



Connecting SHARC® and Blackfin® Processors over SPI

Contributed by Jeyanthi Jegadeesan and Andreas Pellkofer

Rev 1 – July 8, 2008

Introduction

This EE-Note describes how to connect Analog Devices Blackfin® processors to SHARC® processors over the Serial Peripheral Interface (SPI).

This document includes:

- SPI description
- Description of the physical layer setup
- Configuration of the SPI
- Programming model for the Blackfin processor SPI interface
- Programming model for the SHARC processor SPI interface
- Code examples as separate files

Motivation

Today's embedded systems often require multiple processors, each for a special application range. A typical system using both Analog Devices Blackfin and SHARC processors could involve an audio environment.

For SHARC processors, a typical application might involve high-dynamic/high-performance, floating-point audio processing. Blackfin processors, which combine both a DSP and a microcontroller, can act as a host to control SHARC processors, for example:

- Booting SHARC processors (SPI slave booting)
- Sending messages (e.g., bass, treble, volume, fader)

Figure 1 shows an example signal flow diagram for a multiprocessor system.

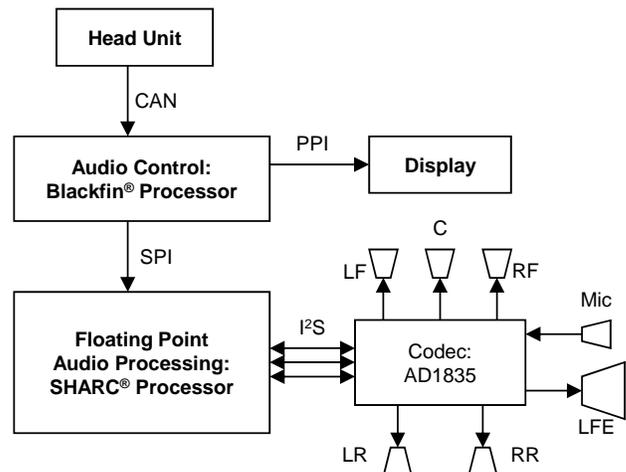


Figure 1. System setup

SPI Description

SPI® is an industry-standard synchronous serial data link named by Motorola. The standard is not fully specified. The specification includes the hardware, but not the software protocol.

It supports communication with multiple SPI-compatible devices. Unlike other serial interfaces like I²C, the SPI peripheral is a four-wire interface – I²C uses two lines – consisting of two

data pins (MOSI and MISO), one device select pin (/SPISS), and a gated clock pin (SCK). The two data pins allow full-duplex operation to other SPI-compatible devices. SPI also includes programmable baud rates, clock phase, and clock polarity. The devices communicate in master/slave mode, where the master device initiates the data frame. Multiple slave devices are allowed with individual slave/chip select lines.

A restricted subset of SPI is Microwire™ (μWire) from National Semiconductor, which is based on SPI and is compatible to it. In this EE-Note, SPI refers to Motorola SPI.

Typical SPI-compatible peripheral devices include:

- Microcontrollers
- Codecs
- A/D and D/A converters
- Sensors
- Flash memory devices
- SP/DIF and AES/EBU digital audio transmitters and receivers
- LCD displays

The SPI interface on Blackfin processors and SHARC processors provide the following features:

- Full-duplex synchronous serial interface
- 8- or 16-bit word sizes (Blackfin processors)
- 32-bit word sizes (SHARC processors)
- Little endian or big endian formats
- Programmable baud rates, clock polarities, and phases
- Master, slave, and multi-master modes
- Open drain outputs to avoid possible driver conflict due to data contention and to support multi-master scenarios
- Master or slave booting from an SPI device

- DMA capability to allow data transfers without core overhead

Figure 2 and Figure 3 show SPI block diagrams.

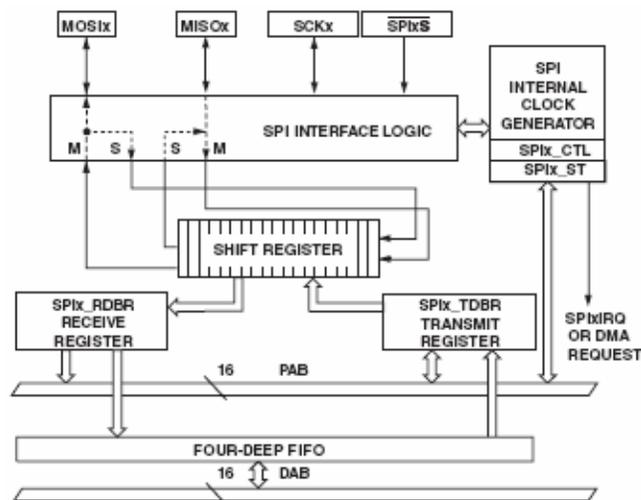


Figure 2. Blackfin SPI block diagram

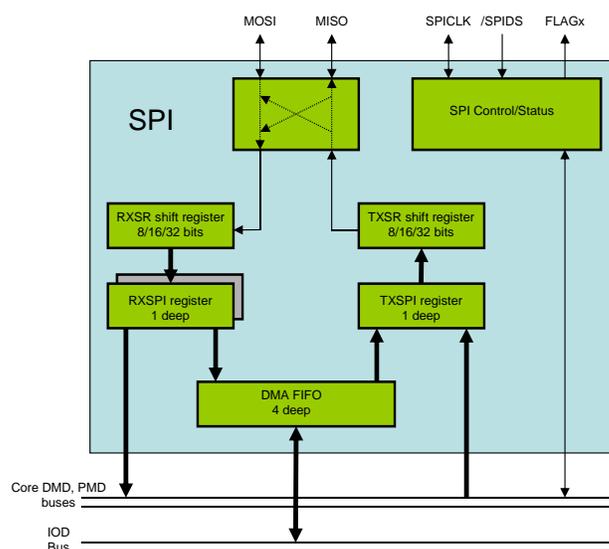


Figure 3. SHARC SPI block diagram

For details, refer to the SPI chapter in the *Hardware Reference* manual (HRM) of your Blackfin processor derivative and SHARC processor derivative, respectively.

Setting Up the Physical Layer

Figure 4 shows the connection and direction of the signals for a SHARC SPI slave device and a Blackfin SPI master device.

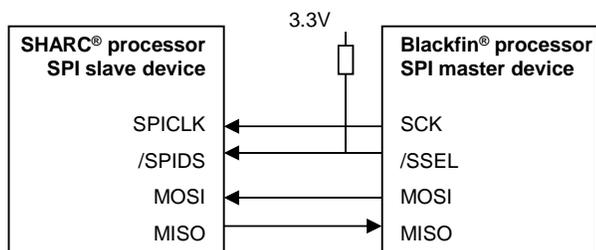


Figure 4. Master-slave connection

An SPI master device always drives both the clock and slave select line signals. An SPI slave device is active only when two signals are present: a clock and an active slave select line.

For both processors, the difference between an SPI master device and an SPI slave device is one bit (`MSTR`) in the SPI control register.

For a master SPI device, the `/SPISS` pin can act as an error signal input in a multi-master environment. A pull-up resistor is recommended.

The SPI slave select enable output signals (`/SSEL`) are always active low in the SPI protocol. Since the respective pins are not driven during reset, it is recommended to pull them up by a resistor.

 Blackfin processors require a pull-up resistor for the `SCK` line. The SHARC processor's DPI provide an internal pull-up for the `SPICLK` pin, so no external pull-up is required here.

Programming Model for Blackfin Processors

The programming model of the Blackfin processor's SPI interface is described in the *Hardware Reference* manual of your Blackfin derivative. The SPI flowcharts provided in the

ADSP-BF537 HRM provide a good reference for setting up an SPI transfer.

This section describes specific details of the programming model and is provided as an addendum and a brief summary of the current documentation.

 This EE-Note focuses on a single master/single slave environment, where the ADSP-BF537 processor acts as an SPI master and the ADSP-21369 processor acts as an SPI slave device. The provided examples cover other set-ups based on these two processors.

Preparation

Most Blackfin processors use pin multiplexing to reduce pin count. On the ADSP-BF537 processor, most of the SPI signals are accessible through Port F. The five most important signals (`SCK`, `MISO`, `MOSI`, `/SPISS`, and `SPISEL1`) are not multiplexed with other peripherals.

For ADSP-BF537 processors, you must write to `PORTF_FER` to enable these SPI signals. By default, GPIO functionality is enabled.

For using `SPISEL2` – `SPISEL7`, refer to the SPI chapter and the General-Purpose PORT chapter of the HRM. These signals – if required – must also be enabled.

Interrupts are used to signal the end of a transfer from/to a peripheral. Otherwise, status bits must be polled periodically by the processor to detect the end of a transfer. If the clock ratio between the core and the system/peripheral is high (e.g., SPI clock = 1 MHz, and core clock = 500 MHz), polling is probably not an issue. This allows other operations to be performed in the meantime.

Both the Core Event Controller and the System Interrupt Controller (SIC) must be configured as well. Refer to the “Program Sequencer” chapter in the *Blackfin Processor Programming Reference* manual (PRM) and the “System Interrupts” chapter of the HRM for details.

SPI Initialization

The SPI interface consists of a set (SPI_FLG, SPI_BAUD, and SPI_CTL) of system memory-mapped registers (MMRs) for configuration.

SPI status can be read from the SPI_STAT register.

For core-driven transfers, the SPI transmit/receive data buffer registers (SPI_TDBR/SPI_RDBR) and SPI_SHADOW are also required.

SPI_FLG Register

For an SPI master device, a slave is selected by writing to SPI_FLG to set the appropriate slave select enable (FLSx) bits.

The slave select value (FLGx) bits determine the value driven onto the slave select line.

If CPHA = 1 (CPHA bit set in SPI_CTL), the output value is set by software control of the FLGx bits.

If CPHA = 0, the SPI hardware sets the output value, and the FLGx bits are ignored. This means hardware selects and deselects the slave (/SSEL) every single word (see Figure 5 and Figure 7).

The following figures show a transfer of 16-bit words over SPI. The first two words are illustrated. The Blackfin processor is the master and drives the SPI clock and slave select line. The SHARC processor acts as a slave, receiving data from the master SPI device.

Additionally, Figure 7 and Figure 8 show an active low SCK version of the transmission; clock polarity is inverted in this case.

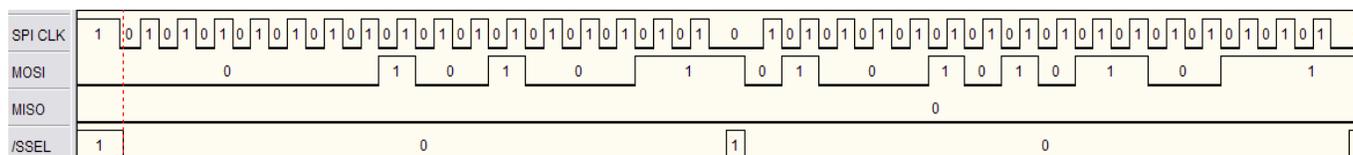


Figure 5. CPHA = 0, active high SCK (CPOL = 0)

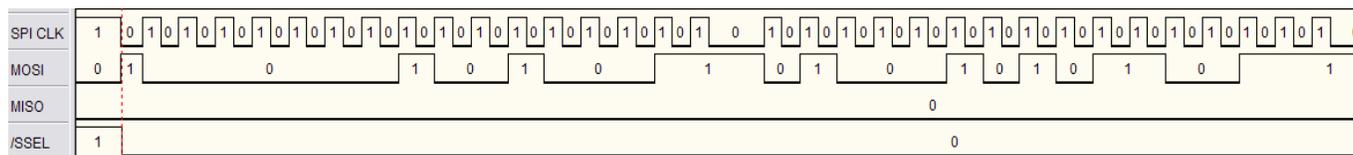


Figure 6. CPHA = 1, active high SCK (CPOL = 0)

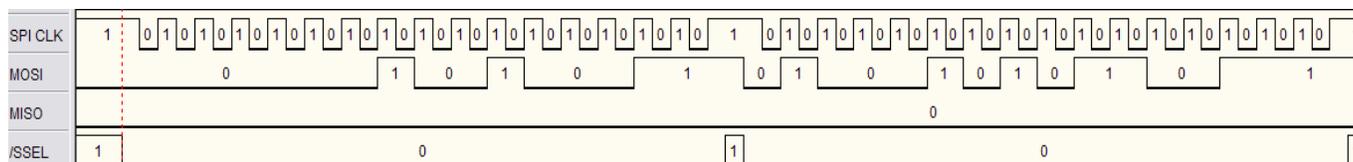


Figure 7. CPHA = 0, active low SCK (CPOL = 1)

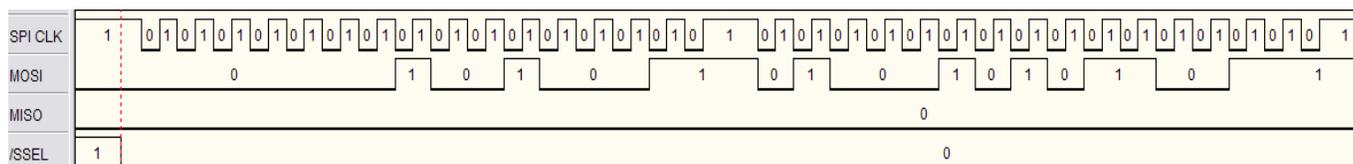


Figure 8. CPHA = 1, active low SCK (CPOL = 1)

SPI_BAUD

The `SPI_BAUD` register allows you to set the SPI clock frequency. It is derived from the Blackfin processor's system clock (`SCK`). If `SCK` is maximized to 133 MHz, this translates to an SPI frequency limitation of ~33 MHz (`SCK/4`). If higher frequencies are required, the Blackfin processor's SPORT interface can be used to emulate SPI. Serial clock speeds of up to ~66 MHz can be achieved, as the maximum SPORT clock frequency is `SCK/2`. Refer to EE-304^[4].

SPI_CTL

The `SPI_CTL` register is described in the processor's HRM.

As mentioned in “SPI Description”, Motorola did not specify a (timing) protocol for SPI, and four different modes are typically used. These modes are set with the two bits: `CPOL` (clock polarity) and `CPHA` (clock phase).

The SPI interface of current Blackfin processors – compared to the SHARC SPI interface – does not support 32-bit data words.

SPI_STAT

The SPI status register (`SPI_STAT`) can be divided in two groups of bits. The first group is for detecting errors, such as buffer under-run, buffer overflow, and conflicts in multi-master environments. The other group of bits shows the status of the transmit buffer or the receive buffer and the SPI status (SPI finished).

For details, see “SPI DMA Transfer and Interrupt Servicing”, which shows how to use these status bits.

SPI DMA Initialization

The SPI has a single dedicated DMA channel for doing one transmit or one receive operation at the same time.

Refer to the Direct Memory Access chapter in the HRM for details.

SPI Transfer Start

For an SPI master device, select a slave by writing to `SPI_FLG` to set the appropriate slave select enable (`FLSx`) bits.

In SPI DMA mode, the SPI interface and the DMA should not be enabled during initialization. If both are configured properly, first start the DMA (`DMAEN = 1`) and then the SPI (`SPEN = 1`).

In SPI core mode, we can enable the SPI during initialization, if we are running in `TIMOD = 00`. In `TIMOD = 01`, the SPI starts immediately when the SPI bit is set. Either fill `SPI_TDBR` first and then enable the SPI, or use also `TIMOD = 00` for SPI transmit operations. For the last case, a dummy read access to `SPI_RDBR` starts the transfer.

SPI DMA Transfer and Interrupt Servicing

	DMA Tx	DMA Rx	Core Tx	Core Rx
DMA_RUN	1x	-	-	-
TXS	2x	-	1x	-
RXS	-	-	-	≥1x
SPIF	1x	-	1x	1x

Table 1. Status bits to be polled

For an SPI DMA operation, you must wait for the DMA interrupt to occur. That's when the `DMA_DONE` bit is set in the `DMAx_IRQ_STATUS` register.



The `DMA_DONE` interrupt is asserted when the last memory access (read or write) has completed. To transmit to a peripheral (memory read), there may be up to four data words in the channel's DMA FIFO when the interrupt occurs.

The service routine for an SPI receive or SPI transmit must first clear the interrupt source. Write a 1 to the `DMA_DONE` bit (*WIC – write-1-to-clear*). Otherwise, an interrupt will be latched again.

 If, however, the application needs to know when the final data item is actually transferred to the peripheral, the application can test or poll the `DMA_RUN` bit. As long as there is pending data in the FIFO, the `DMA_RUN` bit is 1.

Poll the `DMA_RUN` bit (`DMA_RUN = 1`) in the `DMAx_IRQ_STATUS` register until it goes low (`DMA_RUN = 0`).

 When using DMA for SPI transmit, the `DMA_DONE` interrupt signifies that the DMA FIFO is empty. However, at this point there may still be data in the PERIPHERAL (SPI) DMA FIFO waiting to be transmitted. Therefore, software needs to poll `TXS` in the `SPI_STAT` register until it goes low for two successive reads, at which point the SPI DMA FIFO will be empty.

The `TXS` bit defines when the transmit buffer can be filled, but the data might be still in the SPI shift register.

The `SPIF` bit is set one-half the `SCK` period after the last `SCK` edge. That's when the last bit of the word is shifted out, and the SPI transfer is finished.

The `RXS` bit defines when the receive buffer can be read. The end of a single word transfer occurs when the `RXS` bit is set, indicating that a new word has just been received and latched into the receive buffer, `SPI_RDBR`. For a master SPI, `RXS` is set shortly after the last sampling edge of `SCK`. For a slave SPI, `RXS` is set shortly after the last `SCK` edge, regardless of `CPHA` or `CPOL`. The latency is typically a few `SCLK` cycles and is independent of `TIMOD` and the baud rate. If configured to generate an interrupt when `SPI_RDBR` is full (`TIMOD = 00`), the interrupt goes active one `SCLK` cycle after `RXS` is set. When not relying on this interrupt, the end of a transfer can be detected by polling the `RXS` bit.

Polling is not required for a DMA receive operation as the interrupt occurs after the last

memory write operation of the DMA. For a DMA transmit operation, you need to take care of the data latency between the DMA FIFOs and SPI DMA FIFOs.

SPI Core Transfer

For an SPI core transfer, you can also use interrupts. But that's not very meaningful if the SPI clock to core clock ratio is close to one. You would have an interrupt for every single word, and the MIPS performance would be reduced dramatically.

Therefore, it is good practice to set up a hardware loop (a feature of the Blackfin Sequencer) for data transfer. Inside this loop, the appropriate status bits must be polled every single word between every single transfer.

SPI Transfer Stop

Stopping the SPI is similar to the start procedure. For SPI DMA transfer, stop the components in the reversed order. Deselect the slave by clearing the appropriate slave select enable (`FLSx`) bit. Then stop the DMA, and at least disable the SPI.

Additionally, you can reset the `SPI_STATUS` register by clearing error bits (`WIC`). To clear the `RXS` bit, do a dummy read from `SPI_RDBR`.

Summary: DMA and Core SPI Transfer

The Blackfin SPI Controller provides two ways to perform a transfer. You can set up a DMA-based transfer or use the processor core to access the `SPI_TDBR` or `SPI_RDBR` registers, driving the transfer, set up with the two `TIMOD` (transfer initiation mode) bits.

A DMA-based transfer reduces the processor's load – other computations or tasks can be done in parallel. If you need an acknowledgement that the transfer (the DMA) has finished, enable interrupts; otherwise, DMA status bits must be polled by the processor.

DMA-based transfer is preferred, especially for large transfers.

There is one disadvantage regarding a DMA-based transfer when the Blackfin processor is both in DMA receive mode ($TIMOD = 10$) and a master ($MSTR = 1$). In this case, the SPI still drives the SPI clock and slave select line even when the last word is received and the transfer should be completed; the value in the current inner loop count register is zero ($DMAx_CURR_X_XOUNT = 0$). This is because the SPI and DMA are not synchronized with regard to word count. Every read access to SPI_RDBR (never mind if DMA or core is doing this) drives the SPI to fetch new data on the $MISO$ line until the SPI is stopped ($SPE = 0$).

If the connected slave has no clue about the number of words to be transmitted and/or no counter is running to stop the slave at the right time, the slave might accidentally be driven to transfer more words than required.

i In receive mode with DMA ($TIMOD = 10$), as long as there is data in the SPI DMA FIFO (i.e., the FIFO is not empty), the SPI continues to request a DMA write to memory. The DMA engine continues to read a word from the SPI DMA FIFO and writes to memory until the SPI DMA word count register transitions from 1 to 0. The SPI continues receiving words until SPI DMA mode is disabled.

i The whole FIFO depth for SPI DMA transfers is 6: DMA peripheral (SPI) FIFO (4 16/8 bit words) + SPI_TDBR / SPI_RDBR + SPI interface shift register. Refer to Figure 10.

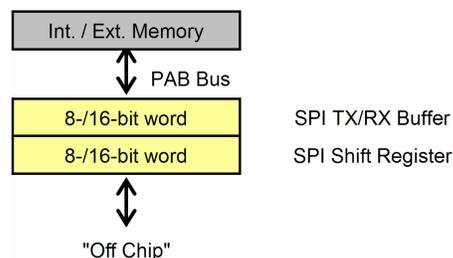


Figure 9. SPI core FIFO / bus structure

In this case, or if only a few data words (e.g. short control messages) have to be send, a core-driven transfer is the preferred method. There are two transfer initiation modes: one mode starts the transfer with a read of SPI_RDBR ($TIMOD = 00$), and the other mode starts with a write to SPI_TDBR ($TIMOD = 01$). The second mode should not be used as the transfer starts immediately when the SPI is enabled ($SPE = 1$).

This means a core-based SPI master transmit and receive should be done with $TIMOD$ bits set to 00. For an SPI receive, simply read the SPI_RDBR data buffer. For an SPI transmit, first fill up the SPI_TDBR data buffer followed by a read access to SPI_RDBR which starts the transfer.

With regard to debugging, a core SPI transfer can be monitored more easily than an SPI DMA transfer. For example, every single word can be transferred in single-step mode. A running DMA cannot be stopped. If the core is in “halt” status, the DMA is running in the background.

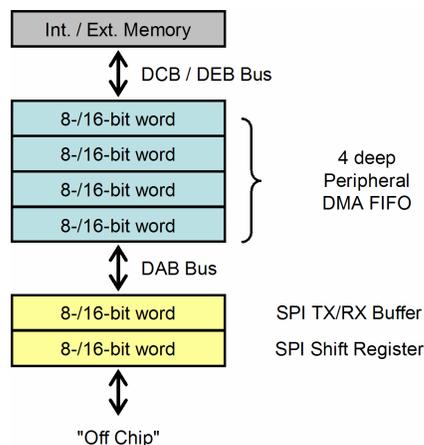


Figure 10. SPI DMA FIFO / bus structure

 The very first access to `SPI_RDBR` is a dummy access; the first word is invalid. Every access to `SPI_RDBR` drives the SPI interface to fetch the next data word.

 After the last transfer, the final word should be read out of the `SPI_SHADOW` data buffer (which does not drive another read transfer), or disable first SPI (`SPE = 0`) before accessing `SPI_RDBR` the last time. Only an access to `SPI_RDBR` will clear `RXS` (read data buffer full) bit.

Blackfin Processor SPI Examples

The examples provided in the associated .ZIP file are for an SPI master and an SPI slave implementation for ADSP-BF537 Blackfin processors. For more details, refer to the 'README.txt' file.

Programming Model for SHARC Processors

The SPI port is available on ADSP-21161N, ADSP-2126x, ADSP-2136x, and ADSP-2137x SHARC processors. The programming model discussed below applies mainly to ADSP-2126x, ADSP-2136x, and ADSP-2137x processors. For ADSP-21161N processors, the SPI programming model is different. This section describes some *specific details* of the programming model. It serves as an addendum and brief summary of the current documentation in the HRM.

Preparation

Each SPI port on SHARC processors has the following signals:

- SPICLK
- SPIDS#
- MOSI
- MISO
- SPI slave select signals (`SPIFLG3-0`)

ADSP-2116x and ADSP-2126x processors have one SPI port, and the SPI signals are available on the dedicated hardware pins. The Flag 3 – 0 pins are used as the slave select signals when the SPI is the master.

ADSP-21362/3/4/5/6 processors have two SPI ports in which the primary SPI signals are available on the dedicated hardware pins and the secondary SPI can be routed through the Signal Routing Unit (SRU) to the Digital Application Interface (DAI) pins of the processors. The primary SPI port uses the Flag 3-0 pins as the slave select signals where as the secondary SPI port has 4 dedicated `SPIFLGB3-0` signals which can be routed to any of the DAI pins.

For ADSP-21367/8/9 and ADSP-2137x processors, both SPI ports can be routed through the Signal Routing Unit 2 (SRU2) to the Digital Peripheral Interface (DPI) pins of the processors. Both the SPI ports on the ADSP-21367/8/9 and ADSP-2137x processors have dedicated `SPIFLG3-0` signals as slave select signals when the SPI is used as master.

The DAI pins and DPI pins on SHARC processors have an internal pull-up resistor. By default, this pull-up resistor is enabled and can be disabled by writing into the corresponding pull-up enable register. When the SPI signals are routed to the DAI or DPI pins, adding the external pull-up may not be needed.

For ADSP-21367/8/9 and ADSP-2137x processors, when the SPI is used as the master the following must be taken care of in the application while routing the `SPICLK` signal using SRU2. When `CLKPL = 0`, the pin enable signal of the DPI pin to which the `SPICLK` is routed should be configured as follows:

```
SRU(SPI_CLK_O, DPI_PB03_I);
SRU(HIGH, DPI_PBen03_I);
```

When `CLKPL = 1`, the pin enable signal of the DPI pin to which the `SPICLK` is routed should be configured as follows:

```
SRU(SPI_CLK_O, DPI_PB03_I);
SRU(SPI_CLK_PBen_O, DPI_PBen03_I);
```

SPI Initialization

The SPI port of SHARC processors use the following control and status registers to configure the SPI:

- SPICTL
- SPIFLG
- SPIBAUD
- SPISTAT

SPICTL Register

The SPICTL register has control bits for selecting the SPI configuration and enabling the SPI. Some of the control bits include:

- Word length selection
- Master or slave operation
- Data format (little or big endian)
- Clock phase and polarity
- Open drain output for data pins
- Packing enable

Refer to the processor's HRM for a complete description of the SPICTL register.

SPIFLG Register

The SPIFLG register is used only when the SPI is configured as the master. This register is not used for slave mode operation. This register allows you to select one or all of the SPIFLG3-0 signals as slave select signals. Setting the DSXEN bit on the SPIFLG register selects the corresponding SPIFLGX signal as the slave select signal.

When CPHASE = 0, the slave select signal will be generated by the hardware automatically. The SPIFLGX signal will be asserted low before each data transfer, and between successive transfers this signal will be asserted high.

When CPHASE = 1, the slave select signal should be generated manually by the user application. The SPIFLGX bit of the SPIFLG register can be used for this purpose. In this mode, setting or clearing the SPIFLGX bit on the SPIFLG register reflects on the SPIFLGX signal. Before SPI is

enabled, ensure that this pin is high by setting the SPIFLGX bit of the SPIFLG register. As soon as the transfer is started, clear the SPIFLGX bit of the SPIFLG register before starting the transfer.

SPIBAUD Register

The SPIBAUD register is used only when the SPI is configured as the master. This register is not used for slave mode operation. This register allows you to select the baud rate of the SPI master. The BAUDR bits (15-1) can be used to configure the SPI baud rate. The SPI baud rate is derived from the core clock of the processor. For ADSP-2136x and ADSP-2137x processors, the SPI baud rate is calculated as follows:

$$\text{SPI baud rate} = \text{CCLK} / (8 * (\text{BAUDR} - 1))$$

For ADSP-2126x processors, the SPI baud rate is calculated as follows:

$$\text{SPI baud rate} = \text{CCLK} / (4 * (\text{BAUDR} - 1))$$

SPISTAT Register

The SPISTAT register is the read-only register that provides the status of transmit and receive buffer FIFOs, and the current transfer completion. It also has the bits to indicate the transmission/reception errors and multi-master errors. The bits on this register are write-1-to-clear bits.

SPI DMA Initialization

The SPI has the following registers for initializing the DMA mode of operation:

- SPIDMAC
- IISPI
- IMSPI
- CSPI
- CPSPI

The IISPI, IMSPI, CSPI, and CPSPI registers are the DMA parameter registers. The IISPI, IMSPI, and CSPI registers hold the value of the internal memory address from/to which the data needs to be transferred, modifier value, and the count value, respectively. The CPSPI register holds the

value of the next DMA sequence when the DMA chaining is enabled.

The `SPIDMAC` register has the control bits to configure the transmit or receive DMA, enable the interrupt, enable the DMA, and DMA chaining. It also has the status bits which show the current DMA transfer status and some error conditions. SPI DMA is initialized by setting the `SPIDEN` bit of the `SPIDMAC` register.

SPI Transfer Start

SPI transfer can be in core mode or in DMA mode. The `TIMOD1-0` bits on the SPI control register select the mode of data transfer. In core mode, the transmit buffer (`TXSPI`) and receive buffer (`RXSPI`) registers are accessed by the core directly. In DMA mode, the DMA controller accesses the receive buffer or the transmit buffer and does the data transfer.

When the `TIMOD1-0` bits are configured as '00', reading the `RXSPI` buffer after enabling the SPI master initiates a data transfer. When these bits are configured as '01', writing to the `TXSPI` buffer after enabling the SPI master initiates a data transfer. The `TIMOD1-0` bits are configured as '10' for DMA mode of operation. Depending upon the direction of the DMA (transmit or receive), the DMA controller initiates a data transfer by writing into the transmit buffer or reading from the receive buffer.

 The DMA controller has 4-deep FIFO, which is used for the DMA data transfer. The internal DMA request is generated for a group of 4 data words on the FIFO. If DMA count is not a multiple of 4, the requests are generated for each group of 4 words, and finally one request is generated for the rest of the words. For example, if the DMA count is 7, the DMA controller generates two DMA requests (one for the DMA count of 4, and the other for the rest of the data words).

For all the cases, the master generates the SPI clock to the slave device. The slave select signals are generated automatically or manually, depending on the `CPHASE` bit configuration.

SPI Transfer and Interrupt Servicing

Core-driven transfers can be in interrupt driven mode or in polling mode. In polling mode, the user application must continuously poll for the status of the transmit buffer or receive buffer by reading the `SPISTAT` register. When the transmit buffer is empty or the receive buffer has data, the application can write data to the transmit buffer or read data from the receive buffer.

In interrupt-driven mode, an interrupt is generated automatically when the transmit buffer is empty or the receive buffer has one complete word, depending on the `TIMOD1-0` bits configuration. Inside the interrupt service routine, the user application must write to the transmit buffer or read from the receive buffer.

In general, the user application can use the combination of the polling and interrupt-driven modes where full-duplex communication is needed.

For DMA-driven transfers, an interrupt is generated at the end of a block transfer. For DMA chaining mode, the interrupt can be generated at the end of each sequence or at the end of all the sequences. The DMA interrupt can be to process the block of data received over the SPI or it can be used to start a new transfer.

 When the SPI master is configured for initiating a data transfer by reading the receive buffer in core mode, initially the master must do a dummy read. This dummy read generates the clock for the slave device to transmit the first data. To receive `N` words of data, the master must do `N+1` data transfers in core mode. For an SPI receive DMA operation, the DMA controller takes care of this dummy read.

SPI Transfer Stop

SPI transfers are stopped when the SPI is disabled. The SPI can be disabled inside the interrupt service routine. For core mode operation, after transmitting or receiving the required number of words, the SPI can be disabled by clearing the `SPICTL` register contents. The `SPIFE` bit of the `SPISTAT` register bit must be polled before disabling the SPI. This bit indicates whether the current transfer is completed. Ensure that the current data transfer is completed before the SPI is disabled.

For DMA mode operation, inside the ISR, the `SPIDMAS` bit of the `SPIDMAC` register must be polled. This bit indicates the DMA completion status and is set when the DMA transfer is in progress (and cleared once it is completed). The `SPIFE` bit of the `SPISTAT` register must be polled to ensure that the last data transfer is completed. Then the SPI and the `SPIDMA` register can be disabled by clearing the contents of the `SPICTL` and `SPIDMAC` registers.

For DMA chaining mode, SPI transfers can be stopped by writing a zero to the chain pointer register. After writing a zero, poll for the `SPIDMAS` bit of the `SPIDMAC` register and the `SPIFE` bit of the `SPICTL` register. Once the current data transfer is completed, the SPI and the `SPIDMA` register can be disabled by clearing the contents of the `SPICTL` and `SPIDMAC` registers.

SHARC Processor SPI Examples

The examples provided with this EE-Note are implemented for ADSP-21369 processors. The SPI master code configures the primary SPI of the ADSP-21369 processor as a master in DMA mode. The divisor value for the `SPIBAUD` register is calculated during runtime, based on the core clock and the selected SPI baud rate. The SPI slave code configures the primary SPI of the ADSP-21369 processor as a slave in DMA mode. For both code examples, the direction of the DMA transfer can be selected using the macros. Macros are also available for selecting the clock phase and polarity configuration.

The code can be used for ADSP-21367/8 and ADSP-2137x processors without modification. For ADSP-21362/3/4/5/6 and ADSP-2126x processors, the SPI configuration code can be used as is, but the `InitSRU` function need not be called. The calculation of the `SPIBAUD` rate must be changed for ADSP-2126x processors.

Conclusion

This EE-Note discusses the SPI programming model on SHARC and Blackfin processors. It also provides example code for serial communication over SPI between SHARC and Blackfin processors. The examples provided with this EE-Note are tested between the SPI ports of the ADSP-21369 SHARC processors and ADSP-BF537 Blackfin processors.

Appendix

The .ZIP file associated with this document contains the following code examples:

- [1] Example code for a Blackfin processor SPI master device
- [2] Example code for a Blackfin processor SPI slave device
- [3] Example code for a SHARC processor SPI master device
- [4] Example code for a SHARC processor SPI slave device

References

- [1] *ADSP-BF53x/BF56x Blackfin Processor Programming Reference*. Rev. 1.0, June 2005. Analog Devices, Inc.
- [2] *ADSP-BF537 Blackfin Processor Hardware Reference*. Rev. 2.0, December 2005. Analog Devices, Inc.
- [3] *ADSP-BF534/ADSP-BF536/ADSP-BF537 Blackfin Embedded Processor Data Sheet*. Rev. C, February 2007. Analog Devices, Inc.
- [4] *Using the Blackfin Processor SPORT to Emulate a SPI Interface (EE-304)*. Rev. 1, November 10, 2006. Analog Devices, Inc.
- [5] *ADSP-21368 SHARC Processor Hardware Reference*. Rev. 1.0, September 2006. Analog Devices, Inc.
- [6] *ADSP-2136x SHARC Processor Hardware Reference for ADSP-21362/3/4/5/6 Processors*. Rev. 1.0, October 2005. Analog Devices, Inc.
- [7] *ADSP-2126x SHARC Processor Peripherals Manual*. Rev 3.0, December 2005. Analog Devices, Inc.
- [8] *ADSP-21371: SHARC Processor Data Sheet*. Rev. 0, July 2007. Analog Devices, Inc.
- [9] *ADSP-21375: 266 MHz High Performance SHARC Processor Preliminary Data Sheet*. Rev. PrB, December 2005. Analog Devices, Inc.
- [10] *ADSP-21367/ADSP-21368/ADSP-21369 SHARC Processors Data Sheet*. Rev. A, August 2006. Analog Devices, Inc.
- [11] *ADSP-21362/ADSP-21363/ADSP-21364/ADSP-21365/ADSP-21366 SHARC Processors Data Sheet*. Rev. B, June 2007. Analog Devices, Inc.
- [12] *ADSP-21261: 3rd Generation, Low-Cost, 150 MHz SHARC Processor Data Sheet*. Rev 0, April 2006. Analog Devices, Inc.
- [13] *ADSP-21262: 3rd Generation Low Cost 32-Bit Floating-Point SHARC Processor Data Sheet*. Rev. B, October 2005. Analog Devices, Inc.
- [14] *ADSP-21266: High Performance SHARC Audio Processor Data Sheet*. Rev. B, May 2005. Analog Devices, Inc.
- [15] *SHARC SPI Slave Booting (EE-177)*. Rev. 3, January 19, 2007. Analog Devices, Inc.

Document History

Revision	Description
Rev 1 – July 8, 2008 by Jeyanthi Jegadeesan and Andreas Pellkofer	Initial release.