Technical notes on using Analog Devices DSPs, processors and development tools
Contact our technical support at dsp.support@analog.com and at dsptools.support@analog.com
Or visit our on-line resources http://www.analog.com/ee-notes and http://www.analog.com/processors

In-Circuit Programming of an SPI Flash with ADSP-2126x SHARC® DSPs

Contributed by Brian M.

Rev 1 – March 2, 2004

Introduction

Unlike previous SHARC® DSPs, ADSP-2126x DSPs are able to boot from SPI Flash devices, providing a cheap booting alternative to parallel flash, SPI EEPROM, or host processor schemes. Since the details of the general booting process are discussed in the System Design chapter of the *ADSP-2126x SHARC DSP Peripheral Manual* [2], this EE-Note describes how to program the desired code or data into the SPI Flash in-circuit.

Default SPI Settings

The defaults of the ADSP-21262 SHARC DSP do not match those supported by most SPI Flash devices. Two issues arise when interfacing an SPI memory device to this DSP: word transfer order and SPI mode.

The ADSP-21262 DSP defaults to transfer words in least significant bit first (LSBF) format, while most SPI memory devices transfer in most significant bit first (MSBF) format. In most applications, this will not be a problem because the SPI of the DSP can be set up to transfer in MSBF format. However, while booting, the word format setting cannot be changed. Therefore, it is necessary to program boot data into the SPI memory device in a bit-reversed manner for devices that support MSBF format only.

There are two ways to transfer bit-reversed words to the flash. The loader tool included with VisualDSP++® 3.0 SP1 has an option to automatically bit-reverse the boot data. The SPI Master and SPI Prom loader options both use this format. This format should be used only when the SPI memory will be programmed by a dedicated memory programmer (such as those used to program an SPI EEPROM).

Alternatively, it is possible to communicate with the SPI Flash in an LSBF format, while bit reversing the commands to look like they are in MSBF format. For debugging purposes, this format is much easier to read, since this matches the format shown in the VisualDSP++ memory windows. This is the method used in this example.

The other issue is much more problematic, but does not apply to the ST M25P80 SPI flash used in this example. By default, the ADSP-21262 SHARC DSP uses SPI mode 3 (the SPI clock toggles at the start of the first data bit, and the SPI clock is active-low). Some SPI memory devices support SPI mode 0 only, or provide only partial mode 3 support. If the device does not support SPI mode 3 at all, the part can not boot the ADSP-21262 DSP.

If partial support for SPI mode 3 is available (as in the Atmel AT25F512 found on the ADSP-21262 EZ-KIT Lite™ development board), a workaround may be available. For the Atmel part mentioned above, reading from the flash in SPI mode 3 works correctly, but programming to the flash exhibits a bit error on the last word of each page if a certain pattern exists in the previous word. This can be addressed in two ways. The easiest way to avoid this problem is to append a

final (8- or 32-bit) word, each bit existing entirely of 1s (0xFF or 0xFFFFFFFF). If this final word wraps around to the beginning of the page, it will not overwrite what was previously programmed there, nor does it waste that word, since the flash cannot overwrite bits that are set to 1 without an erase command. This method is used by the Flash Programmer utility included with VisualDSP++. The alternate method presented herein avoids using problematic words. Since the SPI Flash is programmed on a page-by-page basis, it is possible to shift the position of the starting word to ensure that the offending bit pattern will not be matched. For an example of how to accomplish this, refer to the Atmel SPI Flash Programmer code included with the ADSP-21262 EZ-KIT Lite installation.

Flash Programmer Functions

An SPI flash operates by receiving commands from the Master Out Slave In (MOSI) line, and returning any response on the Master In Slave Out (MISO) line. The following is a quick discussion of the commands implemented in the included example code. The code is attached in the Appendix.

Write Enable (0x06)

This function enables modification to the status register settings or contents of the flash. Since no address or additional data is required, this command is sent directly by the core as a 1-byte, MSBF word. The Write Enable function is called automatically by each function that requires the Write Enable Latch bit in the flash's status register to be set to function properly. These functions are the Write Status Register, Page Program, and Erase (Bulk and Sector) commands.

Write Disable (0x04)

This function disables modification of the status register settings or the contents of the flash after the Write Enable command has already been registered. Since no address or additional data is required, this command is sent directly by the core as a 1-byte, MSBF word.

Read Status Register (0x05)

This function fetches the value of the byte-wide status register. It is implemented as a 2-word DMA using a 1-byte MSBF word. The first returned byte corresponds to the command sent from the DSP, and can be ignored. The second returned byte contains the value of the status register placed in the lowest byte of the destination.

Write Status Register (0x01)

This function writes to the byte-wide status register. It is implemented as a 2-word DMA using a 1-byte MSBF word. The first byte sent is the command from the DSP, and second byte is the desired value of the status register. The Write Enable command must be sent before using the this command. Before calling this function, place the desired value of the status register in the second location of the status register buffer. This is the same location that returns the value of the status register in the Read Status Register function. The first location in this buffer is reserved for the command being sent to the device.

Read Data Bytes (0x03)

This function reads any number of words from the flash. The destination address in the DSP's internal memory, the source address in the flash, and the number of 32-bit words to be read are passed to the function using the dedicated memory locations. It is implemented as an N-word DMA using 32-bit LSBF words, where N is the number of words passed to the function. The first word in the destination buffer will be garbage corresponding to the 1-byte command plus 3-byte address that must precede the data (one 32-bit word).

Page Program (0x02)

This function complements the Read function. The destination address in the flash, source address in the internal memory of the DSP, and number of 32-bit words are passed to this function using the dedicated memory locations.

The flash requires that it be programmed one page at a time. Before being sent to the flash, the buffer referenced in the call is transferred into a temporary buffer corresponding to the page size.

After each page has been sent, the lowest bit in the flash's status register, Write-In-Progress (WIP), is tested in a loop to determine when the flash has finished programming the new data into memory.

The function included in this example also verifies each page after completing the write by calling the Read Data Bytes function and comparing the result of the call to what was sent. The number of 32-bit words that do not match is tracked.

Because this example programs the flash for booting, which requires LSBF format, both the Read Data Bytes command and the Page Program command use LSBF format. Therefore, the command and address need to be bit-reversed before transfer so that the flash receives them in MSBF format. (This is accomplished using the BITREV instruction. See the *ADSP-2126x SHARC DSP Core Reference Manual* [1] for more information.)

Sector Erase (0xD8)

This function erases one sector of the flash according to the memory layout in the SPI flash's data sheet. An address in the sector to be erased is passed to the function with the call. This command uses a 1-byte command and 3-byte address, comprising one 32-bit word. Since the Page Program and Read Data Bytes function use a function that joins the command and address into a single 32-bit word, the Sector Erase command is sent in LSBF format. After sending the command, the status register is polled until the WIP bit is clear.

Bulk Erase (0xC7)

This function erases the entire flash. No address is passed to this function. Therefore, the function is implemented as a 1-byte command sent in MSBF format. After sending the command, the status register is polled until the WIP bit is clear.

Deep Powerdown (0xB9)

This function puts the flash into a low-power state. Since no address or additional data is required, this command is sent directly by the core as a 1-byte, MSBF word.

Deep Powerdown Release/Electronic Signature (0xAB)

This function returns the flash from the low-power state. It is also used when the flash is not in the low-power state to retrieve the electronic signature of the part. It is implemented as a 5-word DMA using a 1-byte MSBF word. The first byte corresponds to the command from the DSP, and can be ignored. The middle three bytes, which are garbage returned by the flash, can also be ignored. The final byte contains the value of the status register placed in the lowest byte of the destination. For the M25P80, the lowest byte is 0x13.

Generating the Loader File

To generate a loader file compatible with this example, it is necessary to use SPI Slave format. Though it may seem logical to use the SPI PROM format, SPI PROM format produces an image that is bit-reversed. In addition, an SPI PROM only uses a 16-bit address; thus, an extra byte is prepended to the image to make the boot stream compatible with the DSP. For details, refer to the booting section in the *ADSP-2126x SHARC DSP Peripheral Manual* [2].

Conclusion

This document provides an example project used to program the SPI flash. Each function included in the example is described. To compare this project to a similar example, refer to the Atmel SPI Flash Programmer included with the ADSP-21262 EZ-KIT Lite installation for the VisualDSP++ development tools.

Appendix

main.asm

```
/*-----------------------------------------------------------------------------
//
// NAME:     main.asm (ST SPI Flash)
// DATE:     2/20/04
// PURPOSE:  Program the SPI Flash for the ADSP-21262
//
// USAGE:    Use this file to call the chip functions contained in SPIflash.asm.
//
-----------------------------------------------------------------------------*/

#include "SPIflash.h"

.global main;

.extern flash_curr_page;
.extern sector_base_addr;
.extern read_base_addr;
.extern read_buffer_addr;
.extern read_buffer_length;
.extern write_buffer_addr;
.extern write_base_addr;
.extern write_buffer_length;
.extern file_data;
.extern file_data_verf;
.extern flash_data_out;
.extern flash_data_verf;
.extern spi_setting;
.extern spi_dma_setting;
.extern status_register;
.extern electronic_signature;
.extern curr_command;
.extern verf_errors;

.extern write_enable;
.extern write_disable;
.extern read_status_register;
.extern write_status_register;
.extern read_data_bytes;
.extern read_data_bytes_hs;
.extern page_program;
.extern flash_program;
.extern sector_erase;
.extern bulk_erase;
.extern deep_powerdown;
.extern deep_powerdown_release;
```

```
.extern setup_spi;
.extern setup_spi_dma;
.extern wait_for_SPIF;
.extern wait_for_WIP;
.extern bit_reverse_command;

.section/pm seg_pmco;
main:
r15=0; r11=0;

// ---------------------------------------
// Choose FLASH algorithms to run here.

// Reset the state of the device with a dummy read
call read_status_register;

// Check to make sure that the correct flash is being used
// Deep_powerdown_release also reads the electronic signature
// of the device.
call deep_powerdown_release;
r0=dm(electronic_signature+4);
r1=0x13;    // For ST M25P80, electronic signature is hex13
comp(r0,r1);
if EQ jump (pc,2);
jump(pc,0);

// Sector erase call
// Either use chip_erase for the entire device
// or use sector_erase to erase sectors individually
// This example uses sector erase.
r0=0;     // Initialize address of sector to be erased.
dm(sector_base_addr)=r0;
call sector_erase;

// Bulk Erase Call
// call bulk_erase;

// Write to FLASH
// The flash_program subroutine requires the user to pass:
// write_base_addr - the base address to write to in the SPI Flash
// write_buffer_addr - the address of buffer holding the data to program
// write_buffer_length - the length of the buffer to program in 32-bit words
// This routine also returns a number of errors encountered in the programming
// process.
// This is stored in the verf_errors with units in 32-bit words.
r0=0;               // Address to write to in the flash
dm(write_base_addr)=r0;
r0=file_data;  // Address of the buffer to write from the DSP
dm(write_buffer_addr)=r0;
r0=@file_data; // Length of the buffer to write from the DSP
dm(write_buffer_length)=r0;
call flash_program;

// Read from FLASH
// The read_data_bytes subroutine requires the user to pass:
// read_base_addr - the base address to write to in the SPI Flash
// read_buffer_addr - the address of buffer holding the data to program
// read_buffer_length - the length of the buffer to program in 32-bit words
```

```
r0=0;               // Address to read from in the flash
dm(read_base_addr)=r0;
r0=file_data_verf;  // Address of the buffer to read from in the DSP
dm(read_buffer_addr)=r0;
r0=@file_data_verf; // Length of the buffer to read from in DSP
dm(read_buffer_length)=r0;
call read_data_bytes;

// ---------------------------------------
// Use FLAGS to show that the process completed.
// Indicate whether the process was error-free
r0=dm(verf_errors);
r0=pass r0;
if ne jump error_detected;
BIT SET FLAGS 0xAAAAAAAA;
BIT SET FLAGS 0x55555555;
main.end: jump(pc,0);

error_detected:
BIT SET FLAGS 0xAAAAAAAA;
BIT CLR FLAGS 0x55555555;
error_detected.end: nop;
jump(pc,0);
```

Listing 1. main.asm

buffers.asm

```
/*-----------------------------------------------------------------------------

// NAME:     buffers.asm (ST SPI Flash)
// DATE:     2/20/04
// PURPOSE:  Program the SPI Flash for the ADSP-21262
//
// USAGE:    This file contains the buffer declarations for use in this project.
//
-----------------------------------------------------------------------------*/

#include "SPIflash.h"

.global flash_curr_page;
.global sector_base_addr;
.global read_base_addr;
.global read_buffer_addr;
.global read_buffer_length;
.global write_buffer_addr;
.global write_base_addr;
.global write_buffer_length;
.global file_data;
.global file_data_verf;
.global flash_data_out;
.global flash_data_verf;
.global spi_setting;
.global spi_dma_setting;
.global status_register;
.global electronic_signature;
.global curr_command;
```

```
.global verf_errors;

.section/dm seg_dmda;
// Buffer to hold the (byte) address to send to the SPI Flash
// Used in bit_reverse_command subroutine and those that call it
.var flash_curr_page;

// Buffer to hold the (byte) address to indicate which sector to erase
// Used in sector_erase
.var sector_base_addr;

// Buffer to hold the (byte) address to indicate where to begin reading from the
// flash
// Used in read_data_bytes
.var read_base_addr;

// Buffer to hold the (normal word) address of the buffer to store after reading
// Used in read_data_bytes
.var read_buffer_addr;

// Buffer to hold the length (in 32-bit words) of the buffer to store after reading
// Used in read_data_bytes
.var read_buffer_length;

// Buffer to hold the (normal word) address of the buffer to write to the flash
// Used in page_program
.var write_buffer_addr;

// Buffer to hold the (byte) address to indicate where to begin writing to the
// flash
// Used in page_program
.var write_base_addr;

// Buffer to hold the length (in 32-bit words) of the buffer to send to the flash
// Used in page_program
.var write_buffer_length;

// Buffer initialized with the data to write to the flash
// Used in main.asm
.var file_data[]=FILENAME;

// Buffer to verify that the entire buffer was programmed as expected
// Length is one location more than file_data because the buffer will hold an extra
// word
// Corresponding to the read command and address.
// Used in main.asm
.var file_data_verf[]=FILENAME,0;

// Temporary buffer used to hold the current page to program to the flash
// First location is Write command + address
// Used in page_program
.var flash_data_out[PAGE_LENGTH+1];

// Temporary buffer used to verify the current page programmed to the flash
// First location corresponds Read command + address
// Used in page_program
.var flash_data_verf[PAGE_LENGTH+1];
```

```
// Buffer to hold the desired SPICTL setting
// Used in setup_spi
.var spi_setting;

// Buffer to hold the desired SPIDMAC setting
// Used in setup_spi_dma
.var spi_dma_setting;

// Buffer to hold the current value of the flash's status register on a read
// or the desired value of the flash's status register on a write.
// Used in read_status_register and write_status_register
.var status_register[2] =   0,  // dummy location for RX, command word for TX
                            0;  // status location

// Buffer to hold the product ID of the flash
// Used in read_device_id
.var electronic_signature[5] =  0,0,0, // dummy locations
                                0,     // Manufacturer ID
                                0;     // Device code

// Buffer to hold the command to send to the flash
// Used in bit_reverse_command
.var curr_command;

// Buffer to hold the number of 32-bit words that don't match what was programmed
// Used in page_program
.var verf_errors=0;
```

Listing 2. buffers.asm

SPIflash.asm

```
/*-----------------------------------------------------------------------------
//
// NAME:     SPIflash.asm (ST SPI Flash)
// DATE:     2/20/04
// PURPOSE:  Program the SPI Flash for the ADSP-21262
//
// USAGE:    This file contains the subroutines used to program and verify the
//           SPI Flash
//
-----------------------------------------------------------------------------*/

#include "SPIflash.h"

.global write_enable;
.global write_disable;
.global read_status_register;
.global write_status_register;
.global read_data_bytes;
.global read_data_bytes_hs;
.global page_program;
.global flash_program;
.global sector_erase;
.global bulk_erase;
.global deep_powerdown;
.global deep_powerdown_release;
```

```
.global setup_spi;
.global setup_spi_dma;
.global wait_for_SPIF;
.global wait_for_WIP;
.global bit_reverse_command;

.extern flash_inst;
.extern flash_curr_page;
.extern sector_base_addr;
.extern read_base_addr;
.extern read_buffer_addr;
.extern read_buffer_length;
.extern write_buffer_addr;
.extern write_base_addr;
.extern write_buffer_length;
.extern file_data;
.extern file_data_verf;
.extern flash_data_out;
.extern flash_data_verf;
.extern spi_setting;
.extern spi_dma_setting;
.extern status_register;
.extern electronic_signature;
.extern curr_command;
.extern verf_errors;

.section/pm seg_pmco;
// ---------------------------------------
// WRITE ENABLE SUBROUTINE (1 Byte)
// Enables the Write Enable Latch of the Flash
// Inputs - none
// Outputs- none
// Core sends the command
write_enable:
r0=COMMON_SPI_SETTINGS|TIMOD1|WL8|MSBF;
dm(spi_setting)=r0;
call setup_spi;

r0=SPI_WREN;    //  Write Enable Command
dm(TXSPI)=r0;

call wait_for_SPIF;
dm(SPICTL)=r15;
bit set flags FLG0;

write_enable.end: rts;

// ---------------------------------------
// WRITE DISABLE SUBROUTINE (1 Byte)
// Disables the Write Enable Latch of the Flash
// Inputs - none
// Outputs- none
// Core sends the command
write_disable:
r0=COMMON_SPI_SETTINGS|TIMOD1|WL8|MSBF;
dm(spi_setting)=r0;
call setup_spi;
```

```
r0=SPI_WRDI;    //  Write Disable Command
dm(TXSPI)=r0;

call wait_for_SPIF;

dm(SPICTL)=r15;
bit set flags FLG0; // Disable SPI

write_disable.end: rts;

// ---------------------------------------
// READ STATUS REGISTER SUBROUTINE (2 Bytes)
// Returns the 8-bit value of the status register.
// Inputs - none
// Outputs- second location of status_register[2],
//          first location is garbage.
// Core sends the command
read_status_register:
r0=COMMON_SPI_SETTINGS|TIMOD2|WL8|MSBF;
dm(spi_setting)=r0;
call setup_spi;

r0=SPI_RDSR;    // Read Status Register command
dm(TXSPI)=r0;

r0=2;
dm(CSPI)=r0;
r0=status_register;
dm(IISPI)=r0;
r0=1;
dm(IMSPI)=r0;
r0=COMMON_SPI_DMA_SETTINGS|SPIRCV;
dm(spi_dma_setting)=r0;

call setup_spi_dma;

dm(SPICTL)=r15;
bit set flags FLG0; // Disable SPI

read_status_register.end: rts;

// ---------------------------------------
// WRITE STATUS REGISTER SUBROUTINE (2 Bytes)
// Writes 8-bit value of the status register.
// Inputs - second location of status_register[2]
//          first location will be corrupted.
// Outputs- none
// 1st word of DMA buffer is the command
write_status_register:
call write_enable;

r0=COMMON_SPI_SETTINGS|TIMOD2|WL8|MSBF;
dm(spi_setting)=r0;
call setup_spi;

r0=SPI_WRSR;    // Write Status Register Command
dm(status_register)=r0;
```

```
r0=2;
dm(CSPI)=r0;
r0=status_register;
dm(IISPI)=r0;

r0=1;
dm(IMSPI)=r0;
r0=COMMON_SPI_DMA_SETTINGS;
dm(spi_dma_setting)=r0;

call setup_spi_dma;

dm(SPICTL)=r15; // Disable SPI
bit set flags FLG0;

write_status_register.end: rts;

// ---------------------------------------
// READ DATA BYTES SUBROUTINE (read_buffer_length 32-bit words)
// Reads the specified number of words from the Flash.
// Inputs - read_base_addr - Starting Flash address
//        - read_buffer_addr - Address of buffer to write
//        - read_buffer_length - Number of 32-bit words to read
//          plus one
// Outputs- Buffer specified by read_buffer_addr will be updated
//          by the specified number of words plus one. The first
//          32-bit word in the buffer is not valid
// Core sends the command
read_data_bytes:
r0=SPI_READ;
dm(curr_command)=r0;
r0=COMMON_SPI_SETTINGS|TIMOD2|WL32;
dm(spi_setting)=r0;
call setup_spi;

r0=dm(read_base_addr);
dm(flash_curr_page)=r0;

call bit_reverse_command;
r0=dm(curr_command);
dm(TXSPI)=r0;

r0=dm(read_buffer_length);
dm(CSPI)=r0;
r0=dm(read_buffer_addr);
dm(IISPI)=r0;
r0=1;
dm(IMSPI)=r0;
r0=COMMON_SPI_DMA_SETTINGS|SPIRCV;
dm(spi_dma_setting)=r0;

call setup_spi_dma;

dm(SPICTL)=r15; // Disable SPI
bit set flags FLG0;

read_data_bytes.end: rts;
```

```
// ----------------------------------------
// HIGH SPEED READ DATA BYTES SUBROUTINE (read_buffer_length 32-bit words)
// Reads the specified number of words from the Flash.
// Inputs - read_base_addr - Starting Flash address
//        - read_buffer_addr - Address of buffer to write
//        - read_buffer_length - Number of 32-bit words to read
//          plus one
// Outputs- Buffer specified by read_buffer_addr will be updated
//          by the specified number of words plus one. The first
//          32-bit word in the buffer is not valid
// Core sends the command
read_data_bytes_hs:// This mode is not implemented.
jump read_data_bytes;

read_data_bytes_hs.end: rts;

// ----------------------------------------
// PAGE PROGRAM SUBROUTINE (write_buffer_length 32-bit words)
// Reads the specified number of words from the Flash.
// Inputs - write_base_addr - Starting Flash address
//        - write_buffer_addr - Address of buffer to read
//        - write_buffer_length - Number of 32-bit words to write
// Outputs- verf_errors - number of errors detected
// 1st word of DMA buffer is the command
page_program:
flash_program:
bit set mode1 CBUFEN;
b5=dm(write_buffer_addr);
b6=flash_data_out+1;
m5=1;
l6=PAGE_LENGTH;
l5=dm(write_buffer_length);
r14=dm(write_buffer_length);

r0=dm(write_base_addr);
dm(flash_curr_page)=r0;

page_write:
call write_enable;

// Page is PAGE_LENGTH bytes, only 1 page can be programmed
// per command. This loop breaks the data into pages for transmission.
get_page:
lcntr = PAGE_LENGTH, do get_page.end until lce;
r0=dm(i5,m5);
get_page.end: dm(i6,m5)=r0;

r0=COMMON_SPI_SETTINGS|TIMOD2|WL32;
dm(spi_setting)=r0;
call setup_spi;

r0=SPI_PP;
dm(curr_command)=r0;
call bit_reverse_command;
r0=dm(curr_command);
dm(flash_data_out)=r0;

// Determine if this is the last page
```

```
r13=PAGE_LENGTH;
r14=r14-r13;
if GT jump (pc,2);
r13=r13+r14;

// Send an entire page, or if this is the last page
// only send remaining words.
r12=r13+1;
r0=r12+1;
dm(CSPI)=r0;
r0=flash_data_out;
dm(IISPI)=r0;
r0=1;
dm(IMSPI)=r0;
r0=COMMON_SPI_DMA_SETTINGS;
dm(spi_dma_setting)=r0;

call setup_spi_dma;

ustat1=dm(SPISTAT);
bit tst ustat1 TXS;
if tf jump (pc,-2);

dm(SPICTL)=r15; // Disable SPI
bit set flags FLG0;

call wait_for_WIP;

// Wait 5 ms for part to be available in all cases.
// 1000000 cycles @ 200 MHz
lcntr=1000000; do (pc,1) until lce;
nop;

r0=dm(flash_curr_page);
dm(read_base_addr)=r0;
r0=flash_data_verf;
dm(read_buffer_addr)=r0;
r0=@flash_data_verf;
dm(read_buffer_length)=r0;
call read_data_bytes;

b1=flash_data_out+1;
b2=flash_data_verf+1;
m1=1;
l1=PAGE_LENGTH;
l2=PAGE_LENGTH;

error_check:
lcntr=r13, do error_check.end until lce;
r0=dm(i1,m1);
r1=dm(i2,m1);
comp(r0,r1);
if eq jump error_check.end;
r0=dm(verf_errors);
r0=r0+1;
dm(verf_errors)=r0;
error_check.end: nop;
```

```
// Update the page address
r1=PAGE_LENGTH*4;
r0=dm(flash_curr_page);
r0=r0+r1;
dm(flash_curr_page)=r0;

// If r14 < 0, no more data
r14 = pass r14;
if gt jump page_write;


page_write.end:
flash_program.end:
page_program.end: rts;

// ----------------------------------------
// SECTOR ERASE SUBROUTINE (4 Bytes)
// Erases the sector of the Flash corresponding to the specified address..
// Inputs - sector_base_addr - Starting 24-bit Flash address
// Outputs- none
// Core sends the command
sector_erase:
call write_enable;

r0=COMMON_SPI_SETTINGS|TIMOD1|WL32;
dm(spi_setting)=r0;
call setup_spi;

r0=dm(sector_base_addr);
dm(flash_curr_page)=r0;

r0=SPI_SE;
dm(curr_command)=r0;
call bit_reverse_command;
r0=dm(curr_command);
dm(TXSPI)=r0;

call wait_for_SPIF;

dm(SPICTL)=r15; // Disable SPI
bit set flags FLG0;

call wait_for_WIP;

sector_erase.end: rts;

// ----------------------------------------
// BULK ERASE SUBROUTINE (1 Byte)
// Erases the entire Flash
// Inputs - none
// Outputs- none
// Core sends the command
bulk_erase:
call write_enable;

r0=COMMON_SPI_SETTINGS|TIMOD1|WL8|MSBF;
dm(spi_setting)=r0;
call setup_spi;
```

```
r3=SPI_BE;
dm(TXSPI)=r3;

call wait_for_SPIF;

dm(SPICTL)=r15; // Disable SPI
bit set flags FLG0;

call wait_for_WIP;

bulk_erase.end: rts;


// ----------------------------------------
// DEEP POWERDOWN SUBROUTINE (1 Byte)
// Places the Flash into Deep Powerdown mode
// Inputs - none
// Outputs- none
// Core sends the command
deep_powerdown:
r0=COMMON_SPI_SETTINGS|TIMOD1|WL8|MSBF;
dm(spi_setting)=r0;
call setup_spi;

r0=SPI_DP;    //  Write Enable Command
dm(TXSPI)=r0;

call wait_for_SPIF;

dm(SPICTL)=r15; // Disable SPI
bit set flags FLG0;

call wait_for_WIP;

deep_powerdown.end: rts;

// ----------------------------------------
// DEEP POWERDOWN RELEASE SUBROUTINE (5 Bytes)
// READ ELECTRONIC SIGNATURE
// Awakens the Flash from Deep Powerdown mode
// Also returns the electronic signature of the Flash
// Inputs - none
// Outputs- Fifth location of electronic_signature
//          First four locations are garbage
// Core sends the command
deep_powerdown_release:
r0=COMMON_SPI_SETTINGS|TIMOD2|WL8|MSBF;
dm(spi_setting)=r0;
call setup_spi;

r0=5;
dm(CSPI)=r0;
r0=electronic_signature;
dm(IISPI)=r0;
r0=1;
dm(IMSPI)=r0;
r0=COMMON_SPI_DMA_SETTINGS|SPIRCV;
```

```
dm(spi_dma_setting)=r0;

r0=SPI_RES;
dm(TXSPI)=r0;

call setup_spi_dma;

dm(SPICTL)=r15; // Disable SPI
bit set flags FLG0;

// Wait 5 us for part to be available in all cases.
// 1000 cycles @ 200 MHz
lcntr=1000, do (pc,1) until lce;
nop;

deep_powerdown_release.end: rts;

// ----------------------------------------
// SETUP SPI SUBROUTINE (1 Byte)
// Sets up the SPI for mode specified in spi_setting
// Inputs - spi_setting
// Outputs- none
setup_spi:

r0=0xFE01;
dm(SPIFLG)=r0;

r0=BAUD_RATE_DIVISOR;
dm(SPIBAUD)=r0;

r0=dm(spi_setting);
dm(SPICTL)=r0;

setup_spi.end: rts;

// ----------------------------------------
// WAIT FOR SPIF SUBROUTINE (1 Byte)
// Polls the SPIF (SPI single word transfer complete) bit
// of SPISTAT until the transfer is complete.
// Inputs - none
// Outputs- none
wait_for_SPIF:
ustat1=dm(SPISTAT);
bit tst ustat1 SPIF;
if not tf jump (pc,-2);
wait_for_SPIF.end: rts;

// ----------------------------------------
// SETUP SPI DMA SUBROUTINE (1 Byte)
// Sets up the SPI DMA for mode specified in spi_dma_setting
// Inputs - spi_dma_setting
// Outputs- none

setup_spi_dma:

ustat1=dm(spi_dma_setting);
dm(SPIDMAC)=ustat1;
```

```
// Wait for SPI DMA to complete using polling------
ustat1=dm(SPIDMAC);
bit tst ustat1 SPIDMAS; //  Check SPI DMA Status bit
IF TF jump (pc,-2);     //  SPIDMAS = 1 when DMA in progress

ustat1=0;   // Disable DMA
dm(SPIDMAC)=ustat1;
setup_spi_dma.end: rts;

// ----------------------------------------
// BIT REVERSE COMMAND SUBROUTINE (1 Byte)
// For booting the SHARC EX requires that the data in the Flash
// be sent LSB first, but all PROM/FLASH devices are MSB first
// so the data must be programmed into the Flash LSBF, but the
// command word for the flash must be bit reversed for the Flash
// to recognize the command in LSBF mode.
// Inputs - curr_command - current command to send
//        - flash_curr_page - address of the page to access
// Outputs- curr_command
bit_reverse_command:
i7=dm(curr_command);
bitrev(i7,0);
r0=i7;
r0=rot r0 by -24;
i7=dm(flash_curr_page);
bitrev(i7,0);
r1=i7;
r2=BITREV_ADDRESS_MASK;
r1=r1 and r2;
r0=r0 or r1;
dm(curr_command)=r0;
bit_reverse_command.end: rts;

// ----------------------------------------
// WAIT FOR WIP SUBROUTINE (1 Byte)
// Polls the WIP (Write In Progress) bit of the Flash's status
// register until the Flash is finished with its access. Accesses
// that are affected by a latency are Page_Program, Sector_Erase,
// and Block_Erase.
// Inputs - none
// Outputs- none
wait_for_WIP:
call read_status_register;

ustat1=dm(status_register+1);
bit tst ustat1 WIP;
if tf jump wait_for_WIP;

wait_for_WIP.end: rts;
```

Listing 3. SPIflash.asm

SPIflash.h

```
/*-----------------------------------------------------------------------------

SPIflash.h
```

```
This file contains the commands specified for the SPI Flash Being used.

------------------------------------------------------------------------------*/
#include <def21262.h>

// There is an error in the definition for the Master/Slave SPI setting
// We will redefine MS to SPIMS to avoid a conflict with a conditional flag
// This will be fixed in VDSP 3.5
// Comment out this definition if you are using VDSP 3.5
#define SPIMS MS

// Filename holding data to be programmed
// Change this as needed, but always include the quotation marks
#define FILENAME "SPIovlyascii.ldr"

// Flash commands
#define SPI_WREN            (0x06)
#define SPI_WRDI            (0x04)
#define SPI_RDSR            (0x05)
#define SPI_WRSR            (0x01)
#define SPI_READ            (0x03)
#define SPI_FAST_READ       (0x0B)
#define SPI_PP              (0x02)
#define SPI_SE              (0xD8)
#define SPI_BE              (0xC7)
#define SPI_DP              (0xB9)
#define SPI_RES             (0xAB)

// Flash Properties
#define WIP                 BIT_0
#define PAGE_LENGTH         64  // (in 32-bit words)

// Application definitions.
#define COMMON_SPI_SETTINGS (SPIEN|SPIMS|SENDZ|CPHASE|CLKPL)
#define COMMON_SPI_DMA_SETTINGS (INTEN|SPIDEN)
#define BITREV_ADDRESS_MASK (0xFFFFFF00)
#define BAUD_RATE_DIVISOR 100
```

Listing 4. SPIflash.h

References

[1] *ADSP-2126x SHARC DSP Core Manual.* Rev 1.0, November 2003. Analog Devices, Inc.

[2] *ADSP-2126x SHARC DSP Peripheral Manual.* Rev 1.0, November 2003. Analog Devices, Inc.

[3] *ST M25P80 Serial Flash Memory Datasheet.* Rev 1.2, December 2003. STMicroelectronics, Inc.

Document History

Revision	Description
Rev 1 – March 02, 2004 by Brian M.	Initial Release