constant Definitions
;-----------------------------------------------------------------------

;Data Memory Boundary Addresses
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
B0_SADDR  .set   00200h    ;Block B0 start address
B0_EADDR  .set   002FFh    ;Block B0 end address
```

```
B1_SADDR  .set   00300h    ;Block B1 start address
B1_EADDR  .set   003FFh    ;Block B1 end address
B2_SADDR  .set   00060h    ;Block B2 start address
B2_EADDR  .set   0007Fh    ;Block B2 end address
XDATA_SADDR   .set   08000h    ;External Data Space start address
XDATA_EADDR   .set   09FFFh    ;External Data Space end address

;Frequently Used Data Pages
;~~~~~~~~~~~~~~~~~~~~~~~~~~
DP_0      .set 0              ;page 0 of data space
DP_PF1    .set 224            ;page 1 of peripheral file (7000h/80h)
DP_PF2    .set 225            ;page 2 of peripheral file (7080h/80h)
DP_PF3    .set 226            ;page 3 of peripheral file (7100h/80h)
DP_EV     .set 232            ;page 1 of EV reg file (7400h/80h)

;Bit codes for Test Bit instruction (BIT)
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
BIT15     .set   0000h     ;Bit Code for 15
BIT14     .set   0001h     ;Bit Code for 14
BIT13     .set   0002h     ;Bit Code for 13
BIT12     .set   0003h     ;Bit Code for 12
BIT11     .set   0004h     ;Bit Code for 11
BIT10     .set   0005h     ;Bit Code for 10
BIT9      .set   0006h     ;Bit Code for 9
BIT8      .set   0007h     ;Bit Code for 8
BIT7      .set   0008h     ;Bit Code for 7
BIT6      .set   0009h     ;Bit Code for 6
BIT5      .set   000Ah     ;Bit Code for 5
BIT4      .set   000Bh     ;Bit Code for 4
BIT3      .set   000Ch     ;Bit Code for 3
BIT2      .set   000Dh     ;Bit Code for 2
BIT1      .set   000Eh     ;Bit Code for 1
BIT0      .set   000Fh     ;Bit Code for 0

;Bit masks used by the CBIT & SBIT Macros
;~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
B15_MSK   .set   8000h     ;Bit Mask for 15
B14_MSK   .set   4000h     ;Bit Mask for 14
B13_MSK   .set   2000h     ;Bit Mask for 13
```

```
B12_MSK   .set   1000h     ;Bit Mask for 12
B11_MSK   .set   0800h     ;Bit Mask for 11
B10_MSK   .set   0400h     ;Bit Mask for 10
B9_MSK    .set   0200h     ;Bit Mask for 9
B8_MSK    .set   0100h     ;Bit Mask for 8
B7_MSK    .set   0080h     ;Bit Mask for 7
B6_MSK    .set   0040h     ;Bit Mask for 6
B5_MSK    .set   0020h     ;Bit Mask for 5
B4_MSK    .set   0010h     ;Bit Mask for 4
B3_MSK    .set   0008h     ;Bit Mask for 3
B2_MSK    .set   0004h     ;Bit Mask for 2
B1_MSK    .set   0002h     ;Bit Mask for 1
B0_MSK    .set   0001h     ;Bit Mask for 0

;-----------------------------------------------------------------------
; M A C R O - Definitions
;-----------------------------------------------------------------------

CBIT      .macro DMA, MASK    ;Clear bit Macro
          LACC   DMA
          AND #(0FFFFh-MASK)
          SACL   DMA
          .endm

SBIT      .macro DMA, MASK    ;Set bit Macro
          LACC   DMA
          OR  #(MASK)
          SACL   DMA
          .endm

KICK_DOG  .macro           ;Watchdog reset macro
          LDP #00E0h       ;DP-->7000h-707Fh
          SPLK   #055h, WDKEY     ;WDCNTR is enabled to be reset by next AAh
          SPLK   #0AAh, WDKEY     ;WDCNTR is reset
          LDP #0h          ;DP-->0000h-007Fh
          .endm
```

Appendix D. ’F240 Slave Mode Program Files

SPI.ASM

```
 ;*******************************************************************
 ;*
 ;*    DESCRIPTION:  SPI Slave Code for Point-to-Point Communication
 ;*
 ;*    AUTHOR:  Jeff Stafford
 ;*
;*******************************************************************
;----------------------------------------------------------------------
; Debug directives
;----------------------------------------------------------------------

             .include "..\..\f240regs.h"

; 1st Data Page of peripheral registers (7000h/80h)
DP_PF1                  .set 224

;BAD_CHKSUM             .set   004h
INVALID_CHKSUM          .set   004h
VALID_CHKSUM            .set   000h
DATA_IN_LEN             .set   010h
DATA_OUT_LEN            .set   010h
STATUS_INFO_BYTES       .set   005h
CONTEXT_MEM_PTR_BYTES   .set   020h
CHKSM_RQST_MSG          .set   008h
STATUS_RQST_MSG         .set   002h

                    .bss   CONTEXT_MEM_PTR, CONTEXT_MEM_PTR_BYTES
                    .bss   STATUS_INFO, STATUS_INFO_BYTES
                    .bss   DATA_IN_PTR,1
                    .bss   DATA_IN, DATA_IN_LEN
                    .bss   DATA_OUT_PTR,1
                    .bss   DATA_OUT, DATA_OUT_LEN

RCV_MSG_TYPE        .set   DATA_IN
```

```
RCV_BYTE_NO         .set   DATA_IN + 1
RCV_DATA_START      .set   DATA_IN + 2
PARAMS              .usect "VARS", 020h

                    .bss   VAR1, 1      ;Scratchpad
                    .bss   NEW_BYTE_FLAG, 1
                    .bss   COPIED, 1

                    .data
ST0_TEMP            .word 0
ST1_TEMP            .word 0

KICK_DOG      .macro
          LDP    #00E0H
          SPLK   #055H, WDKEY
          SPLK   #0AAH, WDKEY
          LDP    #DATA_IN_PTR
                 .endm

          .text

START:    CLRC   SXM        ; Clear Sign Extension Mode
          CLRC   OVM        ; Reset Overflow Mode

       ; Set Data Page pointer to  page 1 of the peripheral frame
          LDP    #DP_PF1          ; Page DP_PF1 includes WET through EINT frames

       ; initialize WDT registers
          SPLK   #06Fh, WDCR  ; clear WDFLAG, Disable WDT, set WDT for 1 second
                              ; overflow (max)

          SPLK   #07h, RTICR  ; clear RTI Flag, set RTI for 1 second
                              ; overflow (max)

       ; EVM 10MHz oscillator settings.  (XTAL2 open, OSCBYP_=GND)
          SPLK   #00B1h,CKCR1     ; CLKIN(OSC)=10MHz, Mult by 2, Div by 1.
          SPLK   #00C3h,CKCR0     ; CLKMD=PLL Enable,SYSCLK=CPUCLK/2,
       ; Clear reset flag bits in SYSSR (PORRST, PLLRST, ILLRST, SWRST, WDRST)
          LACL   SYSSR     ; ACCL <= SYSSR
```

```
    AND    #00FFh       ; Clear upper 8 bits of SYSSR
    SACL   SYSSR     ; Load new value into SYSSR

; initialize B1 RAM to zero's.
    LAR    AR1,#B1_SADDR  ; AR1 <= B1 start address
    MAR    *,AR1          ; use B1 start address for next indirect
    ZAC                  ; ACC <= 0
RPT #(CONTEXT_MEM_PTR_BYTES+STATUS_INFO_BYTES+DATA_OUT_LEN+DATA_IN_LEN+2)
; set repeat counter for sizeof(.bss)-1 loops
    SACL   *+             ; write zeros to B1 RAM

; initialize B2 RAM to zero's.
    LAR    AR1,#B2_SADDR  ; AR1 <= B2 start address
    MAR    *,AR1          ; use B2 start address for next indirect
    ZAC                  ; ACC <= 0
    RPT    #1fh           ; set repeat counter for 1fh+1=20h or 32 loops
    SACL   *+             ; write zeros to B2 RAM

; initialize STATUS_INFO
    LAR    AR1,#STATUS_INFO    ; AR1 <= STATUS_INFO start address
    MAR    *,AR1               ;
    SPLK   #01,*+          ; STAT1 = 1
    SPLK   #02,*+          ; STAT2 = 2
    SPLK   #03,*+          ; STAT3 = 3
    SPLK   #04,*+          ; STAT4 = 4
    SPLK   #05,*+          ; STAT5 = 5

;Initialize NEW_BYTE_FLAG
    LDP    #NEW_BYTE_FLAG
    SPLK   #01h, NEW_BYTE_FLAG
    SPLK   #0, COPIED

;Initialize DSP for interrupts
    LAR    AR6,#IMR       ;
    LAR    AR7,#IFR       ;
    MAR    *,AR6
    LACL   #010h          ;
    SACL   *,AR7          ; Enable interrupt 5 only (SPI low priority)
    LACL   *         ; Clear IFR by reading and
```

```
          SACL   *,AR6     ; writing contents back into itself

       ; call SPI initialization routine
          CALL   INIT_SPI

          LDP    #DATA_IN_PTR

       ;Initialize RX and TX buffers
          SPLK   #DATA_IN, DATA_IN_PTR  ;Reset RX buffer pointer
          SPLK   #DATA_OUT, DATA_OUT_PTR;Reset TX buffer pointer
          SPLK   #0FFH, DATA_OUT     ;Initialize default TX byte = ACKnowledge

          CLRC   INTM          ; Enable DSP interrupts
MAIN:
          LDP    #NEW_BYTE_FLAG
          MAR    *, AR5
          LAR    AR5, NEW_BYTE_FLAG  ;UPDATE AR5
          BANZ   MAIN,*

RD_MSG:
          LDP    #IMR/128
          LACL   IMR              ;  Mask SPI interrupt to avoid changes
          AND    #0FFEFh             ;to DATA_IN_PTR.  NEW_BYTE_FLAG is used
          SACL   IMR              ;as a handshake between SPI_ISR & RD_MSG.

          LDP    #NEW_BYTE_FLAG
          SPLK   #1, NEW_BYTE_FLAG   ;UPDATE FLAG, NEW BYTE HAS BEEN READ

          LDP    #IMR/128
          LACL   IMR              ;  Unmask SPI interrupt to avoid changes
          OR     #010h            ;to DATA_IN_PTR
          SACL   IMR

CHECKSUM_RQST:
          LDP    #DATA_IN
          LACC   DATA_IN             ;  Is this 1st byte a Checksum Request
          SUB    #CHKSM_RQST_MSG     ;message from the Master SPI?  If so,
          BCND   RESET_RCV_BFR, EQ   ;then no processing is required, reset
                                  ;receive buffer and return.
```

```
STATUS_INFO_RQST:                ;  Is status info requested?  It is if
          BIT    RCV_MSG_TYPE, BIT1  ;bit #1 is set in the first rcvd byte,
          BCND   STATUS_RQST, NTC ;the msg-type byte.  If status info is
                                 ;requested then copy it into DATA_OUT.
COPY_STATUS_INFO:
          BIT    COPIED, BIT0
          BCND   MSG_COMPLETE, TC ;Don't copy if it was copied before

          LDP    #IMR/128
          LACL   IMR              ;  Mask SPI interrupt until DATA_OUT
          AND    #0FFEFh             ;has been updated.
          SACL   IMR

          LDP    #DATA_OUT_PTR    ;Copy status info into DATA_OUT buffer.
          MAR    *, AR6
          LAR    AR6, DATA_OUT_PTR
          ADRK   #01
          SAR    AR6, DATA_OUT_PTR
          LDP    #STATUS_INFO
          RPT    #(STATUS_INFO_BYTES - 1)
          BLDD   #STATUS_INFO, *+
          SBRK   #1
          SAR    AR6, DATA_OUT_PTR
          SPLK   #01, COPIED         ;Set flag to copy only once

          LDP    #IMR/128
          LACL   IMR              ;Unmask SPI interrupt
          OR     #010h
          SACL   IMR

STATUS_RQST:
          LDP    #DATA_IN
          LACC   DATA_IN             ;  Is this 1st byte a Status Request
          SUB    #STATUS_RQST_MSG    ;message from the Master SPI?  If so,
          BCND   RESET_RCV_BFR, EQ   ;then no further processing is required,
                                     ;reset receive buffer and return.
MSG_COMPLETE:
          LDP    #DATA_IN_PTR
```

```
          LACC   DATA_IN_PTR      ;Are all bytes rcvd for the current msg?
          SUB    #DATA_IN         ;If not, then do not process. Min bytes in
          SUB    #04h             ;message = 4, excluding ChksumRqst.
          BCND   MAIN, LT
          LACC   DATA_IN_PTR
          SUB    #DATA_IN
          LAR    AR6, #RCV_BYTE_NO
          MAR    *, AR6
          SUB    *
          BCND   MAIN, LT

DATA_OUT_EMPTY:
          LACC   DATA_OUT_PTR     ;  Is the DATA_OUT buffer empty?
          SUB    #DATA_OUT        ;If not, then do not process.
          BCND   MAIN, NEQ
          SPLK   #0, COPIED          ;Reset COPIED flag for next status rqst msg

VERIFY_CHECKSUM:
          LACL   *-             ;  Calculate checksum and compare with received
          SUB    #02H           ;value.  Checksum is calculated by adding all
          SACL   VAR1           ;received bytes except the received checksum.
          LACC   #0             ;Only the LSB of the result is used.
          RPT    VAR1
          ADD    *+
          AND    #0FFh
          SACL   VAR1
          SUB    *
          BCND   STORE_CHECKSUM, EQ
          LACL   #INVALID_CHKSUM

STORE_CHECKSUM:                   ;  Store result of checksum comparison into
          LAR    AR6, DATA_OUT_PTR   ;DATA_OUT buffer.
          ADRK   #01h
          SAR    AR6, DATA_OUT_PTR
          SACL   *+

PARAMS_RQST:                    ;  Were new parameters downloaded?
          BIT    RCV_MSG_TYPE, BIT0  ;If so, transfer them from DATA_OUT buffer
          BCND   RESET_RCV_BFR, NTC  ;to memory.
```

```
STORE_PARAMS:
          LACL   RCV_BYTE_NO             ;Get # of data bytes
          SUB    #04H
          SACL   VAR1
          LAR    AR6, #RCV_DATA_START   ;Point to 1st data byte
          RPT    VAR1                 ;Copy data bytes to mem location PARAMS
          BLDD   *+, #PARAMS

RESET_RCV_BFR:
          LDP    #DATA_IN
          SPLK   #DATA_IN, DATA_IN_PTR  ;Reset rcv buffer pointer
          B      MAIN

 ;*******************************************************************
 ;*
 ;** INIT_SPI
 ;*
 ;*  REGISTER USAGE:   assumes DP = 224 (addresses 0x7000 - 0x707f)
 ;*
 ;*  DESCRIPTION:   SPI initialization subroutine.  This SR initializes
 ;*             the SPI for data stream transfer to a master SPI.
 ;*             The '240 SPI is configured for 8-bit transfers as a slave.
 ;*
 ;*******************************************************************
INIT_SPI:
       ; initialize SPI in slave mode
          LDP    #DP_PF1
          SPLK   #00C7h,SPICCR ; Reset SPI by writing 1 to SWRST
          SPLK   #0000h,SPICTL ; Disable ints & TALK, normal clock, SLAVE mode
          SPLK   #0040h,SPIPRI ; Set SPI interrupt to low priority.
                               ; For emulation purposes, allow the SPI
                               ; to continue after an XDS suspension.
                        ; HAS NO EFFECT ON THE ACTUAL DEVICE.
          SPLK   #0000h,SPISTS ; Clear the SPI interrupt status bits

          ; falling edge with No delay
          SPLK   #0013h,SPICTL ; Enable TALK, ena SPI int, CLK ph 0, slave mode
          SPLK   #0002h,SPIPC1
```

```
          SPLK   #0022h,SPIPC2 ; Set SIMO & SOMI functions to serial I/O

          SPLK   #0047h,SPICCR ; Release SWRST, clock polarity 1, 8 bits

          RET                  ; Return to MAIN routine.

 ;******************************************************************
 ;*
 ;** SPI_ISR
 ;*
 ;*  REGISTER USAGE:   ST0, ST1, ACC, AR6, AR7
 ;*
 ;*  DESCRIPTION:   SPI interrupt service routine.
 ;*
;******************************************************************

SPI_ISR:
          SST    #0, ST0_TEMP ;Auto page-0 DP
          SST    #1, ST1_TEMP
          LDP    #CONTEXT_MEM_PTR
          SACL   CONTEXT_MEM_PTR
          SACH   CONTEXT_MEM_PTR+1
          SAR    AR7, CONTEXT_MEM_PTR+2
          SAR    AR6, CONTEXT_MEM_PTR+3
          LDP    #IMR/128
          LACL   IMR
          LDP    #CONTEXT_MEM_PTR
          SACL   CONTEXT_MEM_PTR+4

OVER_RUN:
          LDP    #DP_PF1          ; Page DP_PF1 includes SPI
          BIT    SPISTS, 8        ;Overrun flag (SPISTS.7) set?
          BCND   CLEAR_FLAG, TC   ;If set, clear & return

       ;Send next TX byte
NEXT_TX:
          LDP    #DATA_OUT_PTR
          LAR    AR6, DATA_OUT_PTR
          MAR    *,AR6
```

```
          LACL   *-            ;ACC = next byte out
          LDP    #DP_PF1
          SACL   SPIDAT        ;Send next byte out, ACK is default

          LDP    #DATA_OUT_PTR
          LACC   #DATA_OUT     ;If the TX pointer is not at the start pos.
          SUB    DATA_OUT_PTR ;  then point it at next byte to be send.

          BCND   READ_SPI, EQ
          SAR    AR6, DATA_OUT_PTR   ;Save TX pointer back to memory

READ_SPI:
          LDP    #DP_PF1       ; Page DP_PF1 includes SPI
          LACC   SPIBUF        ;load rcvd data into ACC

          LDP    #DATA_IN_PTR     ; Set data page pointer to B1 page
          LAR    AR7, DATA_IN_PTR ; update RX ptr
          MAR    *, AR7
          SACL   *+
          SAR    AR7, DATA_IN_PTR ;Set RX pointer to next entry

          LDP    #NEW_BYTE_FLAG      ;Trigger RD_MSG
          SPLK   #0, NEW_BYTE_FLAG

SPI_DONE:
          LDP    #CONTEXT_MEM_PTR
          LACL   CONTEXT_MEM_PTR+4
          LDP    #IMR/128
          SACL   IMR
          LDP    #CONTEXT_MEM_PTR
          LAR    AR6, CONTEXT_MEM_PTR+3
          LAR    AR7, CONTEXT_MEM_PTR+2
          LACC   CONTEXT_MEM_PTR+1,16
          ADDS   CONTEXT_MEM_PTR
          LDP    #0
          LST    #1, ST1_TEMP
          LST    #0, ST0_TEMP
          CLRC   INTM
```

```
          RET

CLEAR_FLAG:
          SPLK   #0H, SPISTS
          B      SPI_DONE



 ;*******************************************************************
 ;*
 ;** Interrupt Vectors
 ;*
 ;*  DESCRIPTION:   Used by linker to place this section at the reset vector
location.
 ;*
 ;*******************************************************************

          .sect   "vectors"
          B       START           ;reset
          B       START           ;Int level 1 not used
          B       START           ;Int level 2 not used
          B       START           ;Int level 3 not used
          B       START           ;Int level 4 not used
          B       SPI_ISR         ;Int level 5, low priority SPI
```

SPI1.TAK

```
cd c:\dspcode\x240\f240evm\spi_app\slave

load spi.out

WA TEST_MSG,,x
WA *TEST_MSG_PTR,,x
WA *TEST_MSG_END,,x

WA DATA_IN,,x
WA DATA_IN_PTR,,x
WA *DATA_IN_PTR,,x

WA DATA_OUT,,x
```

```
WA DATA_OUT_PTR,,x
WA *DATA_OUT_PTR,,x

WA STATUS_INFO,,x
WA *STATUS_INFO,,x

WA *VAR1,,x
WA *NEW_BYTE_FLAG,,x
WA *COPIED


;ba XINT1_ISR
;ba SPI_ISR
;ba RD_MSG
BA VERIFY_CHECKSUM

MEM DATA_IN_PTR
MEM1 DATA_OUT_PTR
;MEM2 TEST_MSG
MEM2 PARAMS
MEM3 STATUS_INFO

bl
```