TMS320x281x DSP Boot ROM Reference Guide

Literature Number: SPRU095B
May 2003 − November 2004

Texas Instruments

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third−party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products & application solutions:

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
		Telephony	www.ti.com/telephony
		Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments
Post Office Box 655303 Dallas, Texas 75265

Copyright © 2003, Texas Instruments Incorporated

Contents

1 Boot ROM Overview 8
1.1 Effect of XMPNMC on the Boot ROM 8
1.2 On-Chip ROM Description 8

2 Boot ROM Version and Checksum Information 11

3 CPU Vector Table 12

4 Bootloader Features 14
4.1 Bootloader Functional Operation 14
4.2 Bootloader Device Configuration 16
4.2.1 PLL Multiplier Selection 17
4.2.2 Watchdog Module 17
4.2.3 PIE Configuration 17
4.2.4 Reserved Memory 17
4.3 Bootloader Modes 17
4.4 Bootloader Data Stream Structure 21
4.5 General Structure of Source Program Data Stream in 8-Bit Mode 25
4.6 Basic Transfer Procedure 28
4.7 InitBoot Assembly Routine 30
4.8 SelectBootMode Function 30
4.9 SCI_Boot Function 34
4.10 Parallel_Boot Function (GPIO) 37
4.11 SPI_Boot Function 43
4.12 ExitBoot Assembly Routine 47

5 Building the Boot Table 50

6 Bootloader Code Listing 52

Figures

1. Memory Map of On-Chip ROM 10
2. Vector Table Map 12
3. Bootloader Flow Diagram 15
4. Boot ROM Function Overview 19
5. Jump to Flash Flow Diagram 20
6. Flow Diagram of Jump to H0 SARAM 20
7. Flow Diagram of Jump to OTP Memory 20
8. F2810/12 Boot Loader Basic Transfer Procedure 29
9. Overview of InitBoot Assembly Function 30
10. Overview of the SelectBootMode Function 33
11. Overview of SCI Boot Loader Operation 34
12. Overview of SCI_Boot Function 35
13. Overview of SCIA_CopyData Function 36
14. Overview of SCI_GetWordData Function 37
15. Overview of Parallel GPIO Boot Loader Operation 37
16. Parallel GPIO Boot loader Handshake Protocol 38
17. Parallel GPIO Mode Overview 39
18. Parallel GPIO Mode − Host Transfer Flow 40
19. Overview of Parallel_CopyData Function 41
20. Parallel GPIO Boot Loader Word Fetch 42
21. SPI Loader 43
22. Data Transfer From EEPROM Flow 45
23. Overview of SPIA_CopyData Function 46
24. Overview of SPIA_GetWordData Function 47
25. ExitBoot Procedure Flow 48

Tables

1. Memory Addresses 11
2. Vector Locations 13
3. Configuration for Device Modes 17
4. Boot Mode Selection 18
5. General Structure Of Source Program Data Stream In 16-Bit Mode 22
6. LSB/MSB Loading Sequence in 8-Bit Data Stream 25
7. GPIO Pin Status 31
8. SPI 8-Bit Data Stream 43
9. CPU Register Values 49
10. Boot-Loader Options 51

This page intentionally left blank.

Boot ROM

This reference guide is applicable for the Boot ROM found on the TMS320x281x generation of processors in the TMS320C2000 platform. This includes all Flash-based, ROM-based, and RAM-based devices within the 281x generation.

The Boot ROM is factory-programmed with boot-loading software. Boot-mode signals (general purpose I/Os) are used to tell the bootloader software what mode to use on power up. The 281x Boot ROM also contains standard math tables, such as SIN/COS waveforms for use in IQ math related algorithms.

This guide describes the purpose and features of the bootloader. It also describes other contents of the device on-chip boot ROM and identifies where all of the information is located within that memory.

1 Boot ROM Overview

The boot ROM on the F281x, C281x, and R281x devices is a 4K x 16 block located in memory from 0x3F F000 − 0x3F FFC0. This memory block is mapped only when the MPNMC status bit in the XINTCNF2 register is 0. The same Boot ROM is included in Flash, ROM, and RAM 281x devices.

1.1 Effect of XMPNMC on the Boot ROM

In this document XMPNMC refers to the input signal to the device while MPNMC refers to the status bit in the external interface (XINTF) configuration register XINTFCNF2. On devices without an XINTF, the XMPNMC input signal is tied low internal to the device.

The MPNMC status bit in the XINTFCNF2 register switches the device between microprocessor and microcomputer mode. When high, XINTF Zone 7 is enabled on the external interface and the internal boot ROM is disabled. When low, XINTF Zone 7 is disabled from the external interface, and the on-chip boot ROM memory can be accessed instead. The XMPNMC input signal is latched into the XINTF configuration register XINTCNF2 on a reset. After reset, the state of the XMPNMC input signal is ignored and you can then modify the state of this mode in software.

On devices with an XINTF, like the F2812, the XMPNMC input signal is available externally and therefore, you can control this signal to boot from the internal boot ROM or from XINTF Zone 7. On devices without the XINTF, such as the F2810, XMPNMC is tied low internal to the device and, therefore, boot ROM is automatically enabled at reset.

The remainder of this document assumes that the XMPNMC input signal is pulled low at reset to boot from internal ROM unless stated otherwise.

1.2 On-Chip ROM Description

On the 281x devices, the 4K x 16 on-chip ROM is factory programmed with the boot-load routine and additional features. Appendix A contains the code for each of the following items:

- Bootloader functions
- Version number, release date and checksum
- Reset vector
- CPU vector table (Used for test purposes only)
- IQmath Tables

  3K x 16 of boot ROM memory is reserved for math tables and future upgrades. These math tables and in the future math functions are intended to help with performance and save RAM space.

The 281x boot ROM includes math tables that are used by the Texas Instruments™ TMS320C28x™ IQmath Library. The 28x IQmath Library is a collection of highly optimized and high precision mathematical functions for C/C++ programmers to seamlessly port a floating-point algorithm into fixed-point code on TMS320C28x devices.

These routines are typically used in computational-intensive real-time applications where optimal execution speed and high accuracy is critical. By using these routines you can achieve execution speeds that are considerably faster than equivalent code written in standard ANSI C language. In addition, by providing ready-to-use high precision functions, the TI IQmath Library can shorten significantly your DSP application development time. The *28x IQmath Library* (literature number SPRC087), can be downloaded from the TI website.

The following math tables are included in the 281x Boot ROM.

- Sin/Cos Table:

  Table size: 1282 words

  Q format: Q30

  Contents: 32-bit samples for one and a quarter period sin wave

  This is useful for accurate sin wave generation and 32-bit FFTs. This can also be used for 16-bit math, just skip over every second value.

- Normalized Inverse Table:

  Table size: 528 words

  Q format: Q29

  Contents: 32-bit normalized inverse samples plus saturation limits

  This table is used as an initial estimate in the Newton-Raphson inverse algorithm. By using a more accurate estimate the convergence is quicker and hence cycle time is faster.

- Normalized Square Root Table:

  Table size: 274 words

  Q format: Q30

  Contents: 32-bit normalized inverse square root samples plus saturation

  This table is used as an initial estimate in the Newton-Raphson square-root algorithm. By using a more accurate estimate the convergence is quicker and hence cycle time is faster.

- Normalized Arctan Table:

  Table size: 452 words

  Q format: Q30

  Contents 32-bit 2nd order coefficients for line of bset fit plus normalization table

  This table is used as an initial estimate in the Arctan iterative algorithm. By using a more accurate estimate the convergence is quicker and hence cycle time is faster.

- Rounding and Saturation Table:

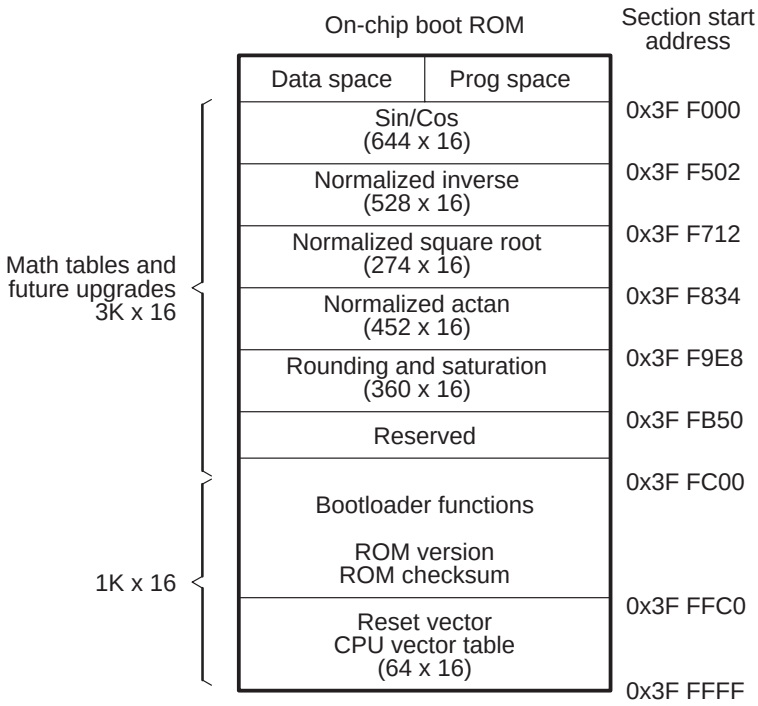
  Table size: 360 words

  Q format: Q30

  Contents 32-bit rounding and saturation limits for various Q values

Figure 1 shows the memory map of the on-chip ROM for the F2810/12. The memory block is 4Kx16 in size and is located at 0x3F F000 − 0x3F FFFF in both program and data space when the MPNMC status bit is low.

Figure 1. Memory Map of On-Chip ROM



2 Boot ROM Version and Checksum Information

The boot ROM contains its own version number located at address 0x3F FFBA. This version number starts at 1 and will be incremented any time the boot ROM code is modified. The next address, 0x3F FFBB contains the month and year (MM/YY in decimal) that the boot code was released. The next four memory locations contain a checksum value for the boot ROM. Taking a 64-bit summation of all addresses within the ROM, except for the checksum locations, generates this checksum.

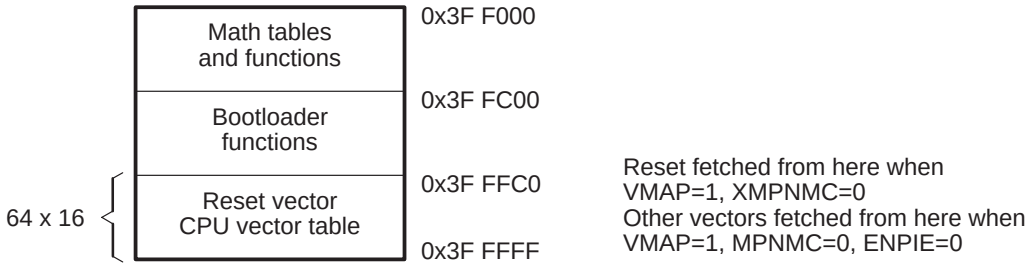
Table 1. Memory Addresses

Address	Contents
0x3F FFBA	Boot ROM Version Number
0x3F FFBB	MM/YY of release (in decimal)
0x3F FFBC	Least significant word of checksum
0x3F FFBD	...
0x3F FFBE	...
0x3F FFBF	Most significant word of checksum

3 CPU Vector Table

A CPU vector table resides in boot ROM memory from address 0x3F FFC0 − 0x3F FFFF. This vector table is active when VMAP = 1, ENPIE = 0 (PIE vector table disabled) and MPNMC = 0 (internal Boot ROM memory enabled, XINTF Zone 7 disabled).

Figure 2. Vector Table Map



Notes: 1) The VMAP bit is located in Status Register 1 (ST1). On the 281x devices, VMAP is always 1 on reset. It can be changed after reset by software, however the normal operating mode will be to leave VMAP = 1.

2) On the 2812, XMPNMC is available externally; on the 2810 and 2811, it is tied low internally. On reset, the state of the XMPNMC input signal is latched into the MPNMC status bit in the XINTCNF2 register where it can be changed by software.

3) The ENPIE bit is located in the PIECTRL register. The default state of this bit at reset is 0, which disables the Peripheral Interrupt Expansion block (PIE).

The only vector that will normally be handled from the internal boot ROM memory is the reset vector located at 0x3F FFC0. The reset vector is factory programmed to point to the InitBoot function. This function starts the boot load process. A series of checking operations is performed on General Purpose I/O (GPIO I/O) pins to determine which boot mode to use. This boot mode selection is described in the next section of this document.

The remaining vectors in the ROM are not used during normal operation on the 281x devices. After the boot process is complete, you should initialize the Peripheral Interrupt Expansion (PIE) vector table and enable the PIE block. From that point on, all vectors, except reset, will be fetched from the PIE module and not the CPU vector table shown here.

For TI silicon debug and test purposes the vectors located in the boot ROM memory point to locations in the M0 SARAM block as described in the following table. During silicon debug, you can program the specified locations in M0 with branch instructions to catch any vectors fetched from boot ROM. This is not required for normal device operation.

Table 2. Vector Locations

Vector	Location in Boot ROM	Contents (i.e., points to)	Vector	Location in Boot ROM	Contents (ie points to)
RESET	0x3F FFC0	InitBoot (0x3F FC00)	RTOSINT	0x3F FFE0	0x00 0060
INT1	0x3F FFC2	0x00 0042	Reserved	0x3F FFE2	0x00 0062
INT2	0x3F FFC4	0x00 0044	NMI	0x3F FFE4	0x00 0064
INT3	0x3F FFC6	0x00 0046	ILLEGAL	0x3F FFE6	0x00 0066
INT4	0x3F FFC8	0x00 0048	USER1	0x3F FFE8	0x00 0068
INT5	0x3F FFCA	0x00 004A	USER2	0x3F FFEA	0x00 006A
INT6	0x3F FFCC	0x00 004C	USER3	0x3F FFEC	0x00 006C
INT7	0x3F FFCE	0x00 004E	USER4	0x3F FFEE	0x00 006E
INT8	0x3F FFD0	0x00 0050	USER5	0x3F FFF0	0x00 0070
INT9	0x3F FFD2	0x00 0052	USER6	0x3F FFF2	0x00 0072
INT10	0x3F FFD4	0x00 0054	USER7	0x3F FFF4	0x00 0074
INT11	0x3F FFD6	0x00 0056	USER8	0x3F FFF6	0x00 0076
INT12	0x3F FFD8	0x00 0058	USER9	0x3F FFF8	0x00 0078
INT13	0x3F FFDA	0x00 005A	USER10	0x3F FFFA	0x00 007A
INT14	0x3F FFDC	0x00 005C	USER11	0x3F FFFC	0x00 007C
DLOGINT	0x3F FFDE	0x00 005E	USER12	0x3F FFFE	0x00 007E

4 Bootloader Features

This section describes in detail the boot mode selection process, as well as the specifics of boot loader operation.

4.1 Bootloader Functional Operation

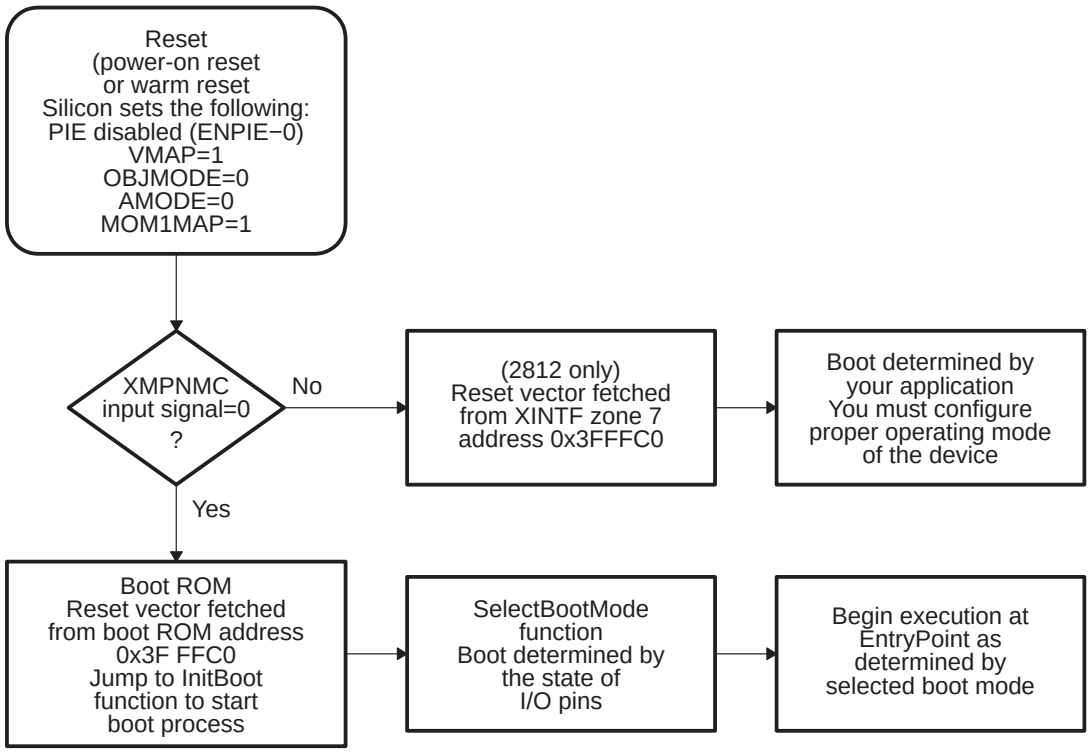
The 281x bootloader is the program located in the 281x ROM that is executed following a reset when the device is in microcomputer mode.

The bootloader is used to transfer code from an external source into internal or external interface (XINTF) memory following power up. This allows code to reside in slow non-volatile memory externally, and be transferred to high-speed memory to be executed.

The bootloader provides a variety of different ways to download code to accommodate different system requirements. These modes are only available if the processor boots in microcomputer mode (XMPNMC device input signal = low).

The bootloader uses various GPIO signals to determine which boot mode to use. The boot mode selection process as well as the specifics of each boot loader operation are described in the remainder of this document. Figure 3 shows the basic bootloader flow.

Figure 3. Bootloader Flow Diagram



Note: On the F2810 the XMPNMC input signal is tied low internally on the device, and therefore boot from reset is always from the internal boot ROM.

At reset, the value of the XMPNMC pin is sampled. The state of this pin determines whether boot ROM or XINTF Zone 7 is enabled at reset.

If XMPNMC = 1 (Micro-Processor Mode) then Zone 7 is enabled and the reset vector will be fetched from external memory. In this case, you must ensure that the reset vector points to a valid memory location for code execution. This option is only available on devices with an XINTF.

If XMPNMC = 0 (Micro-Computer Mode) then the boot ROM memory is enabled and XINTF Zone 7 is disabled. In this case, the reset vector is fetched from the internal boot ROM memory. All devices without an XINTF module have the XMPNMC signal tied low internal to the device such that the boot ROM memory is always enabled at reset.

The reset vector in boot ROM redirects program execution to the InitBoot function. After performing device initialization the boot loader will check the state of GPIO pins to determine which boot mode you want to execute. Options include: Jump to Flash, Jump to H0 SARAM, Jump to OTP or call one of the on-chip boot loading routines.

After the selection process and if required boot loading is complete, the processor will continue execution at an entry point determined by the boot mode selected. If a boot loader was called, then the input stream loaded by the peripheral determines this entry address. This data stream is described in section 2.4.3. If, instead, you choose to boot directly to Flash, OTP, or H0 SARAM, the entry address is predefined for each of these memory blocks.

The following sections discuss in detail the different boot modes available and the process used for loading data code into the device.

4.2 Bootloader Device Configuration

At reset, any 28x™ CPU-based device is in 27x™ object-compatible mode. It is up to the application to place the device in the proper operating mode before execution proceeds.

On the 28x devices, when booting from the internal boot ROM (XMPNMC = 0), the device is configured for 28x operating mode by the boot ROM software. You are responsible for any additional configuration required.

For example:

- If your application includes C2xLP™ source, then you are responsible for configuring the device for C2xLP source compatibility prior to execution of code generated from C2xLP source.
- If you boot from external memory (MPNMC = 1) then the application must configure the device for 28x operating mode or C2xLP source compatible mode as appropriate.

The configuration required for each operating mode is summarized in Table 3.

27x, 28x, and C2xLP are trademarks of Texas Instruments.

Table 3. Configuration for Device Modes

	281x C27x Mode (Reset)	28x Mode	C2xLP Source Compatible Mode
OBJMODE	0	1	1
AMODE	0	0	1
PAGE0	0	0	0
M0M1MAP†	1	1	1
Other Settings			SXM = 1 C = 1 SPM = 0

† Normally for 27x compatibility, the M0M1MAP would be 0. On the 281x; however, it is tied off high internally. Thus at reset M0M1MAP is always configured for 28x mode on these devices.

4.2.1 PLL Multiplier Selection

The Boot ROM does not change the state of the PLL. Note that the PLL multiplier is not affected by a reset from the debugger. Therefore, a boot that is initialized from a reset from Code Composer Studio™ may be at a different speed than booting by pulling the external reset line ($\overline{\text{XRS}}$) low.

4.2.2 Watchdog Module

When branching directly to flash, H0 single-access RAM (SARAM), or one-time-programmable (OTP) memory, the watchdog will not be touched. In the other boot modes, the watchdog will be disabled before booting and then re-enabled and cleared before branching to the final destination address.

4.2.3 PIE Configuration

The boot modes do not enable the PIE. It is left in its default state, which is disabled.

4.2.4 Reserved Memory

The first 80 words of the M1 memory block (address 0x400 − 0x450) are reserved for stack use during the boot load process. If code is bootloaded into this region there is no error checking to prevent it from corrupting the Boot ROM stack.

4.3 Bootloader Modes

To accommodate different system requirements, the 281x boot ROM offers a variety of different boot modes. This section describes the different boot

modes and gives brief summary of their functional operation. The state of four GPIO pins are used to determine the boot mode desired as shown in Table 4.

Table 4. Boot Mode Selection

GPIOF4	**GPIOF12**	**GPIOF3**	**GPIOF2**	
(SCITXDA)	(MDXA)	(SPISTEA)	(SPICLK)	
PU	No PU	No PU	No PU	Mode Selected
1	x	x	x	Jump to Flash address 0x3F 7FF6 You must have programmed a branch instruction here prior to reset to re-direct code execution as desired.
0	1	x	x	Call SPI_Boot to load from an external serial SPI EEPROM
0	0	1	1	Call SCI_Boot to load from SCI-A
0	0	1	0	Jump to H0 SARAM address 0x3F 8000
0	0	0	1	Jump to OTP address 0x3D 7800
0	0	0	0	Call Parallel_Boot to load from GPIO Port B

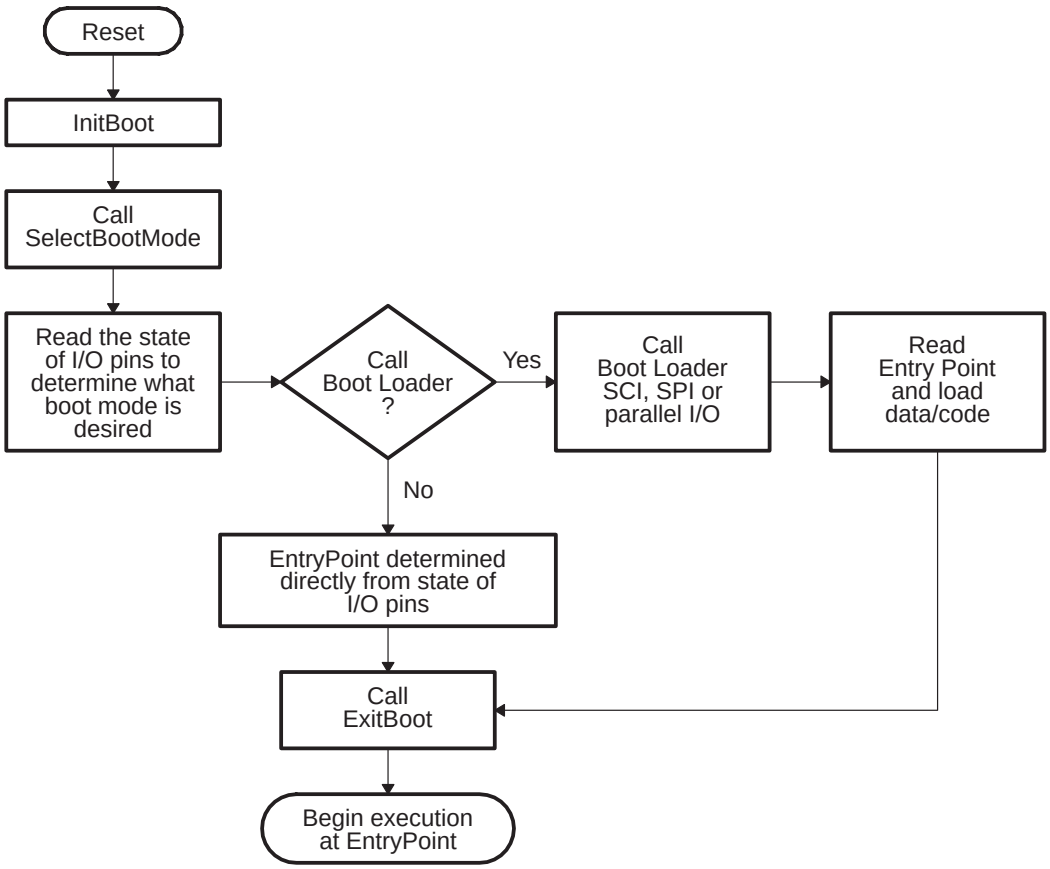
Notes: 1) PU = pin has an internal pullup No PU = pin does not have an internal pullup

2) You must take extra care due to any affect toggling SPICLK in order to select a boot mode may have on external logic.

3) If the boot mode selected is Flash, H0 or OTP, then no external code is loaded by the bootloader.

Figure 4 shows an overview of the boot process. Each step is described in greater detail in following sections.

Figure 4. Boot ROM Function Overview



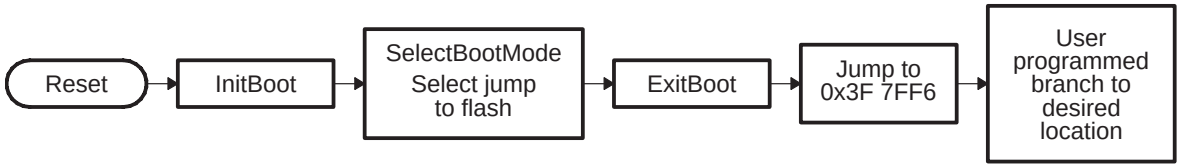
The following boot modes do not call a boot loader. Instead, they jump to a predefined location in memory:

- Jump to branch instruction in Flash Memory:

  In this mode, the boot ROM software will configure the device for ‘28x operation and then branch directly to location 0x3F 7FF6 in Flash memory. This location is just before the 128-bit code security module (CSM) password locations. You are required to have previously programmed a branch instruction at location 0x3F 7FF6 that will redirect code execution to either a custom boot-loader or the application code.

  On 281x RAM devices, the boot-to-Flash option jumps to reserved memory and should not be used. On 281x ROM devices, the boot-to-Flash option jumps to the location 0x3F 7FF6 in ROM.

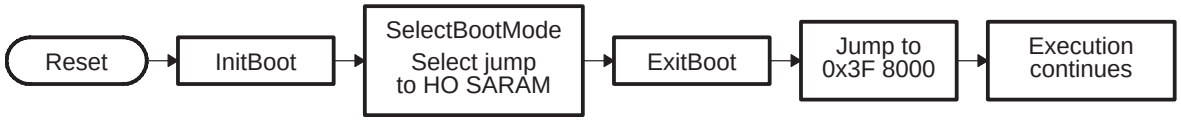
Figure 5. Jump to Flash Flow Diagram



- Jump to H0 SARAM

  In this mode, the boot ROM software will configure the device for ‘28x operation and then branch directly to 0x3F 8000; the first address in the H0 SARAM memory block.

Figure 6. Flow Diagram of Jump to H0 SARAM

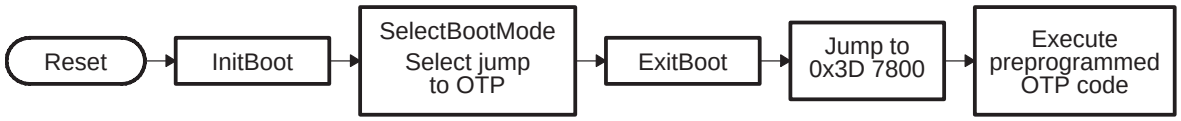


- Jump to OTP Memory

  In this mode, the boot ROM software will configure the device for C28x operation and then branch directly to at 0x3D 7800; the first address in the OTP memory block.

  On 281x ROM devices, the boot-to-OTP option jumps to address 0x3D 7800 in OTP. On 281x RAM devices, the boot-to-OTP option jumps to reserved memory and should not be used.

Figure 7. Flow Diagram of Jump to OTP Memory



- Standard Serial Boot Mode (SCI):

  In this mode, the boot ROM will load code to be executed into on-chip memory via the SCI-A port.

- SPI EEPROM Boot Mode:

  In this mode, the boot ROM will load code and data into on-chip memory from an external EEPROM via the SPI port.

- Boot from GPIO Port:

  In this mode, the boot ROM uses a GPIO port B to load code and data from an external source. This mode supports both 8-bit and 16-bit data

streams. Since this mode requires a number of GPIO pins, it would typically be used for downloading code only for flash programming when the device is connected to a platform explicitly for flash programming and not a target board.

4.4 Bootloader Data Stream Structure

The following two tables and associated examples show the structure of the data stream incoming to the boot loader. The basic structure is the same for all the boot loaders and is based on the C54x source data stream generated by the C54x hex utility. The C28x hex utility has been updated to support this structure. All values in the data stream structure are in hex.

The first 16-bit word in the data stream is known as the key value. The key value is used to tell the boot loader the width of the incoming stream: 8 or 16 bits. Note that not all boot loaders will accept both 8 and 16-bit streams. Please refer to the detailed information on each loader for the valid data stream width. For an 8-bit data stream, the key value is 0x08AA and for a 16-bit stream it is 0x10AA. If a boot loader receives an invalid key value, then the load is aborted. In this case, the entry point for the Flash memory will be used.

The next 8 words are used to initialize register values or otherwise enhance the boot loader by passing values to it. If a boot loader does not use these values then they are reserved for future use and the boot loader simply reads the value and then discards it. Currently only the SPI boot loader uses one word to initialize registers.

The next 10^{th} and 11^{th} words comprise the 22-bit entry point address. This address is used to initialize the PC after the boot load is complete. This address is most likely the entry point of the program downloaded by the boot loader.

The twelfth word in the data stream is the size of the first data block to be transferred. The size of the block is defined for both 8 and 16-bit data stream formats as the number of 16-bit words in the block. For example, to transfer a block of 20 8-bit data values from an 8-bit data stream, the block size would be 0x000A to indicate 10 16-bit words.

The next two words tell the loader the destination address of the block of data. Following the size and address will be the 16-bit words that makeup that block of data.

This pattern of block size/destination address repeats for each block of data to be transferred. Once all the blocks have been transferred, a block size of 0x0000 signals to the loader that the transfer is complete. At this point the

loader will return the entry point address to the calling routine which in turn will cleanup and exit. Execution will then continue at the entry point address as determined by the input data stream contents.

Table 5. General Structure Of Source Program Data Stream In 16-Bit Mode

Word	Contents
1	10AA (KeyValue for memory width = 16bits)
2	Register initialization value or reserved for future use
3	Reserved for future use
	…
9	Reserved for future use
10	Entry point PC[22:16]
11	Entry point PC[15:0]
12	Block size (number of words) of the first block of data to load. If the block size is 0, this indicates the end of the source program. Otherwise another section follows.
13	Destination address of first block Addr[31:16]
14	Destination address of first block Addr[15:0]
15	First word of the first block in the source being loaded
.	…
.	…
.	Last word of the first block of the source being loaded
.	Block size of the 2nd block to load.
.	Destination address of second block Addr[31:16]
.	Destination address of second block Addr[15:0]
.	First word of the second block in the source being loaded
.	…
.	…
.	Last word of the second block of the source being loaded
	…
	…

Table 5. General Structure Of Source Program Data Stream In 16-Bit Mode (Continued)

Word	Contents
.	Block size of the last block to load
.	Destination address of last block Addr[31:16]
.	Destination address of last block Addr[15:0]
.	First word of the last block in the source being loaded
.	…
.	…
.	Last word of the last block of the source being loaded
n	0000h − indicates end of the source program

Example 1. Data stream structure 16bit:

```
10AA   ; 0x10AA 16-bit key value
0000   ;      8 reserved words
0000
0000
0000
0000
0000
0000
0000
003F   ; 0x003F8000 EntryAddr, starting point after boot load completes
8000
0005   ; 0x0005 – First block consists of 5 16-bit words
003F   ; 0x003F9010 – First block will be loaded starting at 0x3F9010
9010
0001   ; Data loaded = 0x0001 0x0002 0x0003 0x0004 0x0005
0002
0003
0004
0005
0002   ; 0x0002 – 2nd block consists of 2 16-bit words
003F   ; 0x003F8000 – 2nd block will be loaded starting at 0x3F8000
8000
7700   ; Data loaded = 0x7700 0x7625
7625
0000   ; 0x0000 – Size of 0 indicates end of data stream
```

After load has completed the following memory values will have been initialized as follows:

```
Location  Value
0x3F9010  0x0001
0x3F9011  0x0002
0x3F9012  0x0003
0x3F9013  0x0004
0x3F9014  0x0005
0x3F8000  0x7700
0x3F8001  0x7625
PC    Begins execution at 0x3F8000
```

4.5 General Structure of Source Program Data Stream in 8-Bit Mode

In 8-bit mode, the LSB of the word is sent first followed by the MSB. The boot loaders take this into account when loading an 8-bit data stream.

Table 6. LSB/MSB Loading Sequence in 8-Bit Data Stream

Byte	Contents
1	LSB = AA (KeyValue for memory width = 8 bits)
2	MSB = 08h (KeyValue for memory width = 8 bits)
3	LSB = Register initialization value or reserved for future use
4	MSB= Register initialization value or reserved for future use
…	…
17	LSB = reserved for future use
18	MSB= reserved for future use
19	LSB: Upper half of Entry point PC[23:16]
20	MSB: Upper half of Entry point PC[31:24] (Note: Always 0x00)
21	LSB: Lower half of Entry point PC[7:0]
22	MSB: Lower half of Entry point PC[15:8]
23	LSB: Block size in words of the first block to load. If the block size is 0, this indicates the end of the source program. Otherwise another block follows. For example, a block size of 0x000A would indicate 10 words or 20 bytes in the block.
24	MSB: block size
25	LSB: Upper half of Destination address of first block Addr[23:16]
26	MSB: Upper half of Destination address of first block Addr[31:24]
27	LSB: Lower half of Destination address of first block Addr[7:0]
28	MSB: Lower half of Destination address of first block Addr[15:8]
29	LSB: First word of the first block being loaded
30	MSB: First word of the first block being loaded
.	…
	…
.	LSB: Last word of the first block of the source being loaded

Table 6. LSB/MSB Loading Sequence in 8-Bit Data Stream (Continued)

Byte	Contents
.	MSB: Last word of the first block of the source being loaded
.	LSB: Block size of the second block
.	MSB: Block size of the second block
.	LSB: Upper half of Destination address of second block Addr[23:16]
.	MSB: Upper half of Destination address of second block Addr[31:24]
.	LSB: Lower half of Destination address of second block Addr[7:0]
.	MSB: Lower half of Destination address of second block Addr[15:8]
.	LSB: First word of the second block being loaded
.	MSB: First word of the second block being loaded
.	…
.	…
.	LSB: Last word of the second block of the source being loaded
.	MSB: Last word of the second block of the source being loaded
.	…
.	…
.	…
.	LSB: Block size of the last block
.	MSB: Block size of the last block
.	LSB: Upper half of Destination address of last block Addr[23:16]
.	MSB: Upper half of Destination address of last block Addr[31:24]
.	LSB: Lower half of Destination address of last block Addr[7:0]
.	MSB: Lower half of Destination address of last block Addr[15:8]
.	LSB: First word of the last block being loaded
.	MSB: First word of the last block being loaded…
.	…
.	…

Table 6. LSB/MSB Loading Sequence in 8-Bit Data Stream (Continued)

Byte	Contents
.	LSB: Last word of the last block of the source being loaded
.	MSB: Last word of the last block of the source being loaded
n	LSB: 00h
n+1	MSB: 00h − indicates the end of the source

Example 2. Example 1: Data stream structure 8 bit:

```
AA      ; 0x08AA 8-bit key value
08
00      ; 8 reserved words
00
00
00
00
00
00
00
00
00
00
00
00
00
3F      ; 0x003F8000 EntryAddr, starting point after boot load completes
00
00
80
05      ; 0x0005 – First block consists of 5 16-bit words
00
3F      ; 0x003F9010 – First block will be loaded starting at 0x3F9010
00
10
90
01      ; Data loaded = 0x0001 0x0002 0x0003 0x0004 0x0005
00
02
00
03
00
04
00
05
00
```

```
02     ; 0x0002 – 2nd block consists of 2 16-bit words
00
3F     ; 0x003F8000 – First block will be loaded starting at 0x3F8000
00
00
80
00     ; Data loaded = 0x7700 0x7625
77
25
76
00     ; 0x0000 – Size of 0 indicates end of data stream
00
```

After load has completed the following memory values will have been initialized as follows:

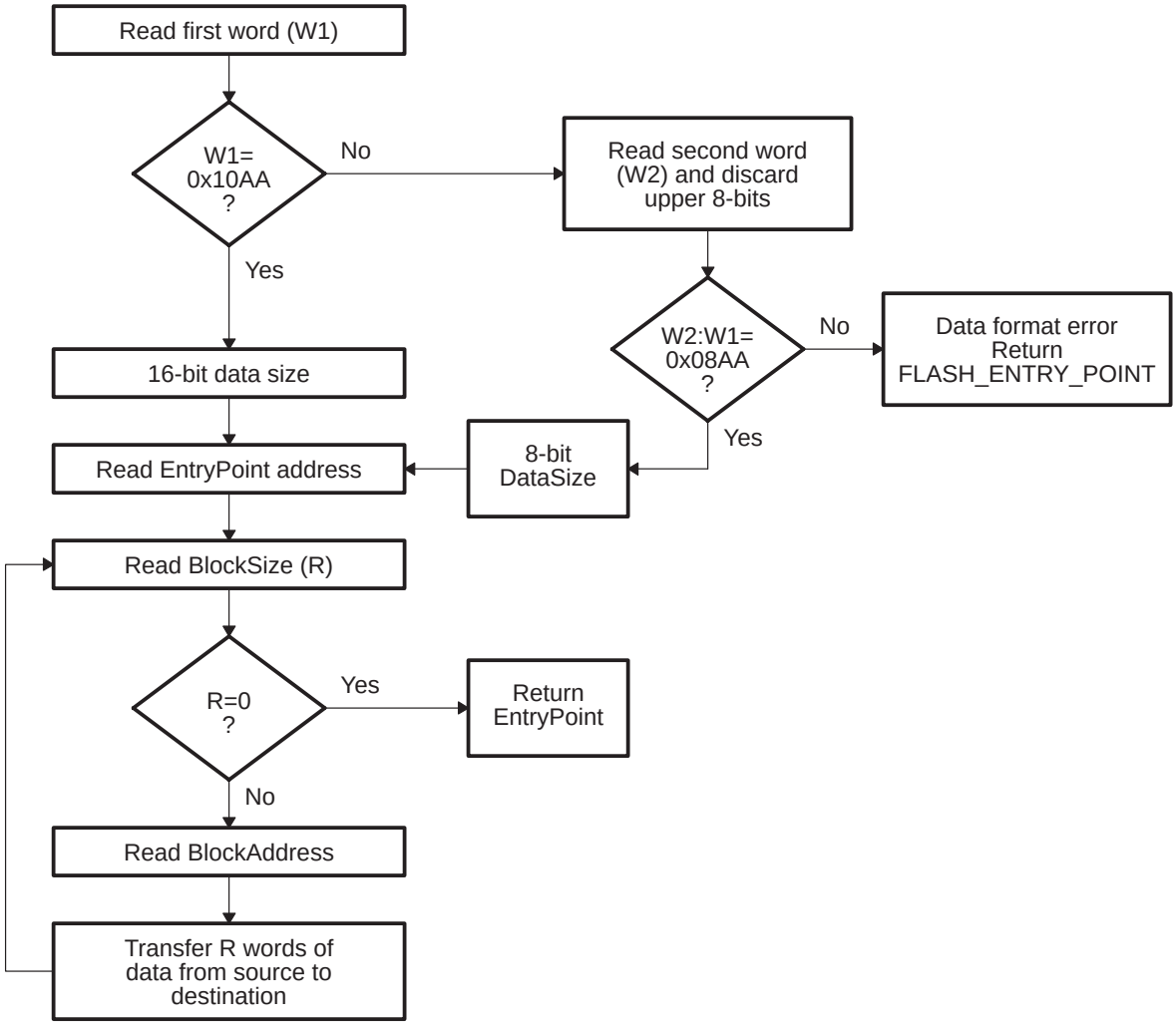
```
Location  Value
0x3F9010  0x0001
0x3F9011  0x0002
0x3F9012  0x0003
0x3F9013  0x0004
0x3F9014  0x0005
0x3F8000  0x7700
0x3F8001  0x7625
PC        Begins execution at 0x3F8000
```

4.6 Basic Transfer Procedure

Figure 8 illustrates the basic process a boot loader uses to determine whether 8-bit or 16-bit data stream has been selected, transfer that data, and start program execution. This process occurs after the boot loader finds the valid boot mode selected by the state of GPIO pins.

The loader first compares the first value sent by the host against the 16-bit key value of 0x10AA. If the value fetched does not match then the loader will read a second value. This value will be combined with the first value to form a word. This will then be checked against the 8-bit key value of 0x08AA. If the loader finds that the header does not match either the 8-bit or 16-bit key value, or if the value is not valid for the given boot mode then the load will abort. In this case the loader will return the entry point address for the flash to the calling routine.

Figure 8. F2810/12 Boot Loader Basic Transfer Procedure



Notes: 1) 8-bit and 16-bit transfers are not valid for all boot modes. Refer to the info specific to a particular boot loader for any limitations.

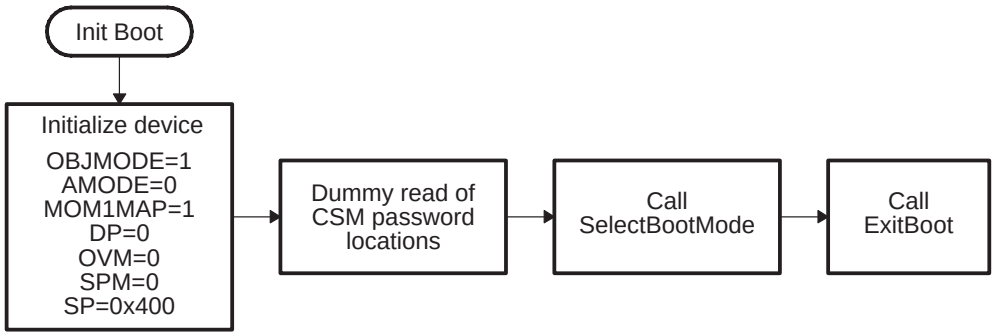
2) In 8-bit mode the LSB of the 16-bit word is read first followed by the MSB.

4.7 InitBoot Assembly Routine

The first routine called after reset is the InitBoot assembly routine. This routine initializes the device for operation in C28x object mode. Next it performs a dummy read of the Code Security Module (CSM) password locations. If the CSM passwords are erased (all 0xFFFFs) then this has the effect of unlocking the CSM. Otherwise the CSM will remain locked and this dummy read of the password locations will have no effect. This can be useful if you have a new device that you want to boot load.

After the dummy read of the CSM password locations, the InitBoot routine calls the SelectBootMode function. This function will then determine the type of boot mode desired by the state of certain GPIO pins. Once the boot is complete, the SelectBootMode function passes back the EntryAddr to the InitBoot function. InitBoot then calls the ExitBoot routine that then restores CPU registers to their reset state and exits to the EntryAddr that was determined by the boot mode.

Figure 9. Overview of InitBoot Assembly Function



4.8 SelectBootMode Function

To determine the desired boot mode, the SelectBootMode function examines the state of 4 GPIO pins as shown in Table 7. These pins are all part of GPIO Port F and are normally output pins when used in their peripheral function (shown in parenthesis).

Table 7. GPIO Pin Status

GPIOF4	GPIOF12	GPIOF3	GPIOF2	
(SCITXDA)	(MDXA)	(SPISTEA)	(SPICLK)	
PU	No PU	No PU	No PU	Mode Selected
1	x	x	x	Jump to Flash address 0x3F 7FF6 You must have programmed a branch instruction here prior to reset to redirect code execution as desired
0	1	x	x	Call SPI_Boot to load from external EEPROM
0	0	1	1	Call SCI_Boot to load from SCI-A
0	0	1	0	Jump to H0 SARAM address 0x3F 8000
0	0	0	1	Jump to OTP address 0x3D 7800
0	0	0	0	Call Parallel_Boot to load from GPIO Port B

Notes:
1) When booting directly to Flash is assumed that you have previously programmed a branch statement at 0x3F 7FF6 to redirect program flow as desired.
2) When booting directly to OTP or H0, it is assumed that you have previously programmed or loaded code starting at the entry point location.
3) x = don't care
4) You must take extra care due to any affect toggling SPICLK in order to select a boot mode may have on external logic.
5) PU = pin has an internal pull-up resister. No PU = pin does not have an internal pull-up resistor

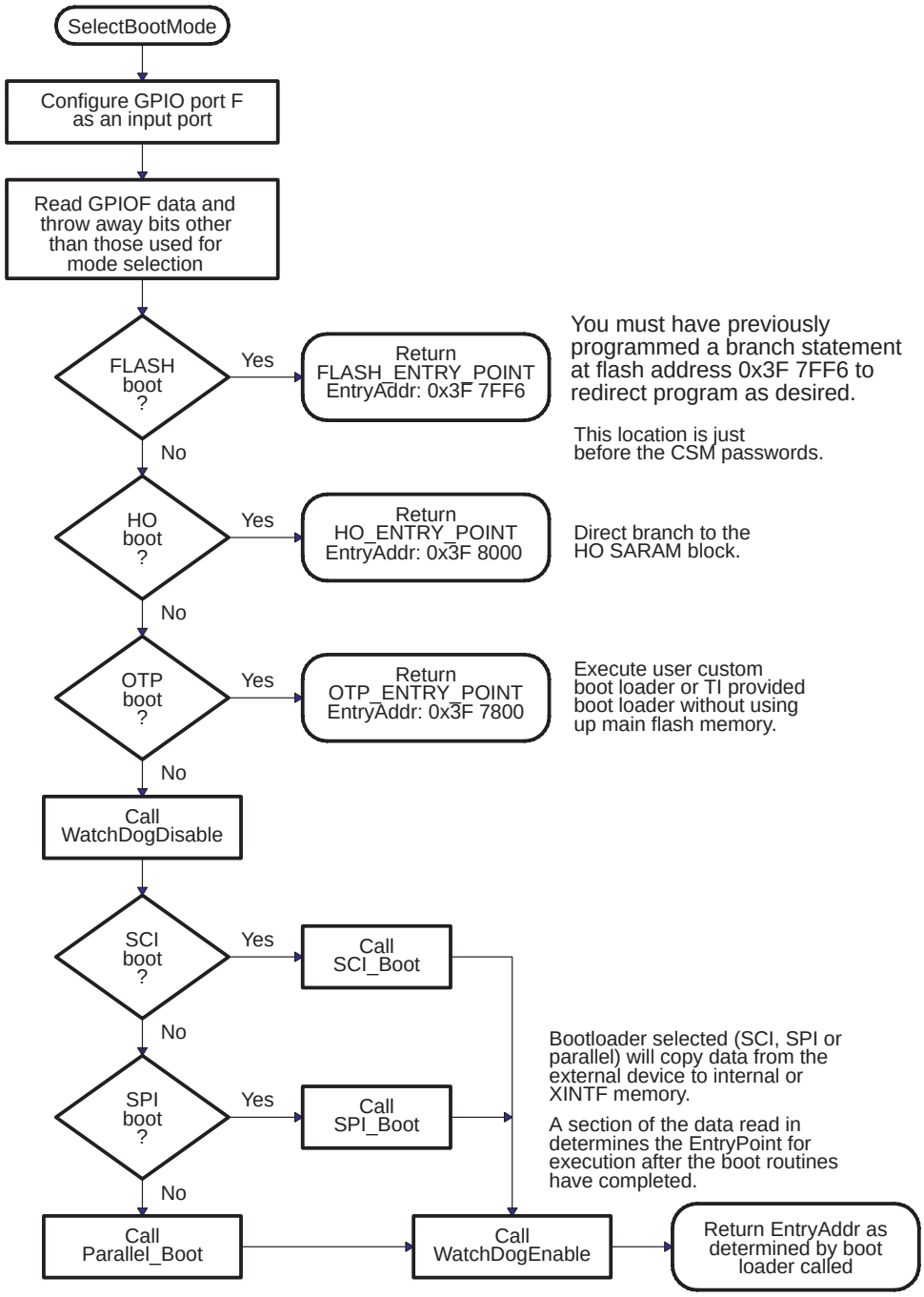
For a boot mode to be selected, the pins corresponding to the desired boot mode have to be pulled low or high until the selection process completes. Note that the state of the selection pins is not latched at reset; they are sampled some cycles later in the SelectBootMode function.

The SelectBootMode routine disables the watchdog before calling the SCI, SPI or Parallel boot loader. If a boot loader is not going to be called, then the watchdog is left untouched. **The boot loaders do not service the watchdog and assume that it is disabled.** Before exiting the SelectBootMode routine will re-enable the watchdog and reset its timer.

When selecting a boot mode, the pins should be pulled high or low through a weak pulldown or weak pull-up such that the DSP can drive them to a new state when required. For example, if you wanted to boot from the SCI one of the pins you pull low is the SCITXDA pin. This pulldown must be weak so that when the SCI boot process begins the DSP will be able to properly transmit through the TX pin. Likewise for the remaining boot mode selection pins.

You must take extra care if you use SPICLK to select a boot mode. Toggling of this signal may have an affect on external logic and this must be taken into account.

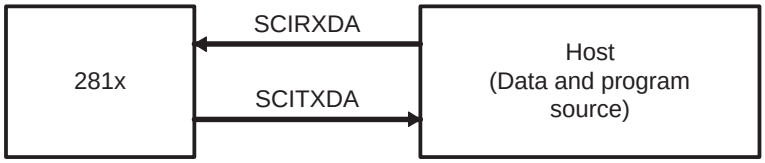
Figure 10. Overview of the SelectBootMode Function



4.9 SCI_Boot Function

The SCI boot mode asynchronously transfers code from SCI-A to internal or XINTF memory. This boot mode only supports an incoming 8-bit data stream and follows the same data flow as outlined in Example 1.

Figure 11. Overview of SCI Boot Loader Operation



The F2810/12 communicates with the external host device by communication through the SCI-A Peripheral. The autobaud feature of the SCI port is used to lock baud rates with the host. For this reason the SCI loader is very flexible and you can use a number of different baud rates to communicate with the DSP.

After each data transfer, the DSP will echo back the 8-bit character received to the host. In this manner, the host can perform checks that each character was received by the DSP.

At higher baud rates, the slew rate of the incoming data bits can be effected by transceiver and connector performance. While normal serial communications may work well, this slew rate may limit reliable auto-baud detection at higher baud rates (typically beyond 100kbaud) and cause the auto-baud lock feature to fail. To avoid this, the following is recommended:

1) Achieve a baud-lock between the host and 28x SCI boot loader using a lower baud rate.

2) Load the incoming 28x application or custom loader at this lower baud rate.

3) The host may then handshake with the loaded 28x application to set the SCI baud rate register to the desired high baud rate.

Figure 12. Overview of SCI_Boot Function

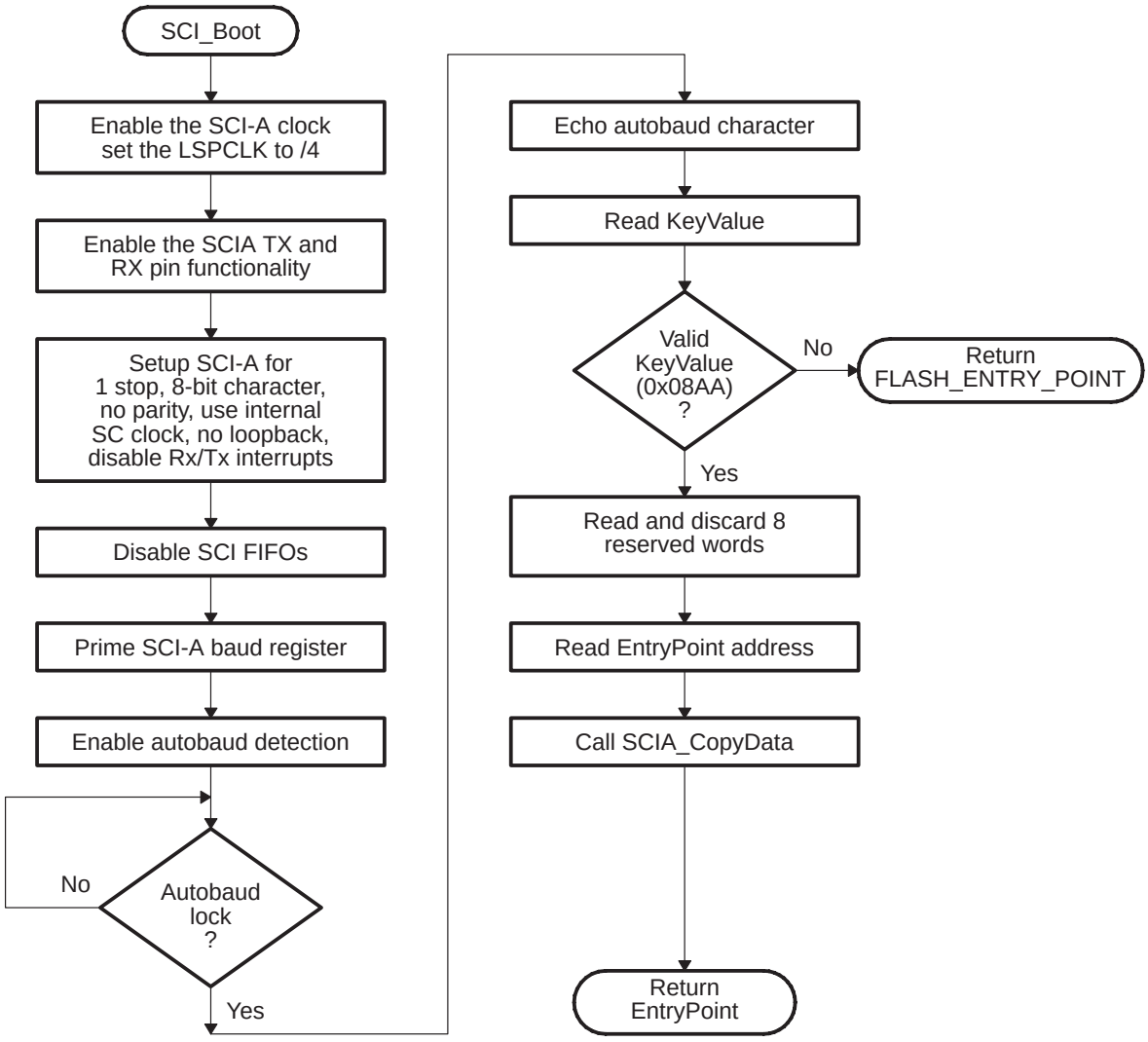


Figure 13. Overview of SCIA_CopyData Function

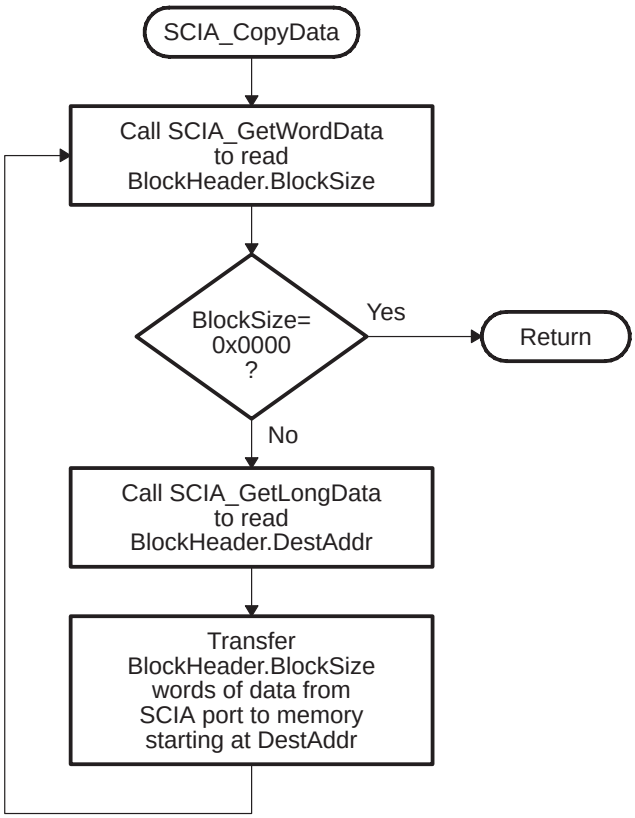
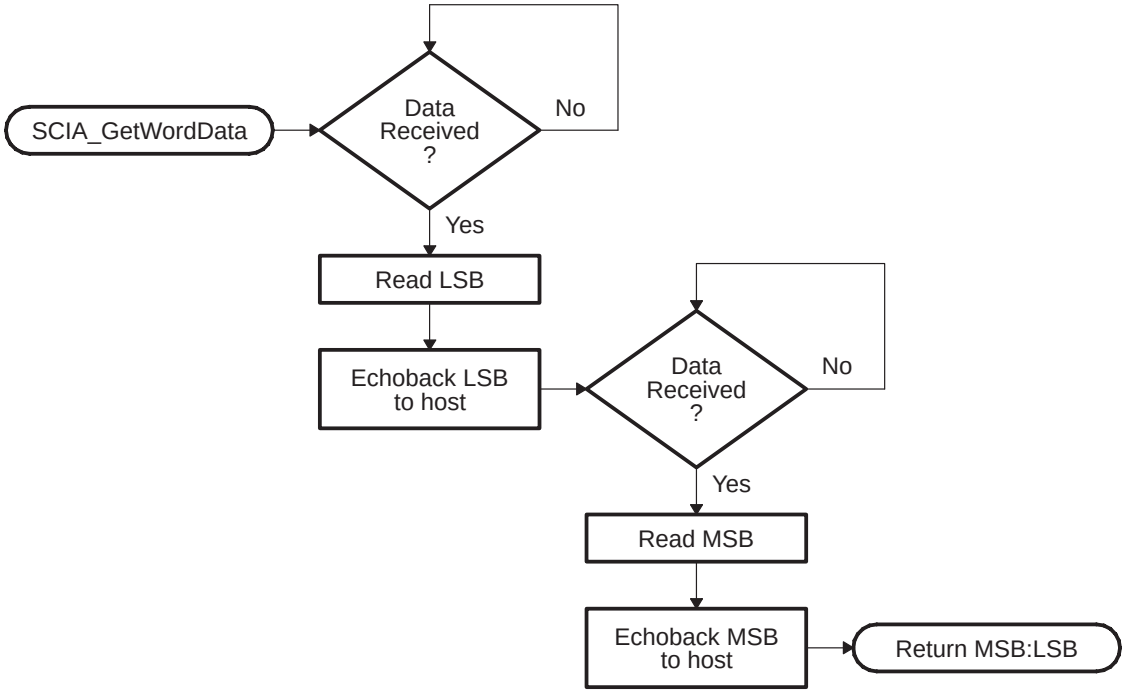


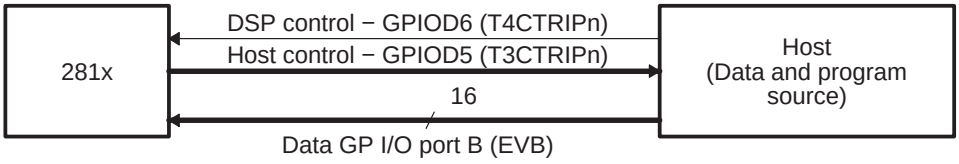
Figure 14. Overview of SCI_GetWordData Function



4.10 Parallel_Boot Function (GPIO)

The parallel general purpose I/O (GPIO) boot mode asynchronously transfers code from GPIO port B to internal or XINTF memory. Each value can be 16 bits or 8 bits long and follows the same data flow as outlined in Data Stream Structure.

Figure 15. Overview of Parallel GPIO Boot Loader Operation



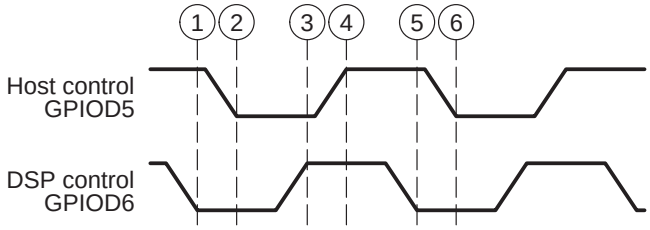
The 28x communicates with the external host device by polling/driving the GPIOD5 and GPIOD6 lines. The handshake protocol shown in Figure 16 must be used to successfully transfer each word via GPIO port B. This protocol is very robust and allows for a slower or faster host to communicate with the 281x device.

If the 8-bit mode is selected, two consecutive 8-bit words are read to form a single 16-bit word. The most significant byte (MSB) is read first followed by the

least significant byte (LSB). In this case, data is read from the lower eight lines of GPIO port B ignoring the higher byte.

The DSP first signals the host that the DSP is ready to begin data transfer by pulling the GPIOD6 pin low. The host load then initiates the data transfer by pulling the GPIOD5 pin low. The complete protocol is shown in the diagram below:

Figure 16. Parallel GPIO Boot loader Handshake Protocol



1) The DSP indicates it is ready to start receiving data by pulling the GPIOD6 pin low.

2) The boot loader waits until the host puts data on GPIO port B. The host signals to the DSP that data is ready by pulling the GPIOD5 pin low.

3) The DSP reads the data and signals the host that the read is complete by pulling GPIOD6 high.

4) The Boot loader waits until the Host acknowledges the DSP by pulling GPIOD5 high.

5) The DSP again indicates it is ready for more data by pulling the GPIOD6 pin low.

This process is repeated for each data value to be sent.

Figure 17 shows an overview of the Parallel GPIO boot loader flow.

Figure 17. Parallel GPIO Mode Overview

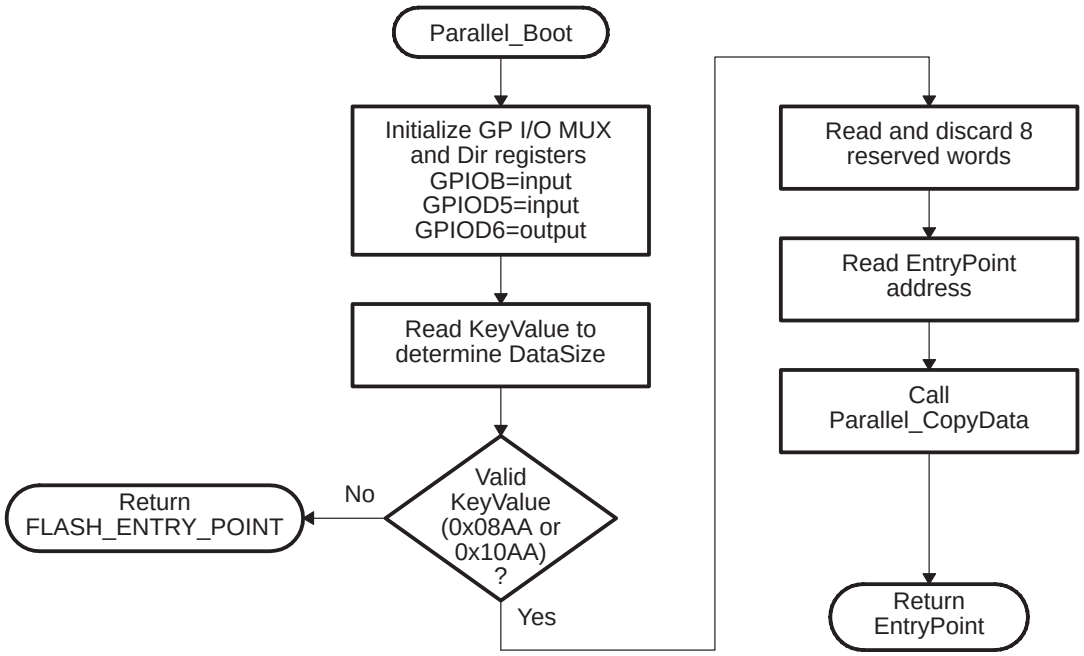


Figure 18 shows the transfer flow from the Host side. The operating speed of the CPU and Host are not critical in this mode as the host will wait for the DSP and the DSP will in turn wait for the host. In this manner the protocol will work with both a host running faster and a host running slower then the DSP.

Figure 18. Parallel GPIO Mode − Host Transfer Flow

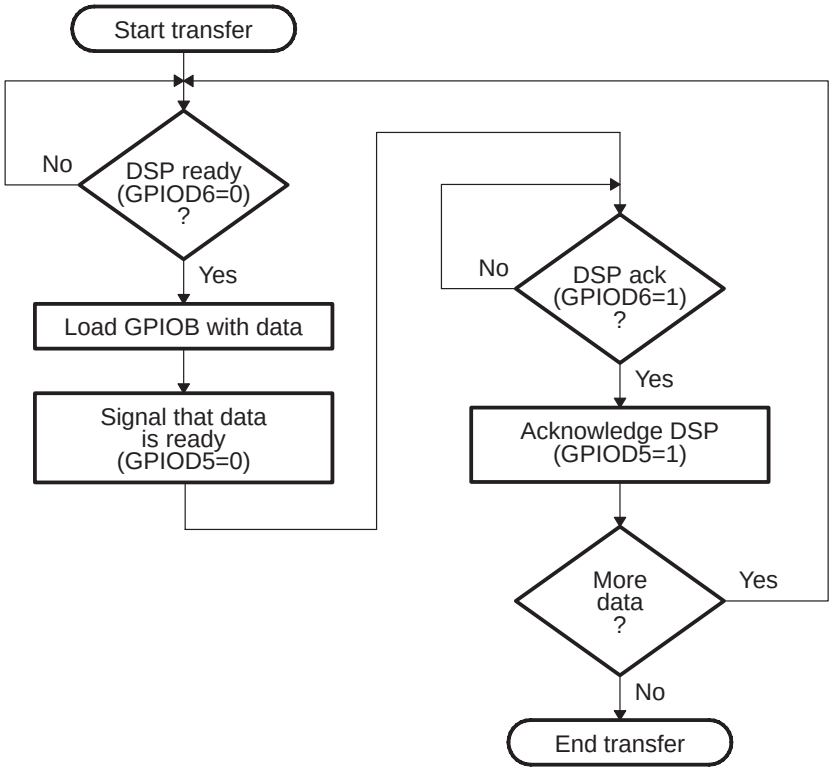


Figure 19. Overview of Parallel_CopyData Function

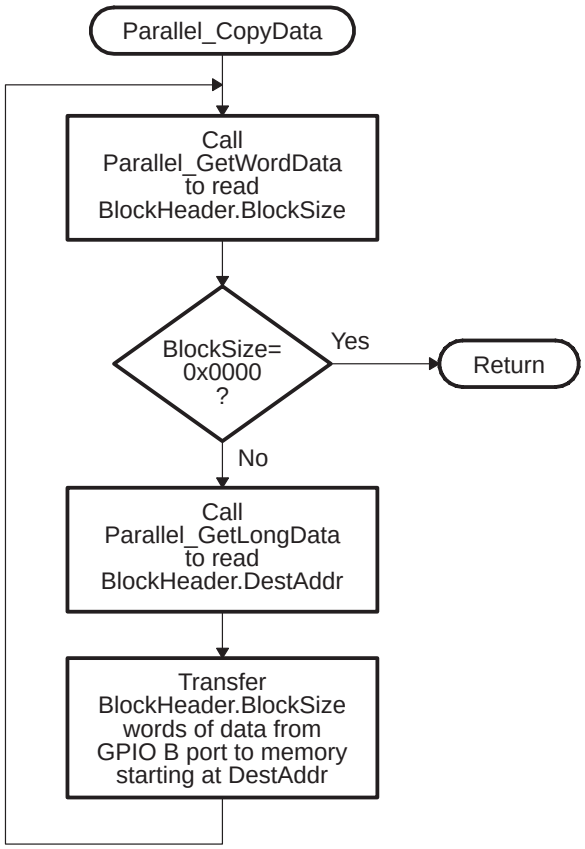
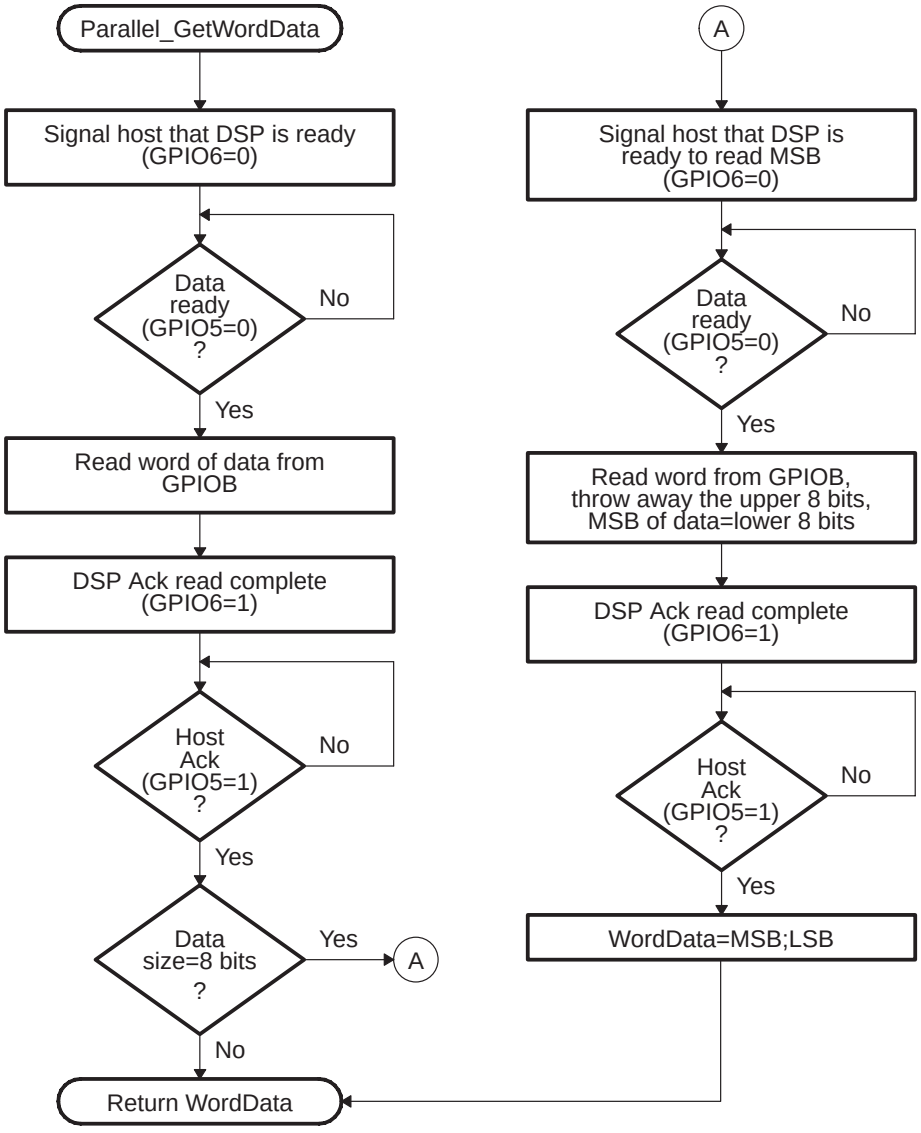


Figure 20 shows the flow for fetching a single word of data from the parallel port.

The routine is passed a DataSize parameter of 8 bits or 16 bits. The routine follows the previously defined protocol flow to read 16 bits from the GPIO B port. If the DataSize is 16 bits, then the routine will pass this 16-bit value back to the calling routine.

If the DataSize parameter is 8 bits, then the routine will and discard the upper 8 bits of the first read and treat the lower 8 bits as the least significant byte (LSB) of the word to be fetched. The routine will then perform a 2nd fetch to read the most significant byte (MSB). It then combines the MSB and LSB into a single 16-bit value to be passed back to the calling routine.

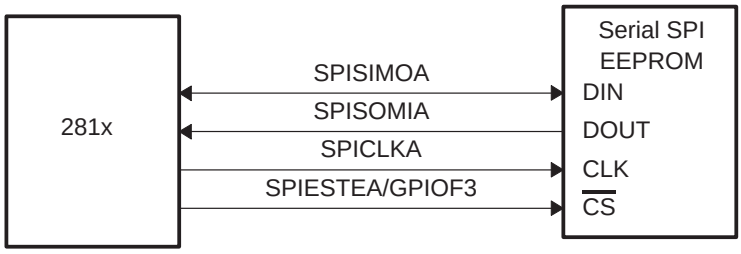
Figure 20. Parallel GPIO Boot Loader Word Fetch



4.11 SPI_Boot Function

The SPI loader expects an 8-bit wide SPI-compatible serial EEPROM device to be present on the SPI pins as indicated in Figure 21. The SPI bootloader does not support a 16-bit data stream.

Figure 21. SPI Loader



The SPI boot ROM loader initializes the SPI module to interface to a serial SPI EEPROM. Devices of this type include, but are not limited to, the Xicor X25320 (4Kx8) and Xicor X25256 (32Kx8) SPI serial SPI EEPROMs.

The SPI boot ROM loader initializes the SPI with the following settings: FIFO enabled, 8-bit character, internal SPICLK master mode and talk mode, clock phase = 0, polarity = 0, using the slowest baud rate.

If the download is to be preformed from an SPI port on another device, then that device must be setup to operate in the slave mode and mimic a serial SPI EEPROM. Immediately after entering the SPI_Boot function, the pin functions for the SPI pins are set to primary and the SPI is initialized. The initialization is done at the slowest speed possible. Once the SPI is initialized and the key value read, you could specify a change in baud rate or low speed peripheral clock.

Table 8. SPI 8-Bit Data Stream

Byte	Contents
1	LSB = AA (KeyValue for memory width = 8-bits)
2	MSB = 08h (KeyValue for memory width = 8-bits)
3	LSB = LOSPCP
4	MSB= SPIBRR
5	LSB = reserved
6	MSB = reserved

Table 8. SPI 8-Bit Data Stream (Continued)

Byte	Contents
17	LSB = reserved for future use
18	MSB= reserved for future use
19	LSB: Upper half of Entry point PC[23:16]
20	MSB: Upper half of Entry point PC[31:24] (Note: Always 0x00)
21	LSB: Lower half of Entry point PC[7:0]
22	MSB: Lower half of Entry point PC[15:8]
	Blocks of data in the format size/destination address/data as shown in the generic data stream description
	LSB: 00h
	MSB: 00h − indicates the end of the source

The data transfer is done in “burst” mode from the serial SPI EEPROM. The transfer is carried out entirely in byte mode (SPI at 8 bits/character). A step-by step description of the sequence follows:

1) The SPI-A port is initialized
2) The GPIOF3 pin is now used as a chip-select for the serial SPI EEPROM
3) The SPI-A outputs a read command for the serial SPI EEPROM
4) The SPI-A sends the serial SPI EEPROM an address 0x0000; that is, the host requires that the EEPROM must have the downloadable packet starting at address 0x0000 in the EEPROM.
5) The next word fetched must match the key value for an 8-bit data stream (0x08AA).

   The least significant byte of this word is the byte read first and the most significant byte is the next byte fetched. This is true of all word transfers on the SPI.

   If the key value does not match then the load is aborted and the entry point for the Flash (0x3F 7FF6) is returned to the calling routine.
6) The next 2 bytes fetched can be used to change the value of the low speed peripheral clock register (LOSPCP) and the SPI Baud rate register (SPIBRR). The first byte read is the LOSPCP value and the 2nd byte read is the SPIBRR value.

   The next 7 words are reserved for future enhancements. The SPI boot loader reads these 7 words and discards them.

7) The next 2 words makeup the 32-bit entry point address where execution will continue after the boot load process is complete. This is typically the entry point for the program being downloaded through the SPI port.

8) Multiple blocks of code and data are then copied into memory from the external serial SPI EEPROM through the SPI port. The blocks of code are organized in the standard data stream structure presented earlier. This is done until a block size of 0x0000 is encountered. At that point in time the entry point address is returned to the calling routine that then exits the boot loader and resumes execution at the address specified.

Figure 22. Data Transfer From EEPROM Flow

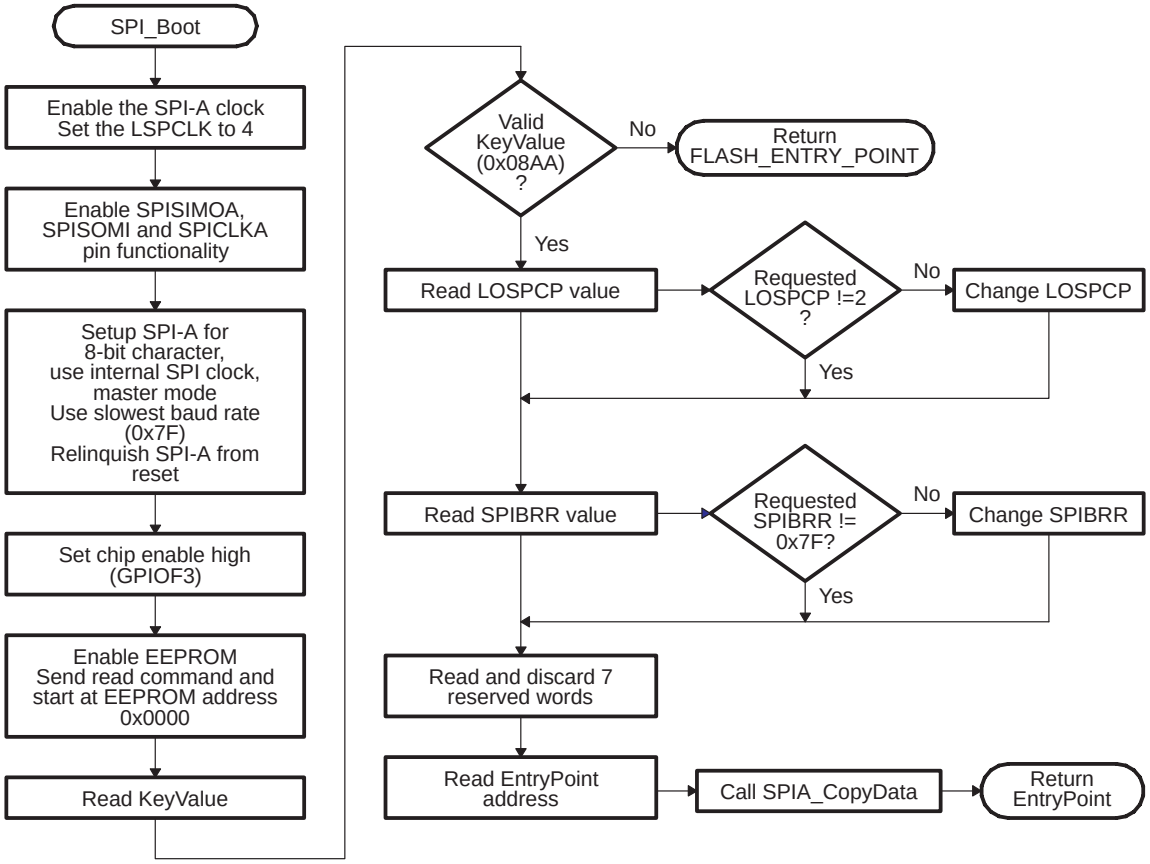


Figure 23. Overview of SPIA_CopyData Function

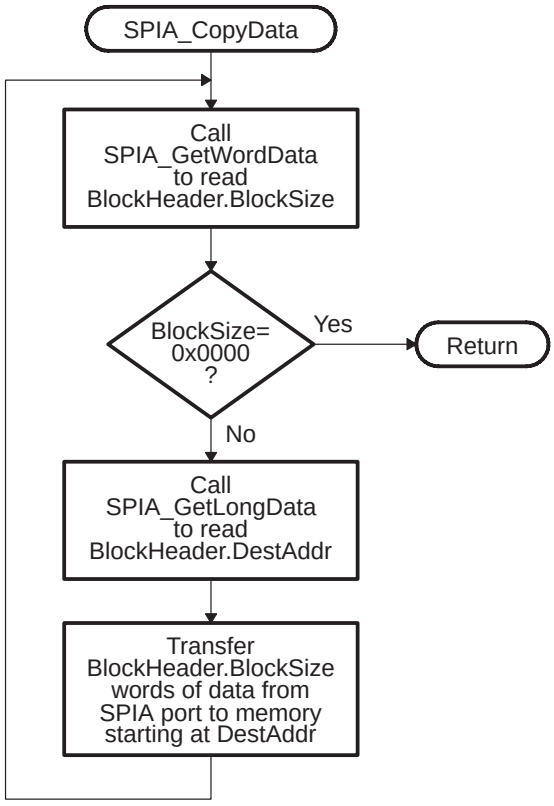
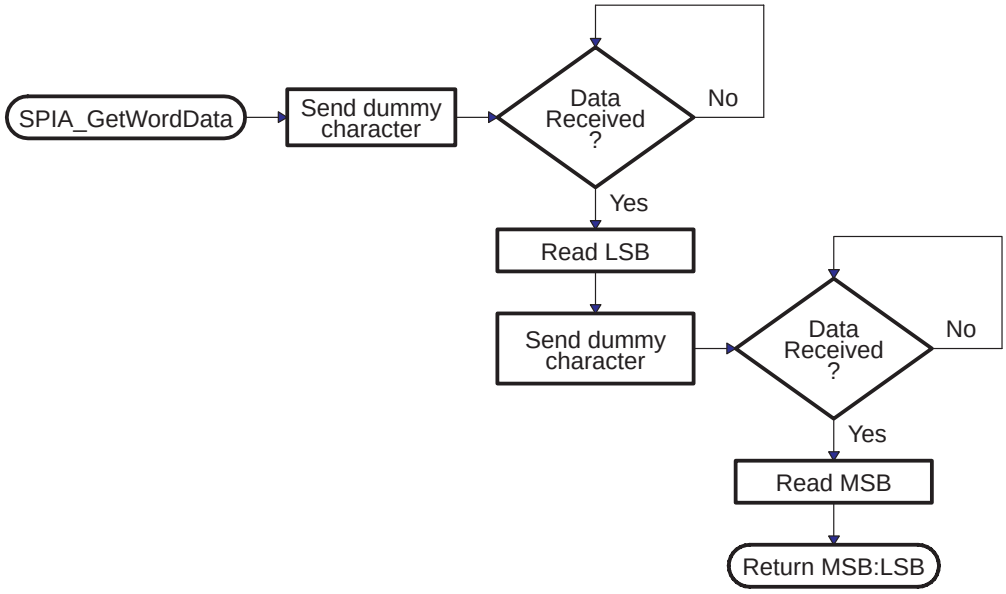


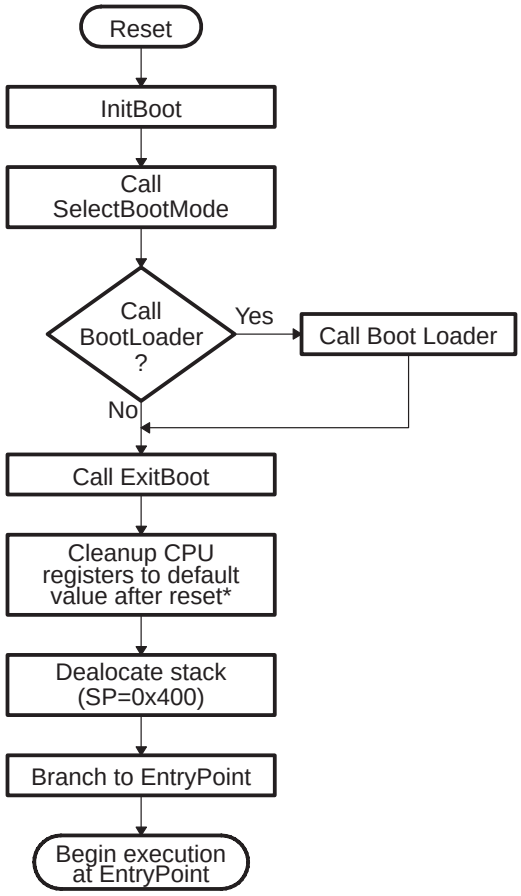
Figure 24. Overview of SPIA_GetWordData Function



4.12 ExitBoot Assembly Routine

The 281x Boot Rom includes a ExitBoot routine that restores the CPU registers to their default state at reset. This is performed on all registers with one exception. The OBJMODE bit in ST1 is left set so that the device remains configured for C28x operation. This flow is detailed in the following diagram:

Figure 25. ExitBoot Procedure Flow



The following CPU registers are restored to their default values:

ACC = 0x0000 0000

RPC = 0x0000 0000

P = 0x0000 0000

XT = 0x0000 0000

ST0 = 0x0000

ST1 = 0x0A0B

XAR0 = XAR7 = 0x0000 0000

After the ExitBoot routine completes and the program flow is redirected to the entry point address, the CPU registers will have the following values:

Table 9. CPU Register Values

Register	Value	Register	Value
ACC	0x0000 0000	P	0x0000 0000
XT	0x0000 0000	RPC	0x00 0000
XAR0–XAR7	0x0000 0000	DP	0x0000
ST0	0x0000 15:10 OVC = 0 9:7 PM = 0 6 V = 0 5 N = 0 4 Z = 0 3 C = 0 2 TC = 0 1 OVM = 0 0 SXM = 0	ST1	0x0A0B 15:13 ARP = 0 12 XF = 0 11 M0M1MAP = 1 10 reserved 9 OBJMODE = 1 8 AMODE = 0 7 IDLESTAT = 0 6 EALLOW = 0 5 LOOP = 0 4 SPA = 0 3 VMAP = 1 2 PAGE0 = 0 1 DBGM = 1 0 INTM = 1

5 Building the Boot Table

To use the features of the 281x bootloader, you must first generate a boot table that contains the complete data stream the bootloader needs. The hex conversion utility tool included with the 28x code generation tools generates the boot table. The contents of the boot table vary slightly depending on the boot mode and the options selected when running the hex conversion utility.

The hex utility supports creation of the boot table required for the SCI, SPI, and parallel I/O loaders. That is, the hex utility adds the required information to the file such as the key value, reserved bits, entry point, address, block start address, block length and terminating value. The actual file format required by the host (ASCII, binary, hex, etc.) will differ from one specific application to another and some additional conversion may be required.

To build the 281x boot table, follow these steps:

Step 1: Assemble (or compile) the code.

Step 2: Link the file. Each block of the boot table data corresponds to an initialized section in the COFF file. Uninitialized sections are not converted by the hex conversion utility.

Step 3: Run the hex conversion utility. Choose the appropriate options for the desired boot mode and run the hex conversion utility to convert the COFF file produced by the linker to a boot table.

This table summarizes the hex conversion utility options available for the boot loader**.** See the *TMS320C28x Assembly Language Tools User's Guide* (SPRU513) for a detailed description for the procedure for generating a boot table and the required options.

Table 10. Boot-Loader Options

Option	Description
−boot	Convert all sections into bootable form (use instead of a SECTIONS directive)
−sci8	Specify the source of the boot loader table as the SCI-A port, 8-bit mode
−spi8	Specify the source of the boot loader table as the SPI-A port, 8-bit mode
−gpio8	Specify the source of the boot loader table as the GP I/O port, 8-bit mode
−gpio16	Specify the source of the boot loader table as the GP I/O port, 16-bit mode
−bootorg value	Specify the source address of the boot loader table
−lospcp value	Specify the initial value for the LOSPCP register. This value is used only for the spi8 boot table format and ignored for all other formats. If the value is greater than 0x7F, the value is truncated to 0x7F.
−spibrr value	Specify the initial value for the SPIBRR register. This value is used only for the spi8 boot table format and ignored for all other formats. If the value is greater than 0x7F, the value is truncated to 0x7F.
−e value	Specify the entry point at which to begin execution after boot loading. The value can be an address or a global symbol. This value is optional. The entry point can be defined at compile time using the linker −e option to assign the entry point to a global symbol. The entry point for a C program is normally −c.intOO unless defined otherwise by the −e linker option.

6 Bootloader Code Listing

```
//###########################################################################
//
// FILE:      F2812_Boot.h
//
// TITLE:     F2812 Boot ROM Definitions.
//
//###########################################################################
//
//  Ver | dd mmm yyyy |     Who      | Description of changes
// =====|=============|==============|======================================
//  0.1 | 30 Jan 2002 | LH           | Original Release.
// -----|-------------|--------------|--------------------------------------
//      |             |              |
//
//###########################################################################
#ifndef F2812_BOOT_H
#define F2812_BOOT_H
//---------------------------------------------------------------------------
// Fixed boot entry points:
//
#define FLASH_ENTRY_POINT 0x3F7FF6
#define OTP_ENTRY_POINT   0x3D7800
#define H0_ENTRY_POINT    0x3F8000
#define PASSWORD_LOCATION 0x3F7FF8
// Misc definitions
#define ERROR                   1
#define NO_ERROR                0
#define EIGHT_BIT               8
#define SIXTEEN_BIT            16
#define EIGHT_BIT_HEADER   0x08AA
#define SIXTEEN_BIT_HEADER 0x10AA
//---------------------------------------------------------------------------
// Common CPU Definitions:
//
#define        EALLOW      asm(" EALLOW");
#define        EDIS        asm(" EDIS");
typedef int                i16;
typedef long               i32;
typedef unsigned int       ui16;
typedef unsigned long      ui32;
//---------------------------------------------------------------------------
// Include Peripheral Header Files:
//
#include "SysCtrl_Boot.h"
#include "SPI_Boot.h"
#include "SCI_Boot.h"
#include "Parallel_Boot.h"
#endif  // end of DSP28_DEVICE_H definition

;;###########################################################################
;;
;; FILE:    Init_Boot.asm
```

```
;;
;; TITLE:   F2810/12 Boot Rom Initialization and Exit routines.
;;
;; Functions:
;;
;;     _InitBoot
;;     _ExitBoot
;;
;; Notes:
;;
;;############################################################################
;;
;;  Ver | dd mmm yyyy |     Who      | Description of changes
;; =====|=============|==============|======================================
;;  1.0 | 12 Mar 2002 | LH           | PG Release.
;;      |             |              |
;;############################################################################
    .def _InitBoot
    .ref _SelectBootMode
    .sect ".Version"
    .word 0x0001      ; F2810/12 Boot ROM Version 1
    .word 0x0302      ; Month/Year: 3/02

    .sect ".Checksum" ; 64-bit Checksum
    .long 0x70F3099C  ; least significant 32-bits
    .long 0x00000402  ; most significant 32-bits

    .sect ".InitBoot"
;---------------------------------------------
; _InitBoot
;---------------------------------------------
;---------------------------------------------
; This function performs the initial boot routine
; for the F2810/12 boot ROM.
;
; This module performs the following actions:
;
;     1) Initializes the stack pointer
;     2) Sets the device for C28x operating mode
;     3) Calls the main boot functions
;     4) Calls an exit routine
;---------------------------------------------
_InitBoot:
; Initialize the stack pointer.
__stack:    .usect ".stack",0
    MOV SP, #__stack ; Initialize the stack pointer
                     ; Initialize the device for running in C28x mode.
    C28OBJ           ; Select C28x object mode
    C28ADDR          ; Select C27x/C28x addressing
    C28MAP           ; Set blocks M0/M1 for C28x mode
    CLRC PAGE0       ; Always use stack addressing mode
    MOVW DP,#0       ; Initialize DP to point to the low 64 K
    CLRC OVM
```

```
; Set PM shift of 0
    SPM 0
; Read the password locations - this will unlock the
; CSM only if the passwords are erased. Otherwise it
; will not have an effect.
  MOVL XAR1,#0x3F7FF8;
  MOVL XAR0,*XAR1++
  MOVL XAR0,*XAR1++
  MOVL XAR0,*XAR1++
  MOVL XAR0,*XAR1
; Decide which boot mode to use
    LCR  _SelectBootMode
; Cleanup and exit.  At this point the EntryAddr
; is located in the ACC register
    BF  _ExitBoot,UNC
;-----------------------------------------------
; _ExitBoot
;-----------------------------------------------
;-----------------------------------------------
;This module cleans up after the boot loader
;
; 1) Make sure the stack is deallocated.
;    SP = 0x400 after exiting the boot
;    loader
; 2) Push 0 onto the stack so RPC will be
;    0 after using LRETR to jump to the
;    entry point
; 2) Load RPC with the entry point
; 3) Clear all XARn registers
; 4) Clear ACC, P and XT registers
; 5) LRETR - this will also clear the RPC
;    register since 0 was on the stack
;-----------------------------------------------
_ExitBoot:
;-----------------------------------------------
;   Ensure that the stack is deallocated
;-----------------------------------------------
    MOV SP,#__stack
;-----------------------------------------------
; Clear the bottom of the stack.  This will endup
; in RPC when we are finished
;-----------------------------------------------
    MOV  *SP++,#0
    MOV  *SP++,#0
;-----------------------------------------------
; Load RPC with the entry point as determined
; by the boot mode.  This address will be returned
; in the ACC register.
;-----------------------------------------------
    PUSH ACC
    POP  RPC
;-----------------------------------------------
; Put registers back in their reset state.
```

```
;
; Clear all the XARn, ACC, XT, and P and DP
; registers
;
; NOTE: Leave the device in C28x operating mode
;       (OBJMODE = 1, AMODE = 0)
;-----------------------------------------------
    ZAPA
    MOVL  XT,ACC
    MOVZ  AR0,AL
    MOVZ  AR1,AL
    MOVZ  AR2,AL
    MOVZ  AR3,AL
    MOVZ  AR4,AL
    MOVZ  AR5,AL
    MOVZ  AR6,AL
    MOVZ  AR7,AL
    MOVW  DP, #0
;-----------------------------------------------
;   Restore ST0 and ST1.  Note OBJMODE is
;   the only bit not restored to its reset state.
;   OBJMODE is left set for C28x object operating
;   mode.
;
;  ST0 = 0x0000     ST1 = 0x0A0B
;  15:10 OVC = 0    15:13      ARP = 0
;   9: 7  PM = 0       12       XF = 0
;      6   V = 0       11  M0M1MAP = 1
;      5   N = 0       10  reserved
;      4   Z = 0        9  OBJMODE = 1
;      3   C = 0        8    AMODE = 0
;      2  TC = 0        7 IDLESTAT = 0
;      1 OVM = 0        6   EALLOW = 0
;      0 SXM = 0        5     LOOP = 0
;                       4      SPA = 0
;                       3     VMAP = 1
;                       2    PAGE0 = 0
;                       1     DBGM = 1
;                       0     INTM = 1
;-----------------------------------------------
    MOV  *SP++,#0
    MOV  *SP++,#0x0A0B
    POP  ST1
    POP  ST0
;-----------------------------------------------
;   Jump to the EntryAddr as defined by the
;   boot mode selected and continue execution
;-----------------------------------------------
    LRETR

;eof ----------

//###########################################################################
```

```
//
// FILE:    SelectMode_Boot.c
//
// TITLE:   F2810/12 Boot Mode selection routines
//
// Functions:
//
//      ui32   SelectBootMode(void)
//      inline void SelectMode_GPOISelect(void)
//
// Notes:
//
//###########################################################################
//
//  Ver | dd mmm yyyy |     Who      | Description of changes
// =====|=============|==============|======================================
//  1.0 | 12 Mar 2002 | LH           | PG Release.
//      |             |              |
//###########################################################################
#include "F2812_Boot.h"
inline void SelectMode_GPIOSelect(void);
// Define mask for mode selection
// all mode select pins are on GPIOF
// These definitions define which pins
// are used:
//GPIOF4     GPIOF12  GPIOF3    GPIOF2
//(SCITXDA)  (MDXA)  (SPISTEA)  (SPICLKA) Mode Selected
//   1          x       x         x       Jump to Flash address 0x3F 7FF6
//   0          1       x         x       Call SPI_Boot
//   0          0       1         1       Call SCI_Boot
//   0          0       1         0       Jump to H0 SARAM
//   0          0       0         1       Jump to OTP
//   0          0       0         0       Call  Parallel_Boot
#define MODE_PIN1   0x0010  // GPIOF4
#define MODE_PIN2   0x1000  // GPIOF12
#define MODE_PIN3   0x0008  // GPIOF3
#define MODE_PIN4   0x0004  // GPIOF2
// On GP I/O port F use only the following pins to determine the boot mode
#define MODE_MASK    (MODE_PIN1 | MODE_PIN2 | MODE_PIN3 | MODE_PIN4)
// Define which pins matter to which modes.  Note that some modes
// do not check the state of all 4 pins.
#define FLASH_MASK     (MODE_PIN1)
#define SPI_MASK       (MODE_PIN1 | MODE_PIN2)
#define SCI_MASK       (MODE_PIN1 | MODE_PIN2 | MODE_PIN3 | MODE_PIN4)
#define H0_MASK        (MODE_PIN1 | MODE_PIN2 | MODE_PIN3 | MODE_PIN4)
#define OTP_MASK       (MODE_PIN1 | MODE_PIN2 | MODE_PIN3 | MODE_PIN4)
#define PARALLEL_MASK  (MODE_PIN1 | MODE_PIN2 | MODE_PIN3 | MODE_PIN4)
// Define which pins (out of the ones being examined) must be set
// to boot to a particular mode
#define FLASH_MODE     (MODE_PIN1)
#define SPI_MODE       (MODE_PIN2)
#define SCI_MODE       (MODE_PIN3 | MODE_PIN4)
#define H0_MODE        (MODE_PIN3)
```

```
#define OTP_MODE       (MODE_PIN4)
#define PARALLEL_MODE  0x0000
ui32 SelectBootMode()
{
  ui32 EntryAddr;
  ui16 BootMode;

  SelectMode_GPIOSelect();
  BootMode = GPIODataRegs.GPFDAT.all & MODE_MASK;

  // First check for modes which do not require
  // a boot loader (Flash/H0/OTP)

  if( (BootMode & FLASH_MASK) == FLASH_MODE ) return FLASH_ENTRY_POINT;
  if( (BootMode & H0_MASK) == H0_MODE) return H0_ENTRY_POINT;
  if( (BootMode & OTP_MASK) == OTP_MODE) return OTP_ENTRY_POINT;

  // Otherwise, disable the watchdog and check for the
  // other boot modes that requre loaders
  WatchDogDisable();
  if( (BootMode & SCI_MASK) == SCI_MODE) EntryAddr = SCI_Boot();
  else if( (BootMode & SPI_MASK) == SPI_MODE) EntryAddr = SPI_Boot();
  else EntryAddr = Parallel_Boot();
   WatchDogEnable();
   return EntryAddr;
}
//#####################################################
// inline void SelectMode_GPIOSelect(void)
//-----------------------------------------
// Enable GP I/O port F as an input port.
//-----------------------------------------
inline void SelectMode_GPIOSelect()
{
    EALLOW;

    // GPIO Port F is all I/O pins
    // 0 = I/O pin  1 = Peripheral pin
     GPIOMuxRegs.GPFMUX.all = 0x0000;

    // Port F is all input
    // 0 = input   1 = output
    GPIOMuxRegs.GPFDIR.all = 0x0000;
    EDIS;
}
;EOF ----------
//###########################################################################
//
// FILE:    SCI_Boot.c
//
// TITLE:   F2810/12 SCI Boot mode routines
//
// Functions:
//
```

```
//     ui32 SCI_Boot(void)
//     inline void SCIA_GPIOSelect(void)
//     inline void SCIA_SysClockEnable(void)
//     inline void SCIA_Init(void)
//     inline void SCIA_AutobaudLock(void)
//     inline ui16 SCIA_CheckKeyVal(void)
//     inline void SCIA_ReservedFn(void)
//     ui32 SCIA_GetLongData(void)
//     ui32 SCIA_GetWordData(void)
//     void SCIA_CopyData(void)
//
// Notes:
//
//###########################################################################
//
//  Ver | dd mmm yyyy |     Who      | Description of changes
// =====|=============|==============|======================================
//  1.0 | 12 Mar 2002 | LH           | PG Release.
//      |             |              |
//###########################################################################
#include "F2812_Boot.h"
// Private functions
inline void SCIA_GPIOSelect(void);
inline void SCIA_Init(void);
inline void SCIA_AutobaudLock(void);
inline ui16 SCIA_CheckKeyVal(void);
inline void SCIA_ReservedFn(void);
inline void SCIA_SysClockEnable(void);
ui32 SCIA_GetLongData(void);
ui16 SCIA_GetWordData(void);
void SCIA_CopyData(void);
// Data section where SCIA control registers
// reside
#pragma DATA_SECTION(SCIARegs,".SCIARegs");
volatile struct SCI_REGS SCIARegs;
//###########################################################
// ui32 SCI_Boot(void)
//-------------------------------------------
// This module is the main SCI boot routine.
// It will load code via the SCI-A port.
//
// It will return a entry point address back
// to the InitBoot routine which in turn calls
// the ExitBoot routine.
//-------------------------------------------
ui32 SCI_Boot()
{
   ui32 EntryAddr;
   ui16 ErrorFlag;
   SCIA_SysClockEnable();
   SCIA_GPIOSelect();
   SCIA_Init();
   SCIA_AutobaudLock();
```

```
   // If the KeyValue was invalid, abort the load
   // and return the flash entry point.
   ErrorFlag = SCIA_CheckKeyVal();
   if (ErrorFlag == ERROR) return FLASH_ENTRY_POINT;
   SCIA_ReservedFn();
   EntryAddr = SCIA_GetLongData();
   SCIA_CopyData();
   return EntryAddr;
}
//###########################################################
// void SCIA_GPIOSelect(void)
//-----------------------------------------
// Enable pins for the SCI-A module for SCI
// peripheral functionality.
//-----------------------------------------
inline void SCIA_GPIOSelect()
{
    EALLOW
    GPIOMuxRegs.GPFMUX.all = 0x0030;
    EDIS
}
//###########################################################
// void SCIA_Init(void)
//-------------------------------------------
// Initialize the SCI-A port for communications
// with the host.
//-------------------------------------------
inline void SCIA_Init()
{
    // Enable FIFO reset bit only
    SCIARegs.SCIFFTX.all=0x8000;
    // 1 stop bit, No parity, 8-bit character
    // No loopback
    SCIARegs.SCICCR.all = 0x0007;
    // Enable TX, RX, Use internal SCICLK
    SCIARegs.SCICTL1.all = 0x0003;
    // Disable RxErr, Sleep, TX Wake,
    // Diable Rx Interrupt, Tx Interrupt
    SCIARegs.SCICTL2.all = 0x0000;
    // Relinquish SCI-A from reset
    SCIARegs.SCICTL1.all = 0x0023;
    return;
}
//###########################################################
// void SCIA_AutobaudLock(void)
//-----------------------------------------------
// Perform autobaud lock with the host.
// Note that if autobaud never occurs
// the program will hang in this routine as there
// is no timeout mechanism included.
//-----------------------------------------------
inline void SCIA_AutobaudLock()
{
```

```
    ui16 byteData;
    // Must prime baud register with >= 1
    SCIARegs.SCILBAUD = 1;
    // Prepare for autobaud detection
    // Set the CDC bit to enable autobaud detection
    // and clear the ABD bit
    SCIARegs.SCIFFCT.all = 0x2000;
    // Wait until we correctly read an
    // 'A' or 'a' and lock
    while(SCIARegs.SCIFFCT.bit.ABD != 1) {}
    // After autobaud lock, clear the CDC bit
    SCIARegs.SCIFFCT.bit.CDC = 0;
    while(SCIARegs.SCIRXST.bit.RXRDY != 1) { }
    byteData = SCIARegs.SCIRXBUF.bit.RXDT;
    SCIARegs.SCITXBUF = byteData;
    return;
}
//###################################################
// ui16 SCIA_CheckKeyVal(void)
//---------------------------------------------------
// The header of the datafile should have a proper
// key value of 0x08 0xAA.  If it does not, then
// we either have a bad data file or we are not
// booting properly.  If this is the case, return
// an error to the main routine.
//---------------------------------------------------
inline ui16 SCIA_CheckKeyVal()
{
    ui16 wordData;
    wordData = SCIA_GetWordData();
    if(wordData != EIGHT_BIT_HEADER) return ERROR;
    // No error found
    return NO_ERROR;
}
//###################################################
// void SCIA_ReservedFn(void)
//---------------------------------------------------
// This function reads 8 reserved words in the header.
// None of these reserved words are used by the
// SCI boot loader at this time, they may be used in
// future devices for enhancments.
//---------------------------------------------------
inline void SCIA_ReservedFn()
{
    ui16 i;
    // Read and discard the 8 reserved words.
    for(i = 1; i <= 8; i++)
    {
       SCIA_GetWordData();
    }
    return;
}
//###################################################
```

```
// ui32 SCIA_GetLongData(void)
//--------------------------------------------
// This routine fetches two words from the SCI-A
// port and puts them together to form a single
// 32-bit value.  It is assumed that the host is
// sending the data in the form MSW:LSW.
//--------------------------------------------
ui32 SCIA_GetLongData()
{
    ui32 longData = (ui32)0x00000000;
    // Fetch the upper 1/2 of the 32-bit value
    longData = ( (ui32)SCIA_GetWordData() << 16);
    // Fetch the lower 1/2 of the 32-bit value
    longData |= (ui32)SCIA_GetWordData();
    return longData;
}
//################################################
// ui16 SCIA_GetWordData(void)
//--------------------------------------------
// This routine fetches two bytes from the SCI-A
// port and puts them together to form a single
// 16-bit value.  It is assumed that the host is
// sending the data in the order LSB followed by MSB.
//--------------------------------------------
ui16 SCIA_GetWordData()
{
   ui16 wordData;
   ui16 byteData;

   wordData = 0x0000;
   byteData = 0x0000;

   // Fetch the LSB and verify back to the host
   while(SCIARegs.SCIRXST.bit.RXRDY != 1) { }
   wordData =  (ui16)SCIARegs.SCIRXBUF.bit.RXDT;
   SCIARegs.SCITXBUF = wordData;
   // Fetch the MSB and verify back to the host
   while(SCIARegs.SCIRXST.bit.RXRDY != 1) { }
   byteData =  (ui16)SCIARegs.SCIRXBUF.bit.RXDT;
   SCIARegs.SCITXBUF = byteData;

   // form the wordData from the MSB:LSB
   wordData |= (byteData << 8);
   return wordData;
}
//################################################
// void SCIA_CopyData(void)
//--------------------------------------------------
// This routine copies multiple blocks of data from the host
// to the specified RAM locations.  There is no error
// checking on any of the destination addresses.
// That is it is assumed all addresses and block size
// values are correct.
```

```
//
// Multiple blocks of data are copied until a block
// size of 00 00 is encountered.
//
//-----------------------------------------------------
void SCIA_CopyData()
{
   struct HEADER {
     ui16 BlockSize;
     ui32 DestAddr;
   } BlockHeader;

   ui16 wordData;
   ui16 i;

   // Get the size in words of the first block
   BlockHeader.BlockSize = SCIA_GetWordData();

   // While the block size is > 0 copy the data
   // to the DestAddr.  There is no error checking
   // as it is assumed the DestAddr is a valid
   // memory location

   while(BlockHeader.BlockSize != (ui16)0x0000)
   {
      BlockHeader.DestAddr = SCIA_GetLongData();
      for(i = 1; i <= BlockHeader.BlockSize; i++)
      {
          wordData = SCIA_GetWordData();
          *(ui16 *)BlockHeader.DestAddr++ = wordData;
      }

      // Get the size of the next block
      BlockHeader.BlockSize = SCIA_GetWordData();
   }
   return;
}
//###################################################
// inline void SCIA_SysClockEnable(void)
//-----------------------------------------------------
// This routine enables the clocks to the SCIA Port.
//-----------------------------------------------------
inline void SCIA_SysClockEnable()
{
    EALLOW;
    SysCtrlRegs.PCLKCR.bit.SCIAENCLK=1;
    SysCtrlRegs.LOSPCP.all = 0x0002;
    EDIS;
}
// EOF-------

//===========================================================================
//
```

```
// FILE:    Parallel_Boot.c
//
// TITLE:   F2810/12 Parallel Port I/O boot routines
//
// Functions:
//
//       ui32 Parallel_Boot(void)
//       inline void Parallel_GPIOSelect(void)
//       inline ui16 Parallel_CheckKeyVal(void)
//       invline void Parallel_ReservedFn(void)
//       ui32 Parallel_GetLongData(ui16 DataSize)
//       ui16 Parallel_GetWordData(ui16 DataSize)
//       void Parallel_CopyData(ui16 DataSize)
//       void Parallel_WaitHostRdy(void)
//       void Parallel_HostHandshake(void)
// Notes:
//
//############################################################################
//
//  Ver | dd mmm yyyy |     Who      | Description of changes
// =====|=============|==============|======================================
//  1.0 | 12 Mar 2002 | LH           | PG Release.
//      |             |              |
//############################################################################
#include "F2812_Boot.h"
// Private function definitions
inline void Parallel_GPIOSelect(void);
inline ui16 Parallel_CheckKeyVal(void);
inline void Parallel_ReservedFn();
ui32 Parallel_GetLongData(ui16 DataSize);
ui16 Parallel_GetWordData(ui16 DataSize);
void Parallel_CopyData(ui16 DataSize);
void Parallel_WaitHostRdy(void);
void Parallel_HostHandshake(void);
#define HOST_DATA_NOT_RDY  GPIODataRegs.GPDDAT.bit.GPIOD5!=0
#define WAIT_HOST_ACK      GPIODataRegs.GPDDAT.bit.GPIOD5!=1
// Set (DSP_ACK) or Clear (DSP_RDY) GPIOD6
#define DSP_ACK            GPIODataRegs.GPDSET.all   = 0x0040
#define DSP_RDY            GPIODataRegs.GPDCLEAR.all = 0x0040
#define DATA               GPIODataRegs.GPBDAT.all
// Data section where GPIO control and data registers
// reside
#pragma DATA_SECTION(GPIODataRegs,".GPIODataRegs");
volatile struct GPIO_DATA_REGS GPIODataRegs;
#pragma DATA_SECTION(GPIOMuxRegs,".GPIOMuxRegs");
volatile struct GPIO_MUX_REGS GPIOMuxRegs;
#endif

//###################################################
// ui32 Parallel_Boot(void)
//---------------------------------------
// This module is the main Parallel boot routine.
// It will load code via GP I/O port B.
```

```
//
// This boot mode accepts 8-bit or 16-bit data.
// 8-bit data is expected to be the order LSB
// followed by MSB.
//
// This function returns a entry point address back
// to the InitBoot routine which in turn calls
// the ExitBoot routine.
//-----------------------------------------
ui32 Parallel_Boot()
{
   ui32 EntryAddr;
   ui16 DataSize;

   Parallel_GPIOSelect();
   DataSize = Parallel_CheckKeyVal();
   if (DataSize == ERROR) return FLASH_ENTRY_POINT;
   Parallel_ReservedFn(DataSize);

   EntryAddr = Parallel_GetLongData(DataSize);
   Parallel_CopyData(DataSize);

   return EntryAddr;
}
//#################################################
// void Parallel_GPIOSelect(void)
//-----------------------------------------
// Enable pins for GP I/O on Port B.  Also enable
// the control pins for host ack and DSP ready.
//-----------------------------------------
inline void Parallel_GPIOSelect()
{
    EALLOW;

    // GPIO Port B is all I/O pins
    // 0 = I/O pin  1 = Peripheral pin
     GPIOMuxRegs.GPBMUX.all = 0x0000;

    // GPIO Port D pin 5 and 6 are I/O pins
    GPIOMuxRegs.GPDMUX.all &= 0xFF9F;

    // Port B is all input
    // D5 is an input control from the Host Ack/Rdy
    // D6 is an output for DSP Ack/Rdy
    // 0 = input   1 = output
    GPIOMuxRegs.GPDDIR.bit.GPIOD6 = 1;
    GPIOMuxRegs.GPDDIR.bit.GPIOD5 = 0;
    GPIOMuxRegs.GPBDIR.all = 0x0000;

    EDIS;
}
//#################################################
// void Parallel_CheckKeyVal(void)
```

```
//---------------------------------------
// Determine if the data we are loading is in
// 8-bit or 16-bit format.
// If neither, return an error.
//
// Note that if the host never responds then
// the code will be stuck here.  That is there
// is no timeout mechanism.
//---------------------------------------
inline ui16 Parallel_CheckKeyVal()
{
   ui16 wordData;

   // Fetch a word from the parallel port and compare
   // it to the defined 16-bit header format, if not check
   // for a 8-bit header format.

   wordData = Parallel_GetWordData(SIXTEEN_BIT);
   if(wordData == SIXTEEN_BIT_HEADER) return SIXTEEN_BIT;
   // If not 16-bit mode, check for 8-bit mode
   // Call Parallel_GetWordData with 16-bit mode
   // so we only fetch the MSB of the KeyValue and not
   // two bytes. We will ignore the upper 8-bits and combine
   // the result with the previous byte to form the
   // header KeyValue.

   wordData = wordData & 0x00FF;
   wordData |= Parallel_GetWordData(SIXTEEN_BIT) << 8;
   if(wordData == EIGHT_BIT_HEADER) return EIGHT_BIT;
   // Didn't find a 16-bit or an 8-bit KeyVal header so return an error.
   else return ERROR;
}
//#################################################
// void Parallel_ReservedFn(void)
//-------------------------------------------------
// This function reads 8 reserved words in the header.
// None of these reserved words are used by the
// Parallel boot loader at this time, they may be used in
// future devices for enhancments.
//-------------------------------------------------
inline void Parallel_ReservedFn(ui16 DataSize)
{
    ui16 i;
    // Read and discard the 8 reserved words.
    for(i = 1; i <= 8; i++)
    {
       Parallel_GetWordData(DataSize);
    }
    return;
}
//#################################################
// void Parallel_CopyData(void)
//-------------------------------------------------
```

```
// This routine copies multiple blocks of data from the host
// to the specified RAM locations. There is no error
// checking on any of the destination addresses.
// That is it is assumed all addresses and block size
// values are correct.
//
// Multiple blocks of data are copied until a block
// size of 00 00 is encountered.
//
//-----------------------------------------------------
void Parallel_CopyData(ui16 DataSize)
{
   struct HEADER {
     ui16 BlockSize;
     ui32 DestAddr;
   } BlockHeader;

   ui16 wordData;
   ui16 i;

   // Get the size in words of the first block
   BlockHeader.BlockSize = Parallel_GetWordData(DataSize);

   // While the block size is > 0 copy the data
   // to the DestAddr.  There is no error checking
   // as it is assumed the DestAddr is a valid
   // memory location

   while(BlockHeader.BlockSize != (ui16)0x0000)
   {
      BlockHeader.DestAddr = Parallel_GetLongData(DataSize);
      for(i = 1; i <= BlockHeader.BlockSize; i++)
      {
          wordData = Parallel_GetWordData(DataSize);
          *(ui16 *)BlockHeader.DestAddr++ = wordData;
      }

      // Get the size of the next block
      BlockHeader.BlockSize = Parallel_GetWordData(DataSize);
   }
   return;
}
//###################################################
// ui16 Parallel_GetWordData(ui16 DataSize)
//-----------------------------------------------
// This routine fetches a 16-bit word from the
// GP I/O port. The function is passed a DataSize
// value. If the DataSize is 16, then the input
// stream is 16-bits and the function fetches a
// signle word and returns it to the host.
//
// If the DataSize is 8, then the input stream is
// an 8-bit input stream and the upper 8-bits of the
```

```
// GP I/O port are ignored. In the 8-bit case the
// first fetches the LSB and then the MSB from the
// GPIO port.  These two bytes are then put together to
// form a single 16-bit word that is then passed back
// to the host. Note that in this case, the input stream
// from the host is in the order LSB followed by MSB
//-------------------------------------------------
ui16 Parallel_GetWordData(ui16 DataSize)
{
   ui16 wordData;

   // Get a word of data.  If we are in
   // 16-bit mode then we are done.

   Parallel_WaitHostRdy();
   wordData = DATA;
   Parallel_HostHandshake();
   // If we are in 8-bit mode then the first
   // fetch was only the LSB. Fetch the MSB.

   if(DataSize == EIGHT_BIT) {
      wordData = wordData & 0x00FF;
      Parallel_WaitHostRdy();
      wordData |= (DATA << 8);
      Parallel_HostHandshake();
   }
   return wordData;
}
//#################################################
// ui32 Parallel_GetLongData(ui16 DataSize)
//-------------------------------------------------
// This routine fetches two words from the GP I/O
// port and puts them together to form a single
// 32-bit value.  It is assumed that the host is
// sending the data in the form MSW:LSW.
//-------------------------------------------------
ui32 Parallel_GetLongData(ui16 DataSize)
{
   ui32 longData;
   longData = ( (ui32)Parallel_GetWordData(DataSize) )<< 16;
   longData |=  (ui32)Parallel_GetWordData(DataSize);
   return longData;
}
//#################################################
// void Parallel_WaitHostRdy(void)
//-------------------------------------------------
// This routine tells the host that the DSP is ready to
// recieve data. The DSP then waits for the host to
// signal that data is ready on the GP I/O port.e
//-------------------------------------------------
void Parallel_WaitHostRdy()
{
   DSP_RDY;
```

```
   while(HOST_DATA_NOT_RDY) { }
}
//#################################################
// void Parallel_HostHandshake(void)
//-------------------------------------------------
// This routine tells the host that the DSP has recieved
// the data. The DSP then waits for the host to acknowledge
// the reciept before continuing.
//-------------------------------------------------
void Parallel_HostHandshake()
{
   DSP_ACK;
   while(WAIT_HOST_ACK) { }
}
// EOF --------
//###########################################################################
//
// FILE:    SPI_Boot.c
//
// TITLE:   F2810/12 SPI Boot mode routines
//
// Functions:
//
//     ui32 SPI_Boot(void)
//     inline void SPIA_GPIOSelect(void)
//     inline void SPIA_SysClockEnable(void)
//     inline void SPIA_Init(void)
//     inline void SPIA_Transmit(u16 cmdData)
//     inline ui16 SPIA_CheckKeyVal(void)
//     inline void SPIA_ReservedFn(void);
//     ui32 SPIA_GetLongData(void)
//     ui32 SPIA_GetWordData(void)
//     void SPIA_CopyData(void)
//
// Notes:
//
//###########################################################################
//
//  Ver | dd mmm yyyy |     Who      | Description of changes
// =====|=============|==============|======================================
//  1.0 | 12 Mar 2002 | SS           | PG Release.
//      |             |              |
//###########################################################################
#include "F2812_Boot.h"
#pragma DATA_SECTION(SPIARegs,".SPIARegs");
volatile struct SPI_REGS SPIARegs;
// Private functions
inline void SPIA_GPIOSelect(void);
inline void SPIA_Init(void);
inline ui16 SPIA_Transmit(ui16 cmdData);
inline ui16 SPIA_CheckKeyVal(void);
inline void SPIA_ReservedFn(void);
inline void SPIA_SysClockEnable(void);
```

```
ui32 SPIA_GetLongData(void);
ui16 SPIA_GetWordData(void);
void SPIA_CopyData(void);
//#################################################
// ui32 SPI_Boot(void)
//-----------------------------------------
// This module is the main SPI boot routine.
// It will load code via the SPI-A port.
//
// It will return a entry point address back
// to the ExitBoot routine.
//-----------------------------------------
ui32 SPI_Boot()
{
   ui32 EntryAddr;
   ui16 ErrorFlag;

   SPIA_SysClockEnable();
   SPIA_GPIOSelect();
   SPIA_Init();
   // 1. Enable EEPROM chip enable - low - Bit 5 for the IOPORT
   // Chip enable - high
    GPIODataRegs.GPFCLEAR.bit.GPIOF3 =1;
   // 2. Enable EEPROM and send EEPROM Read Command
   SPIA_Transmit(0x0300);
   // 3. Send Starting for the EEPROM address 16bit
   // Sending 0x0000,0000 will work for address and data packets
   SPIA_GetWordData();
   // 4. Check for 0x08AA data header, else go to flash
   ErrorFlag = SPIA_CheckKeyVal();
   if (ErrorFlag != 0) return FLASH_ENTRY_POINT;
   // 4a.Check for Clock speed change and reserved words
   SPIA_ReservedFn();
   // 5. Get point of Entry address after load
   EntryAddr = SPIA_GetLongData();
   // 6. Receive and copy one or more code sections to  destination addresses
   SPIA_CopyData();
   // 7. Disable EEPROM chip enable - high
   // Chip enable - high
    GPIODataRegs.GPFSET.bit.GPIOF3 =1;
   return EntryAddr;
}
//#################################################
// void SPIA_GPIOSelect(void)
//-----------------------------------------
// Enable pins for the SPI-A module for SPI
// peripheral functionality.
//-----------------------------------------
inline void SPIA_GPIOSelect()
{
  EALLOW
    // Enable SPISIMO/SPISOMI/SPICLK pins
    GPIOMuxRegs.GPFMUX.all = 0x0007;
```

```
    //IOPORT as output pin instead of SPISTE
    GPIOMuxRegs.GPFDIR.all = 0x0008;
    //Set IOPORT high GPIOF3, EEPROM CS enable high
    GPIODataRegs.GPFDAT.all = 0x0008;

    EDIS
}
//###################################################
// void SPIA_Init(void)
//--------------------------------------------
// Initialize the SPI-A port for communications
// with the host.
//--------------------------------------------
inline void SPIA_Init()
{
    // Enable FIFO reset bit only
    SPIARegs.SPIFFTX.all=0x8000;
    // 8-bit character
    SPIARegs.SPICCR.all = 0x0007;
    // Use internal SPICLK master mode and Talk mode
    SPIARegs.SPICTL.all = 0x000E;
    // Use the slowest baud rate
    SPIARegs.SPIBRR     = 0x007f;
    // Relinquish SPI-A from reset
    SPIARegs.SPICCR.all = 0x0087;
    return;
}
//###################################################
// void SPIA_Transmit(void)
//--------------------------------------------
// Send a byte/words through SPI transmit channel
//--------------------------------------------
inline ui16 SPIA_Transmit(ui16 cmdData)
{
    ui16 recvData;
    // Send Read command/dummy word to EEPROM to fetch a byte
    SPIARegs.SPITXBUF = cmdData;
    while( (SPIARegs.SPISTS.bit.INT_FLAG) !=1);
    // Clear SPIINT flag and capture received byte
    recvData = SPIARegs.SPIRXBUF;

    return recvData;

}
//###################################################
// ui16 SPIA_CheckKeyVal(void)
//--------------------------------------------
// The header of the datafile should have a proper
// key value of 0x08 0xAA. If it does not, then
// we either have a bad data file or we are not
// booting properly. If this is the case, return
// an error to the main routine.
//--------------------------------------------
```

```
inline ui16 SPIA_CheckKeyVal()
{
    ui16 wordData;

    wordData = SPIA_GetWordData();
    if(wordData != 0x08AA) return ERROR;
    // No error found
    return NO_ERROR;
}
//#################################################
// void SPIA_ReservedFn(void)
//-------------------------------------------------
// This function reads 8 reserved words in the header.
// The first word has parameters for LOSPCP
// and SPIBRR register 0xMSB:LSB, LSB = is a three
// bit field for LOSPCP change MSB = is a 6bit field
// for SPIBRR register update
//
// If either byte is the default value of the register
// then no speed change occurs. The default values
// are LOSPCP = 0x02 and SPIBRR = 0x7F
// The remaining reserved words are read and discarded
// and then returns to the main routine.
//-------------------------------------------------
inline void SPIA_ReservedFn()
{
    ui16 speedData;
    ui16 i;

    speedData = (ui16)0x0000;
    // update LOSPCP register if 1st reserved byte is not the default
    // value of 0x0002
    speedData = SPIA_Transmit((ui16)0x0000);
    if(speedData != (ui16)0x0002)
    {
        EALLOW;
        SysCtrlRegs.LOSPCP.all = speedData;
        EDIS;
       // Dummy cycles
       asm(" RPT #0x0F ||NOP");
    }

    // update SPIBRR register if 2nd reserved byte is not the default
    // value of 0x7F
    speedData = SPIA_Transmit((ui16)0x0000);

    if(speedData != (ui16) 0x007F)
    {
       SPIARegs.SPIBRR  = speedData;
       // Dummy cycles
       asm(" RPT #0x0F ||NOP");
    }
    // Read and discard the next 7 reserved words.
```

```
    for(i = 1; i <= 7; i++)
    {
       SPIA_GetWordData();
    }
    return;
}
//###########################################################
// ui32 SPIA_GetLongData(void)
//-----------------------------------------------
// This routine fetches two words from the SPI-A
// port and puts them together to form a single
// 32-bit value. It is assumed that the host is
// sending the data in the form MSW:LSW.
// ----------------------------------------------
ui32 SPIA_GetLongData()
{
    ui32 longData = (ui32)0x00000000;
    // Fetch the upper 1/2 of the 32-bit value
    longData = ( (ui32)SPIA_GetWordData() << 16);

    // Fetch the lower 1/2 of the 32-bit value
    longData |= (ui32)SPIA_GetWordData();
    return longData;
}
//###########################################################
// ui16 SPIA_GetWordData(void)
//-----------------------------------------------
// This routine fetches two bytes from the SPI-A
// port and puts them together to form a single
// 16-bit value.  It is assumed that the host is
// sending the data in the form MSB:LSB.
//-----------------------------------------------
ui16 SPIA_GetWordData()
{
   ui16 wordData;
   ui16 byteData;

   wordData = 0x0000;

   // Fetch the LSB and verify back to the host
   wordData =  SPIA_Transmit(0x0000);
   // Fetch the MSB and verify back to the host
   byteData =  SPIA_Transmit(0x0000);
   // Shift the upper byte to be the MSB
    wordData |= (byteData << 8);
   return wordData;
}
//###########################################################
// void SPIA_CopyData(void)
//-----------------------------------------------------
// This routine copies multiple blocks of data from the host
// to the specified RAM locations. There is no error
// checking on any of the destination addresses.
```

```
// That is it is assumed all addresses and block size
// values are correct.
//
// Multiple blocks of data are copied until a block
// size of 00 00 is encountered.
//
//-----------------------------------------------------
void SPIA_CopyData()
{
   struct HEADER {
     ui16 BlockSize;
     ui32 DestAddr;
   } BlockHeader;

   ui16 wordData;
   ui16 i;

   // Get the size in words of the first block
   BlockHeader.BlockSize = SPIA_GetWordData();

   // While the block size is > 0 copy the data
   // to the DestAddr. There is no error checking
   // as it is assumed the DestAddr is a valid
   // memory location

   while(BlockHeader.BlockSize != (ui16)0x0000)
   {
      BlockHeader.DestAddr = SPIA_GetLongData();
      for(i = 1; i <= BlockHeader.BlockSize; i++)
      {
          wordData = SPIA_GetWordData();
          *(ui16 *)BlockHeader.DestAddr++ = wordData;
      }

      // Get the size of the next block
      BlockHeader.BlockSize = SPIA_GetWordData();
   }
   return;
}
//###################################################
// inline void SPIA_SysClockEnable(void)
//-----------------------------------------------------
// This routine enables the clocks to the SPIA Port.
//-----------------------------------------------------
inline void SPIA_SysClockEnable()
{
    EALLOW;
    SysCtrlRegs.PCLKCR.bit.SPIAENCLK=1;
    SysCtrlRegs.LOSPCP.all = 0x0002;
    EDIS;
}
```

Appendix A

Revision History

This document was revised to SPRU095B from SPR095A. The scope of the revisions was limited to technical changes as described in Section A.1. This appendix lists only revisions made in the most recent version.

A.1 Changes Made in This Revision

The following changes were made in this revision:

Global change − Name changed to reflect new devices.

Page	Additions/Modifications/Deletions
7	Added a paragraph to explain the title change and new devices
18	Changed title of Table 4 from GPIO Pin Status to Boot Mode Selection
19	Added paragraph to “Jump to branch instruction in Flash Memory:” bullet under Figure 4
20	Added paragraph to “Jump to OTP Memory” bullet under Figure 6
31	Modified paragraph under Table 7
44	Modified step 5 in the description of the sequence of data transfer by switching least and most significant byte
51	Added to the description of −e value in Table 10