TMS320x280x DSP Boot ROM Reference Guide

Literature Number: SPRU722
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

Contents

Figures

Tables

This page intentionally left blank.

Boot ROM

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 boot ROM for the TMS320x280x devices 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 280x devices is a 4K x 16 block located in memory from 0x3F F000 − 0x3F FFFF.

1.1 On-Chip Boot ROM Description

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 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 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 2^{nd} 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:

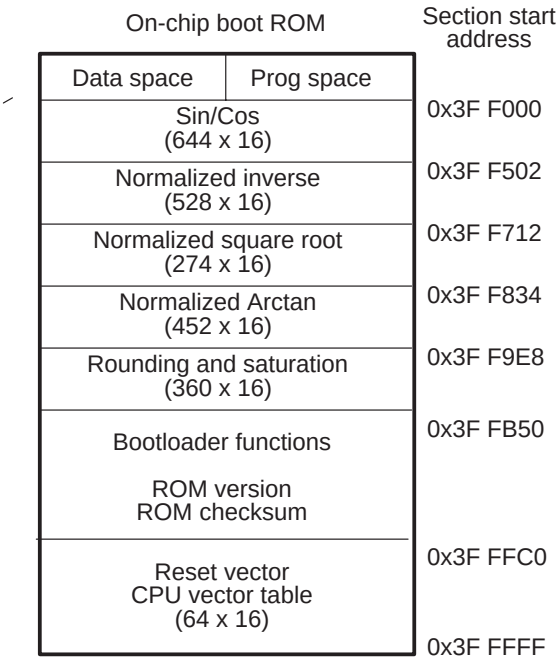
  Table size: 360 words

  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 280x. The memory block is 4Kx16 in size and is located at 0x3F F000 − 0x3F FFFF in both program and data space.

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.

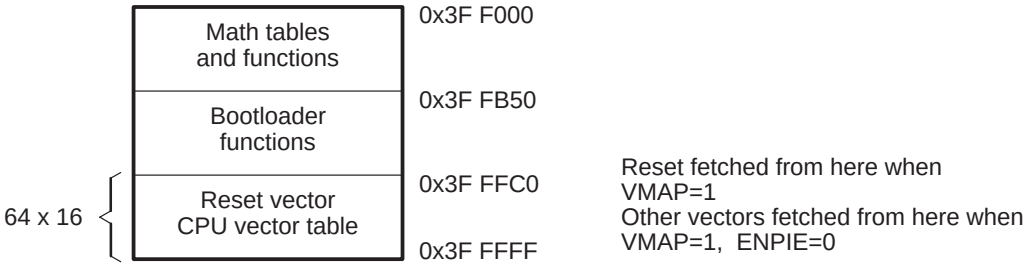
Table 1. Memory Addresses

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 after reset when VMAP = 1, ENPIE = 0 (PIE vector table disabled).

Figure 2. Vector Table Map



Notes: 1) The VMAP bit is located in Status Register 1 (ST1). 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) 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. 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 (i.e., points to)
RESET	0x3F FFC0	InitBoot (0x3F FB50)	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.

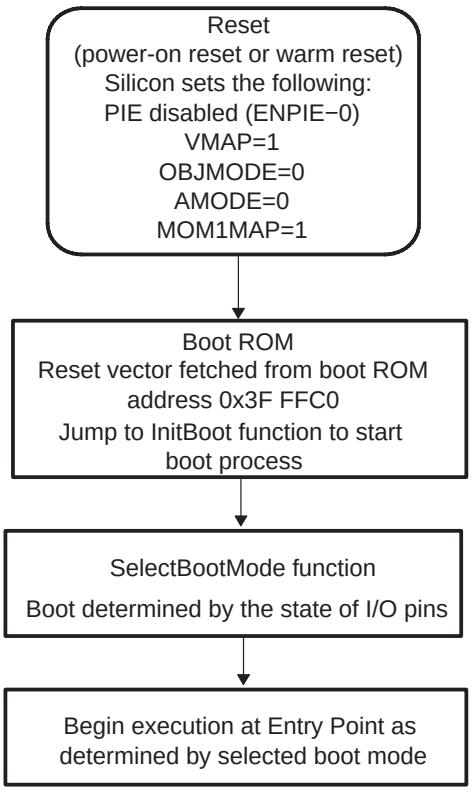
4.1 Bootloader Functional Operation

The bootloader is the program located in the on-chip boot ROM that is executed following a reset.

The bootloader is used to transfer code from an external source into internal 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. 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



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 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 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, 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.

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

	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 280x, however, it is tied off high internally; therefore, 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 ($\overline{XRS}$) low.

4.2.2 Watchdog Module

When branching directly to flash, M0 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 Internal Pullup Resisters

Each GPIO pin on the 280x has an internal pullup resistor that is disabled on reset. The boot mode selection code that reads the GPIO signals to determine the boot mode selection does not enable the pullup resistors. You must configure each of the three boot mode selection pins externally.

The individual boot loaders, SCI, SPI, eCAN, and parallel do, however, enable the pullup resistors for the pins that are used for control and data transfer. The boot loader leaves the resistors enabled for these pins when it exits. For example, the SCI-A boot loader enables the pullup resistors on the SCITXA and SCIRXA pins. It is your responsibility to disable them, if desired, after the boot loader exits.

4.2.4 PIE Configuration

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

4.2.5 Reserved Memory

The first 80 words of the M1 memory block (address 0x400 − 0x450) are reserved for stack and ebss 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

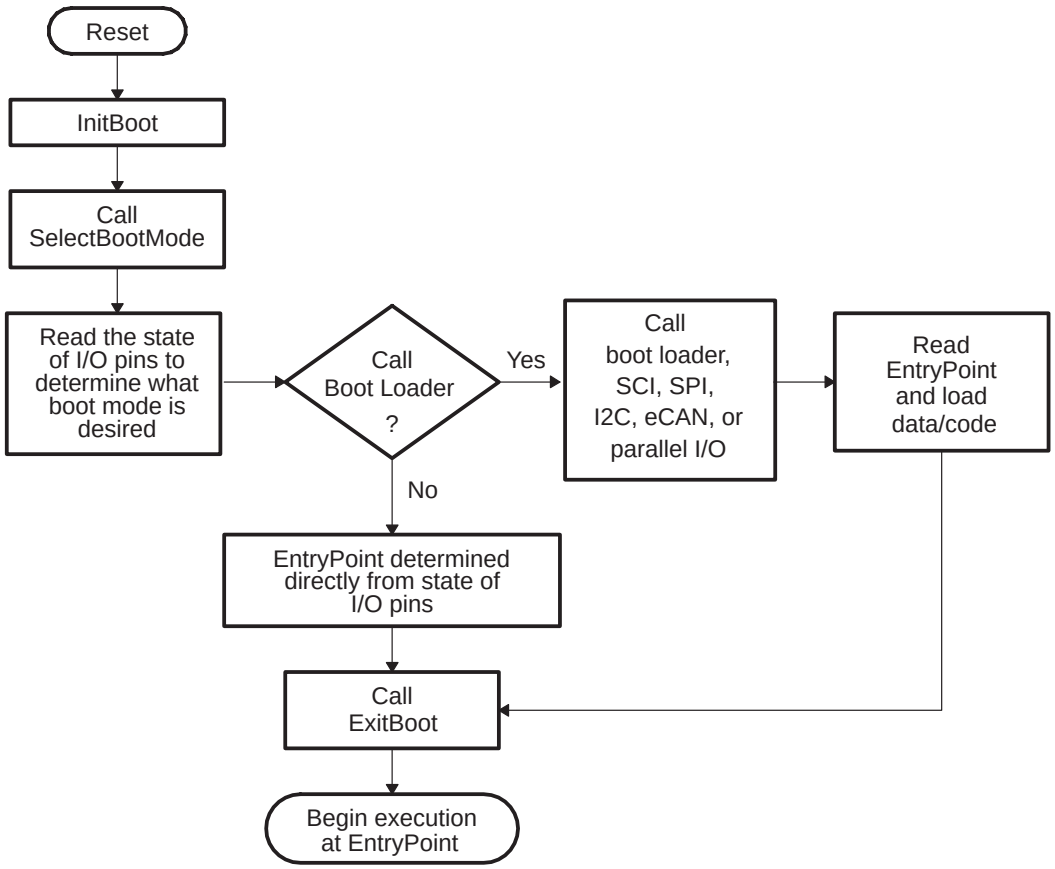
To accommodate different system requirements, the 280x boot ROM offers a variety of different boot modes. This section describes the different boot modes and gives brief summary of their functional operation. The states of three GPIO pins are used to determine the boot mode desired as shown in Table 4.

Table 4. Boot Mode Selection

Mode	Description	GPIO18 SPICLKA SCITXB	GPIO29 SCITXA	GPIO34
Boot to Flash	Jump to Flash address 0x3F 7FF6 You must have programmed a branch instruction here prior to reset to redirect code execution as desired.	1	1	1
SCI−A Boot	Load a data stream from SCI−A.	1	1	0
SPI−A Boot	Load from an external serial SPI EEPROM on SPI−A.	1	0	1
I2C Boot	Load data from an external EEPROM at address 0x50 on the I2C bus.	1	0	0
eCAN−A Boot	Call CAN_Boot to load from eCAN−A mailbox 1.	0	1	1
Boot to M0 SARAM	Jump to M0 SARAM address 0x00 0000.	0	1	0
Boot to OTP	Jump to OTP address 0x3D 7800.	0	0	1
Parallel I/O Boot	Load data from GPIO0 - GPIO15.	0	0	0

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



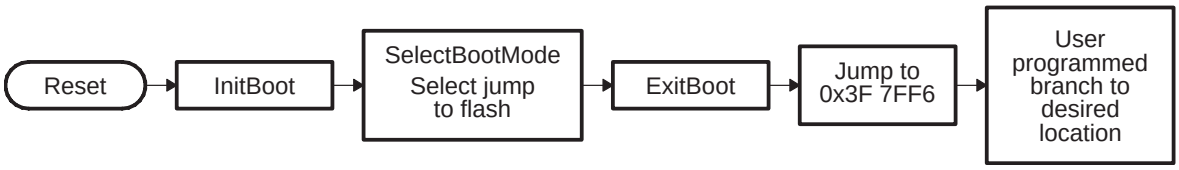
The following boot modes do not call a boot loader. Instead, they jump to a predefined location in memory:

- 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 RAM-only devices, the boot-to-Flash option jumps to reserved memory and should not be used. On ROM-only devices, the boot-to-Flash option jumps to the location 0x3F7FF6 in ROM.

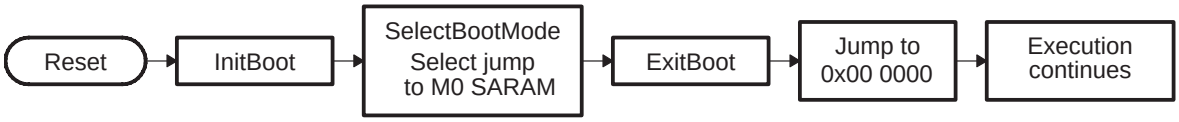
Figure 5. Jump to Flash Flow Diagram



- Jump to M0 SARAM

  In this mode, the boot ROM software will configure the device for 28x operation and then branch directly to 0x00 0000; the first address in the M0 SARAM memory block.

Figure 6. Flow Diagram of Jump to M0 SARAM

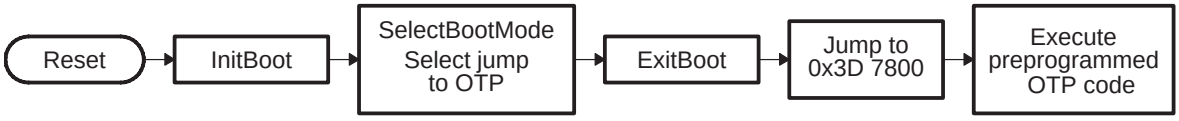


- Jump to OTP Memory

  In this mode, the boot ROM software will configure the device for 28x operation and then branch directly to at 0x3D 7800; the first address in the OTP memory block.

  On ROM devices, the boot-to-OTP option jumps to address 0x3D 7800 in ROM. On 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



The following boot modes call a boot load routine that loads a data stream from the peripheral into 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-A 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-A port.

- I2C-A Boot Mode (I2C-A):

  In this mode, the boot ROM will load code and data into on-chip memory from an external EEPROM at address 0x50 on the I2C-A bus. The EEPROM must adhere to conventional I2C EEPROM protocol with a 16-bit base address architecture.

- eCAN-A Boot Mode (eCAN-A):

  In this mode, the eCAN-A peripheral is used to transfer data and code into the on−chip memory using eCAN-A mailbox 1. The transfer is an 8-bit data stream with two 8-bit values being transferred during each communication.

- Boot from GPIO Port (Parallel Boot):

  In this mode, the boot ROM uses a GPIO port A pins GPIO0−GPIO15 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 and I2C boot loaders use these words to initialize registers.

The next tenth and eleventh 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-bit 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

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

Word	Contents
.	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
	…
	…
	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
.	MSB: 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
.	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…
	…
	…
	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:

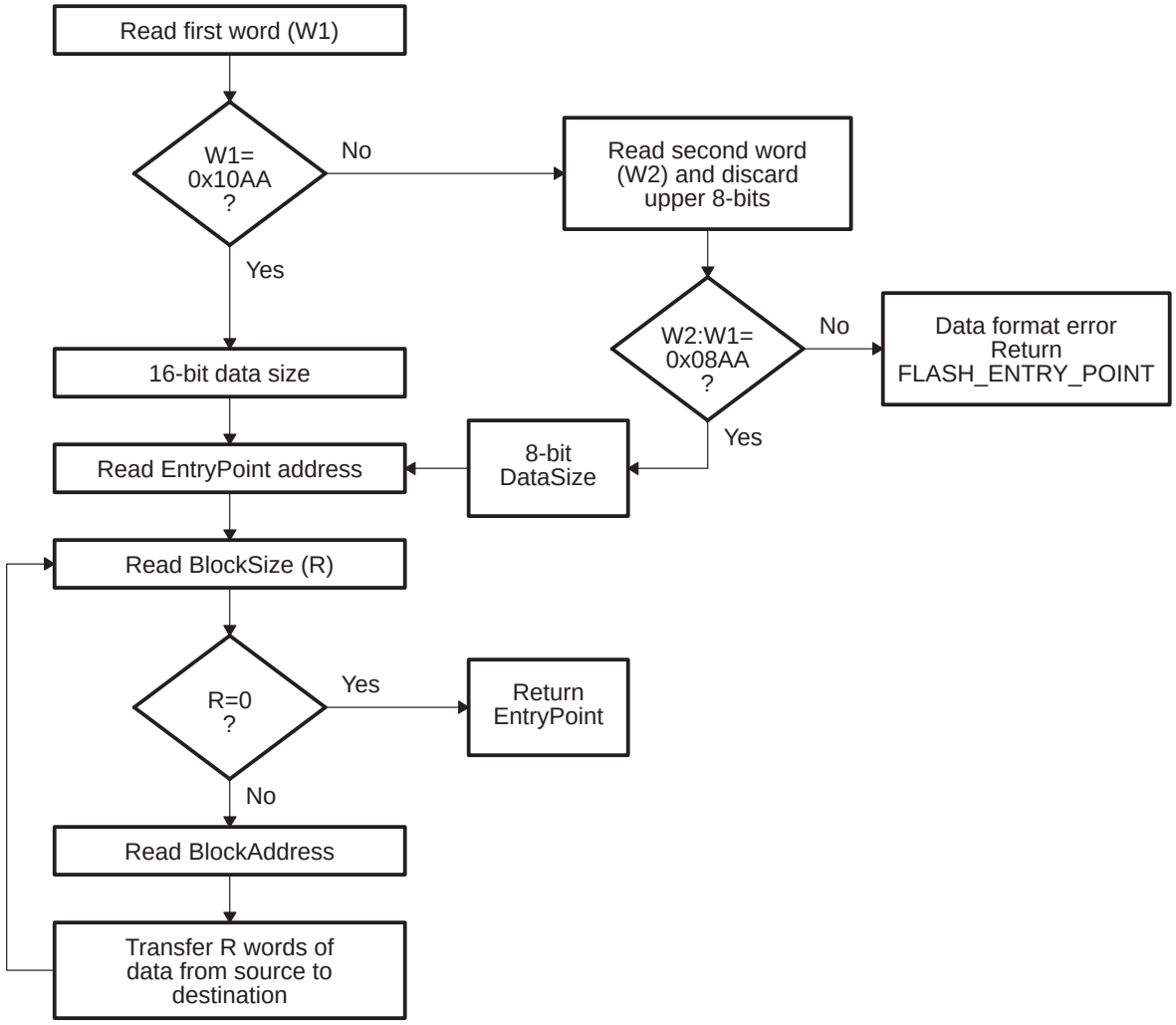
```
Location  Value
0x3F9010  0x0001
0x3F9011  0x0002
0x3F9012  0x0003
0x3F9013  0x0004
0x3F9014  0x0005
0x3F8000  0x7700
0x3F8001  0x7625
PC        Begins execution at 0x3F8000
```

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. Boot Loader Basic Transfer Procedure



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

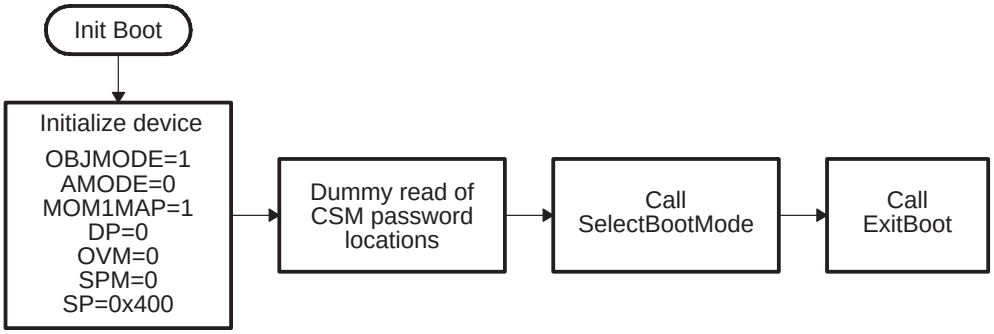
2) In 8-bit mode, the LSB of the 16-bit word is read first followed by the MSB.

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.

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 3 GPIO pins as shown in Table 7.

Table 7. Boot Mode Selection

Mode	Description	GPIO18 SPICLKA SCITXB	GPIO29 SCITXA	GPIO34
Boot to Flash	Jump to Flash address 0x3F 7FF6 You must have programmed a branch instruction here prior to reset to redirect code execution as desired.	1	1	1
SCI−A Boot	Load a data stream from SCI−A.	1	1	0

Notes: 1) When booting directly to Flash, it 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 MO, it is assumed that you have previously programmed or loaded code starting at the entry point location.

3) You must take extra care because of any effect toggling SPICLKA to select a boot mode may have on external logic.

Table 7. Boot Mode Selection (Continued)

Mode	Description	GPIO18 SPICLKA SCITXB	GPIO29 SCITXA	GPIO34
SPI−A Boot	Load from an external serial SPI EEPROM on SPI−A.	1	0	1
I2C Boot	Load data from an external EEPROM at address 0x50 on the I2C bus.	1	0	0
eCAN−A Boot	Call CAN_Boot to load from eCAN−A mailbox 1.	0	1	1
Boot to M0 SARAM	Jump to M0 SARAM address 0x00 0000.	0	1	0
Boot to OTP	Jump to OTP address 0x3D 7800.	0	0	1
Parallel I/O Boot	Load data from GPIO0 - GPIO15.	0	0	0

Notes: 1) When booting directly to Flash, it 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 MO, it is assumed that you have previously programmed or loaded code starting at the entry point location.

3) You must take extra care because of any effect toggling SPICLKA to select a boot mode may have on external logic.

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.

Next, SelectBootMode 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.

The SelectBootMode routine disables the watchdog before calling the SCI, I^2C, eCAN, 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.

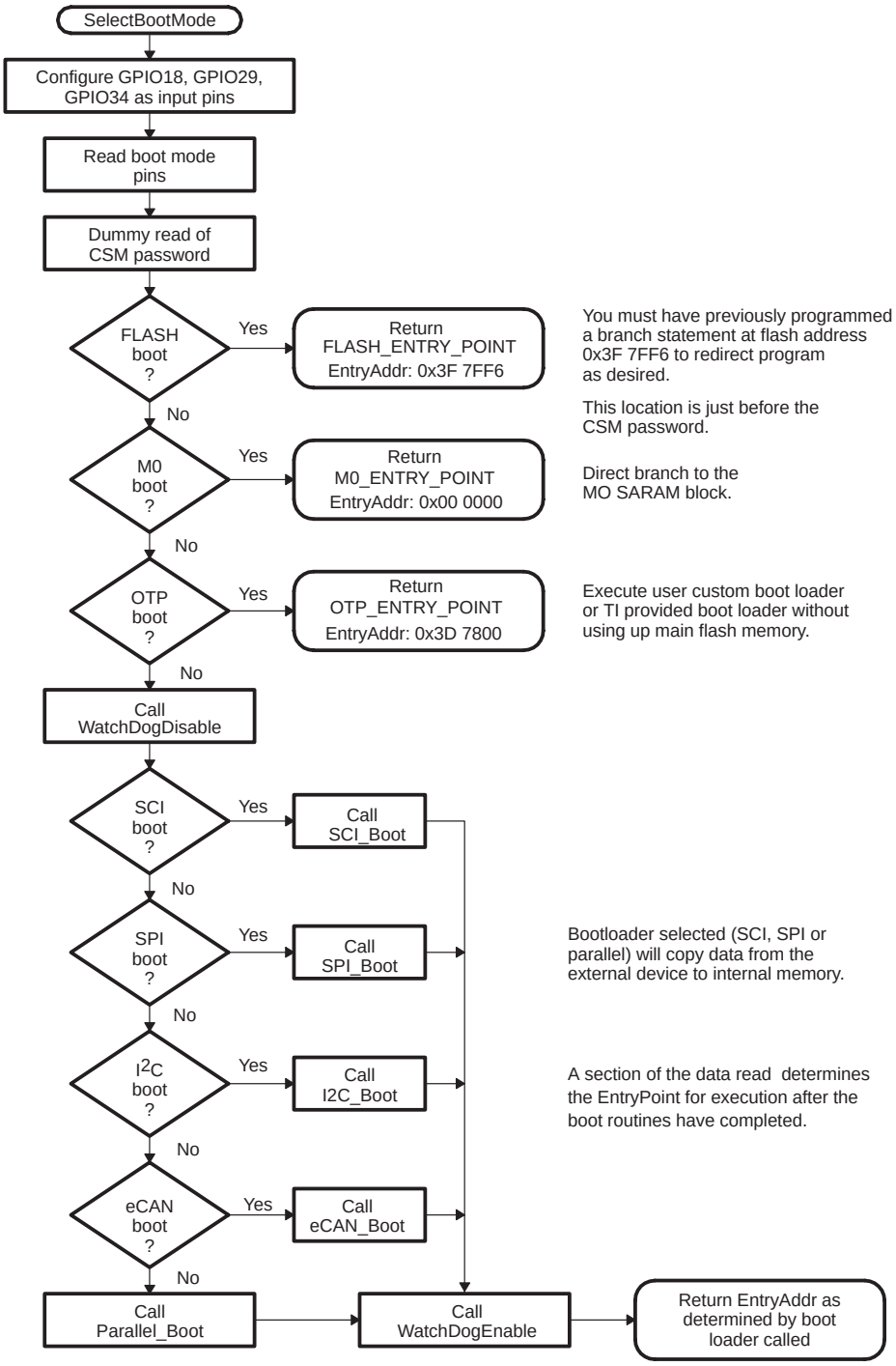
When selecting a boot mode, the pins should be pulled high or low through a weak pulldown or weak pull-up such that the DSP can drive them to a new state when required. For example, if you wanted to boot from the SCI−A one of the pins you pull high is the SCITXDA pin. This pullup must be weak so that 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.

The internal pullup resistors are not enabled for the boot mode selection pins. The configuration must be made externally.

You must take extra care using the SPICLKA signal to select a boot mode. Toggling of this signal may affect external logic and this effect must be taken into account.

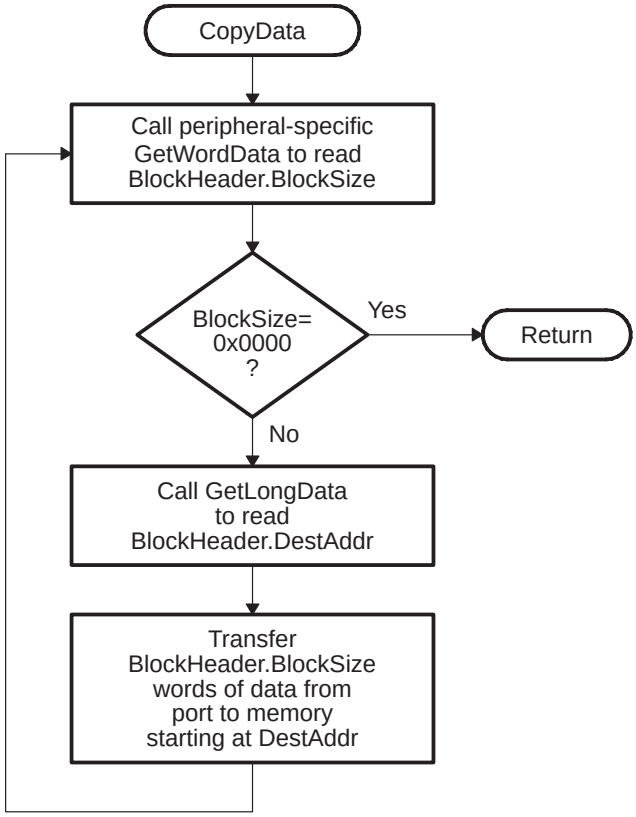
Figure 10. Overview of the SelectBootMode Function



4.9 CopyData Function

Each of the bootloaders uses the same function to copy data from the port to the DSP SARAM. This function is the CopyData() function. This function uses a pointer to a GetWordData function that is initialized by each of the loaders to properly read data from that port. For example, when the SPI loader is evoked, the GetWordData function pointer is initialized to point to the SPI-specific SPI_GetWordData function. Thus when the CopyData() function is called, the correct port is accessed. The flow of the CopyData function is shown in Figure 11.

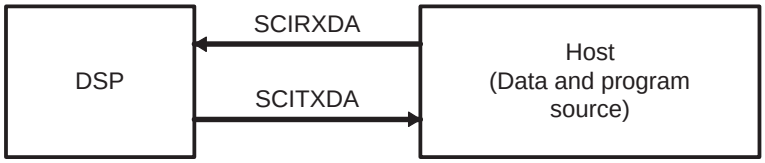
Figure 11. Overview of CopyData Function



4.10 SCI_Boot Function

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

Figure 12. Overview of SCI Boot Loader Operation



The DSP 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 13. Overview of SCI_Boot Function

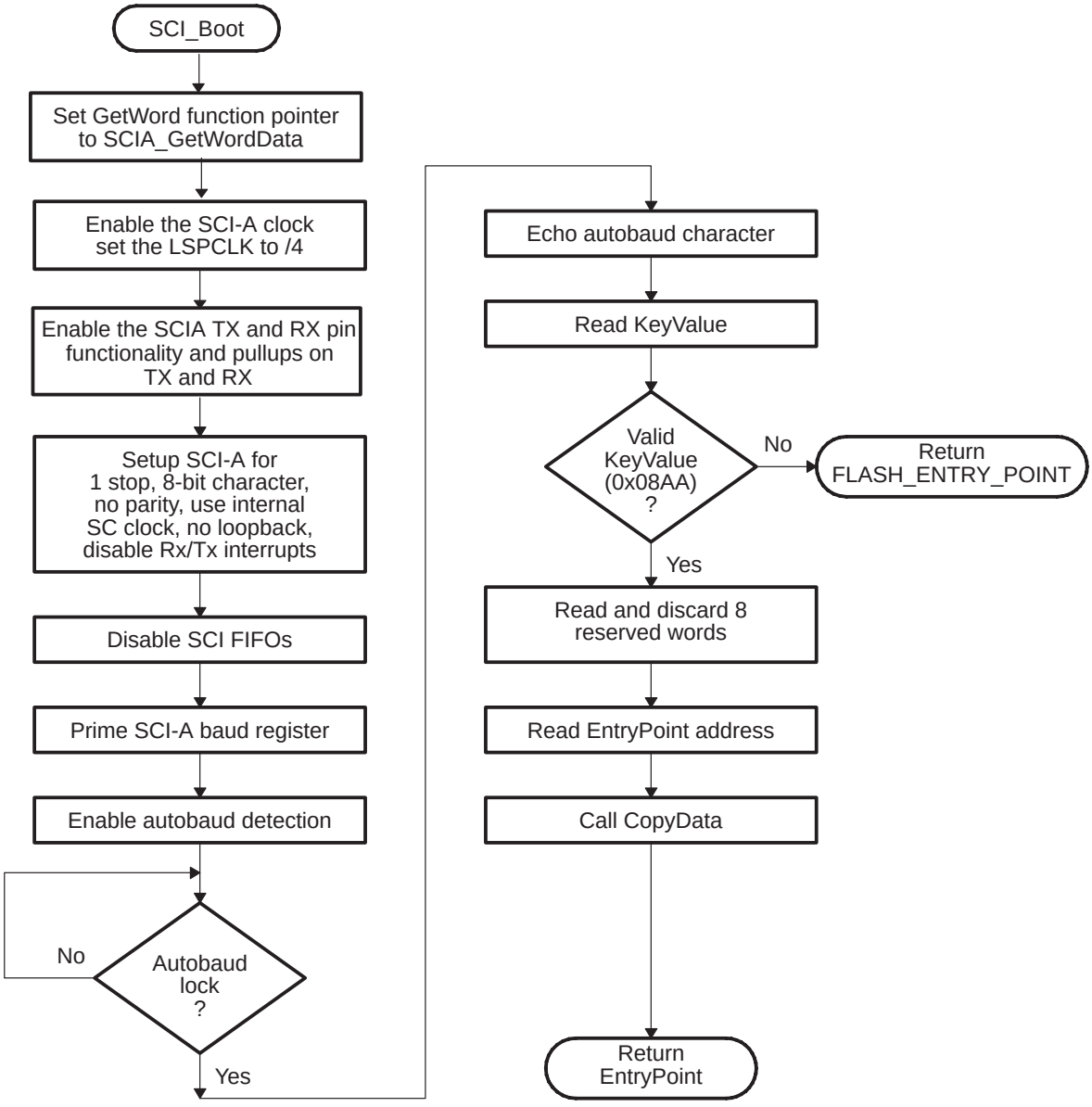
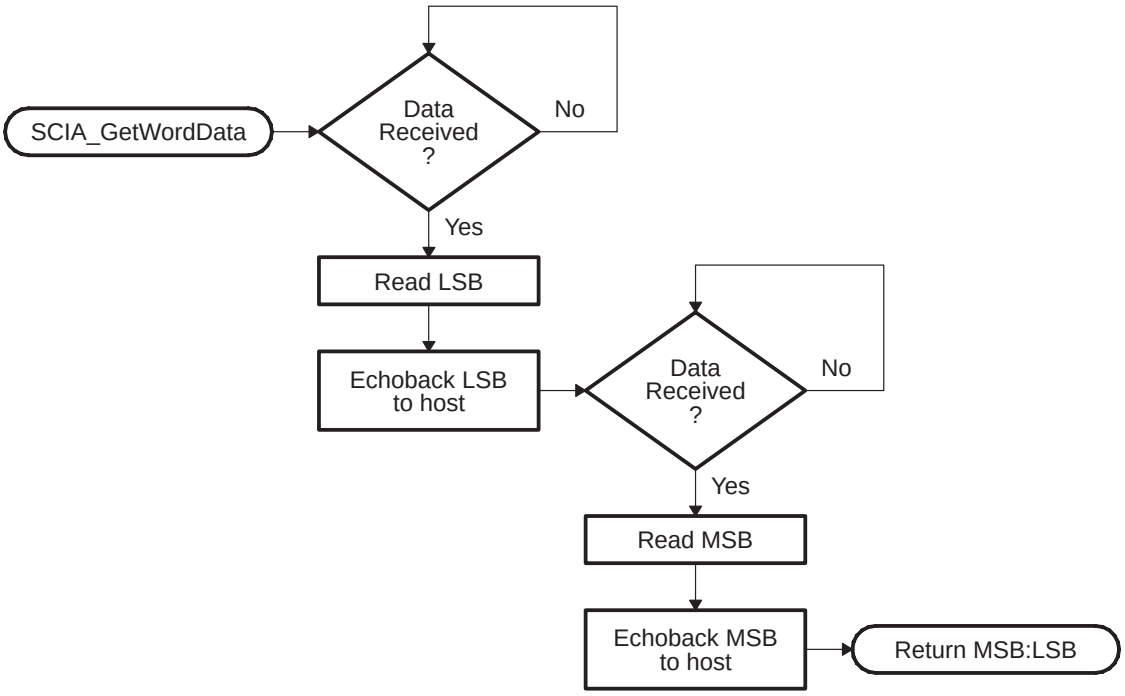


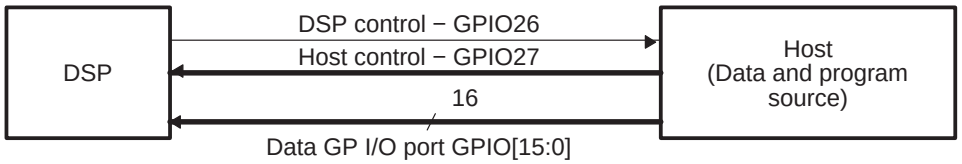
Figure 14. Overview of SCI_GetWordData Function



4.11 Parallel_Boot Function (GPIO)

The parallel general purpose I/O (GPIO) boot mode asynchronously transfers code from GPIO0−GPIO15 to internal 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



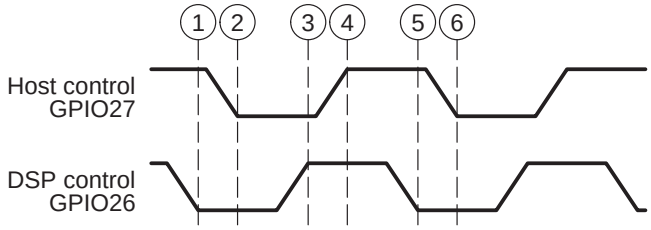
The 28x communicates with the external host device by polling/driving the GPIO27 and GPIO26 lines. The handshake protocol shown in Figure 16 must be used to successfully transfer each word via GPIO[15:0]. This protocol is very robust and allows for a slower or faster host to communicate with the DSP.

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[7:0] ignoring the higher byte.

The DSP first signals the host that the DSP is ready to begin data transfer by pulling the GPIO26 pin low. The host load then initiates the data transfer by pulling the GPIO27 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 GPIO26 pin low.

2) The boot loader waits until the host puts data on GPIO[15:0]. The host signals to the DSP that data is ready by pulling the GPIO27 pin low.

3) The DSP reads the data and signals the host that the read is complete by pulling GPIO26 high.

4) The Boot loader waits until the Host acknowledges the DSP by pulling GPIO27 high.

5) The DSP again indicates it is ready for more data by pulling the GPIO26 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

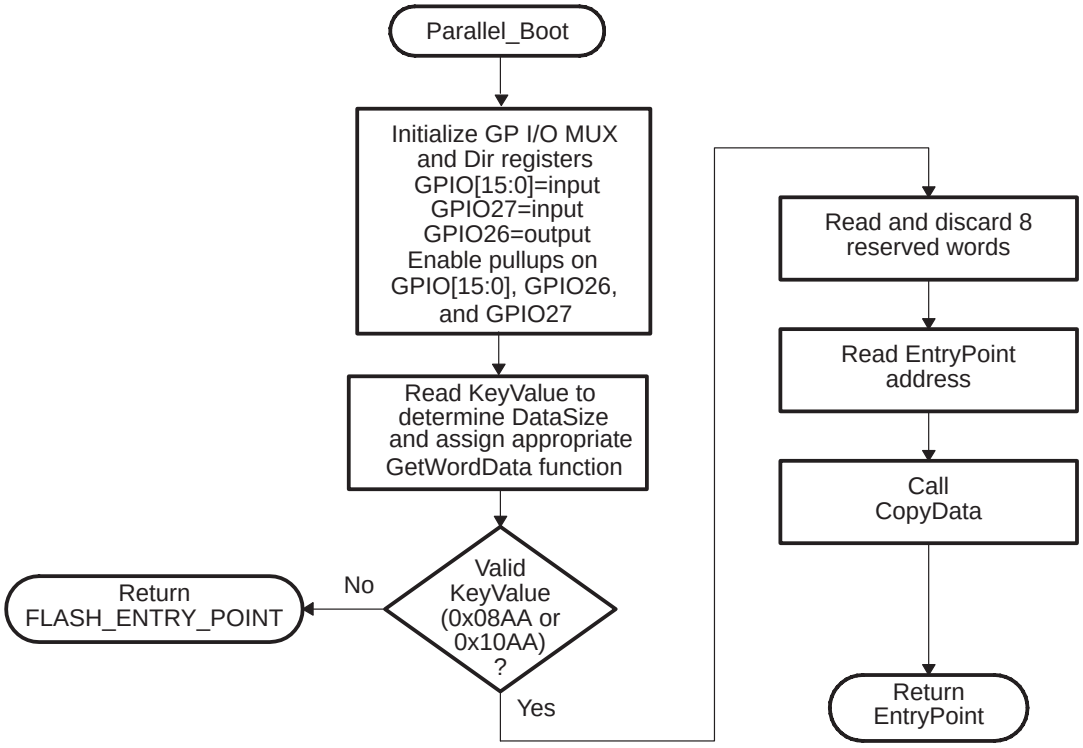


Figure 18 shows the transfer flow from the Host side. The operating speed of the CPU and Host are not critical in this mode as the host will wait for the DSP and the DSP will in turn wait for the host. In this manner the protocol will work with both a host running faster and a host running slower then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

Figure 18. Parallel GPIO Mode − Host Transfer Flow

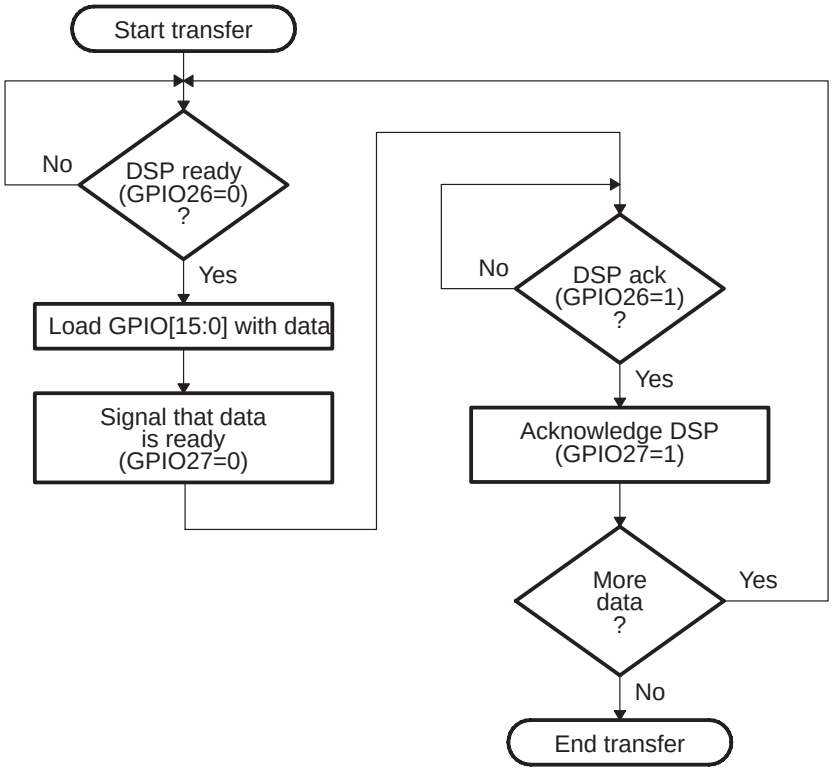


Figure 25 shows an overview of the I2C_Boot function, and Figure 19 and Figure 20 show the 16- and 8-bit parallel GetWord functions, respectively.

For 16-bit reads, the function Parallel_GetWord16bit is used. This routine reads 16 bits from the GPIO[15:0]. See Figure 19.

Figure 19. 16−Bit Parallel GetWord Function

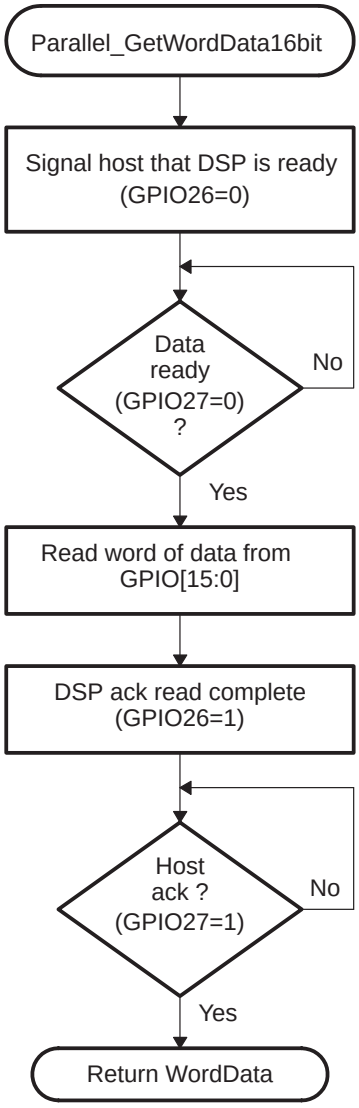
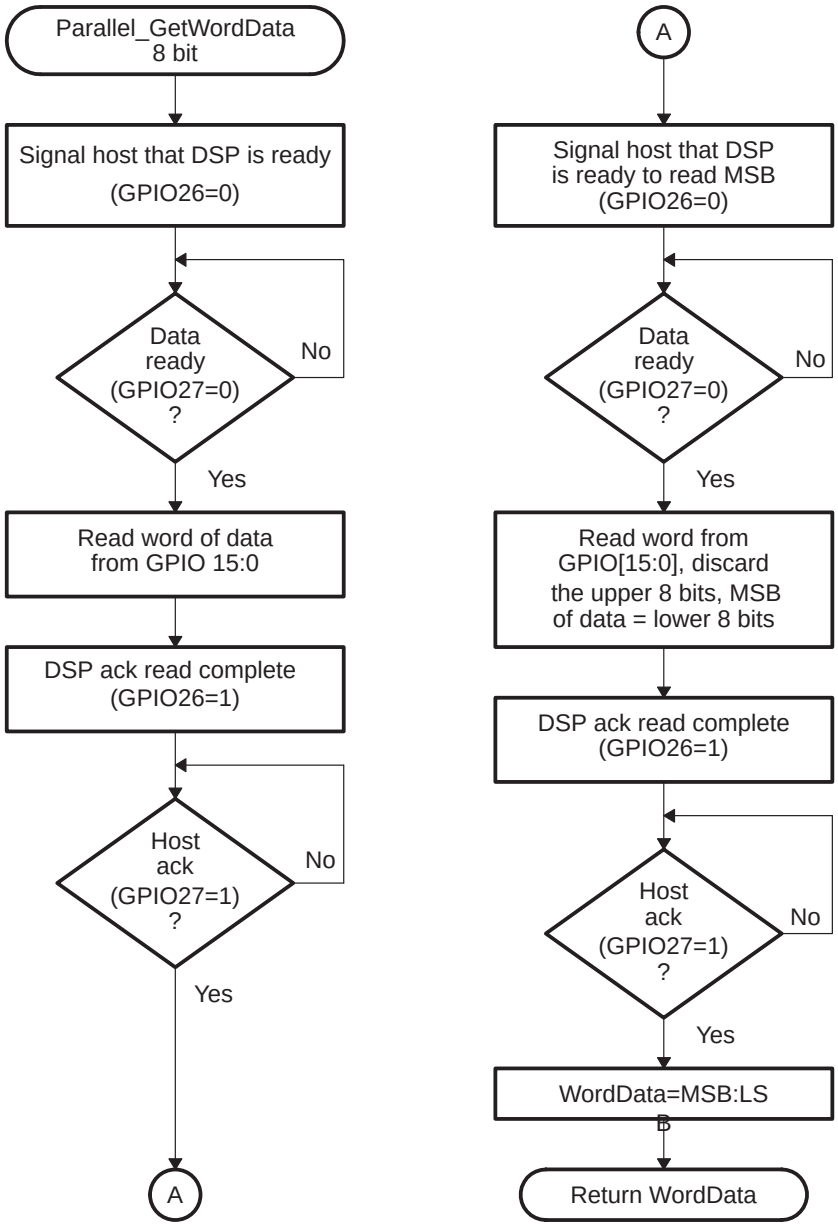


Figure 20 shows the flow for fetching a single word of data from the parallel port. A different GetWordData function is used for the parallel loader, depending on the data size of the incoming data stream.

For 8-bit data, the function Parallel_GetWordData8bit is used. The 8-bit routine will discard the upper 8 bits of the first read from the port and treat the lower 8 bits as the least significant byte (LSB) of the word to be fetched. The routine will then perform a second 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. 8−Bit Parallel GetWord Function



4.12 SPI_Boot Function

The SPI loader expects an 8-bit wide SPI-compatible serial EEPROM device to be present on the SPIA 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

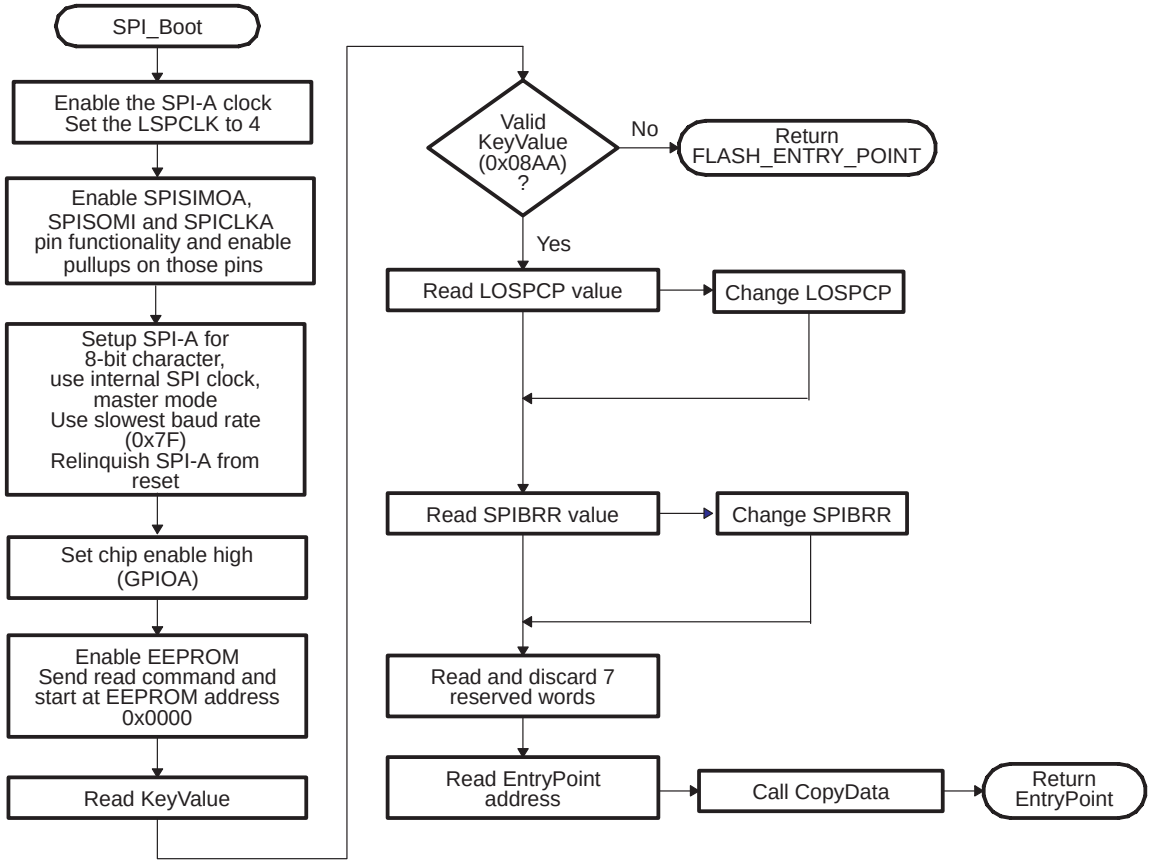
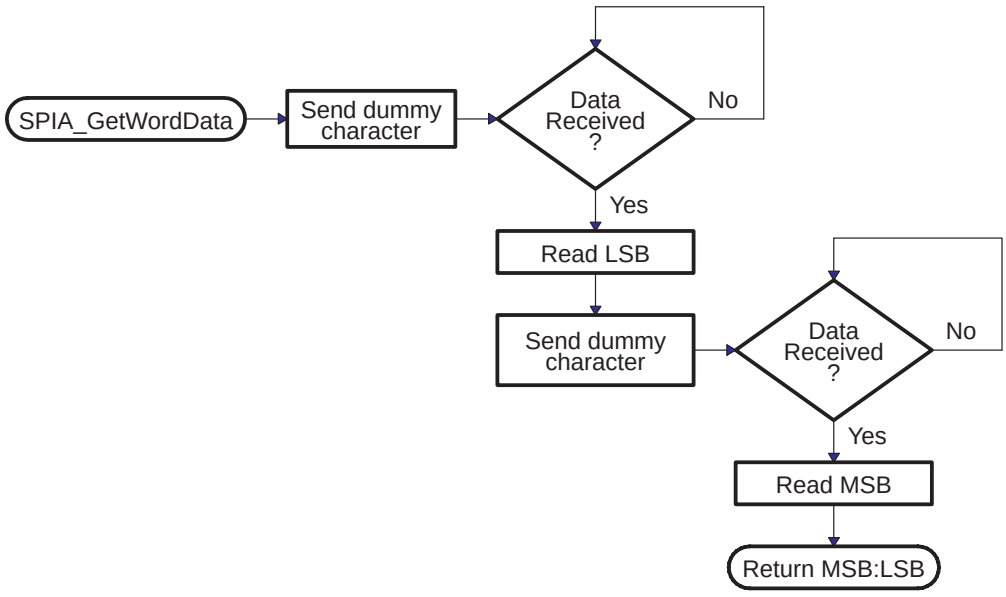


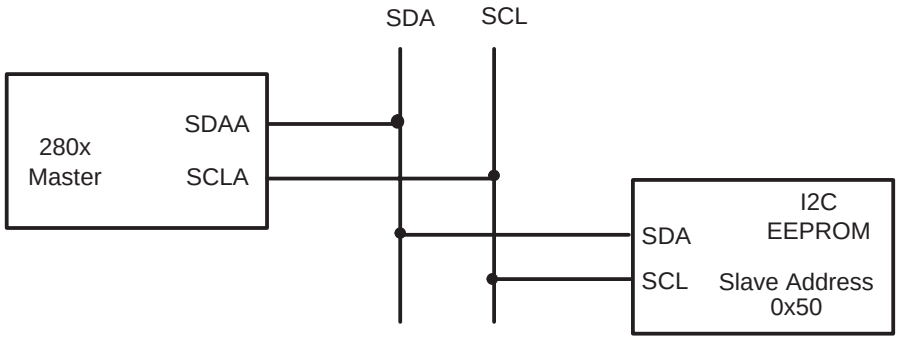
Figure 23. Overview of SPIA_GetWordData Function



4.13 I2C Boot Function

The I2C loader expects an 8-bit wide I2C-compatible EEPROM device to be present at address 0x50 on the I2C-A bus as indicated in Figure 24. The EEPROM must adhere to conventional I2C EEPROM protocol, as described in this section, with a 16-bit base address architecture.

Figure 24. EEPROM Device at Address 0x50

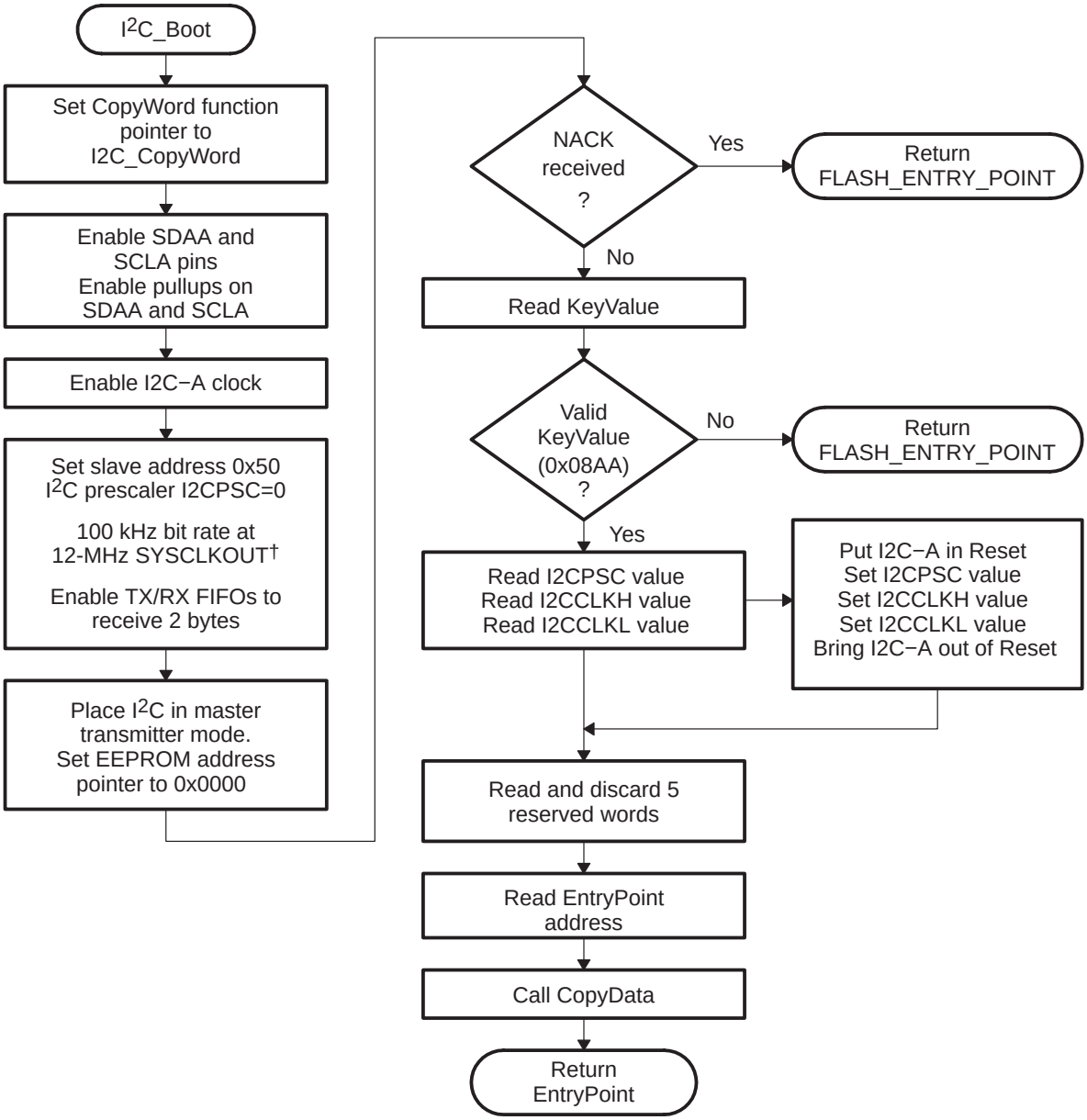


If the download is to be performed from a device other than an EEPROM, then that device must be set up to operate in the slave mode and mimic the I2C EEPROM. Immediately after entering the I2C Boot function, the GPIO pins are

configured for I2C-A operation and the I^2C is initialized. The following requirements must be met when booting from the I^2C module:

- The input frequency to the device must be between 14 MHz and 24 MHz
- The EEPROM must be at slave address 0x50

Figure 25. Overview of I2C_Boot Function



† During device boot, SYSCLKOUT will be the device input frequency divided by two.

To use the I2C-A boot loader, the input clock frequency to the device must be between 14 MHz and 24 MHz. This input clock frequency will result in a default 7 MHz to 12 MHz system clock (SYSCLKOUT). By default, the boot loader sets the I2CPSC prescale value to 0 so that the I2C clock will not be divided down

from SYSCLKOUT. This results in an I2C clock between 7 MHz and 12 MHz, which meets the 280x I2C peripheral clocking specification. The I2CPSC value can be modified after receiving the first few bytes from the EEPROM, but it is not advisable to do this, because this can cause the I2C to operate out of the required specification.

The bit-period prescalers (I2CCLKH and I2CCLKL) are configured by the boot loader to run the I2C at a 50 percent duty cycle at 100-kHz bit rate (standard I2C mode) when the system clock is 12 MHz. These registers can be modified after receiving the first few bytes from the EEPROM. This allows the communication to be increased up to a 400-kHz bit rate (fast I2C mode) during the remaining data reads.

Arbitration, bus busy, and slave signals are not checked. Therefore, no other master is allowed to control the bus during this initialization phase. If the application requires another master during I2C boot mode, that master must be configured to hold off sending any I2C messages until the 280x application software signals that it is past the bootloader portion of initialization.

The nonacknowledgment bit is checked only during the first message sent to initialize the EEPROM base address. This assures that an EEPROM is present at address 0x50 before continuing. If an EEPROM is not present, code will jump to the Flash entry point. The nonacknowledgement bit is not checked during the address phase of the data read messages (I2C_Get Word). If a non acknowledgement is received during the data read messages, the I2C bus will hang. Table 9 shows the 8-bit data stream.

Table 9. I^2C 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:I2CPSC[7:0]
4	Reserved
5	LSB: I2CCLKH[7:0]
6	MSB: I2CCLKH[15:8]
7	LSB: I2CCLKL[7:0]
8	MSB: I2CCLKL[15:8]
…	
17	LSB = Reserved for future use
18	MSB = Reserved for future use
19	LSB: Upper half of entry point PC
20	MSB: Upper half of entry point PC[22:16] (Note: Always 0x00)
21	LSB: Lower half of entry point PC[15:8]
22	MSB: Lower half of entry point PC[7:0]
	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 I^2C EEPROM protocol required by the I^2C boot loader is shown in Figure 26 and Figure 27. The first communication, which sets the EEPROM address pointer to 0x0000 and reads the KeyValue (0x08AA) from it, is shown in Figure 26. All subsequent reads are shown in Figure 27 and are read two bytes at a time.

Figure 26. Random Read

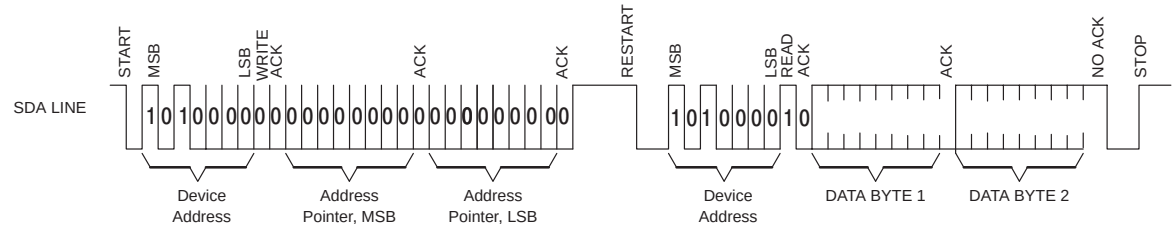
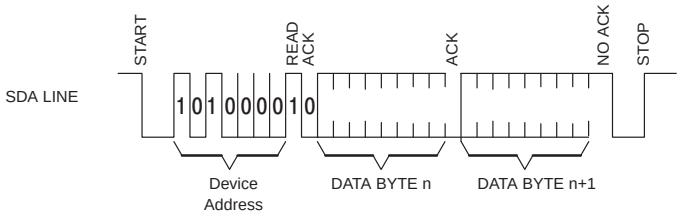


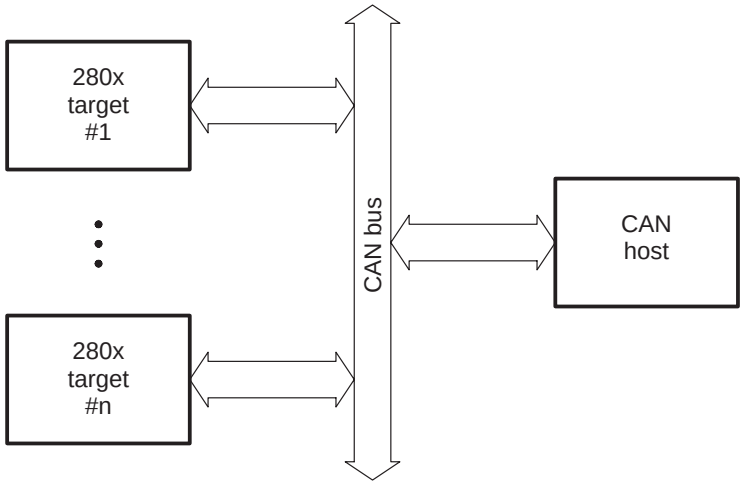
Figure 27. Sequential Read



4.14 CAN Boot Function

The CAN boot mode asynchronously transfers code from eCAN-A to internal memory. The host can be any CAN node. The communication is first done with 11-bit standard identifiers (with a MSGID of 0x1) using two bytes per data frame. The host could then download a kernel to reconfigure the CAN if higher data throughput is desired.

Figure 28. Overview of eCAN-A Boot Loader Operation



The bit-timing registers are programmed in such a way that a valid bit-rate is achieved for different XCLKIN values as shown in Table 10.

Table 10. Bit-Rate Values for Different XCLKIN Values

XCLKIN	SYSCLKOUT	Bit Rate
40 MHz	20 MHz	1 Mbits/s
20 MHz	10 MHz	500 kbits/s
10 MHz	5 MHz	250 kbits/s
5 MHz	2.5 MHz	125 kbits/s

The SYSCLKOUT values show are the reset values with the default PLL setting. The BRP_{reg} and bit-time values are hard coded to 1 and 10, respectively.

Mailbox 1 is programmed with a standard MSGID of 0x1 for boot-loader communication. The CAN host should transmit only 2 bytes at a time, LSB first and MSB next. For example, to transmit the word 0x08AA to the 280x, transmit AA first, followed by 08. The program flow of the CAN boot loader is identical to the SCI boot loader. The data sequence for the CAN bootloader is as follows:

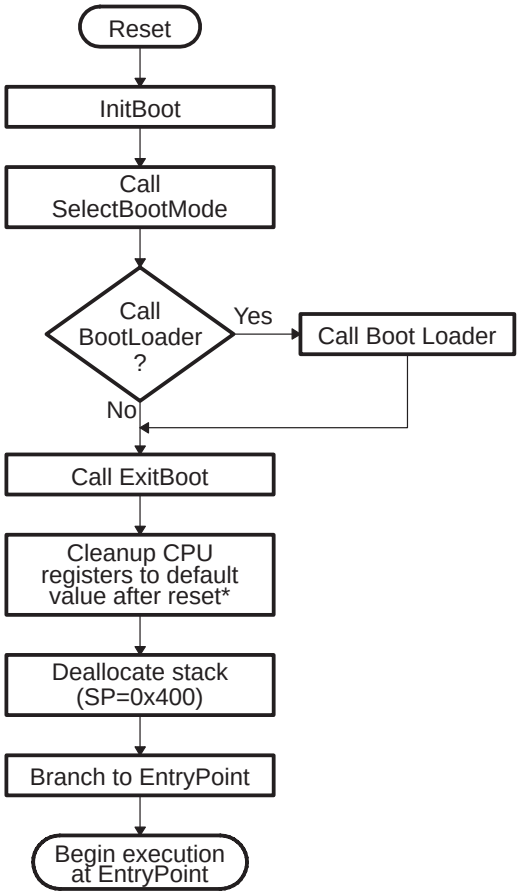
AA 08 −	Keyvalue
00 00 −	Part of 8 reserved words stream
00 00 −	Part of 8 reserved words stream
00 00 −	Part of 8 reserved words stream
00 00 −	Part of 8 reserved words stream
00 00 −	Part of 8 reserved words stream
00 00 −	Part of 8 reserved words stream
00 00 −	Part of 8 reserved words stream
00 00 −	Part of 8 reserved words stream
bb aa −	Most significant (MS) part of 32-bit address (aabb)
dd cc −	Least significant (LS) part of 32-bit address (ccdd) − Final entry−point address = 0xaabbccdd
nn mm	Length of first section (mm nn)

ff ee −	MS part of 32-bit address (eeff)
hh gg −	LS part of 32-bit address (gghh) − Starting address of first section = 0xeeffgghh
xx xx −	First word of first section
xx xx −	Second word of first section
... ...	
xx xx −	Last word of first section
nn mm −	Length of second section (mm nn)
ff ee −	MS part of 32-bit address (eeff)
hh gg	LS part of 32-bit address (gghh) − Starting address of second section = 0xeeffgghh
xx xx −	First word of second section
xx xx −	Second word of second section
... ...	
xx xx −	Last word of second section (more sections if needed)
00 00 −	Section length of zero for next section indicates end of data.

4.15 ExitBoot Assembly Routine

The Boot ROM includes an 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 29. 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 11. 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 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, I^2C, eCAN, 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 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. Updates will be made to support the I^2C boot. See the Codegen release notes for the latest information.

Table 12. 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.
−i2c8	Specify the source of the boot loader table as the I2C-A port, 8-bit
−i2cpsc value	Specify the value for the I2CPSC register. This value will be loaded and take effect after all I2C options are loaded, prior to reading data from the EEPROM. This value will be truncated to the least significant eight bits and should be set to maintain an I2C module clock of 7−12 MHz.
−i2cclkh value	Specify the value for the I2CCLKH register. This value will be loaded and take effect after all I2C options are loaded, prior to reading data from the EEPROM.
−i2cclkl value	Specify the value for the I2CCLKL register. This value will be loaded and take effect after all I2C options are loaded, prior to reading data from the EEPROM.