Preliminary

TMS320x2802x Piccolo Boot ROM

Reference Guide

Literature Number: SPRUFN6
December 2008

Contents

List of Figures

List of Tables

Read This First

This reference guide is applicable for the code and data stored in the on-chip boot ROM on the TMS320x2802x Piccolo™ processors. This includes all devices within this family.

The boot ROM is factory programmed with boot-loading software. Boot-mode signals ($\overline{\text{TRST}}$ and general purpose I/Os) are used to tell the bootloader software which mode to use on power up. The boot ROM also contains standard math tables, such as SIN/COS waveforms, for use in IQ math related algorithms found in the *C28x™ IQMath Library - A Virtual Floating Point Engine* (literature number SPRC087).

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.

This guide refers to associated code that can be downloaded at http://www-s.ti.com/sc/techlit/sprufn6.zip

Notational Conventions

This document uses the following conventions.

- Hexadecimal numbers are shown with the suffix h or with a leading 0x. For example, the following number is 40 hexadecimal (decimal 64): 40h or 0x40.
- Registers in this document are shown in figures and described in tables.
  - Each register figure shows a rectangle divided into fields that represent the fields of the register. Each field is labeled with its bit name, its beginning and ending bit numbers above, and its read/write properties below. A legend explains the notation used for the properties.
  - Reserved bits in a register figure designate a bit that is used for future device expansion.

Related Documentation From Texas Instruments

The following documents describe the related devices and related support tools. Copies of these documents are available on the Internet at www.ti.com. *Tip:* Enter the literature number in the search box provided at www.ti.com.

Data Manual—

SPRS523— TMS320F28022, 28023, 28024, 28025, 28026, 28027 Microcontrollers (MCUs) contains the pinout, signal descriptions, as well as electrical and timing specifications for the 2802x devices.

CPU User's Guides—

SPRU430— TMS320C28x DSP CPU and Instruction Set Reference Guide describes the central processing unit (CPU) and the assembly language instructions of the TMS320C28x fixed-point digital signal processors (DSPs). It also describes emulation features available on these DSPs.

Peripheral Guides—

SPRU566— TMS320x28xx, 28xxx DSP Peripheral Reference Guide describes the peripheral reference guides of the 28x digital signal processors (DSPs).

SPRUFN3— TMS320x2802x Piccolo System Control and Interrupts Reference Guide describes the various interrupts and system control features of the C2802x microcontrollers (MCUs).

SPRUFN6— TMS320x2802x Piccolo Boot ROM Reference Guide describes the purpose and features of the bootloader (factory-programmed boot-loading software) and provides examples of code. It also describes other contents of the device on-chip boot ROM and identifies where all of the information is located within that memory.

SPRUGE5— TMS320x2802x Piccolo Analog-to-Digital Converter (ADC) and Comparator Reference Guide describes how to configure and use the on-chip ADC module, which is a 12-bit pipelined ADC.

SPRUGE9— TMS320x2802x Piccolo Enhanced Pulse Width Modulator (ePWM) Module Reference Guide describes the main areas of the enhanced pulse width modulator that include digital motor control, switch mode power supply control, UPS (uninterruptible power supplies), and other forms of power conversion.

SPRUGE8— TMS320x2802x Piccolo High-Resolution Pulse Width Modulator (HRPWM) describes the operation of the high-resolution extension to the pulse width modulator (HRPWM).

SPRUGH1— TMS320x2802x Piccolo Serial Communications Interface (SCI) Reference Guide describes how to use the SCI.

SPRUFZ8— TMS320x2802x Piccolo Enhanced Capture (eCAP) Module Reference Guide describes the enhanced capture module. It includes the module description and registers.

SPRUG71— TMS320x2802x Piccolo Serial Peripheral Interface (SPI) Reference Guide describes the SPI - a high-speed synchronous serial input/output (I/O) port - that allows a serial bit stream of programmed length (one to sixteen bits) to be shifted into and out of the device at a programmed bit-transfer rate.

SPRUFZ9— TMS320x2802x Piccolo Inter-Integrated Circuit (I2C) Reference Guide describes the features and operation of the inter-integrated circuit (I2C) module.

Tools Guides—

SPRU513— TMS320C28x Assembly Language Tools User's Guide describes the assembly language tools (assembler and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the TMS320C28x device.

SPRU514— TMS320C28x Optimizing C Compiler User's Guide describes the TMS320C28x™ C/C++ compiler. This compiler accepts ANSI standard C/C++ source code and produces TMS320 DSP assembly language source code for the TMS320C28x device.

SPRU608— The TMS320C28x Instruction Set Simulator Technical Overview describes the simulator, available within the Code Composer Studio for TMS320C2000 IDE, that simulates the instruction set of the C28x™ core.

Boot ROM Overview

The boot ROM is a block of read-only memory that is factory programmed.

Topic **Page**

1.1 Boot ROM Memory Map

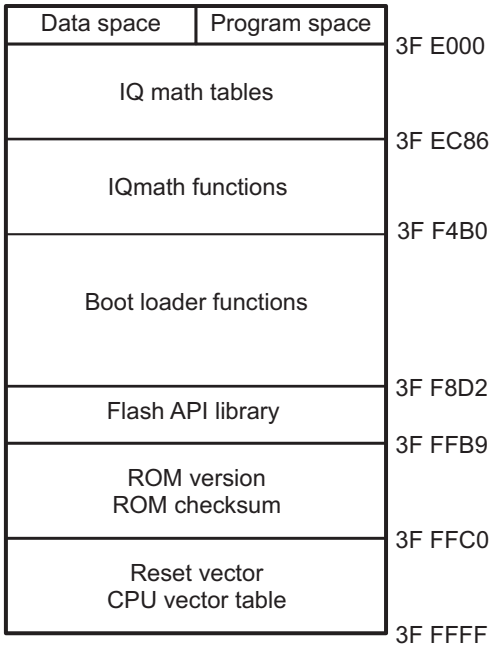
The boot ROM is an 8K x 16 block of read-only memory located at addresses 0x3F E000 - 0x3F FFFF.

The on-chip boot ROM is factory programmed with boot-load routines and fixed-point math tables. These are for use with the *C28x™ IQMath Library - A Virtual Floating Point Engine* (SPRC087). This document describes the following items:

- Bootloader functions
- Version number, release date and checksum
- Reset vector
- Illegal trap vector (ITRAP)
- CPU vector table (Used for test purposes only)
- IQmath Tables
- Selected IQmath functions
- Flash API library

Figure 1-1 shows the memory map of the on-chip boot ROM. The memory block is 8Kx16 in size and is located at 0x3F E000 - 0x3F FFFF in both program and data space.

Figure 1-1. Memory Map of On-Chip ROM



1.2 On-Chip Boot ROM IQmath Tables

The fixed-point math tables and functions included in the boot ROM are used by the Texas Instruments™ *C28x™ IQMath Library - A Virtual Floating Point Engine* (SPRC087). 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.

IQmath library accesses the tables through the IQmathTables and the IQmathTablesRam linker sections. Both of these sections are completely included in the boot ROM. If you do not wish to load a copy of these tables already included in the ROM into the device, use the boot ROM memory addresses and label the sections as “NOLOAD” as shown in Example 1-1. This facilitates referencing the look-up tables without actually loading the section to the target.

The preferred alternative to using the linker command file is to use the IQmath boot ROM symbol library. If this library is linked in the project before the IQmath library, and the linker -priority option is used, then any math tables and IQmath functions within the boot ROM will be used first. Refer to the IQMath Library documentation for more information.

Example 1-1. Linker Command File to Access IQ Tables

```
MEMORY
{
   PAGE 0 :
   ...
   IQTABLES  =)   : origin = 0x3FE000, length = 0x000b50
   IQTABLES2  =)  : origin = 0x3FEB50, length = 0x00008c
   IQTABLES3  =)  : origin = 0x3FEBDC, length = 0x0000AA
   ...
}
SECTIONS
{
   ...
   IQmathTables    : load = IQTABLES, type = NOLOAD, PAGE = 0
   IQmathTables2 > IQTABLES2, type = NOLOAD, PAGE = 0
   {
       IQmath.lib<IQNexpTable.obj> (IQmathTablesRam)
   }
   IQmathTables3   : load = IQTABLES3, PAGE = 0
   {
       IQNasinTable.obj (IQmathTablesRam)
   }
   ...
}
```

The following math tables are included in the Boot ROM:

- **Sine/Cosine Table, IQ Math Table**
  - Table size: 1282 words
  - Q format: Q30
  - Contents: 32-bit samples for one and a quarter period sine wave

  This is useful for accurate sine wave generation and 32-bit FFTs. This can also be used for 16-bit math, just skip over every second value.
- **Normalized Inverse Table, IQ Math 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, IQ Math 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, IQ Math Table**
  - Table size: 452 words
  - Q format: Q30
  - Contents 32-bit second order coefficients for line of best 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, IQ Math Table**
  - Table size: 360 words
  - Q format: Q30
  - Contents: 32-bit rounding and saturation limits for various Q values
- **Exp Min/Max Table, IQMath Table**
  - Table size: 120 words
  - Q format: Q1 - Q30
  - Contents: 32-bit Min and Max values for each Q value
- **Exp Coefficient Table, IQMath Table**
  - Table size: 20 words
  - Q format: Q31
  - Contents: 32-bit coefficients for calculating exp (X) using a taylor series
- **Inverse Sin/Cos Table, IQ Math Table**
  - Table size: 85 x 16
  - Q format: Q29
  - Contents: Coefficient table to calculate the formula f(x) = c4*x^4 + c3*x^3 + c2*x^2 + c1*x + c0.

1.3 On-Chip Boot ROM IQmath Functions

The following IQmath functions are included in the Boot ROM:

- IQNatan2 N= 15, 20, 24, 29
- IQNcos N= 15, 20, 24, 29
- IQNdiv N= 15, 20, 24, 29
- IQisqrt N= 15, 20, 24, 29
- IQNmag N= 15, 20, 24, 29
- IQNsin N= 15, 20, 24, 29
- IQNsqrt N= 15, 20, 24, 29

These functions can be accessed using the IQmath boot ROM symbol library included with the boot ROM source. If this library is linked in the project before the IQmath library, and the linker -priority option is used, then any math tables and IQmath functions within the boot ROM will be used first. Refer to the IQMath Library documentation for more information.

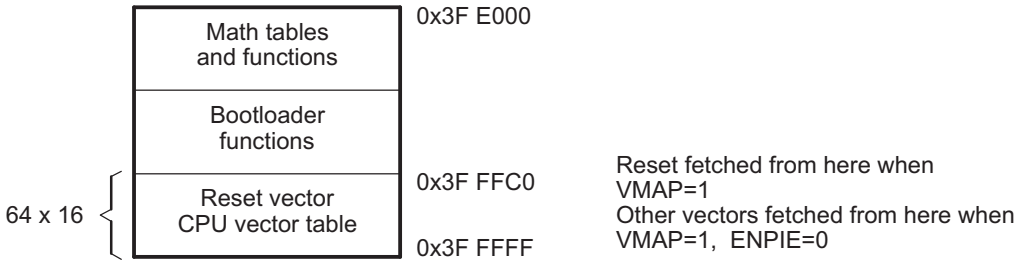
1.4 On-Chip Flash API

The boot ROM contains the API to program and erase the flash. This flash API can be accessed using the boot ROM flash API symbol library released with the boot ROM source. Refer to the 2802x Flash API Library documentation for information on how to use the symbol library.

1.5 CPU Vector Table

A CPU vector table resides in boot ROM memory from address 0x3F E000 - 0x3F FFFF. This vector table is active after reset when VMAP = 1, ENPIE = 0 (PIE vector table disabled).

Figure 1-2. Vector Table Map



A 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.

B 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 stored in the boot ROM. This function starts the boot load process. A series of checking operations is performed on $\overline{\text{TRST}}$ and General-Purpose I/O (GPIO I/O) pins to determine which boot mode to use. This boot mode selection is described in Section 2.9 of this document.

The remaining vectors in the boot 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 in Table 1-1.

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 Table 1-1. 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 1-1. 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	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	ITRAPIsr
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

Bootloader Features

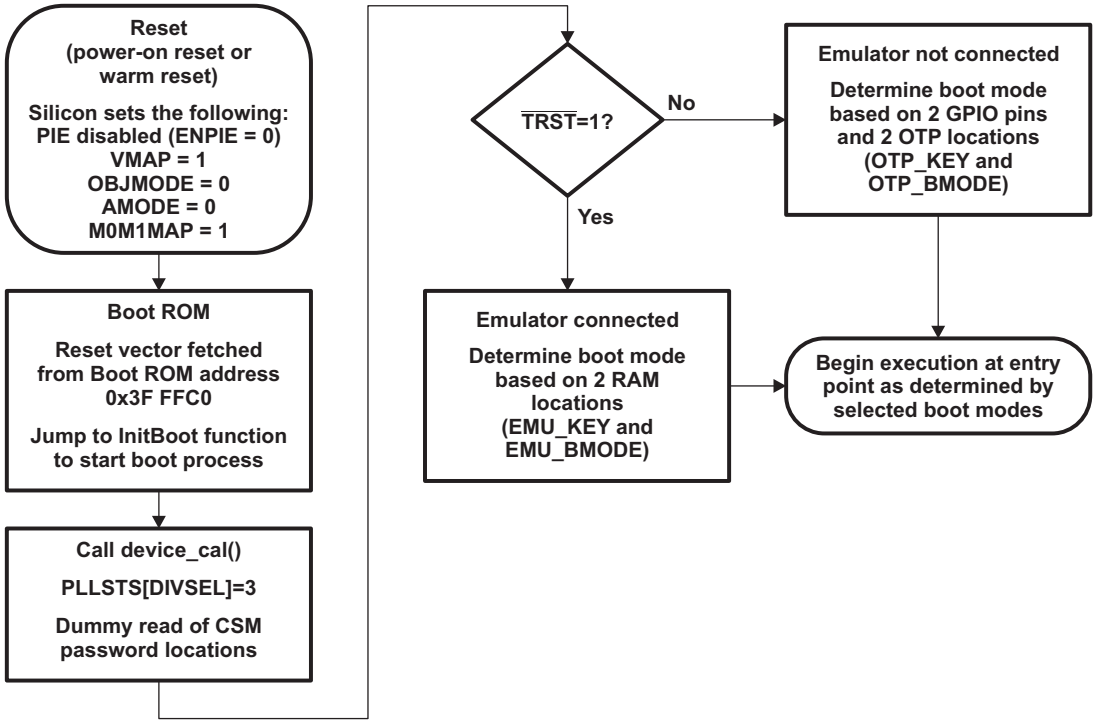
This section describes in detail the boot mode selection process, as well as the specifics of the bootloader operation.

Topic **Page**

2.1 Bootloader Functional Operation

The bootloader uses the state of $\overline{\text{TRST}}$ and two GPIO signals to determine which boot mode to use. The boot mode selection process and the specifics of each bootloader are described in the remainder of this document. Figure 2-1 shows the basic bootloader flow:

Figure 2-1. Bootloader Flow Diagram



The reset vector in boot ROM redirects program execution to the InitBoot function. After performing device initialization the bootloader will check the state of the $\overline{\text{TRST}}$ pin to determine if an emulation pod is connected.

- **Emulation Boot (Emulation Pod is connected and $\overline{\text{TRST}}$ = 1)**

In emulation boot, the boot ROM will check two SARAM locations called EMU_KEY and EMU_BMODE for a boot mode. If the contents of either location are invalid, then the "wait" boot mode is used. All boot mode options can be accessed by modifying the value of EMU_BMODE through the debugger when performing an emulation boot.

- **Stand-alone Boot ($\overline{\text{TRST}}$ = 0)**

If the device is in stand-alone boot mode, then the state of two GPIO pins are used to determine which boot mode execute. Options include: GetMode, wait, SCI, and parallel I/O. Each of the modes is described in detail in Table 2-3. The GetMode option by default boots to flash but can be customized by programming two values into OTP to select another boot loader.

These boot modes mentioned here are discussed in detail in Section 2.9.

After the selection process and if the required boot loading is complete, the processor will continue execution at an entry point determined by the boot mode selected. If a bootloader was called, then the input stream loaded by the peripheral determines this entry address. This data stream is described in Section 2.11. 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.

2.2 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 2-1.

Table 2-1. 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	1
Other Settings			SXM = 1, C = 1, SPM = 0

(1) Normally for C27x compatibility, the M0M1MAP would be 0. On these devices, however, it is tied off high internally; therefore, at reset, M0M1MAP is always configured for 28x mode.

2.3 PLL Multiplier and DIVSEL Selection

The Boot ROM changes the PLL multiplier (PLLCR) and divider (PLLSTS[DIVSEL]) bits as follows:

- **All boot modes:**
  PLLCR is not modified. PLLSTS[DIVSEL] is set to 3 for SYSCLKOUT = CLKIN/1. This increases the speed of the loaders.

Note: The PLL multiplier (PLLSTS) and divider (PLLSTS[DIVSEL]) are 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.

Note: The reset value of PLLSTS[DIVSEL] is 0. This configures the device for SYSCLKOUT = CLKIN/4. The boot ROM will change this to SYSCLKOUT = CLKIN/1 to improve performance of the loaders. PLLSTS[DIVSEL] is left in this state when the boot ROM exits and it is up to the application to change it before configuring the PLLCR register.

Note: The boot ROM leaves PLLSTS[DIVSEL] in the CLKIN/1 state when the boot ROM exits. This is not a valid configuration if the PLL is used. Thus the application must change it before configuring the PLLCR register.

2.4 Watchdog Module

When branching directly to Flash, OTP, or M0 single-access RAM (SARAM) the watchdog is not touched. In the other boot modes, the watchdog is disabled before booting and then re-enabled and cleared before branching to the final destination address. In the case of an incorrect key value passed to the loader, the watchdog will be enabled and the device will boot to flash.

2.5 Taking an ITRAP Interrupt

If an illegal opcode is fetched, the 28x will take an ITRAP (illegal trap) interrupt. During the boot process, the interrupt vector used by the ITRAP is within the CPU vector table of the boot ROM. The ITRAP vector points to an interrupt service routine (ISR) within the boot ROM named ITRAPIsr(). This interrupt service routine attempts to enable the watchdog and then loops forever until the processor is reset. This ISR will be used for any ITRAP until the user's application initializes and enables the peripheral interrupt expansion (PIE) block. Once the PIE is enabled, the ITRAP vector located within the PIE vector table will be used.

2.6 Internal Pullup Resisters

Each GPIO pin has an internal pullup resistor that can be enabled or disabled in software. The pins that are read by the boot mode selection code to determine the boot mode selection have pull-ups enabled after reset by default. In noisy conditions it is still recommended that you configure each of the boot mode selection pins externally.

The peripheral bootloaders all enable the pullup resistors for the pins that are used for control and data transfer. The bootloader leaves the resistors enabled for these pins when it exits. For example, the SCI-A bootloader enables the pullup resistors on the SCITXA and SCIRXA pins. It is your responsibility to disable them, if desired, after the bootloader exits.

2.7 PIE Configuration

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

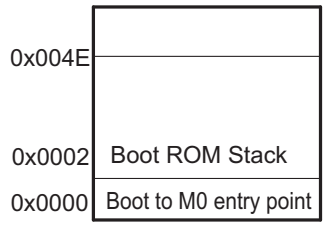
The boot ROM does, however, use the first 6 locations within the PIE vector table for emulation boot mode information and Flash API variables. These locations are not used by the PIE itself and not used by typical applications.

Note: If you are porting code from another 28x processor, check to see if the code initializes the first 6 locations in the PIE vector table to some default value. If it does, then consider modifying the code to not write to these locations so the EMU boot mode will not be over written during debug. Refer to the C2802x C/C++ Header Files and Peripheral Examples.

2.8 Reserved Memory

The M0 memory block address range 0x0002 - 0x004E is reserved for the stack and .ebss code sections 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. Address 0x0000-0x0001 is the boot to M0 entry point. This should be loaded with a branch instruction to the start of the main application when using "boot to SARAM" mode.

Figure 2-2. Boot ROM Stack



Boot ROM loaders on other C28x devices had the stack in M1 memory.

Note: If code or data is bootloaded into the address range address range 0x0002 - 0x004E there is no error checking to prevent it from corrupting the boot ROM stack.

In addition, the first 6 locations of the PIE vector table are used by the boot ROM. These locations are not used by the PIE itself and not used by typical applications. These locations are used as SARAM by the boot ROM and will not effect the behavior of the PIE. Note: Some example code from previous devices may initialize these locations. This will overwrite any boot mode you have populated. These locations are:

Table 2-2. PIE Vector SARAM Locations Used by the Boot ROM

Location	Name	Note
0x0D00 x 16	EMU_KEY	Used for emulation boot
0x0D01 x 16	EMU_BMODE	Used for emulation boot
0x0D02 x 32	Flash_CPUScaleFactor	Used by the flash API
0x0D04 x 32	Flash_CallbackPtr	Used by the flash API

2.9 Bootloader Modes

To accommodate different system requirements, the 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 $\overline{\text{TRST}}$ and two GPIO pins are used to determine the desired boot mode as shown in Table 2-3.

Table 2-3. Boot Mode Selection

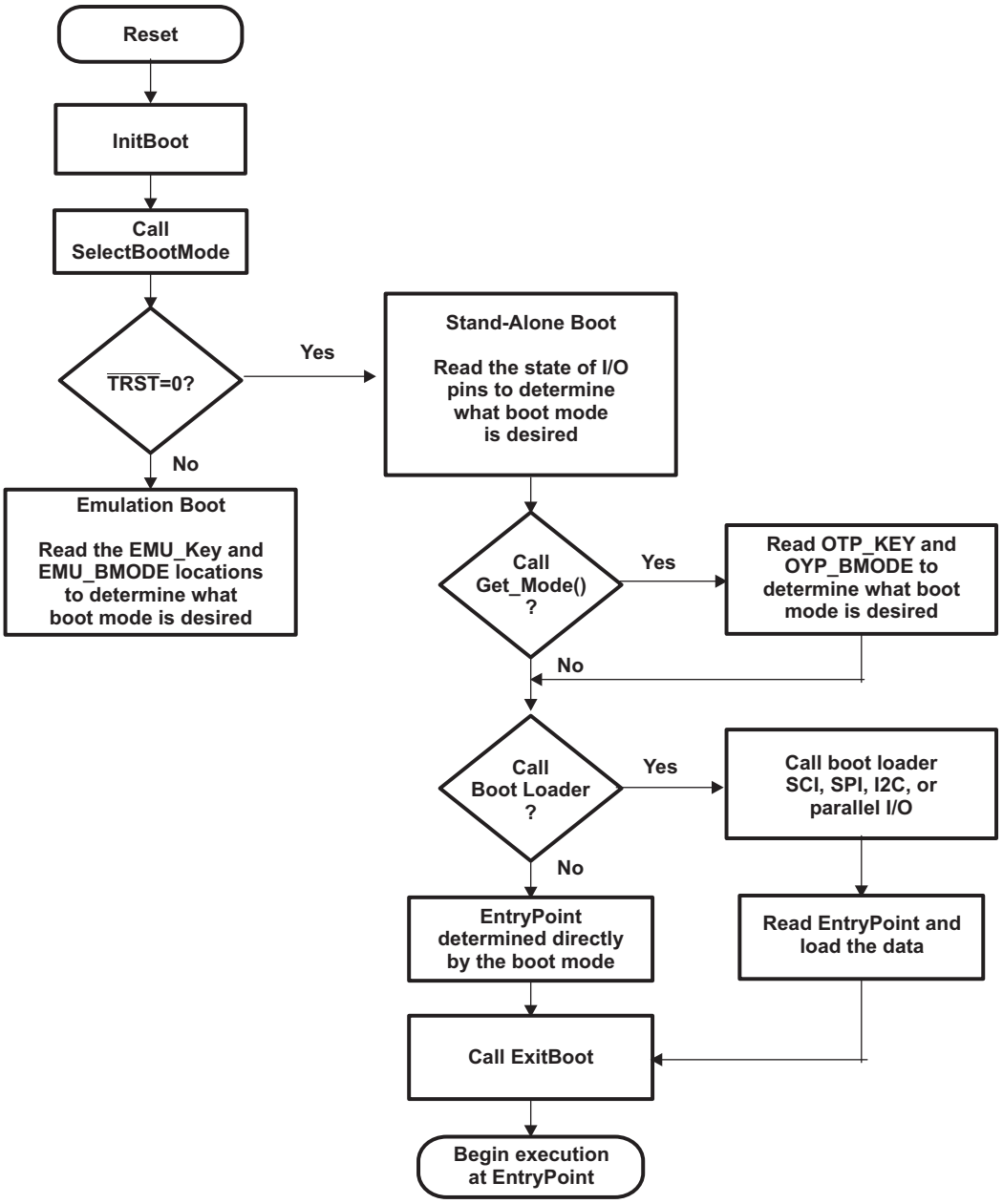
	GPIO37 TDO	GPIO34 CMP2OUT	$\overline{\text{TRST}}$	
Mode EMU	x	x	1	Emulation Boot
Mode 0	0	0	0	Parallel I/O
Mode 1	0	1	0	SCI
Mode 2	1	0	0	Wait
Mode 3	1	1	0	GetMode

Note: The default behavior of the GetMode option on unprogrammed devices is to boot to flash. This behavior can be changed by programming two locations in the OTP as shown in Table 2-5. In addition, if these locations are used by an application, then GetMode will jump to flash as long as OTP_KEY != 0x55AA and/or OTP_BMODE is not a valid value.

Note: The 2802x devices do not support the hardware wait-in-reset mode that is available on other C2000 parts. The "wait" boot mode can be used to emulate a wait-in-reset mode. The "wait" mode is very important for debugging devices with the CSM password programmed (i.e., secured). When the device is powered up, the CPU will start running and may execute an instruction that performs an access to a protected emulation code security logic (ECSL) area. If this happens, the ECSL will trip and cause the emulator connection to be cut. The "wait" mode keeps this from happening by looping within the boot ROM until an emulator is connected.

Figure 2-3 shows an overview of the boot process. Each step is described in greater detail in following sections.

Figure 2-3. Boot ROM Function Overview



The following boot mode is used when an emulator is connected:

- **Emulation Boot**

  In this case an emulation pod is connected to the device ($\overline{TRST}$ = 1) and the boot ROM derives the boot mode from the first two locations in the PIE vector table. These locations, called EMU_KEY and EMU_BMODE, are not used by the PIE module and not typically used by applications. Valid values for EMU_KEY and EMU_BMODE are shown in Table 2-4.

  An EMU_KEY value of 0x55AA indicates the EMU_BMODE is valid. An invalid key or invalid mode will result in calling the wait boot mode. EMU_BMODE and EMU_KEY are automatically populated by the boot ROM when powering up with $\overline{TRST}$ = 0. EMU_BMODE can also be initialized manually through the debugger.

Table 2-4. Valid EMU_KEY and EMU_BMODE Values

Address	Name	Value	
0x0D00	EMU_KEY	if TRST == 1 and EMU_KEY == 0x55AA, then check EMU_BMODE for the boot mode, else { Invalid EMU_KEY Boot mode = WAIT_BOOT }	
0x0D01	EMU_BMODE	0x0000	Boot mode = PARALLEL_BOOT
		0x0001	Boot mode = SCI_BOOT
		0x0002	Boot mode = WAIT_BOOT
		0x0003	Boot mode = GET_BOOT (GetMode from OTP_KEY/OTP_BMODE)
		0x0004	Boot mode = SPI_BOOT
		0x0005	Boot mode = I2C_BOOT (1)
		0x0006	Boot mode = OTP_BOOT
		0x000A	Boot mode = RAM_BOOT
		0x000B	Boot mode = FLASH_BOOT
		Other	Boot mode = WAIT_BOOT

(1) I2C boot uses GPIO32 and GPIO33 which are not available on all packages.

Table 2-6 shows the expanded emulation boot mode table.

Here are two examples of an emulation boot:

Example 2-1. Debug an application that loads through the SCI at boot.

To debug an application that loads through the SCI at boot, follow these steps:

- Configure the pins for mode 1, SCI, and initiate a power-on-reset.
- The boot ROM will detect $\overline{TRST}$ = 0 and will use the two pins to determine SCI boot.
- The boot ROM populates EMU_KEY with 0x55AA and EMU_BMODE with SCI_BOOT.
- The boot ROM sits in the SCI loader waiting for data.
- Connect the debugger. $\overline{TRST}$ will go high.
- Perform a debugger reset and run. The boot loader will use the EMU_BMODE and boot to SCI.

Example 2-2. You want to connect your emulator, but do not want application code to start executing before the emulator connects.

To connect your emulator, but keep application code from executing before the emulator connects:

- Configure GPIO37 and GPIO34 pins for mode 2, WAIT, and initiate a power-on-reset.
- The boot ROM will detect $\overline{\text{TRST}}$ = 0 and will use the two pins to determine wait boot.
- The boot ROM populates EMU_KEY with 0x55AA and EMU_BMODE with WAIT_BOOT.
- The boot ROM sits in the wait routine.
- Connect the debugger; $\overline{\text{TRST}}$ will go high.
- Modify the EMU_BMODE via the debugger to boot to FLASH or other desired boot mode.
- Perform a debugger reset and run. The boot loader will use the EMU_BMODE and boot to the desired loader or location.

Note: **The behavior of emulators with regards to $\overline{\text{TRST}}$ differs.** Some emulators pull $\overline{\text{TRST}}$ high only when Code Composer Studio is in a connected state. For these emulators, if CCS is disconnected $\overline{\text{TRST}}$ will return to a low state. With CCS disconnected, GPIO34 and GPIO37 will be used to determine the boot mode. For these emulators, this is true even if the emulator pod is physically connected.

Some emulators pull $\overline{\text{TRST}}$ high when CCS connects and leave it high as long as the power sense pin is active. $\overline{\text{TRST}}$ will remain high even after CCS disconnects. For these emulators, the EMU mode stored in RAM will be used unless the target is power cycled, causing the state of $\overline{\text{TRST}}$ to reset back to a low state.

The following boot modes are invoked by the state of the boot mode pins if an emulator is not connected:

- **Wait**

  The 2802x devices do not support the hardware wait-in-reset mode that is available on other C2000 parts. The "wait" boot mode can be used to emulate a wait-in-reset mode. The "wait" mode is very important for debugging devices with the CSM password programmed (i.e., secured). When the device is powered up, the CPU will start running and may execute an instruction that performs an access to a emulation code security logic (ECSL) protected area. If this happens, the ECSL will trip and cause the emulator connection to be cut. The "wait" mode keeps this from happening by looping within the boot ROM until an emulator is connected

  This mode writes WAIT_BOOT to EMU_BMODE. Once the emulator is connected you can then manually populate the EMU_BMODE with the appropriate boot mode for the debug session.

- **SCI**

  In this mode, the boot ROM will load code to be executed into on-chip memory via the SCI-A port. When invoked as a stand-alone mode, the boot ROM writes SCI_BOOT to EMU_BMODE.

- **Parallel I/O 8-bit**

  The parallel I/O boot mode is typically used only by production flash programmers.

- **GetMode**

  The GetMode option uses two locations within the OTP to determine the boot mode. On an un-programmed device this mode will always boot to flash. On a programmed device, you can choose to program these locations to change the behavior. If either of these locations is not an expected value, then boot to flash will be used.

  The values used by the Get_Mode() function are shown in Table 2-5.

Table 2-5. OTP Values for GetMode

Address	Name	Value
0x3D 7BFE	OTP_KEY	GetMode will be entered if one of the two conditions is true: Case 1: $\overline{\text{TRST}}$ == 0, GPIO34 == 1 and GPIO37 == 1 Case 2: $\overline{\text{TRST}}$ == 1, EMU_KEY == 0x55AA and EMU_BMODE == GET_BOOT GetMode first checks the value of OTP_KEY: if OTP_KEY == 0x55AA, then check OTP_BMODE for the boot mode else { Invalid key: Boot mode = FLASH_BOOT }
0x3D 7BFF	OTP_BMODE	0x0001 Boot mode = SCI_BOOT 0x0004 Boot mode = SPI_BOOT 0x0005 Boot mode = I2C_BOOT (1) 0x0006 Boot mode = OTP_BOOT Other Boot mode = FLASH_BOOT

(1) The I2C boot loader uses GPIO32 and GPIO33 which are not available on all packages.

The following boot modes are available through the emulation boot option. Some are also available as a programmed get mode option.

- **Jump to M0 SARAM**

  This mode is only available in emulation boot. The boot ROM software configures the device for 28x operation and branches directly to address 0X000000. This is the first address in the M0 memory block.

- **Jump to branch instruction in flash memory.**

  Jump to flash is the default behavior of the Get Mode boot option. Jump to flash is also available as an emulation boot option.

  In this mode, the boot ROM software configures the device for 28x operation and branches directly to location 0x3F 7FF6. 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 0x33 FFF6 that will redirect code execution to either a custom boot-loader or the application code.

- **SPI EEPROM or Flash boot mode (SPI-A)**

  Jump to SPI is available in stand-alone mode as a programmed Get Mode option. That is, to configure a device for SPI boot in stand-alone mode, the OTP_KEY and OTP_BMODE locations must be programmed for SPI_BOOT and the boot mode pins configured for the Get Mode boot option.

  SCI boot is also available as an emulation boot option.

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

- **I2C-A boot mode (I2C-A)**

  Jump to I2C is available in stand-alone mode as a programmed Get mode option. That is, to configure a device for I2C boot in stand-alone mode, the OTP_KEY and OTP_BMODE locations must be programmed for I2C_BOOT and the boot mode pins configured for the Get Mode boot option.

  I2C boot is also available as an emulation boot option.

  In this mode, the boot ROM will load code and data into on-chip memory from an external serial EEPROM or flash at address 0x50 on the I2C-A bus.

Table 2-6. Emulation Boot modes (TRST = 1)

$\overline{\text{TRST}}$	GPIO37 TDO	GPIO34	EMU KEY Read from 0x0D00	EMU BMODE Read from 0x0D01	OTP KEY Read from 0x3D7BFE	OTP BMODE Read from 0x3D7BFF	Boot Mode Selected (1)	EMU KEY Written to 0x0D00	EMU BMODE Written to 0x0D01
1	x (2)	x	!=0x55AA	x	x	x	Wait	-	-
			0x55AA	0x0000	x	x	Parallel I/O	-	-
				0x0001	x	x	SCI	-	-
				0x0002	x	x	Wait	-	-
				0x0003	!= 0x55AA	x	GetMode: Flash	-	-
					0X55AA	0x0001	GetMode: SCI	-	-
						0x0003	GetMode: Flash	-	-
						0x0004	GetMode: SPI	-	-
						0x0005	GetMode: I2C (3)	-	-
						0x0006	GetMode: OTP	-	-
						Other	GetMode: Flash	-	-
				0x0004	x	x	SPI	-	-
				0x0005	x	x	I2C	-	-
				0x0006	x	x	OTP	-	-
				0x000A	x	x	Boot to RAM	-	-
				0x000B	x	x	Boot to FLASH	-	-
				Other	x	x	Wait	-	-

(1) Get Mode indicated the boot mode was derived from the values programmed in the OTP_KEY and OTP_BMODE locations.
(2) x = don't care.
(3) I2C uses GPIO32 and GPIO33 which are not available on all packages.

Table 2-7. Stand-Alone boot Modes with (TRST = 0)

$\overline{\text{TRST}}$	GPIO37 TDO	GPIO34	EMU KEY Read from 0x0D00	EMU BMODE Read from 0x0D01	OTP KEY Read from 0x3D7BFE	OTP BMODE Read from 0x3D7BFF	Boot Mode Selected (1)	(2) EMU KEY Written to 0x0D00	(2) EMU BMODE Written to 0x0D01
0	0	0	x (3)	x	x	x	Parallel I/O	0x55AA	0x0000
0	0	1	x	x	x	x	SCI	0x55AA	0x0001
0	1	0	x	x	x	x	Wait	0x55AA	0x0002
0	1	1	x	x	!=0x55AA	x	GetMode: Flash	0x55AA	0x0003
					0x55AA	0x0001	GetMode: SCI		
						0x0003	GetMode: Flash		
						0x0004	GetMode: SPI		
						0x0005	GetMode: I2C (4)		
						0x0006	GetMode: OTP		
						Other	GetMode: Flash		

(1) Get Mode indicates the boot mode was derived from the values programmed in the OTP_KEY and OTP_BMODE locations.
(2) The boot ROM will write this value to EMU_KEY and EMU_BMODE. This value can be used or overwritten by the user if a debugger is connected.
(3) x = don't care.
(4) I2C uses GPIO32 and GPIO33 which are not available on all packages.

2.10 Device_Cal

The Device_cal() routine is programmed into TI reserved memory by the factory. The boot ROM automatically calls the Device_cal() routine to calibrate the internal oscillators and ADC with device specific calibration data. During normal operation, this process occurs automatically and no action is required by the user.

If the boot ROM is bypassed by Code Composer Studio during the development process, then the calibration must be initialized by application. For working examples, see the system initialization in the *C2802x C/C++ Header Files and Peripheral Examples*.

Note: Failure to initialize these registers will cause the oscillators and ADC to function out of specification. The following three steps describe how to call the Device_cal routine from an application.

Step 1: Create a pointer to the Device_cal function as shown in Example 2-3. This #define is included in the Header Files and Peripheral Examples.

Step 2: Call the function pointed to by Device_cal() as shown in Example 2-3. The ADC clocks must be enabled before making this call.

Example 2-3. Calling the Device_cal() function

```
  //Device call is a pointer to a function
  //that begins at the address shown
  # define Device_cal (void(*)(void))0x3D7C80
  ... ...

   EALLOW;
   SysCtrlRegs.PCLKCR0.bit.ADCENCLK = 1;
   (*Device_cal)();
   SysCtrlRegs.PCLKCR0.bit.ADCENCLK = 0;
   EDIS;
...
```

2.11 Bootloader Data Stream Structure

The following two tables and associated examples show the structure of the data stream incoming to the bootloader. The basic structure is the same for all the bootloaders and is based on the C54x source data stream generated by the C54x hex utility. The C28x hex utility (hex2000.exe) has been updated to support this structure. The hex2000.exe utility is included with the C2000 code generation tools. 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 bootloader the width of the incoming stream: 8 or 16 bits. Note that not all bootloaders 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 bootloader receives an invalid key value, then the load is aborted.

The next 8 words are used to initialize register values or otherwise enhance the bootloader by passing values to it. If a bootloader does not use these values then they are reserved for future use and the bootloader simply reads the value and then discards it. Currently only the SPI and I2C and parallel XINTF bootloaders use these words to initialize registers.

The 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 bootloader.

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 2-8. 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	Register initialization value or reserved for future use
4	Register initialization value or reserved for future use
5	Register initialization value or reserved for future use
6	Register initialization value or reserved for future use
7	Register initialization value or reserved for future use
8	Register initialization value or reserved for future use
9	Register initialization value or 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
.	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
...	...
...	...
n	Last word of the last block of the source being loaded
n+1	Block size of 0000h - indicates end of the source program

Example 2-4. Data Stream Structure 16-bit

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

In 8-bit mode, the least significant byte (LSB) of the word is sent first followed by the most significant byte (MSB). For 32-bit values, such as a destination address, the most significant word (MSW) is loaded first, followed by the least significant word (LSW). The bootloaders take this into account when loading an 8-bit data stream.

Table 2-9. LSB/MSB Loading Sequence in 8-Bit Data Stream

Byte		Contents	
		LSB (First Byte of 2)	MSB (Second Byte of 2)
1	2	LSB: AA (KeyValue for memory width = 8 bits)	MSB: 08h (KeyValue for memory width = 8 bits)
3	4	LSB: Register initialization value or reserved	MSB: Register initialization value or reserved
5	6	LSB: Register initialization value or reserved	MSB: Register initialization value or reserved
7	8	LSB: Register initialization value or reserved	MSB: Register initialization value or reserved
...	...	...	...
...	...	...	...
17	18	LSB: Register initialization value or reserved	MSB: Register initialization value or reserved
19	20	LSB: Upper half of Entry point PC[23:16]	MSB: Upper half of entry point PC[31:24] (Always 0x00)
21	22	LSB: Lower half of Entry point PC[7:0]	MSB: Lower half of Entry point PC[15:8]
23	24	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.	MSB: block size
25	26	LSB: MSW destination address, first block Addr[23:16]	MSB: MSW destination address, first block Addr[31:24]
27	28	LSB: LSW destination address, first block Addr[7:0]	MSB: LSW destination address, first block Addr[15:8]
29	30	LSB: First word of the first block being loaded	MSB: First word of the first block being loaded
...	...	...	...
...	...	...	...
.	.	LSB: Last word of the first block to load	MSB: Last word of the first block to load
.	.	LSB: Block size of the second block	MSB: Block size of the second block
.	.	LSB: MSW destination address, second block Addr[23:16]	MSB: MSW destination address, second block Addr[31:24]
.	.	LSB: LSW destination address, second block Addr[7:0]	MSB: LSW destination address, 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	MSB: Last word of the second block
.	.	LSB: Block size of the last block	MSB: Block size of the last block
.	.	LSB: MSW of destination address of last block Addr[23:16]	MSB: MSW destination address, last block Addr[31:24]
.	.	LSB: LSW destination address, last block Addr[7:0]	MSB: LSW destination address, 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	MSB: Last word of the last block
n	n+1	LSB: 00h	MSB: 00h - indicates the end of the source

Example 2-5. Data Stream Structure 8-bit

```
AA 08       ; 0x08AA 8-bit key value
00 00 00 00 ; 8 reserved words
00 00 00 00
00 00 00 00
00 00 00 00
3F 00 00 80 ; 0x003F8000 EntryAddr, starting point after boot load completes
05 00       ; 0x0005 - First block consists of 5 16-bit words
3F 00 10 90 ; 0x003F9010 - First block will be loaded starting at 0x3F9010
01 00       ; Data loaded = 0x0001 0x0002 0x0003 0x0004 0x0005
02 00
03 00
04 00
05 00
02 00       ; 0x0002 - 2nd block consists of 2 16-bit words
3F 00 00 80 ; 0x003F8000 - 2nd block will be loaded starting at 0x3F8000
00 77       ; Data loaded = 0x7700 0x7625
25 76
00 00       ; 0x0000 - Size of 0 indicates end of data stream
```

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

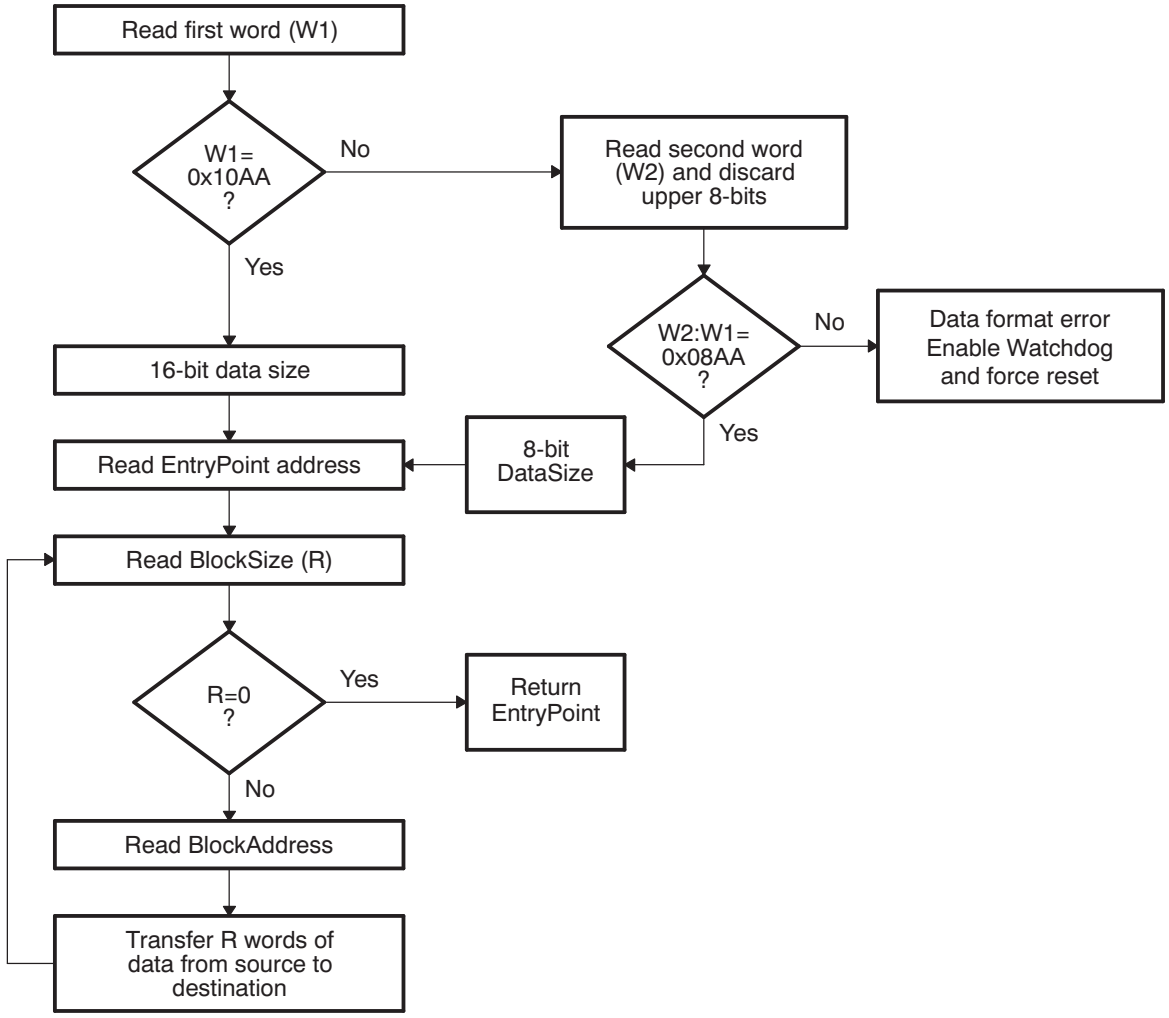
```
Location    Value
0x3F9010    0x0001
0x3F9011    0x0002
0x3F9012    0x0003
0x3F9013    0x0004
0x3F9014    0x0005
0x3F8000    0x7700
0x3F8001    0x7625
PC Begins execution at 0x3F8000
```

2.12 Basic Transfer Procedure

Figure 2-4 illustrates the basic process a bootloader 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 bootloader finds the valid boot mode selected by the state of $\overline{\text{TRST}}$ and 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.

Figure 2-4. Bootloader Basic Transfer Procedure



8-bit and 16-bit transfers are not valid for all boot modes. See the info specific to a particular bootloader for any limitations.

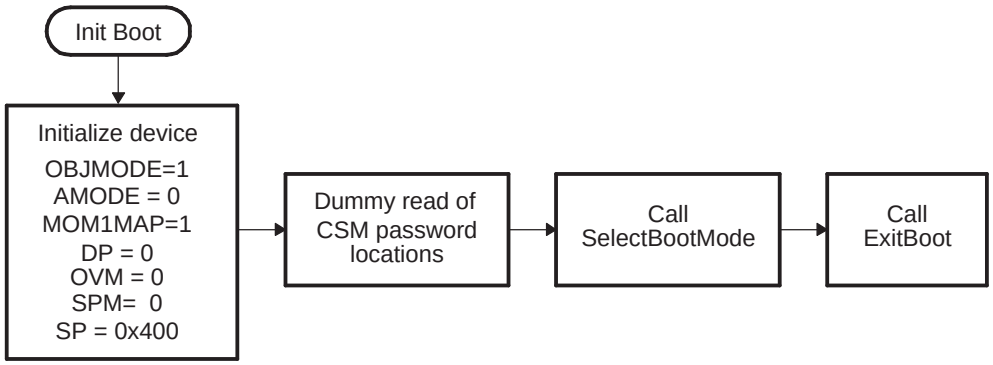
In 8-bit mode, the LSB of the 16-bit word is read first followed by the MSB.

2.13 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. InitBoot also 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 determines the type of boot mode desired by the state of $\overline{\text{TRST}}$ andcertain GPIO pins. This process is described in Section 2.14. Once the boot is complete, the SelectBootMode function passes back the entry point address (EntryAddr) to the InitBoot function. EntryAddr is the location where code execution will begin after the bootloader exits. 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 2-5. Overview of InitBoot Assembly Function



2.14 SelectBootMode Function

To determine the desired boot mode, the SelectBootMode function examines the state of $\overline{\text{TRST}}$ and 2 GPIO pins as shown in Table 2-3.

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 internal pullup resistors are enabled at reset for the boot mode selection pins. It is still suggested that the boot mode configuration be made externally to avoid the effect of any noise on these pins.

Note: The SelectBootMode routine disables the watchdog before calling the SCI, I2C, SPI, or parallel bootloaders. The bootloaders 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.

If a bootloader is not going to be called, then the watchdog is left untouched.

When selecting a boot mode, the pins should be pulled high or low through a weak pulldown or weak pull-up such that the device can drive them to a new state when required.

Figure 2-6. Overview of the SelectBootMode Function

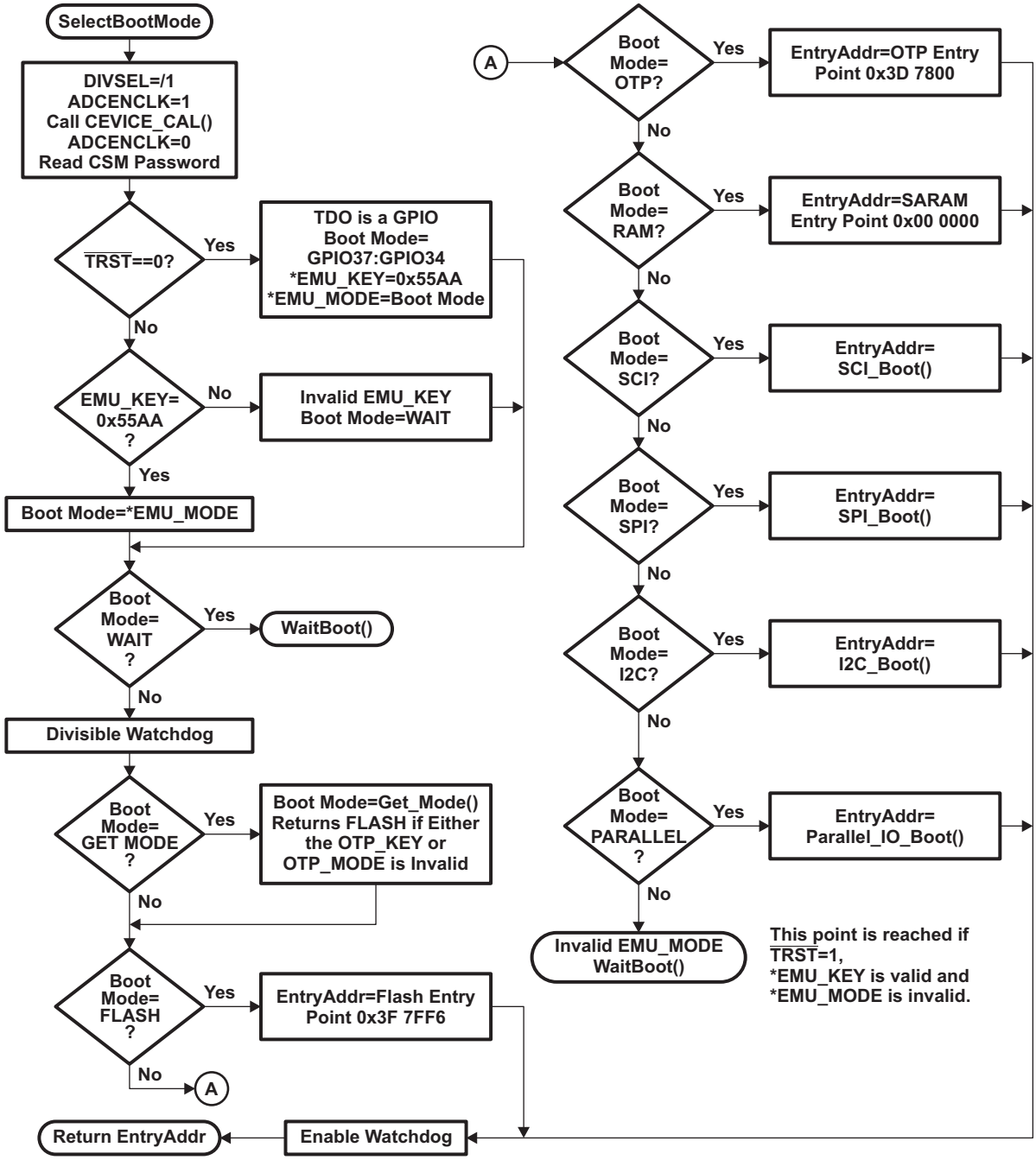
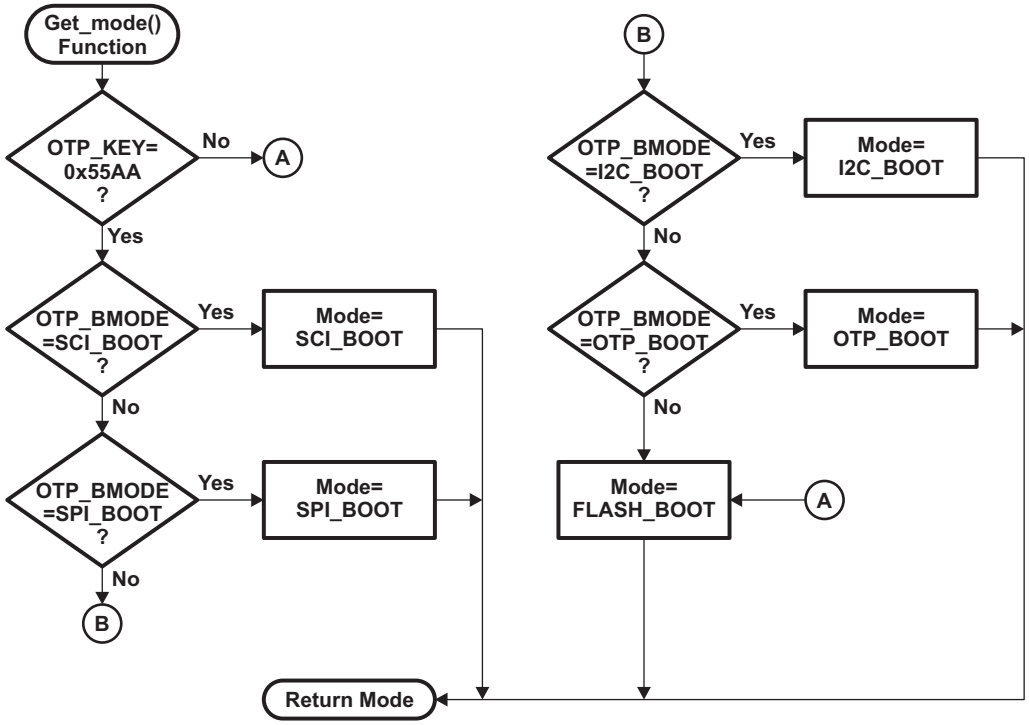


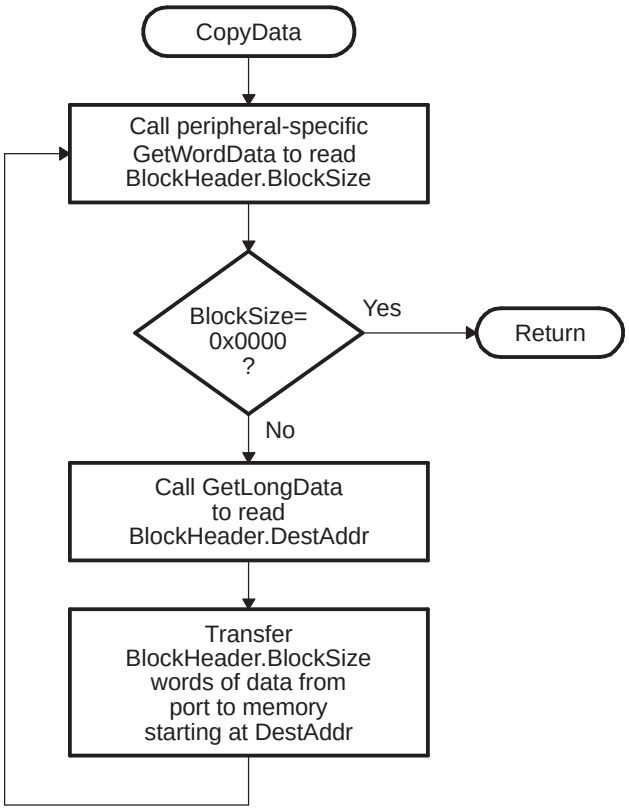
Figure 2-7. Overview of Get_mode() Function



2.15 CopyData Function

Each of the bootloaders uses the same function to copy data from the port to the device's 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 2-8.

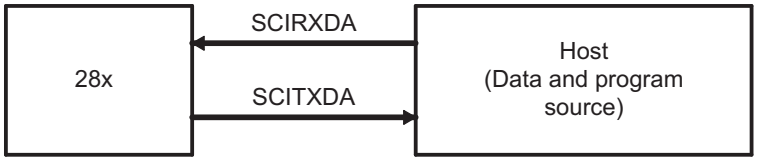
Figure 2-8. Overview of CopyData Function



2.16 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 2-5.

Figure 2-9. Overview of SCI Bootloader Operation



The SCI-A loader uses following pins:

- SCIRXDA on GPIO28
- SCITXDA on GPIO29

The 28x device 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 device.

After each data transfer, the 28x 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 28x.

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 bootloader 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 2-10. Overview of SCI_Boot Function

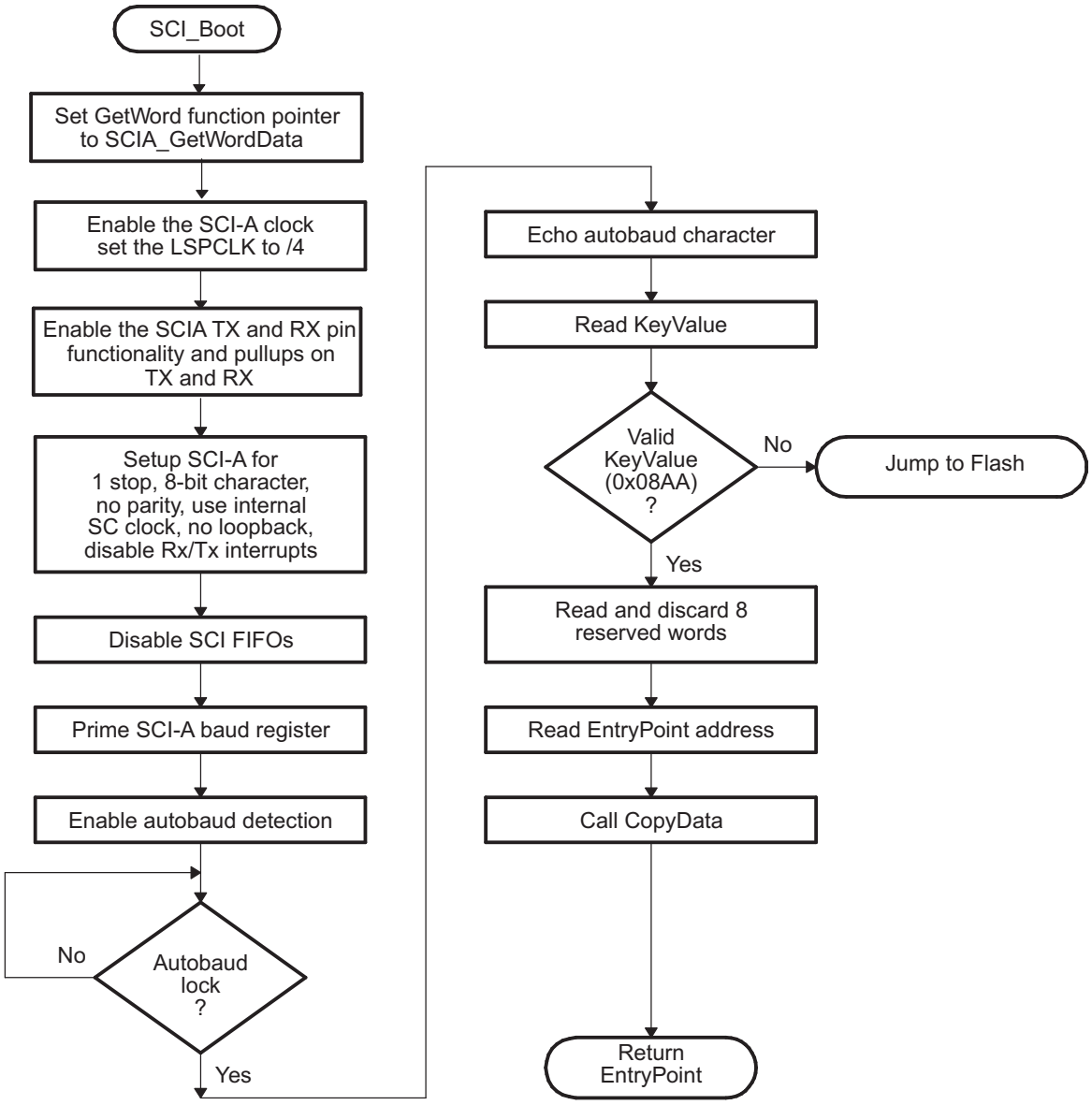
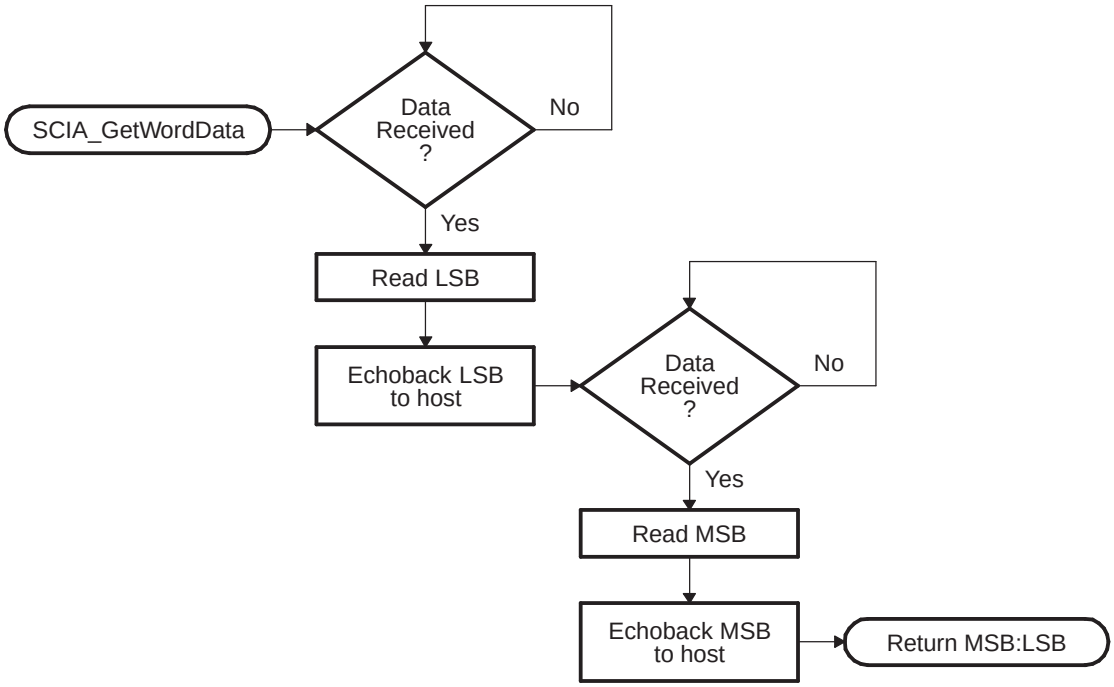


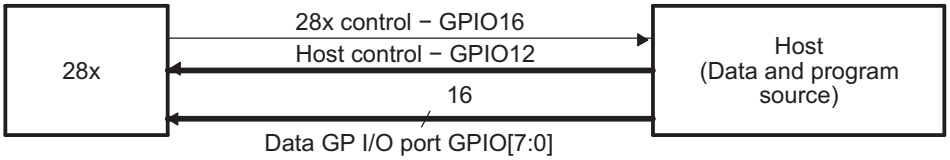
Figure 2-11. Overview of SCI_GetWordData Function



2.17 Parallel_Boot Function (GPIO)

The parallel general purpose I/O (GPIO) boot mode asynchronously transfers code from GPIO0-GPIO7 to internal memory. Each value is 8 bits long and follows the same data flow as outlined in Section 2.11.

Figure 2-12. Overview of Parallel GPIO bootloader Operation



The parallel GPIO loader uses following pins:

- Data on GPIO[7:0]
- 28x Control on GPIO16
- Host Control on GPIO12

The 28x communicates with the external host device by polling/driving the GPIO12 and GPIO16 lines. The handshake protocol shown in Figure 2-13 must be used to successfully transfer each word via GPIO[7:0]. This protocol is very robust and allows for a slower or faster host to communicate with the 28x.

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.

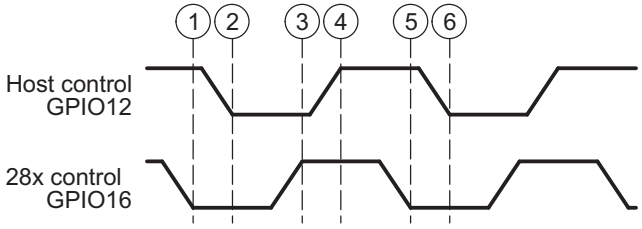
The 8-bit data stream is shown in Table 2-10.

Table 2-10. Parallel GPIO Boot 8-Bit Data Stream

Bytes		GPIO[7:0] (Byte 1 of 2)	GPIO[7:0] (Byte 2 of 2)	Description
1	2	AA	08	0x08AA (KeyValue for memory width = 16bits)
3	4	00	00	8 reserved words (words 2 - 9)
...	...	...	...	...
17	18	00	00	Last reserved word
19	20	BB	00	Entry point PC[22:16]
21	22	DD	CC	Entry point PC[15:0] (PC = 0x00BBCCDD)
23	24	NN	MM	Block size of the first block of data to load = 0xMMNN words
25	26	BB	AA	Destination address of first block Addr[31:16]
27	28	DD	CC	Destination address of first block Addr[15:0] (Addr = 0xAABBCCDD)
29	30	BB	AA	First word of the first block in the source being loaded = 0xAABB
... ...				... Data for this section. ...
.		BB	AA	Last word of the first block of the source being loaded = 0xAABB
.		NN	MM	Block size of the 2nd block to load = 0xMMNN words
.		BB	AA	Destination address of second block Addr[31:16]
.		DD	CC	Destination address of second block Addr[15:0]
.		BB	AA	First word of the second block in the source being loaded
.				…
n	n+1	BB	AA	Last word of the last block of the source being loaded (More sections if required)
n+2	n+3	00	00	Block size of 0000h - indicates end of the source program

The 28x device first signals the host that it is ready to begin data transfer by pulling the GPIO16 pin low. The host load then initiates the data transfer by pulling the GPIO12 pin low. The complete protocol is shown in the diagram below:

Figure 2-13. Parallel GPIO bootloader Handshake Protocol



1. The 28x device indicates it is ready to start receiving data by pulling the GPIO16 pin low.
2. The bootloader waits until the host puts data on GPIO[7:0]. The host signals to the 28x device that data is ready by pulling the GPIO12 pin low.
3. The 28x device reads the data and signals the host that the read is complete by pulling GPIO16 high.
4. The bootloader waits until the host acknowledges the 28x by pulling GPIO12 high.
5. The 28x device again indicates it is ready for more data by pulling the GPIO16 pin low.

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

Figure 2-14 shows an overview of the Parallel GPIO bootloader flow.

Figure 2-14. Parallel GPIO Mode Overview

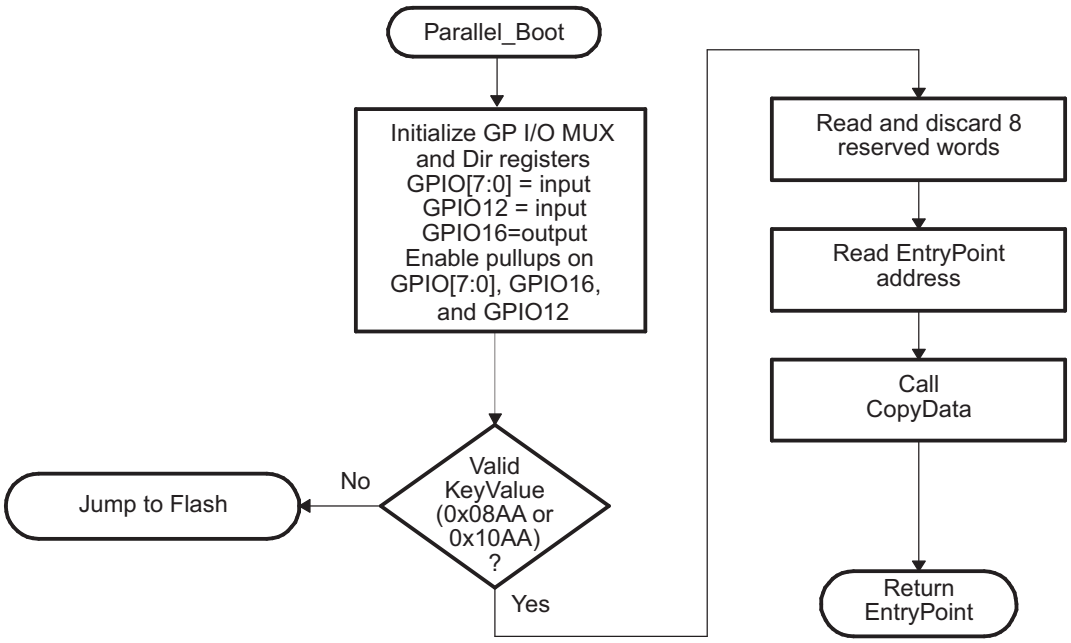


Figure 2-15 shows the transfer flow from the host side. The operating speed of the CPU and host are not critical in this mode as the host will wait for the 28x and the 28x will in turn wait for the host. In this manner the protocol will work with both a host running faster and a host running slower then the 28x.

Figure 2-15. Parallel GPIO Mode - Host Transfer Flow

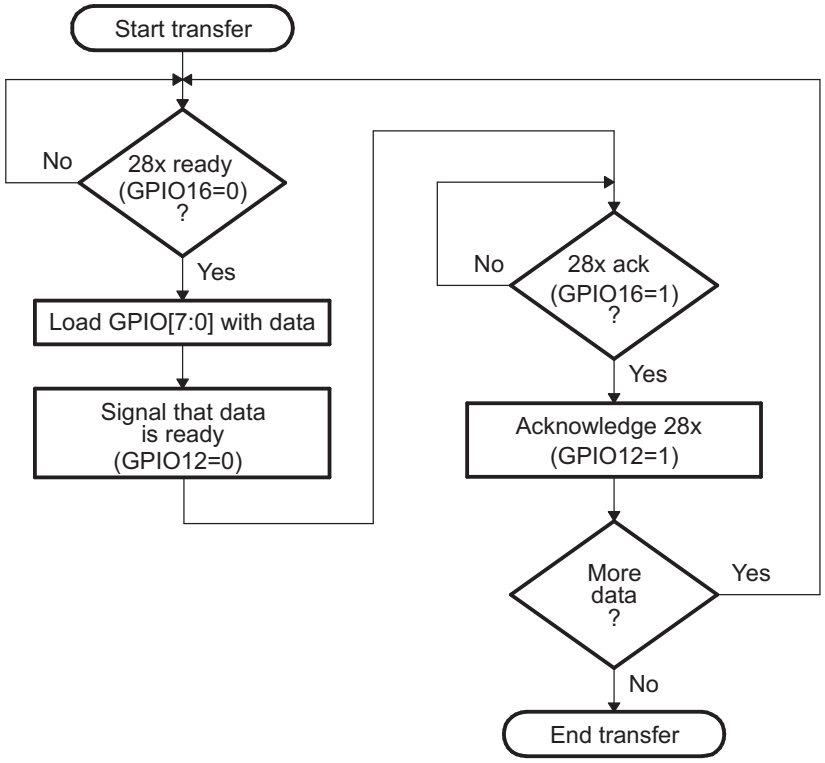
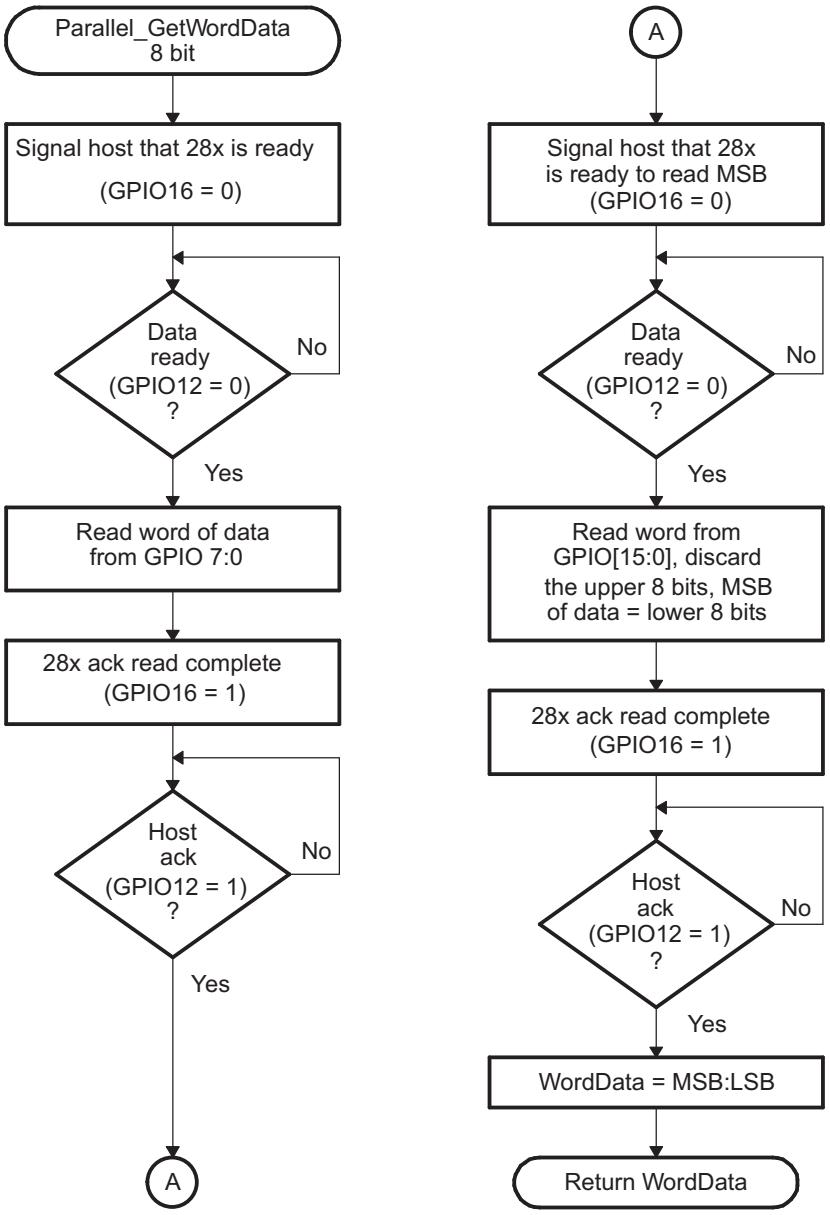


Figure 2-16 show the flow used to read a single word of data from the parallel port.

- **8-bit data stream**

  The 8-bit routine, shown in Figure 2-16, discards the upper 8 bits of the first read from the port and treats the lower 8 bits as the least significant byte (LSB) of the word to be fetched. The routine will then perform a second read to fetch 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 2-16. 8-Bit Parallel GetWord Function



2.18 SPI_Boot Function

The SPI loader expects an SPI-compatible 16-bit or 24-bit addressable serial EEPROM or serial flash device to be present on the SPI-A pins as indicated in Figure 2-17. The SPI bootloader supports an 8-bit data stream. It does not support a 16-bit data stream.

Figure 2-17. SPI Loader

The SPI-A loader uses following pins:

- SPISIMOA on GPIO16
- SPISOMIA on GPIO17
- SPICLKA on GPIO18
- SPISTEA on GPIO19

The SPI boot ROM loader initializes the SPI module to interface to a serial SPI EEPROM or flash. Devices of this type include, but are not limited to, the Xicor X25320 (4Kx8) and Xicor X25256 (32Kx8) SPI serial SPI EEPROMs and the Atmel AT25F1024A serial flash.

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 = 1, polarity = 0, using the slowest baud rate.

If the download is to be performed 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 2-11. 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 for future use
6	MSB: reserved for future use
...	...
...	Data for this section.
	...
17	LSB: reserved for future use
18	MSB: reserved for future use
19	LSB: Upper half (MSW) of Entry point PC[23:16]
20	MSB: Upper half (MSW) of Entry point PC[31:24] (Note: Always 0x00)
21	LSB: Lower half (LSW) of Entry point PC[7:0]
22	MSB: Lower half (LSW) of Entry point PC[15:8]
...	....
...	Data for this section.
	...

Table 2-11. SPI 8-Bit Data Stream (continued)

Byte	Contents
...	Blocks of data in the format size/destination address/data as shown in the generic data stream description
...	...
...	Data for this section.
	...
n	LSB: 00h
n+1	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:

Step 1. The SPI-A port is initialized

Step 2. The GPIO19 (SPISTE) pin is used as a chip-select for the serial SPI EEPROM or flash

Step 3. The SPI-A outputs a read command for the serial SPI EEPROM or flash

Step 4. The SPI-A sends the serial SPI EEPROM an address 0x0000; that is, the host requires that the EEPROM or flash must have the downloadable packet starting at address 0x0000 in the EEPROM or flash. The loader is compatible with both 16-bit addresses and 24-bit addresses.

Step 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

Step 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 second byte read is the SPIBRR value. The next 7 words are reserved for future enhancements. The SPI bootloader reads these 7 words and discards them.

Step 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.

Step 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 bootloader and resumes execution at the address specified.

Figure 2-18. Data Transfer From EEPROM Flow

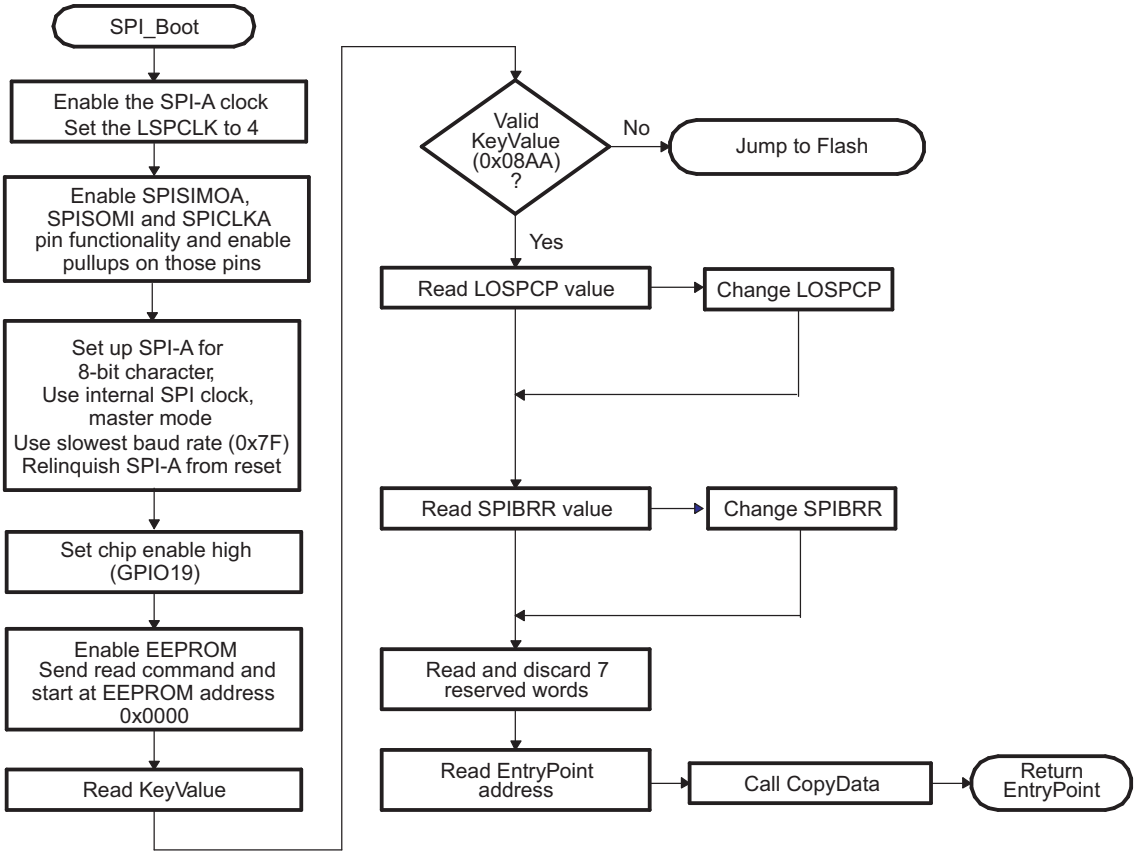
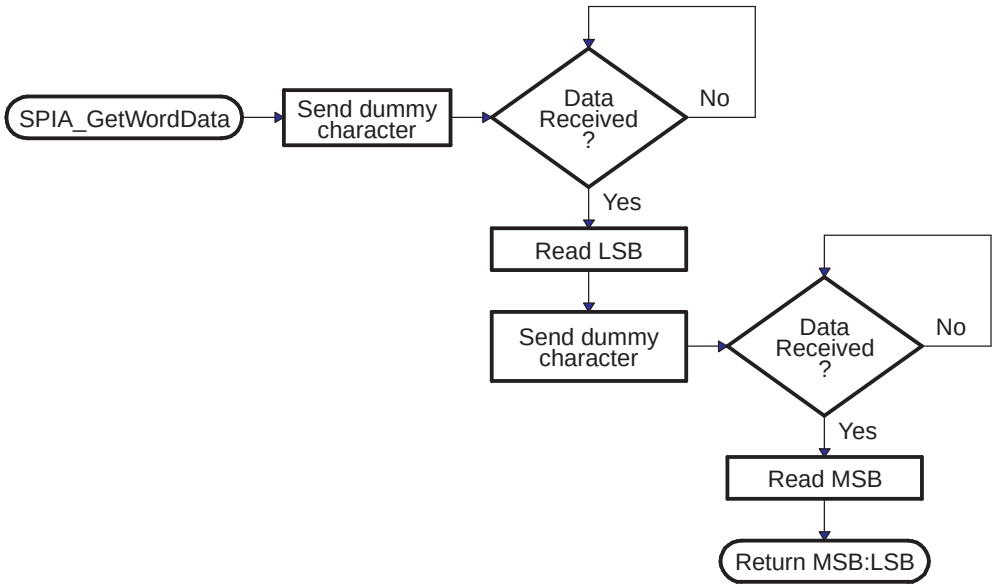


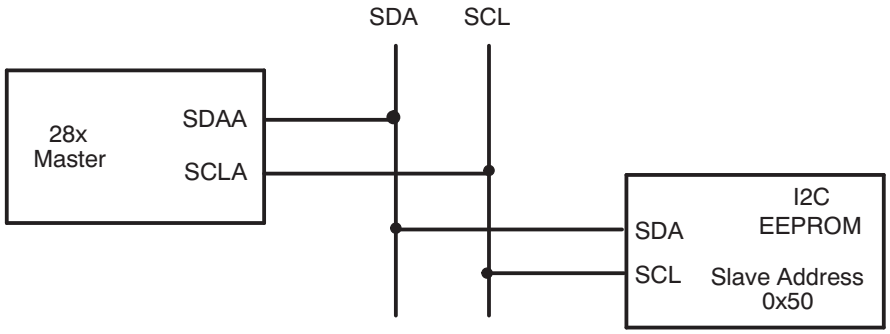
Figure 2-19. Overview of SPIA_GetWordData Function



2.19 I2C Boot Function

The I2C bootloader expects an 8-bit wide I2C-compatible EEPROM device to be present at address 0x50 on the I2C-A bus as indicated in Figure 2-20. The EEPROM must adhere to conventional I2C EEPROM protocol, as described in this section, with a 16-bit base address architecture.

Figure 2-20. EEPROM Device at Address 0x50



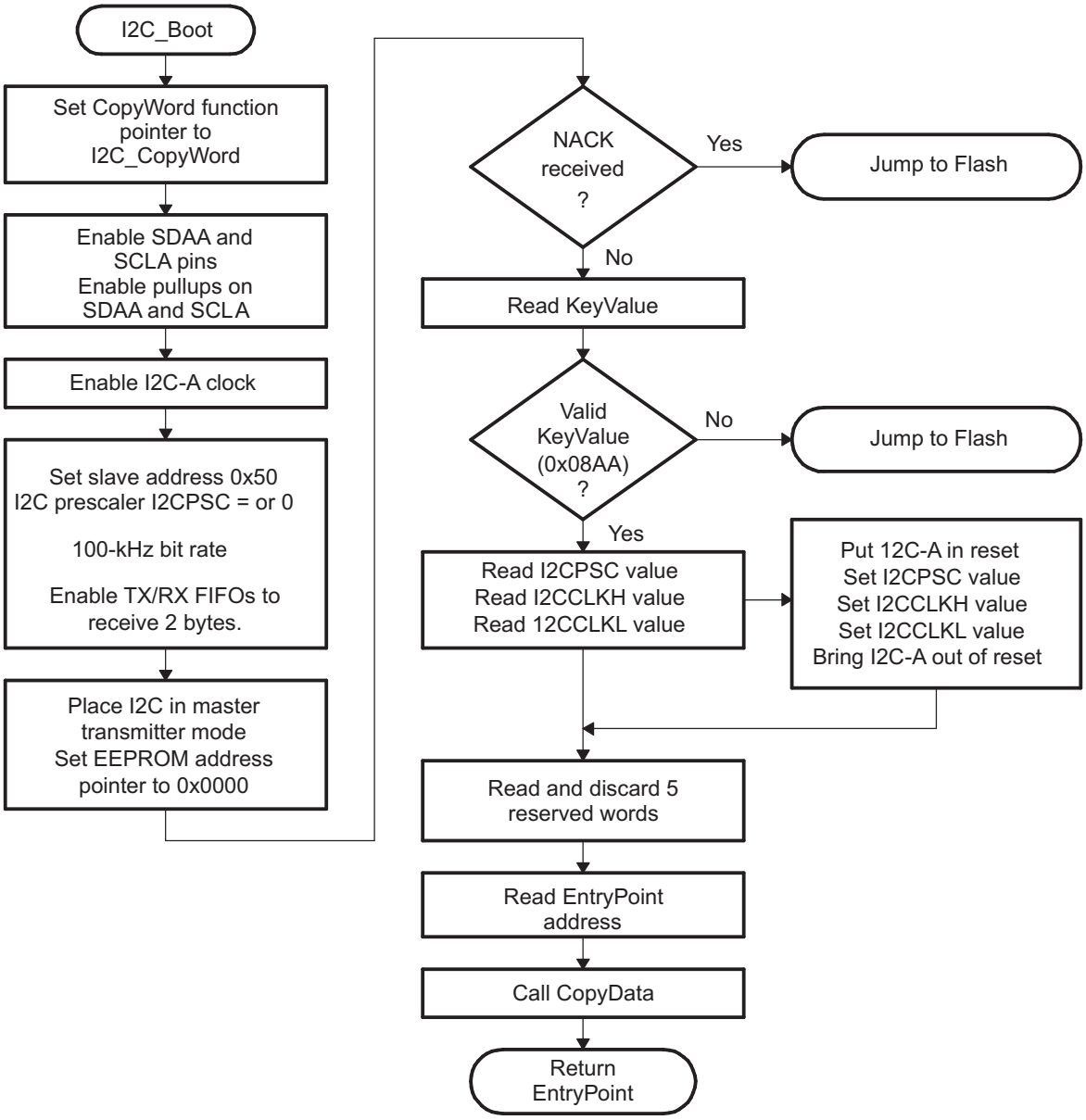
The I2C loader uses following pins:

- SDAA on GPIO32
- SCLA on GPIO33

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 I2C is initialized. The following requirements must be met when booting from the I2C module:

- The input frequency to the device must be in the appropriate range.
- The EEPROM must be at slave address 0x50.

Figure 2-21. Overview of I2C_Boot Function



The bit-period prescalers (I2CCLKH and I2CCLKL) are configured by the bootloader to run the I2C at a 50 percent duty cycle at 100-kHz bit rate (standard I2C mode) when the system clock is 10 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 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 is to make sure that an EEPROM is present at address 0x50 before continuing. If an EEPROM is not present, code will The nonacknowledgment bit is not checked during the address phase of the data read messages (I2C_Get Word). If a non acknowledgment is received during the data read messages, the I2C bus will hang. Table 2-12 shows the 8-bit data stream used by the I2C.

Table 2-12. I2C 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]	
...	...	
...	Data for this section.	
	...	
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]	
...	...	
...	Data for this section.	
	...	
	Blocks of data in the format size/destination address/data as shown in the generic data stream description.	
...	...	
...	Data for this section.	
	...	
n		LSB: 00h
n+1	MSB: 00h - indicates the end of the source	

The I2C EEPROM protocol required by the I2C bootloader is shown in Figure 2-22 and Figure 2-23. The first communication, which sets the EEPROM address pointer to 0x0000 and reads the KeyValue (0x08AA) from it, is shown in Figure 2-22. All subsequent reads are shown in Figure 2-23 and are read two bytes at a time.

Figure 2-22. Random Read

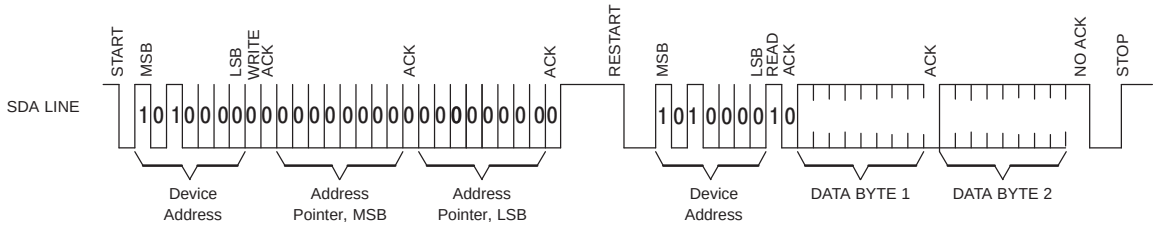
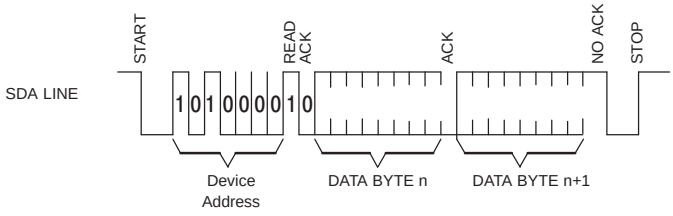


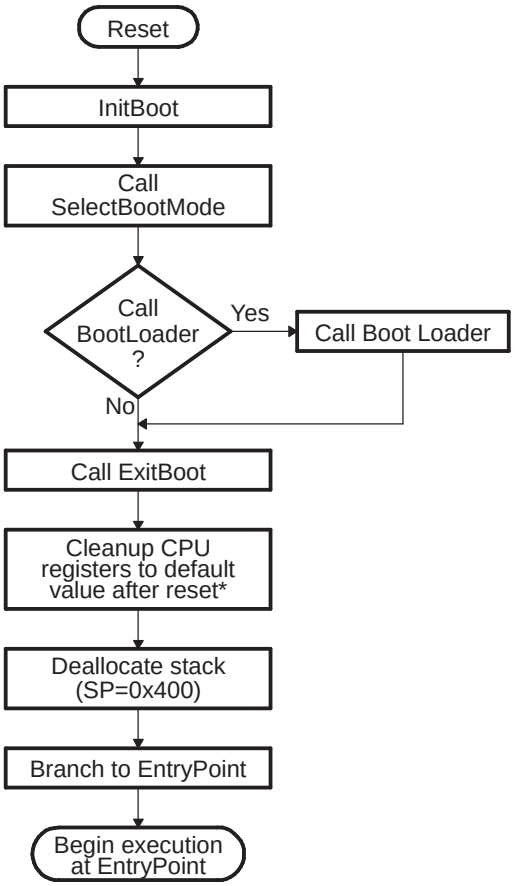
Figure 2-23. Sequential Read



2.20 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 2-24. 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 2-13. CPU Register Restored 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	ST1	0x0A0B	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

Building the Boot Table

This chapter explains how to generate the data stream and boot table required for the bootloader.

Topic **Page**

3.1 The C2000 Hex Utility

To use the features of the bootloader, you must generate a data stream and boot table as described in Section 2.11. The hex conversion utility tool, included with the 28x code generation tools, can generate the required data stream including the required boot table. This section describes the hex2000 utility. An example of a file conversion performed by hex2000 is described in Section 3.2.

The hex utility supports creation of the boot table required for the SCI, SPI, I2C, 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 contents of the boot table vary slightly depending on the boot mode and the options selected when running the hex conversion utility. 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:

1. **Assemble or compile the code.**

   This creates the object files that will then be used by the linker to create a single output file.
2. **Link the file.**

   The linker combines all of the object files into a single output file in common object file format (COFF). The specified linker command file is used by the linker to allocate the code sections to different memory blocks. 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. The following options may be useful:

   The linker -m option can be used to generate a map file. This map file will show all of the sections that were created, their location in memory and their length. It can be useful to check this file to make sure that the initialized sections are where you expect them to be.

   The linker -w option is also very useful. This option will tell you if the linker has assigned a section to a memory region on its own. For example, if you have a section in your code called ramfuncs.
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.

See the *TMS320C28x Assembly Language Tools User's Guide* (SPRU513) and the *TMS320C28x Optimizing C/C++ Compiler User's Guide* (SPRU514) for more information on the compiling and linking process.

Table 3-1 summarizes the hex conversion utility options available for the bootloader. See the *TMS320C28x Assembly Language Tools User's Guide* (SPRU513) for a detailed description of the hex2000 operations used to generate a boot table. Updates will be made to support the I2C boot. See the Codegen release notes for the latest information.

Table 3-1. Boot-Loader Options

Option	Description
-boot	Convert all sections into bootable form (use instead of a SECTIONS directive)
-sci8	Specify the source of the bootloader table as the SCI-A port, 8-bit mode
-spi8	Specify the source of the bootloader table as the SPI-A port, 8-bit mode
-gpio8	Specify the source of the bootloader table as the GPIO port, 8-bit mode
-gpio16	Specify the source of the bootloader table as the GPIO port, 16-bit mode
-bootorg value	Specify the source address of the bootloader 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_int00 unless defined otherwise by the -e linker option.
-i2c8	Specify the source of the bootloader 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.

3.2 Example: Preparing a COFF File For eCAN Bootloading

This section shows how to convert a COFF file into a format suitable for CAN based bootloading. This example assumes that the host sending the data stream is capable of reading an ASCII hex format file. An example COFF file named GPIO34TOG.out has been used for the conversion.

Build the project and link using the -m linker option to generate a map file. Examine the .map file produced by the linker. The information shown in Example 3-1 has been copied from the example map file (GPIO34TOG.map). This shows the section allocation map for the code. The map file includes the following information:

- **Output Section**

  This is the name of the output section specified with the SECTIONS directive in the linker command file.
- **Origin**

  The first origin listed for each output section is the starting address of that entire output section. The following origin values are the starting address of that portion of the output section.
- **Length**

  The first length listed for each output section is the length for that entire output section. The following length values are the lengths associated with that portion of the output section.
- **Attributes/input sections**

  This lists the input files that are part of the section or any value associated with an output section.

See the *TMS320C28x Assembly Language Tools User's Guide* (SPRU513) for detailed information on generating a linker command file and a memory map.

All sections shown in Example 3-1 that are initialized need to be loaded into the DSP in order for the code to execute properly. In this case, the codestart, ramfuncs, .cinit, myreset and .text sections need to be loaded. The other sections are uninitialized and will not be included in the loading process. The map file also indicates the size of each section and the starting address. For example, the .text section has 0x155 words and starts at 0x3FA000.

Example 3-1. GPIO34TOG Map File

```
output                                  attributes/
section   page    origin      length    input sections
--------  ----  ----------  ----------  ----------------
codestart
*          0     00000000    00000002
                 00000000    00000002    DSP280x_CodeStartBranch.obj (codestart)

.pinit     0     00000002    00000000

.switch    0     00000002    00000000    UNINITIALIZED

ramfuncs   0     00000002    00000016
                 00000002    00000016    DSP280x_SysCtrl.obj (ramfuncs)

.cinit     0     00000018    00000019
                 00000018    0000000e    rts2800_ml.lib : exit.obj (.cinit)
                 00000026    0000000a                   : _lock.obj (.cinit)
                 00000030    00000001    --HOLE-- [fill = 0]

myreset    0     00000032    00000002
                 00000032    00000002    DSP280x_CodeStartBranch.obj (myreset)

IQmath     0     003fa000    00000000    UNINITIALIZED

.text      0     003fa000    00000155
                 003fa000    00000046    rts2800_ml.lib : boot.obj (.text)
```

To load the code using the CAN bootloader, the host must send the data in the format that the bootloader understands. That is, the data must be sent as blocks of data with a size, starting address followed by the data. A block size of 0 indicates the end of the data. The HEX2000.exe utility can be used to convert the COFF file into a format that includes this boot information. The following command syntax has been used to convert the application into an ASCII hex format file that includes all of the required information for the bootloader:

Example 3-2. HEX2000.exe Command Syntax

```
C: HEX2000 GPIO34TOG.OUT -boot -gpio8 -a



Where:

- boot    Convert all sections into bootable form.
- gpio8   Use the GPIO in 8-bit mode data format. The eCAN
          uses the same data format as the GPIO in 8-bit mode.
- a       Select ASCII-Hex as the output format.
```

The command line shown in Example 3-2 will generate an ASCII-Hex output file called GPIO34TOG.a00, whose contents are explained in Example 3-3. This example assumes that the host will be able to read an ASCII hex format file. The format may differ for your application. . Each section of data loaded can be tied back to the map file described in Example 3-1. After the data stream is loaded, the boot ROM will jump to the Entrypoint address that was read as part of the data stream. In this case, execution will begin at 0x3FA0000.

Example 3-3. GPIO34TOG Data Stream

```
AA 08                                       ;Keyvalue
00 00 00 00 00 00 00 00                     ;8 reserved words
00 00 00 00 00 00 00 00
3F 00 00 A0                                 ;Entrypoint 0x003FA000
02 00                                       ;Load 2 words - codestart section
00 00 00 00                                 ;Load block starting at 0x000000
7F 00 9A A0                                 ;Data block 0x007F, 0xA09A
16 00                                       ;Load 0x0016 words - ramfuncs section
00 00 02 00                                 ;Load block starting at 0x000002
22 76 1F 76 2A 00 00 1A 01 00 06 CC F0      ;Data = 0x7522, 0x761F etc...
FF 05 50 06 96 06 CC FF F0 A9 1A 00 05
06 96 04 1A FF 00 05 1A FF 00 1A 76 07
F6 00 77 06 00
55 01                                       ;Load 0x0155 words - .text section
3F 00 00 A0                                 ;Load block starting at 0x003FA000
AD 28 00 04 69 FF 1F 56 16 56 1A 56 40      ;Data = 0x28AD, 0x4000 etc...
29 1F 76 00 00 02 29 1B 76 22 76 A9 28
18 00 A8 28 00 00 01 09 1D 61 C0 76 18
00 04 29 0F 6F 00 9B A9 24 01 DF 04 6C
04 29 A8 24 01 DF A6 1E A1 F7 86 24 A7
06 .. ..
.. .. ..
.. .. ..
FC 63 E6 6F
19 00                                       ;Load 0x0019 words - .cinit section
00 00 18 00                                 ;Load block starting at 0x000018
FF FF 00 B0 3F 00 00 00 FE FF 02 B0 3F      ;Data = 0xFFFF, 0xB000 etc...
00 00 00 00 00 FE FF 04 B0 3F 00 00 00
00 00 FE FF .. .. ..
.. .. ..
3F 00 00 00
02 00                                       ;Load 0x0002 words - myreset section
00 00 32 00                                 ;Load block starting at 0x000032
00 00 00 00                                 ;Data = 0x0000, 0x0000
00 00                                       ;Block size of 0 - end of data
```

Bootloader Code Overview

This chapter contains information on the Boot ROM version, checksum, and code.

Topic **Page**

4.1 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 4-1. Bootloader Revision and Checksum Information

Address	Contents
0x3F FFB9	
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

Table 4-2 shows the boot ROM revision per device. A revision history and code listing for the latest boot ROM code can be found in Chapter 4. In addition, a .zip file with each revision of the boot ROM code can be downloaded at

Table 4-2. Bootloader Revision Per Device

Device(s)	Silicon REVID (Address 0x883)	Boot ROM Revision
F2802x	0 (First silicon)	Version 1a

4.2 Bootloader Code Revision History

The associated boot ROM source code can be downloaded at http://www-s.ti.com/sc/techlit/sprufn6.zip.

- **Version: 1a, Released: August 2008:**

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 TI 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. Information of third parties may be subject to additional restrictions.

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.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

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

Products		**Applications**	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DLP® Products	www.dlp.com	Broadband	www.ti.com/broadband
DSP	dsp.ti.com	Digital Control	www.ti.com/digitalcontrol
Clocks and Timers	www.ti.com/clocks	Medical	www.ti.com/medical
Interface	interface.ti.com	Military	www.ti.com/military
Logic	logic.ti.com	Optical Networking	www.ti.com/opticalnetwork
Power Mgmt	power.ti.com	Security	www.ti.com/security
Microcontrollers	microcontroller.ti.com	Telephony	www.ti.com/telephony
RFID	www.ti-rfid.com	Video & Imaging	www.ti.com/video
RF/IF and ZigBee® Solutions	www.ti.com/lprf	Wireless	www.ti.com/wireless

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