

TMS320x2833x, 2823x Boot ROM

Reference Guide



Literature Number: SPRU963A
September 2007–Revised August 2008

Preface	7
1 Boot ROM Overview	11
1.1 Boot ROM Memory Map.....	12
1.2 On-Chip Boot ROM Math Tables.....	13
1.3 CPU Vector Table	15
2 Bootloader Features	17
2.1 Bootloader Functional Operation	18
2.2 Bootloader Device Configuration	19
2.3 PLL Multiplier and DIVSEL Selection.....	19
2.4 Watchdog Module	19
2.5 Taking an ITRAP Interrupt	20
2.6 Internal Pullup Resistors	20
2.7 PIE Configuration.....	20
2.8 Reserved Memory.....	20
2.9 Bootloader Modes	20
2.10 Bootloader Data Stream Structure	25
2.11 Basic Transfer Procedure	29
2.12 InitBoot Assembly Routine	30
2.13 SelectBootMode Function	31
2.14 ADC_cal Assembly Routine.....	33
2.15 CopyData Function.....	34
2.16 McBSP_Boot Function.....	35
2.17 SCI_Boot Function	36
2.18 Parallel_Boot Function (GPIO)	38
2.19 XINTF_Parallel_Boot Function	43
2.20 SPI_Boot Function	51
2.21 I2C Boot Function	54
2.22 eCAN Boot Function	57
2.23 ExitBoot Assembly Routine	59
3 Building the Boot Table	61
3.1 The C2000 Hex Utility	62
3.2 Example: Preparing a COFF File For eCAN Bootloading	63
4 Bootloader Code Overview	67
4.1 Boot ROM Version and Checksum Information	68
4.2 Bootloader Code Revision History	68
4.3 Bootloader Code Listing (V2.0).....	69
A Revision History	111
A.1 Changes Made in Revision A	111

List of Figures

1-1	Memory Map of On-Chip ROM	12
1-2	Vector Table Map	15
2-1	Bootloader Flow Diagram	18
2-2	Boot ROM Stack.....	20
2-3	Boot ROM Function Overview	22
2-4	Jump-to-Flash Flow Diagram	23
2-5	Flow Diagram of Jump to M0 SARAM.....	23
2-6	Flow Diagram of Jump-to-OTP Memory	23
2-7	Flow Diagram of Jump to XINTF x16.....	24
2-8	Flow Diagram of Jump to XINTF x32.....	24
2-9	Bootloader Basic Transfer Procedure	30
2-10	Overview of InitBoot Assembly Function.....	31
2-11	Overview of the SelectBootMode Function	32
2-12	Overview of CopyData Function	34
2-13	Overview of SCI Bootloader Operation.....	36
2-14	Overview of SCI_Boot Function	37
2-15	Overview of SCI_GetWordData Function	38
2-16	Overview of Parallel GPIO bootloader Operation	38
2-17	Parallel GPIO bootloader Handshake Protocol	40
2-18	Parallel GPIO Mode Overview.....	40
2-19	Parallel GPIO Mode - Host Transfer Flow	41
2-20	16-Bit Parallel GetWord Function	42
2-21	8-Bit Parallel GetWord Function.....	43
2-22	Overview of the Parallel XINTF Boot Loader Operation	44
2-23	XINTF_Parallel Bootloader Handshake Protocol	46
2-24	XINTF Parallel Mode Overview.....	47
2-25	XINTF Parallel Mode - Host Transfer Flow	48
2-26	16-Bit Parallel GetWord Function	49
2-27	8-Bit Parallel GetWord Function.....	50
2-28	SPI Loader	51
2-29	Data Transfer From EEPROM Flow	53
2-30	Overview of SPIA_GetWordData Function	53
2-31	EEPROM Device at Address 0x50.....	54
2-32	Overview of I2C_Boot Function	55
2-33	Random Read	56
2-34	Sequential Read.....	57
2-35	Overview of eCAN-A bootloader Operation.....	57
2-36	ExitBoot Procedure Flow	59

List of Tables

1-1	Vector Locations.....	16
2-1	Configuration for Device Modes.....	19
2-2	Boot Mode Selection.....	21
2-3	General Structure Of Source Program Data Stream In 16-Bit Mode	26
2-4	LSB/MSB Loading Sequence in 8-Bit Data Stream	28
2-5	Pins Used by the McBSP Loader	35
2-6	Bit-Rate Values for Different XCLKIN Values	35
2-7	McBSP 16-Bit Data Stream	35
2-8	Parallel GPIO Boot 16-Bit Data Stream	39
2-9	Parallel GPIO Boot 8-Bit Data Stream	39
2-10	XINTF Parallel Boot 16-Bit Data Stream	45
2-11	XINTF Parallel Boot 8-Bit Data Stream	46
2-12	SPI 8-Bit Data Stream	51
2-13	I2C 8-Bit Data Stream	56
2-14	Bit-Rate Values for Different XCLKIN Values	57
2-15	eCAN 8-Bit Data Stream	58
2-16	CPU Register Restored Values	60
3-1	Boot-Loader Options.....	63
4-1	Bootloader Revision and Checksum Information	68
4-2	Bootloader Revision Per Device.....	68
A-1	Additions, Deletions, and Changes.....	111

Read This First

This reference guide is applicable for the code and data stored in the on-chip boot ROM on the TMS320F2833x and TMS320F2823x processors. This includes all devices within these families.

The boot ROM is factory programmed with boot-loading software. Boot-mode signals (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](#)). Floating-point tables for SIN/COS are also included for use with the Texas Instruments™ *C28x FPU Fast RTS Library* ([SPRC664](#)).

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/spru963.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 and Errata—

SPRS439— [TMS320F28335](#), [TMS320F28334](#), [TMS320F28332](#), [TMS320F28235](#), [TMS320F28234](#), [TMS320F28232 Digital Signal Controllers \(DSCs\) Data Manual](#) contains the pinout, signal descriptions, as well as electrical and timing specifications for the F2833x/2823x devices.

SPRZ272— [TMS320F28335](#), [F28334](#), [F28332](#), [TMS320F28235](#), [F28234](#), [F28232 Digital Signal Controllers \(DSCs\) Silicon Errata](#) describes the advisories and usage notes for different versions of silicon.

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.

SPRUE02— [TMS320C28x Floating Point Unit and Instruction Set Reference Guide](#) describes the floating-point unit and includes the instructions for the FPU.

Peripheral Guides—

- SPRU566**— [TMS320x28xx, 28xxx Peripheral Reference Guide](#) describes the peripheral reference guides of the 28x digital signal processors (DSPs).
- SPRUFB0**— [TMS320x2833x, 2823x System Control and Interrupts Reference Guide](#) describes the various interrupts and system control features of the 2833x digital signal controllers (DSCs).
- SPRU812**— [TMS320x2833x, 2823x Analog-to-Digital Converter \(ADC\) Reference Guide](#) describes how to configure and use the on-chip ADC module, which is a 12-bit pipelined ADC.
- SPRU949**— [TMS320x2833x, 2823x External Interface \(XINTF\) User's Guide](#) describes the XINTF, which is a nonmultiplexed asynchronous bus, as it is used on the 2833x devices.
- SPRU963**— [TMS320x2833x, TMS320x2823x Boot ROM User's 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.
- SPRUFB7**— [TMS320x2833x, 2823x Multichannel Buffered Serial Port \(McBSP\) User's Guide](#) describes the McBSP available on the F2833x devices. The McBSPs allow direct interface between a DSP and other devices in a system.
- SPRUFB8**— [TMS320x2833x, 2823x Direct Memory Access \(DMA\) Reference Guide](#) describes the DMA on the 2833x devices.
- SPRUG04**— [TMS320x2833x, 2823x 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.
- SPRUG02**— [TMS320x2833x, 2823x High-Resolution Pulse Width Modulator \(HRPWM\)](#) describes the operation of the high-resolution extension to the pulse width modulator (HRPWM).
- SPRUG4**— [TMS320x2833x, 2823x Enhanced Capture \(eCAP\) Module Reference Guide](#) describes the enhanced capture module. It includes the module description and registers.
- SPRUG05**— [TMS320x2833x, 2823x Enhanced Quadrature Encoder Pulse \(eQEP\) Reference Guide](#) describes the eQEP module, which is used for interfacing with a linear or rotary incremental encoder to get position, direction, and speed information from a rotating machine in high performance motion and position control systems. It includes the module description and registers.
- SPRUEU1**— [TMS320x2833x, 2823x Enhanced Controller Area Network \(eCAN\) Reference Guide](#) describes the eCAN that uses established protocol to communicate serially with other controllers in electrically noisy environments.
- SPRUFZ5**— [TMS320F2833x, 2823x Serial Communication Interface \(SCI\) Reference Guide](#) describes the SCI, which is a two-wire asynchronous serial port, commonly known as a UART. The SCI modules support digital communications between the CPU and other asynchronous peripherals that use the standard non-return-to-zero (NRZ) format.
- SPRUEU3**— [TMS320x2833x, 2823x 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.
- SPRUG03**— [TMS320x2833x, 2823x 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.

SPRU625— [TMS320C28x DSP/BIOS Application Programming Interface \(API\) Reference Guide](#) describes development using DSP/BIOS.

Application Reports and Software—

Key Links Include:

1. C2000 Get Started - www.ti.com/c2000getstarted
2. C2000 Digital Motor Control Software Library - www.ti.com/c2000appsw
3. C2000 Digital Power Supply Software Library - www.ti.com/dpslib
4. DSP Power Management Reference Designs - www.ti.com/dsppower

SPRAAQ7— [TMS320x281x to TMS320x2833x or 2823x Migration Overview](#) describes how to migrate from the 281x device design to 2833x or 2823x designs.

SPRAAQ8— [TMS320x280x to TMS320x2833x or 2823x Migration Overview](#) describes how to migrate from a 280x device design to 2833x or 2823x designs.

SPRAAN9— [C28x FPU Primer](#)

provides an overview of the floating-point unit (FPU) in the TMS320F28335, TMS320F28334, and TMS320F28332 Digital Signal Controller (DSC) devices.

SPRAAM0— [Getting Started With TMS320C28x™ Digital Signal Controllers](#) is organized by development flow and functional areas to make your design effort as seamless as possible. Tips on getting started with C28x™ DSP software and hardware development are provided to aid in your initial design and debug efforts. Each section includes pointers to valuable information including technical documentation, software, and tools for use in each phase of design.

SPRA958— [Running an Application from Internal Flash Memory on the TMS320F28xx DSP](#) covers the requirements needed to properly configure application software for execution from on-chip flash memory. Requirements for both DSP/BIOS™ and non-DSP/BIOS projects are presented. Example code projects are included.

SPRAA85— [Programming TMS320x28xx and 28xxx Peripherals in C/C++](#) explores a hardware abstraction layer implementation to make C/C++ coding easier on 28x DSPs. This method is compared to traditional #define macros and topics of code efficiency and special case registers are also addressed.

SPRAA88— [Using PWM Output as a Digital-to-Analog Converter on a TMS320F280x](#) presents a method for utilizing the on-chip pulse width modulated (PWM) signal generators on the TMS320F280x family of digital signal controllers as a digital-to-analog converter (DAC).

SPRAA91— [TMS320F280x DSC USB Connectivity Using TUSB3410 USB-to-UART Bridge Chip](#) presents hardware connections as well as software preparation and operation of the development system using a simple communication echo program.

SPRAAH1— [Using the Enhanced Quadrature Encoder Pulse \(eQEP\) Module](#) provides a guide for the use of the eQEP module as a dedicated capture unit and is applicable to the TMS320x280x, 28xxx family of processors.

SPRAAI1— [Using Enhanced Pulse Width Modulator \(ePWM\) Module for 0-100% Duty Cycle Control](#) provides a guide for the use of the ePWM module to provide 0% to 100% duty cycle control and is applicable to the TMS320x280x family of processors.

- SPRAAD5**— [Power Line Communication for Lighting Apps using BPSK w/ a Single DSP Controller](#) presents a complete implementation of a power line modem following CEA-709 protocol using a single DSP.
- SPRAAD8**— [TMS320280x and TMS320F2801x ADC Calibration](#) describes a method for improving the absolute accuracy of the 12-bit ADC found on the TMS320280x and TMS3202801x devices. Inherent gain and offset errors affect the absolute accuracy of the ADC. The methods described in this report can improve the absolute accuracy of the ADC to levels better than 0.5%. This application report has an option to download an example program that executes from RAM on the F2808 EzDSP.
- SPRA820**— [Online Stack Overflow Detection on the TMS320C28x DSP](#) presents the methodology for online stack overflow detection on the TMS320C28x™ DSP. C-source code is provided that contains functions for implementing the overflow detection on both DSP/BIOS™ and non-DSP/BIOS applications.
- SPRA806**— [An Easy Way of Creating a C-callable Assembly Function for the TMS320C28x DSP](#) provides instructions and suggestions to configure the C compiler to assist with understanding of parameter-passing conventions and environments expected by the C compiler.

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	12
1.2 On-Chip Boot ROM Math Tables	13
1.3 CPU Vector Table.....	15

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 both fixed-point and floating-point math tables. These are for use with the *C28x™ IQMath Library - A Virtual Floating Point Engine* ([SPRC087](#)) and the *C28x FPU Fast RTS Library* ([SPRC664](#)). [Chapter 4](#) contains the code for each of 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
- Floating-point unit (FPU) math tables

[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

Data space	Program space	
		3F E000
	IQ math tables	
		3F EBDC
	FPU math tables	
		3F F27C
	Reserved	
		3F F34C
	Boot loader functions	
		3F F9EE
	Reserved	
		3F FFB9
	ROM version ROM checksum	
		3F FFC0
	Reset vector CPU vector table	
		3F FFFF

1.2 On-Chip Boot ROM Math Tables

Approximately 4K of the boot ROM is reserved for floating-point and IQ math tables. These tables are provided to help improve performance and save SARAM space.

The floating-point math tables included in the boot ROM are used by the Texas Instruments™ C28x FPU *Fast RTS Library* ([SPRC664](#)). The C28x Fast RTS Library is a collection of optimized floating-point math functions for C programmers of the C28x with floating-point unit. Designers of computationally intensive real-time applications can achieve execution speeds considerably faster than what are currently available without having to rewrite existing code. The functions listed in the features section are specifically optimized for the C28x + FPU controllers. The Fast RTS library accesses the floating-point tables through the FPUmathTables memory section. If you do not wish to load a copy of these tables into the device, use the boot ROM memory addresses and label the section as “NOLOAD” as shown in [Example 1-1](#). This facilitates referencing the look-up tables without actually loading the section to the target.

The following floating-point math tables are included in the Boot ROM:

- **Sine/Cosine Table, Single-precision Floating-point**
 - Table size: 1282 words
 - Contents: 32-bit floating-point samples for one and a quarter period sine wave
- **Normalized Arctan Table, Single-Precision Floating Point**
 - Table Size: 388 words
 - Contents: 32-bit second order coefficients for line of best fit.
- **Exp Coefficient Table, Single-Precision Floating Point**
 - Table size: 20 words
 - Contents: 32-bit coefficients for calculating exp (X) using a taylor series

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

```
MEMORY
{
    PAGE 0 :
    ...
    FPUTABLES : origin = 0x3FEBDC, length = 0x0006A0
    ...
}
SECTIONS
{
    ...
    FPUmathTables : > FPUTABLES, PAGE = 0, TYPE = NOLOAD
    ...
}
```

The fixed-point math tables 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. The IQmathTables section is completely included in the boot ROM. From the IQmathTablesRam section only the IQexp table is included and the remainder must be loaded into the device if used. 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 1-2](#). This facilitates referencing the look-up tables without actually loading the section to the target. Refer to the IQMath Library documentation for more information.

Example 1-2. Linker Command File to Access IQ Tables

```

MEMORY
{
  PAGE 0 :
  ...
  IQTABLES (R) : origin = 0x3FE000, length = 0x000b50
  IQTABLES2 (R) : origin = 0x3FEB50, length = 0x00008c
  ...
}
SECTIONS
{
  ...
  IQmathTables : load = IQTABLES, type = NOLOAD, PAGE = 0
  IQmathTables2 > IQTABLES2, type = NOLOAD, PAGE = 0
  {
    IQmath.lib<IQNexpTable.obj> (IQmathTablesRam)
  }
  IQmathTablesRam : load = DRAML1, PAGE = 1
  ...
}

```

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**

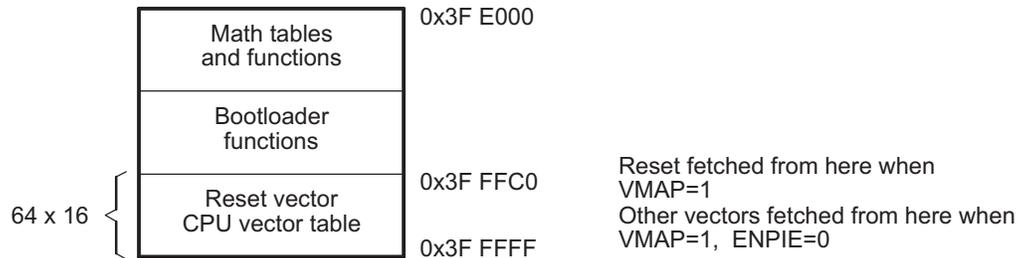
- Table size: 120 words
- Q format: Q1 - Q30
- Contents: 32-bit Min and Max values for each Q value

- **Exp Coefficient Table, IQMath Table**
 - Table size: 20 words
 - Q format: Q31
 - Contents: 32-bit coefficients for calculating exp (X) using a taylor series

1.3 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 PIENTRL 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 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	18
2.2 Bootloader Device Configuration	19
2.3 PLL Multiplier and DIVSEL Selection	19
2.4 Watchdog Module	19
2.5 Taking an ITRAP Interrupt	20
2.6 Internal Pullup Resistors	20
2.7 PIE Configuration	20
2.8 Reserved Memory	20
2.9 Bootloader Modes	20
2.10 Bootloader Data Stream Structure	25
2.11 Basic Transfer Procedure	29
2.12 InitBoot Assembly Routine	30
2.13 SelectBootMode Function	31
2.14 ADC_cal Assembly Routine	33
2.15 CopyData Function	34
2.16 McBSP_Boot Function	35
2.17 SCI_Boot Function	36
2.18 Parallel_Boot Function (GPIO)	38
2.19 XINTF_Parallel_Boot Function	43
2.20 SPI_Boot Function	51
2.21 I2C Boot Function	54
2.22 eCAN Boot Function	57
2.23 ExitBoot Assembly Routine	59

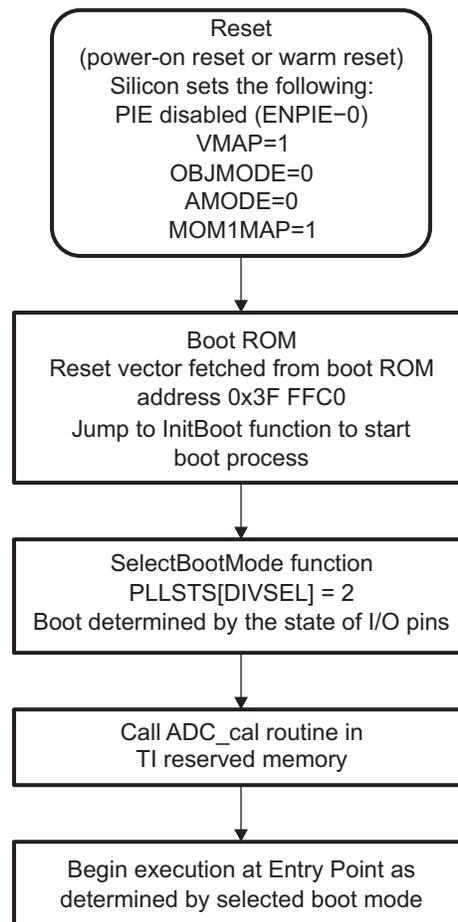
2.1 Bootloader Functional Operation

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 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 GPIO pins to determine which boot mode you want to execute. Options include: jump to flash, jump to OTP, jump to SARAM, jump to XINTF, or call one of the on-chip boot loading routines.

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 2.10](#). If, instead, you choose to boot directly to flash, OTP, XINTF 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:

- **XINTF parallel loader:**
PLLCR and PLLSTS[DIVSEL] are specified by the user as part of the incoming data stream.
- **All other boot modes:**
PLLCR is not modified. PLLSTS[DIVSEL] is set to 2 for SYSCLKOUT = CLKIN/2. 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/2 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.

2.4 Watchdog Module

When branching directly to flash, one-time-programmable (OTP) memory, M0 single-access RAM (SARAM) or external interface (XINTF) 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.

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 Resistors

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.

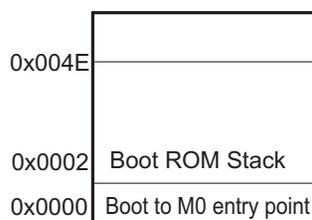
2.7 PIE Configuration

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

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



A Boot ROM loaders on other C28x devices had the stack in M1 memory. This is a change for this boot loader.

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.

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 four GPIO pins are used to determine the desired boot mode as shown in [Table 2-2](#).

Table 2-2. Boot Mode Selection

MODE	GPIO87/XA15	GPIO86/XA14	GPIO85/XA13	GPIO84/XA12	MODE ⁽¹⁾
F	1	1	1	1	Jump to Flash
E	1	1	1	0	SCI-A boot
D	1	1	0	1	SPI-A boot
C	1	1	0	0	I2C-A boot
B	1	0	1	1	eCAN-A boot
A	1	0	1	0	McBSP-A boot
9	1	0	0	1	Jump to XINTF x16
8	1	0	0	0	Jump to XINTF x32
7	0	1	1	1	Jump to OTP
6	0	1	1	0	Parallel GPIO I/O boot
5	0	1	0	1	Parallel XINTF boot
4	0	1	0	0	Jump to SARAM
3	0	0	1	1	Branch to check boot mode
2	0	0	1	0	Branch to Flash, skip ADC calibration
1	0	0	0	1	Branch to SARAM, skip ADC calibration
0	0	0	0	0	Branch to SCI, skip ADC calibration

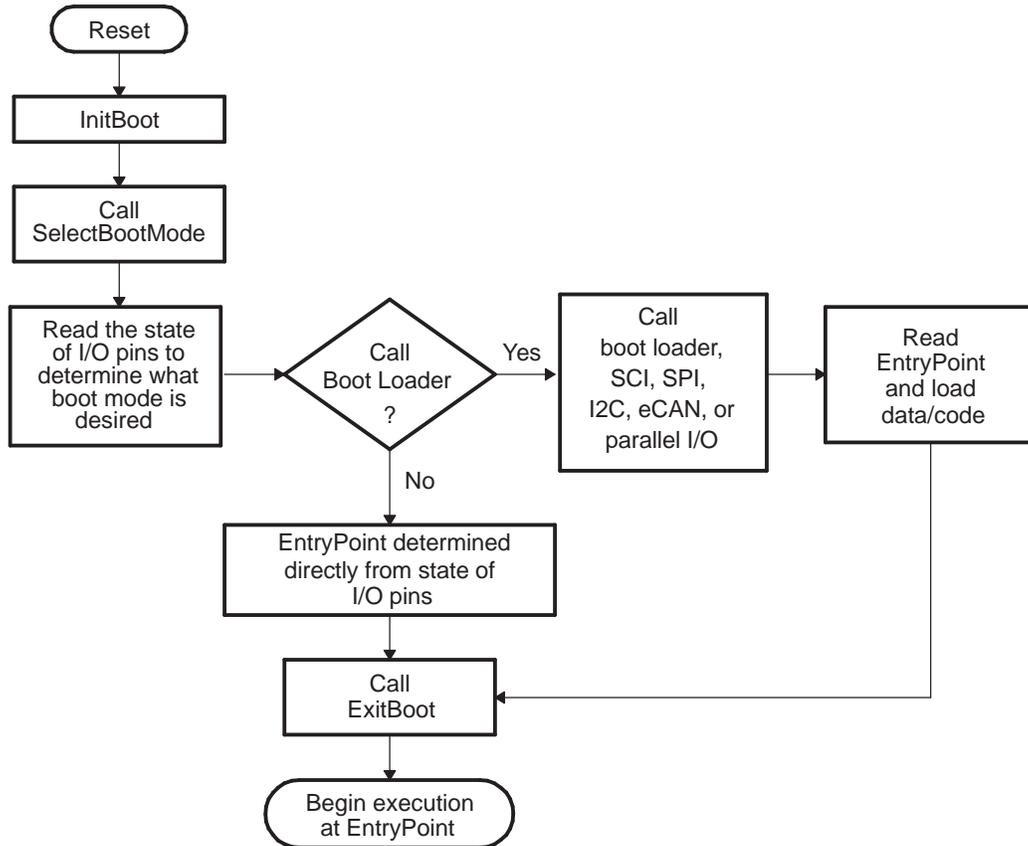
⁽¹⁾ All four GPIO pins have an internal pullup.

Note: Boot modes 0, 1, and 2 shown in [Table 2-2](#) bypass the ADC calibration function call. These boot modes are for TI debug only.

The ADC calibration function initializes the ADCREFSEL and ADCOFFTRIM registers. If these registers are not properly initialized the ADC will operate out of specification. For more information on the ADC calibration function, refer to [Section 2.14](#).

[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 for debug purposes:

- **Branch to check boot mode**

When initially debugging a device with the password locations in flash programmed (i.e., secured), the emulator takes some time to take control of the CPU. During this time, the CPU will start running and may execute an instruction that performs an access to a protected ECSL area. If this happens, the ECSL will trip and cause the emulator connection to be cut. Two solutions to this problem exist:

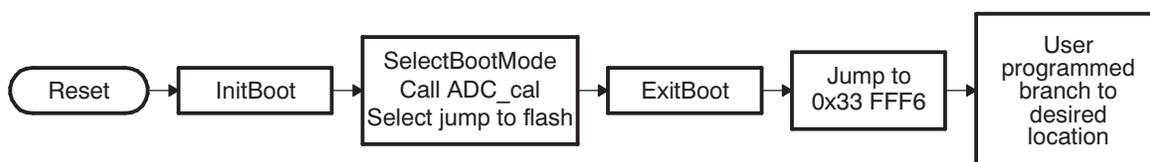
- The first is to use the Wait-In-Reset emulation mode, which will hold the device in reset until the emulator takes control. The emulator must support this mode for this option.
- The second option is to use the “Branch to check boot mode” boot option. This will sit in a loop and continuously poll the boot mode select pins. The user can select this boot mode and then exit this mode once the emulator is connected by re-mapping the PC to another address or by changing the boot mode selection pin to the desired boot mode.

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

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

In this mode, the boot ROM software configures the device for 28x operation and branches directly to location 0x33 FFF6. 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.

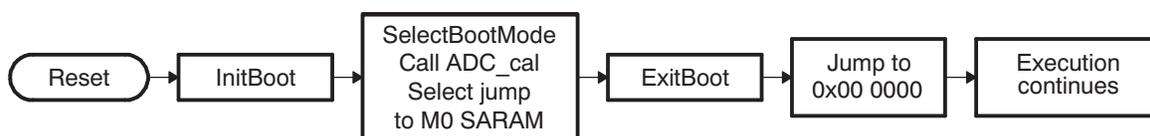
Figure 2-4. Jump-to-Flash Flow Diagram



- **Jump to M0 SARAM**

In this mode, the boot ROM software configures the device for 28x operation and branches directly to 0x00 0000. This is the first address in the M0 SARAM memory block.

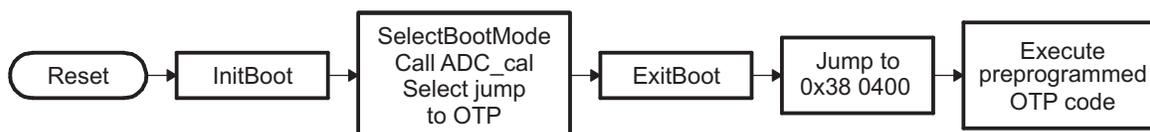
Figure 2-5. Flow Diagram of Jump to M0 SARAM



- **Jump to OTP memory**

In this mode, the boot ROM software configures the device for 28x operation and branches to 0x38 0400. This is the first address in the OTP memory block.

Figure 2-6. Flow Diagram of Jump-to-OTP Memory



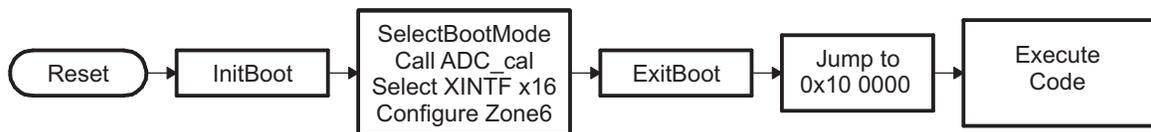
- **Jump to XINTF Zone 6 Configured for 16-bit Data**

The boot ROM configures XINTF zone 6 for 16 bit wide memory, maximum wait states, and sample XREADY in asynchronous mode. This is the default values list after reset:

- XTIMCLK = 1/2 SYSCLKOUT
- XCLKOUT = 1/2 XTIMCLK
- XRDLEAD = XWRLEAD = 3
- XRDACTIVE = XWRACTIVE = 7
- XRDTRAIL = XWRACTIVE = 3
- XSIZE = 16-bit wide
- X2TIMING = 1. Timing values are 2:1.
- USERREADY = 1, READYMODE = 1 (XREADY sampled synchronous mode)

The boot ROM will then jump to the first location within zone 6 at address 0x10 0000.

Figure 2-7. Flow Diagram of Jump to XINTF x16

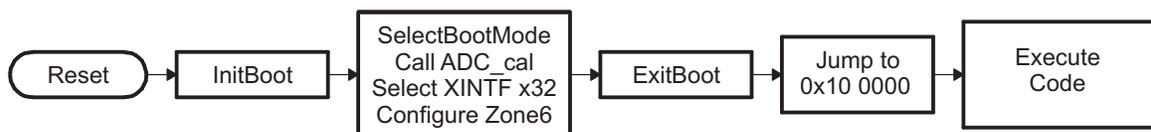


- **Jump to XINTF Zone 6 Configured for 32-bit Data**

In this mode, the boot ROM configures XINTF zone 6 for 32-bit wide memory, maximum wait states, and sample XREADY in asynchronous mode. This is the default mode after reset except the bus width is x32. The boot ROM will then jump to the first location within zone 6 at address 0x10 0000.

- XTIMCLK = ½ SYSCLKOUT
- XCLKOUT = 1/2 XTIMCLK
- XRDLEAD = XWRLEAD = 3
- XRDACTIVE = XWRACTIVE = 7
- XRDTRAIL = XWRACTIVE = 3
- XSIZE = 32-bit wide
- X2TIMING = 1. Timing values are 2:1.
- USERREADY = 1, READYMODE = 1 (XREADY sampled synchronous mode)

Figure 2-8. Flow Diagram of Jump to XINTF x32



The following boot modes call a boot load routine that loads a data stream from the peripheral into memory:

- **Standard serial boot mode (SCI-A)**

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

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

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)**

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. The EEPROM must adhere to conventional I2C EEPROM protocol with a 16-bit base address architecture.

- **eCAN 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.

- **McBSP Boot Mode (McBSP-A)**
Synchronously transfers code from McBSP-A to internal memory. McBSP-A is configured for slave mode operation. i.e. it receives the frame sync and clock from the host. Upon receiving a word, the McBSP echoes the data back to the host.
- **Boot from GPIO Port (Parallel Boot from GPIO0-GPIO15)**
In this mode, the boot ROM uses 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 is typically used to download code for flash programming when the device is connected to a platform explicitly for flash programming and not a target board.
- **Boot From XINTF (Parallel Boot From XD[15:0])**
This mode is similar to the GPIO parallel boot mode except the boot ROM uses XINTF data lines XD[15:0] to load code and data from an external source instead of GPIO pins. This mode supports both 8-bit and 16-bit data streams. The user can specify the PLL configuration as well as XINTF timing through the input data stream.

2.10 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. In this case, the entry point for the flash memory (0x33 7FF6) will be used.

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 2-3. 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 2-1. 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 2-4. 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 2-2. 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:

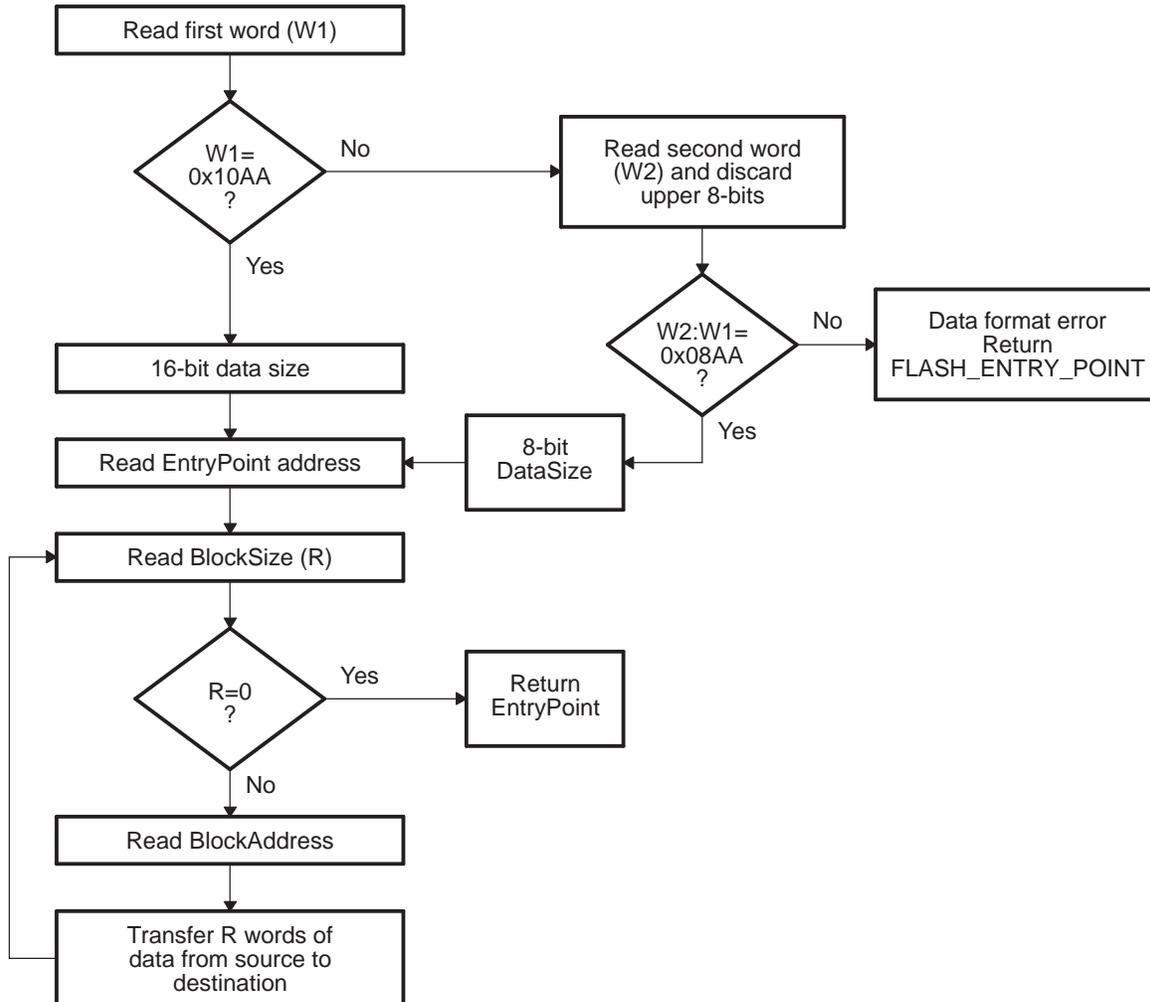
Location	Value
0x3F9010	0x0001
0x3F9011	0x0002
0x3F9012	0x0003
0x3F9013	0x0004
0x3F9014	0x0005
0x3F8000	0x7700
0x3F8001	0x7625

PC Begins execution at 0x3F8000

2.11 Basic Transfer Procedure

Figure 2-9 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 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 2-9. Bootloader Basic Transfer Procedure


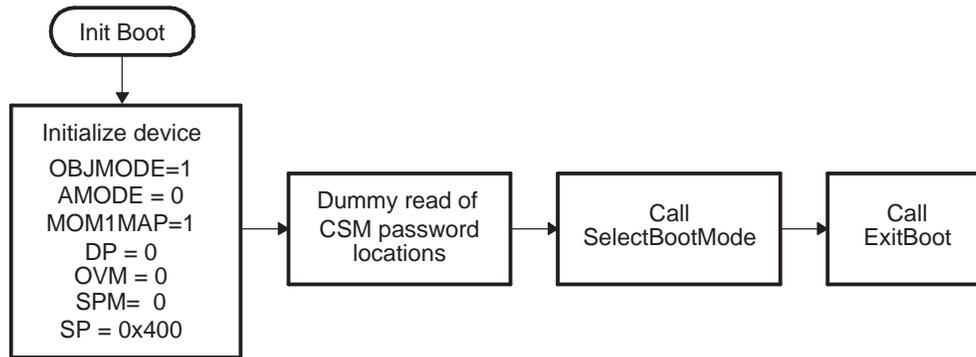
- A 8-bit and 16-bit transfers are not valid for all boot modes. See the info specific to a particular bootloader for any limitations.
- B In 8-bit mode, the LSB of the 16-bit word is read first followed by the MSB.

2.12 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 certain GPIO pins. This process is described in [Section 2.13](#). 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 2-10. Overview of InitBoot Assembly Function



2.13 SelectBootMode Function

To determine the desired boot mode, the SelectBootMode function examines the state of 3 GPIO pins as shown in [Table 2-2](#).

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.

The SelectBootMode function checks the missing clock detect bit (MCLKSTS) in the PLLSTS register to determine if the PLL is operating in limp mode. If the PLL is operating in limp mode, the boot mode select function takes an appropriate action depending on the boot mode selected:

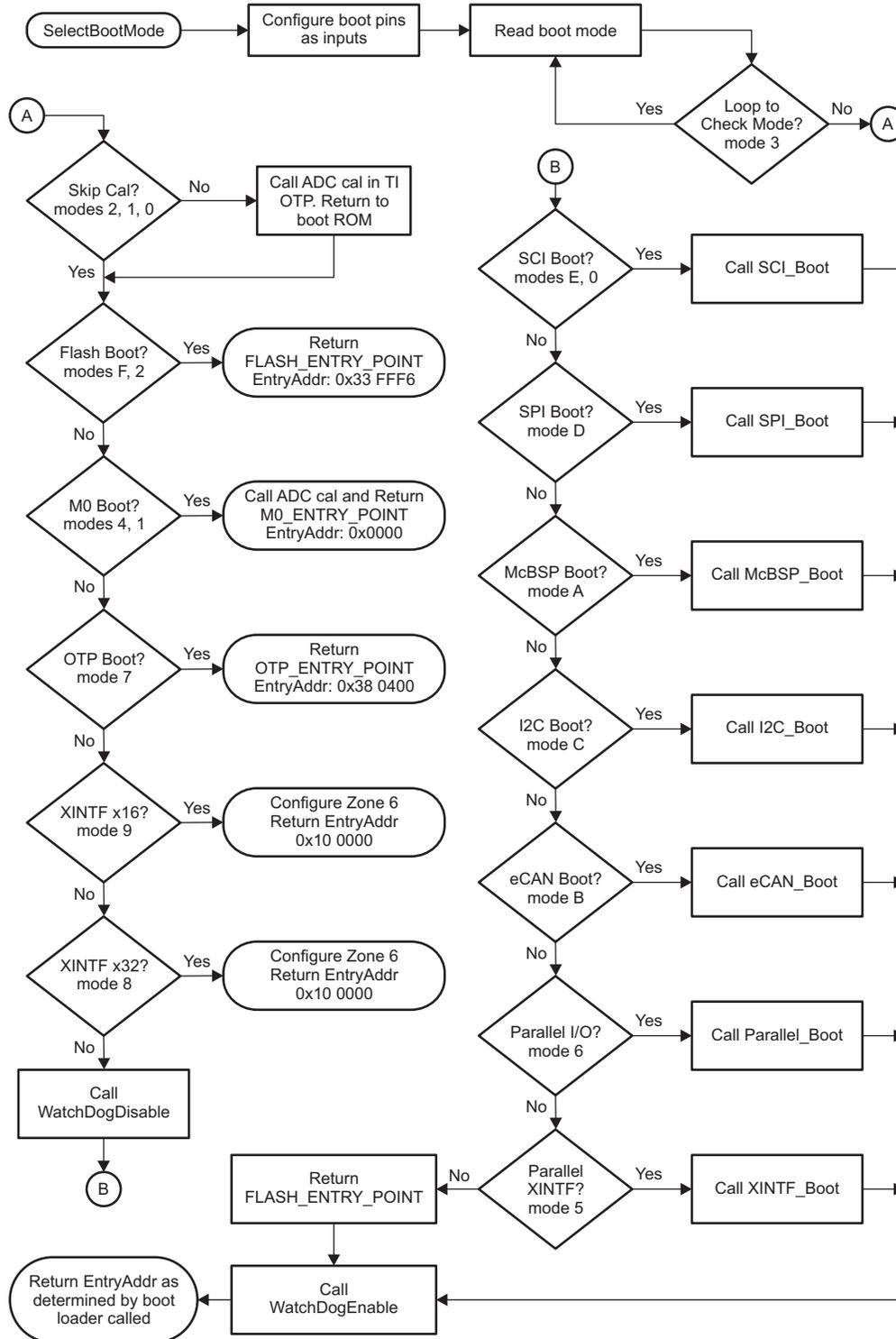
- **Boot to flash, OTP, SARAM, I2C-A, SPI-A, XINTF, and the parallel I/O**
These modes behave as normal. The user's software must check for missing clock status and take the appropriate action if the MCLKSTS bit is set.
- **SCI-A boot**
The SCI bootloader will be called. Depending on the requested baud rate, however, the device may not be able to autobaud lock. In this case the boot ROM software will loop in the autobaud lock function indefinitely. Should the SCI-A boot complete, the user's software must check for a missing clock status and take the appropriate action.
- **eCAN-A boot**
The eCAN bootloader will not be called. Instead the boot ROM will loop indefinitely.
- **McBSP boot**
The McBSP loader will not be called. Instead, the boot ROM will loop indefinitely.

Note: The SelectBootMode routine disables the watchdog before calling the SCI, I2C, eCAN, SPI, McBSP, 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 2-11. Overview of the SelectBootMode Function



2.14 ADC_cal Assembly Routine

The ADC_cal() routine is programmed into TI reserved OTP memory by the factory. The boot ROM automatically calls the ADC_cal() routine to initialize the ADCREFSEL and ADCOFFTRIM registers 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 ADCREFSEL and ADCOFFTRIM must be initialized by the application. For working examples, see the ADC initialization in the *C2833x/C2823x C/C++ Header Files and Peripheral Examples*.

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

- Step 1. Add the ADC_cal assembly function to your project. The source is included with the Header Files and Peripheral Examples. [Example 2-3](#) shows the contents of the ADC_cal function. The values 0xAAAA and 0xB BBB are place holders. The actual values programmed by TI are device specific.
- Step 2. Add the .adc_cal section to your linker command file as shown in [Example 2-4](#).
- Step 3. Call the ADC_cal function before using the ADC. The ADC clocks must be enabled before making this call. See [Example 2-5](#).

Example 2-3. Step 1: Add ADC_cal.asm to the Project

```

;-----
; This is the ADC cal routine. This routine is
; programmed into reserved memory by the factory.
; 0xAAAA and 0xB BBB are place holders. The actual
; values programmed by TI are device specific.
;
; The ADC clock must be enabled before calling
; this function.
;-----
.def _ADC_cal
.asg "0x711C", ADCREFSEL_LOC
.sect ".adc_cal"
_ADC_cal
MOVW DP, #ADCREFSEL_LOC >> 6
MOV @28, #0xAAAA
MOV @29, #0xB BBB
LRETR

```

Example 2-4. Step 2: Modify the Application Linker Command file to Access ADC_cal()

```

MEMORY
{
PAGE 0 :
...
ADC_CAL : origin = 0x380080, length = 0x000009
...
}
SECTIONS
{
...
.adc_cal : load = ADC_CAL, PAGE = 0, TYPE = NOLOAD
...
}

```

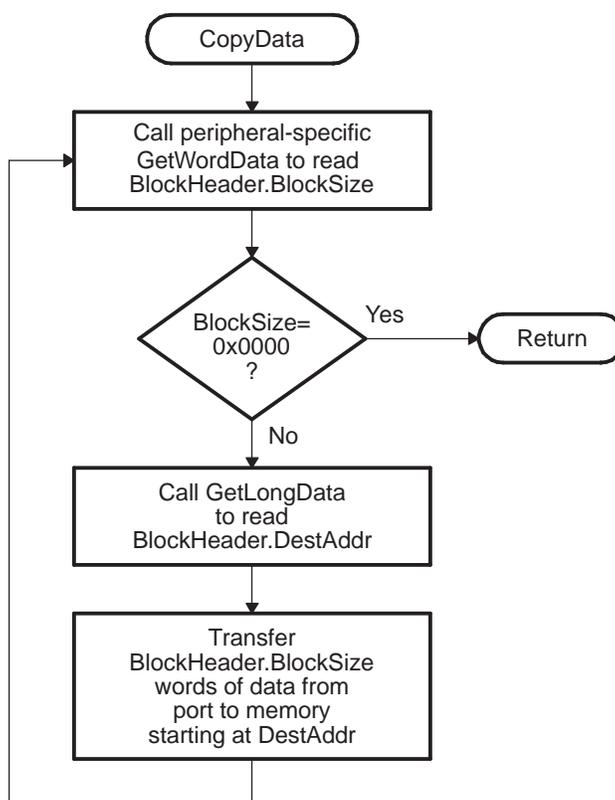
Example 2-5. Step 3: Call the ADC_cal() Function

```
extern void ADC_cal(void);
...
EALLOW;
SysCtrlRegs.PCLKCR0.bit.ADCENCLK = 1;
ADC_cal();
SysCtrlRegs.PCLKCR0.bit.ADCENCLK = 0;
EDIS;
```

2.15 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 2-12](#).

Figure 2-12. Overview of CopyData Function



2.16 McBSP_Boot Function

The McBSP bootloader synchronously transfers code from McBSP-A to internal memory. McBSP-A is configured for slave mode operation. i.e., it receives the frame sync and clock from the host. Upon receiving a word, the McBSP echoes the data back to the host. The host could use this feature to ensure that the previous word was received and copied by the McBSP before transmitting the next word. The host can download a kernel to reconfigure the McBSP if higher data throughput is desired. This can be done by choosing a faster PLL multiplier and also by choosing the /1 divider for the PLL output.

Note: Version 1 of the McBSP loader does not echo back data to the host. This feature is new as of Version 2 included on Rev A devices.

The McBSP-A loader uses pins shown in [Table 2-5](#).

Table 2-5. Pins Used by the McBSP Loader

C28x Slave	2833x/2823x Pin Number	Host Signal
MDXA	GPIO20	MDR
MDRA	GPIO21	MDX
MCLKXA	GPIO22	CLKX
MFSXA	GPIO23	FSR
MCLKRA	GPIO7	CLKX
MFSXA	GPIO5	FSXA

The bit rates achieved for different XCLKIN values as shown in [Table 2-6](#). The SYSCLKOUT values shown are for the default PLLCR of 0 and PLLSTS[DIVSEL] set to 2.

Table 2-6. Bit-Rate Values for Different XCLKIN Values

XCLKIN	SYSCLKOUT	LSPCLK	CLKG
30 MHz	15 MHz	3.75 MHz	1.875 MHz
15 MHz	7.5 MHz	1.875 MHz	937.5 KHz

The host should transmit MSB first and LSB next. For example, to transmit the word 0x10AA to the device, transmit 10 first, followed by AA. The program flow of the McBSP bootloader is identical to the SCI bootloader, with the exception that 16-bit data is used. The data sequence for the McBSP bootloader follows the 16-bit data stream and is shown in [Table 2-7](#)

Table 2-7. McBSP 16-Bit Data Stream

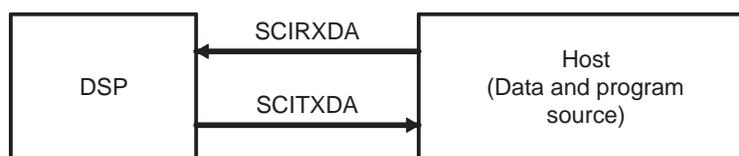
Word	Contents	Description
1	10AA	10AA (KeyValue for memory width = 16bits)
2	0000	8 reserved words (words 2-9)
...
9	0000	Last reserved word
10	AABB	Entry point PC[22:16]
11	CCDD	Entry point PC[15:0] (PC = 0xAABBCCDD)
12	MMNN	Block size (number of words) of the first block of data to load = 0xMMNN words
13	AABB	Destination address of first block Addr[31:16]
14	CCDD	Destination address of first block Addr[15:0] (Addr = 0xAABBCCDD)
15	XXXX	First word of the first block in the source being loaded
...
... Data for this section.
.	XXXX	Last word of the first block of the source being loaded

Table 2-7. McBSP 16-Bit Data Stream (continued)

Word	Contents	Description
.	MMNN	Block size of the 2nd block to load = 0xMMNN words
.	AABB	Destination address of second block Addr[31:16]
.	CCDD	Destination address of second block Addr[15:0]
.	XXXX	First word of the second block in the source being loaded
.
n	XXXX	Last word of the last block of the source being loaded
n+1	0000	Block size of 0000h - indicates end of the source program

2.17 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 2-2](#).

Figure 2-13. 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 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 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 2-14. Overview of SCI_Boot Function

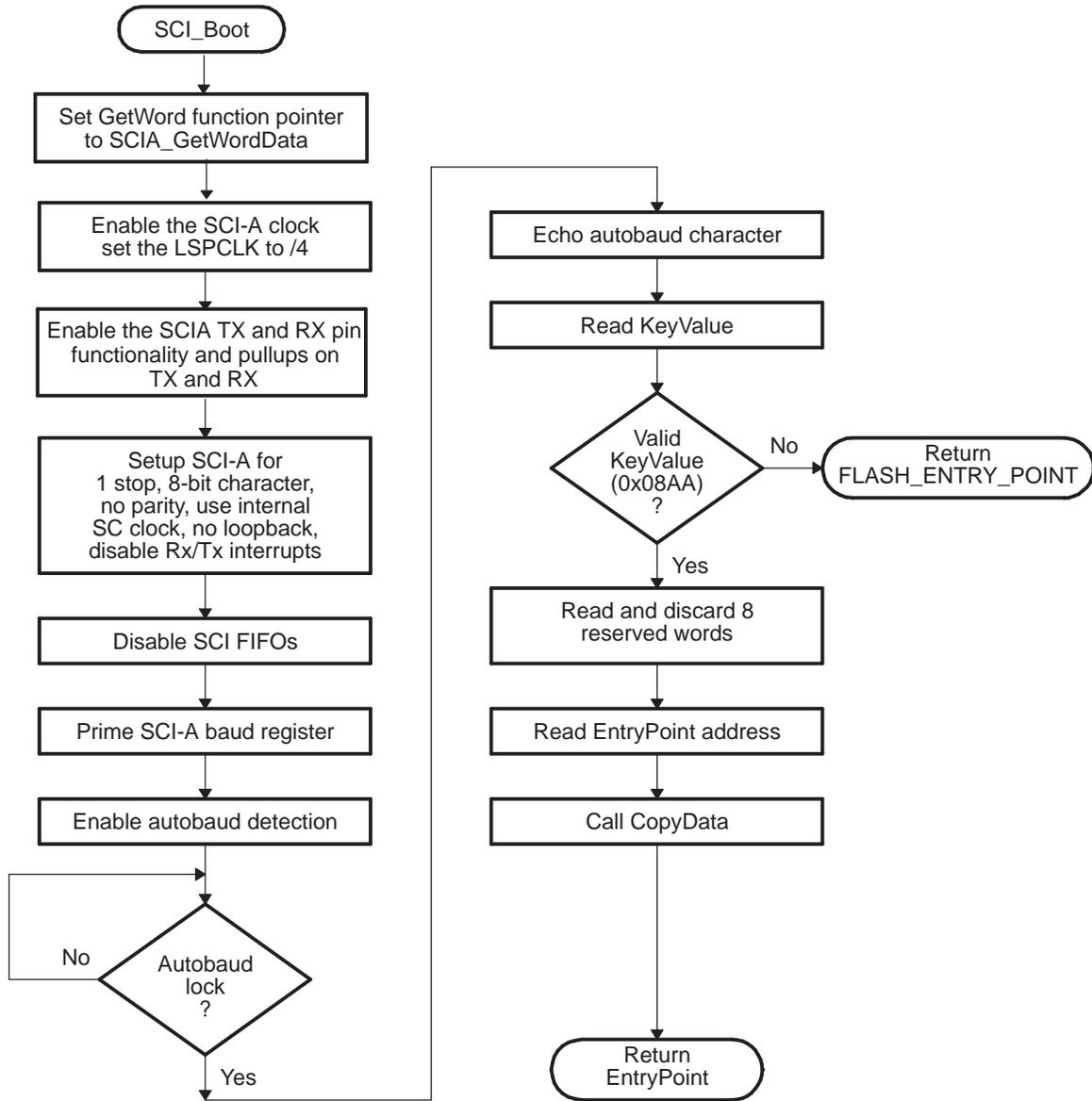
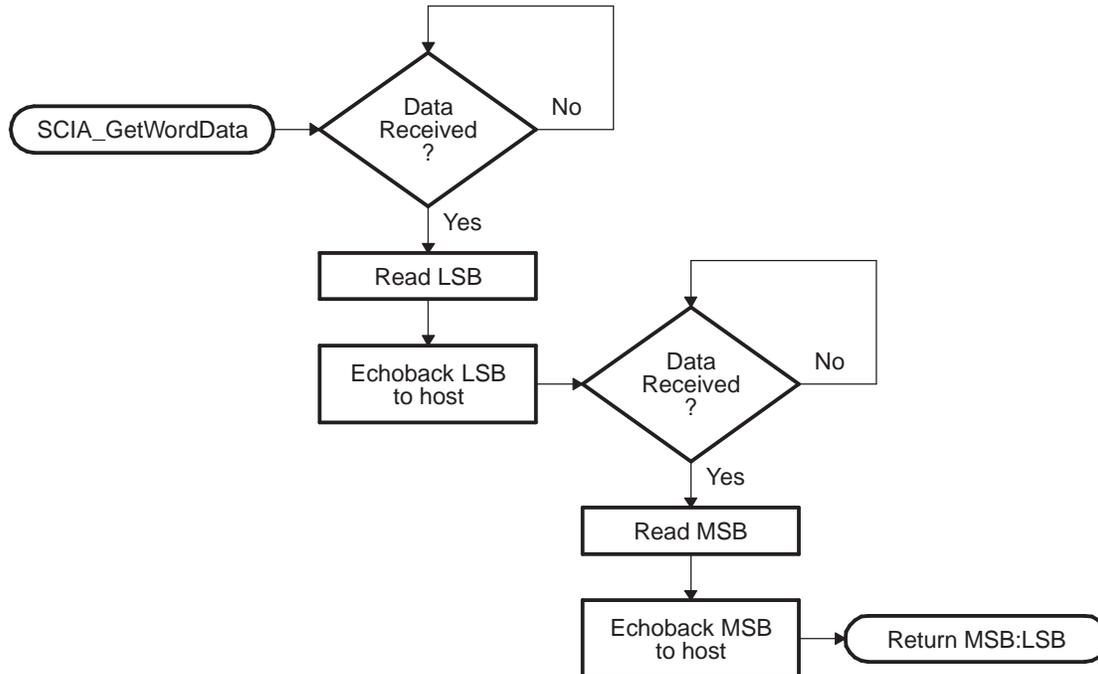
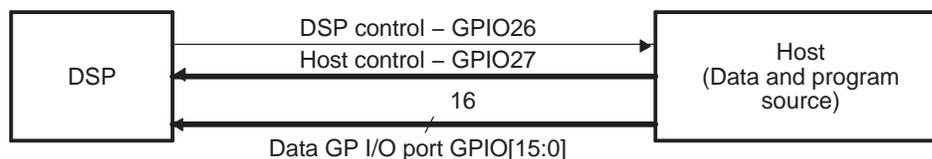


Figure 2-15. Overview of SCI_GetWordData Function


2.18 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 [Section 2.10](#).

Figure 2-16. Overview of Parallel GPIO bootloader Operation


The parallel GPIO loader uses following pins:

- Data on GPIO[15:0] or GPIO[7:0]
- 28x Control on GPIO26
- Host Control on GPIO27

The 28x communicates with the external host device by polling/driving the GPIO27 and GPIO26 lines. The handshake protocol shown in [Figure 2-17](#) 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 16-bit data stream is shown in [Table 2-8](#) and the 8-bit data stream is shown in [Table 2-9](#).

Table 2-8. Parallel GPIO Boot 16-Bit Data Stream

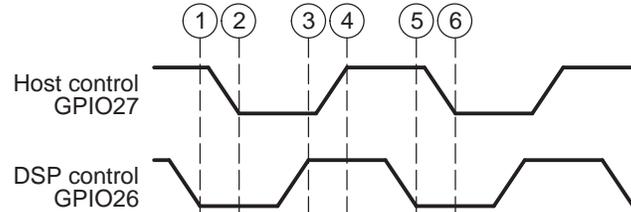
Word	GPIO[15:0]	Description
1	10AA	10AA (KeyValue for memory width = 16bits)
2	0000	8 reserved words (words 2 - 9)
...
9	0000	Last reserved word
10	AABB	Entry point PC[22:16]
11	CCDD	Entry point PC[15:0] (PC = 0xAABBCCDD)
12	MMNN	Block size (number of words) of the first block of data to load = 0xMMNN words
13	AABB	Destination address of first block Addr[31:16]
14	CCDD	Destination address of first block Addr[15:0] (Addr = 0xAABBCCDD)
15	XXXX	First word of the first block in the source being loaded
...
...	...	Data for this section.
...
.	XXXX	Last word of the first block of the source being loaded
.	MMNN	Block size of the 2nd block to load = 0xMMNN words
.	AABB	Destination address of second block Addr[31:16]
.	CCDD	Destination address of second block Addr[15:0]
.	XXXX	First word of the second block in the source being loaded
.
n	XXXX	Last word of the last block of the source being loaded (More sections if required)
n+1	0000	Block size of 0000h - indicates end of the source program

Table 2-9. 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 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 2-17. Parallel GPIO bootloader Handshake Protocol



1. The 28x device indicates it is ready to start receiving data by pulling the GPIO26 pin low.
2. The bootloader waits until the host puts data on GPIO[15:0]. The host signals to the 28x device that data is ready by pulling the GPIO27 pin low.
3. The 28x device reads the data and signals the host that the read is complete by pulling GPIO26 high.
4. The bootloader waits until the host acknowledges the DSP by pulling GPIO27 high.
5. The 28x device 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 2-18 shows an overview of the Parallel GPIO bootloader flow.

Figure 2-18. Parallel GPIO Mode Overview

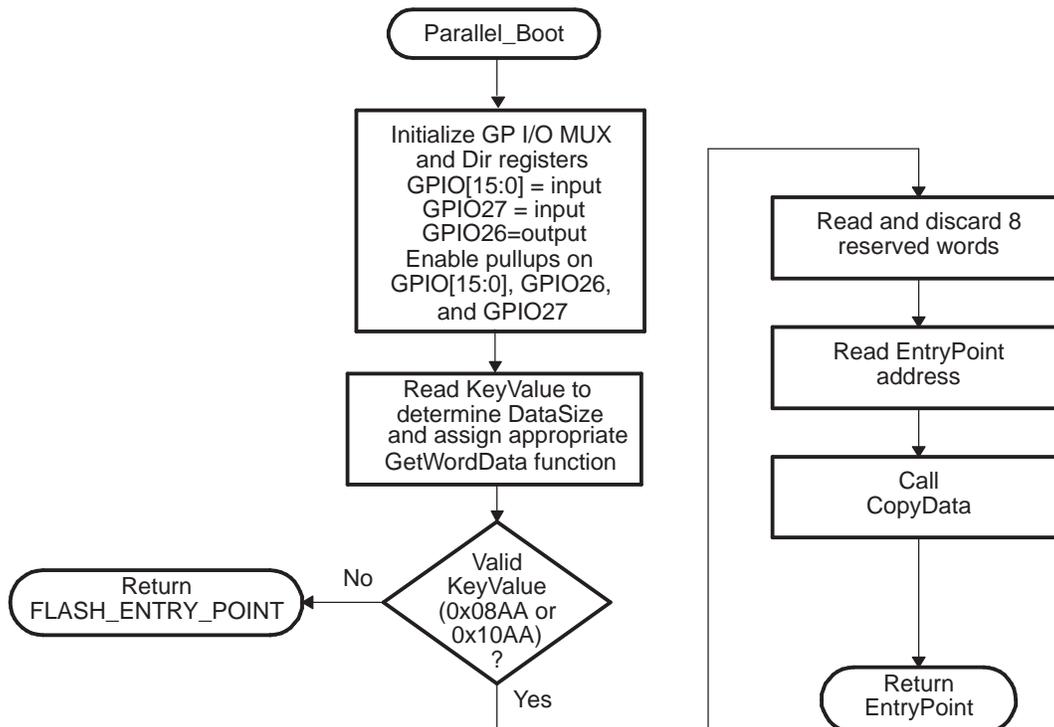


Figure 2-19 shows the transfer flow from the host side. The operating speed of the CPU and host are not critical in this mode as the host will wait for the DSP and the DSP will in turn wait for the host. In this manner the protocol will work with both a host running faster and a host running slower than the DSP.

Figure 2-19. Parallel GPIO Mode - Host Transfer Flow

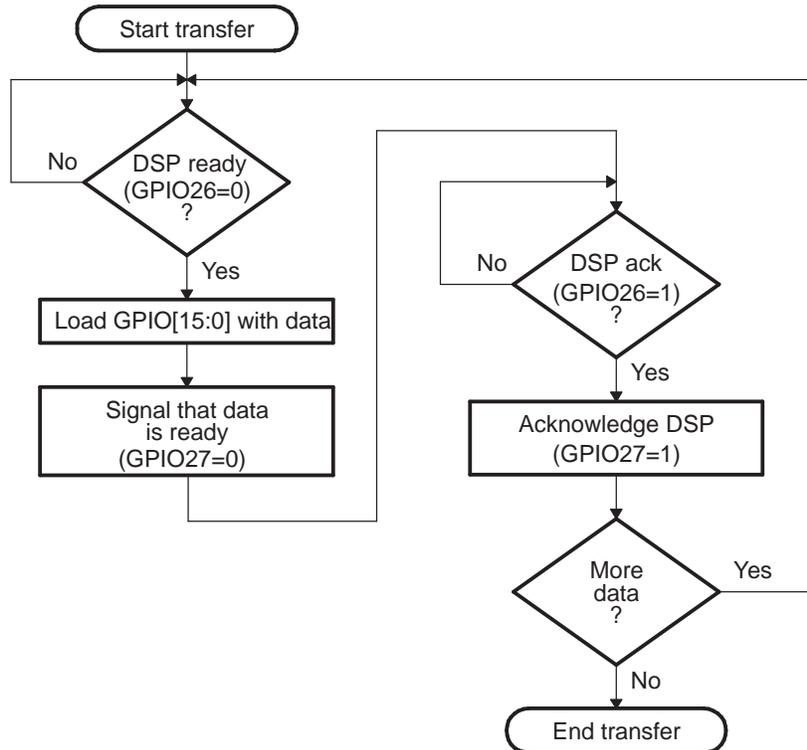


Figure 2-20 and Figure 2-21 show the flow used to read a single word of data from the parallel port. The loader uses the method shown in Figure 2-9 to read the key value and to determine if the incoming data stream width is 8-bit or 16-bit. A different GetWordData function is used by the parallel loader depending on the data size of the incoming data stream.

- **16-bit data stream**

For an 16-bit data stream, the function Parallel_GetWordData16bit is used. This function reads all 16-bits at a time. The flow of this function is shown in Figure 2-20.

- **8-bit data stream**

For an 8-bit data stream, the function Parallel_GetWordData8bit is used. The 8-bit routine, shown in Figure 2-21, 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 2-20. 16-Bit Parallel GetWord Function

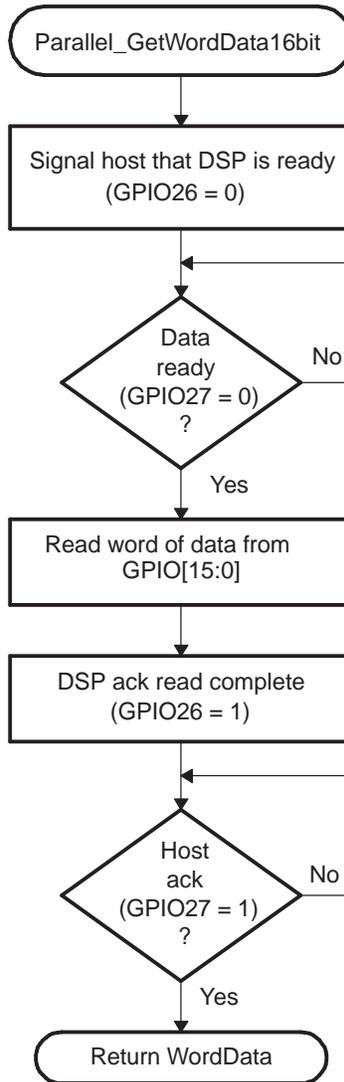
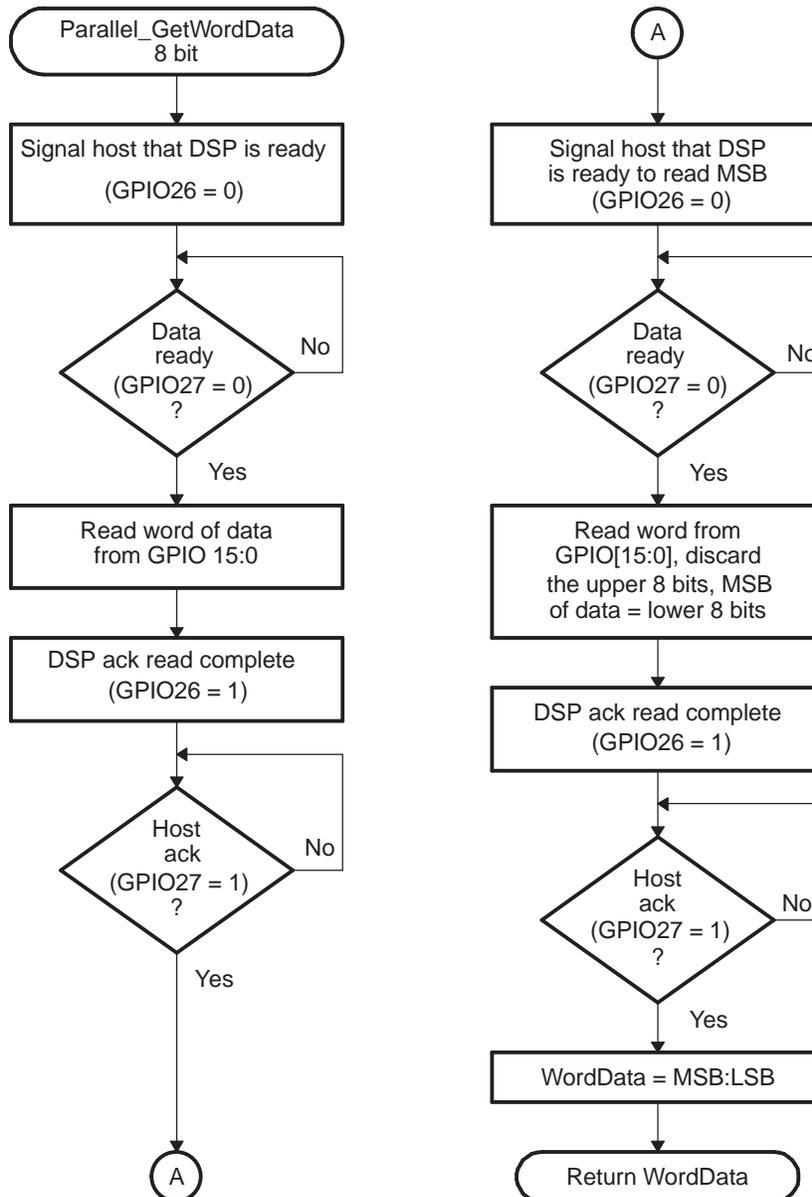


Figure 2-21. 8-Bit Parallel GetWord Function



2.19 XINTF_Parallel_Boot Function

The parallel general purpose I/O (GPIO) boot mode asynchronously transfers code from XD[15:0] to internal memory. Each value can be 16 bits or 8 bits long and follows the same data flow as outlined in [Section 2.10](#). The each word or byte of data is read from address 0x100000 in XINTF zone 6.

Note: This mode loads a stream of data into the SARAM of the device using XINTF resources. If you instead want to configure and jump to the XINTF then use the "Jump to XINTF x16" or "Jump to XINTF x32" boot mode.

The parallel XINTF loader uses following pins:

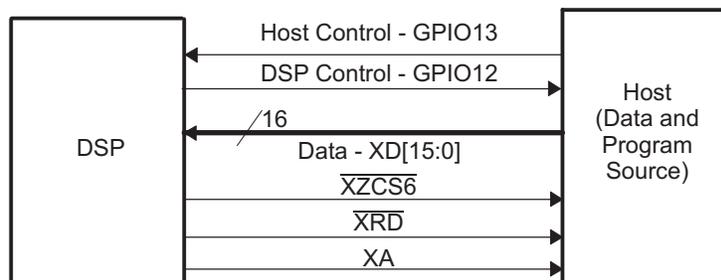
- Data on XD[15:0] or XD[7:0]
- 28x Control on GPIO13
- Host Control on GPIO12

The 28x communicates with the external host device by polling/driving the GPIO13 and GPIO12 lines. The handshake protocol shown in [Figure 2-17](#) must be used to successfully transfer each word via XD[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 XD[7:0] ignoring the higher byte.

The DSP first signals the host that the DSP is ready to begin data transfer by pulling the GPIO12 pin low. The host load then initiates the data transfer by pulling the GPIO13 pin low. The complete protocol is shown in [Figure 2-22](#).

Figure 2-22. Overview of the Parallel XINTF Boot Loader Operation



The DSP communicates with the external host device by polling/driving the GPIO13 and GPIO12 lines. The handshake protocol shown below must be used to successfully transfer each word via the first address location within XINTF zone 6. 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 XD[7:0] ignoring the higher byte.

To begin the transfer, the DSP will use the default XINTF timing for zone 6. This is the maximum wait states, slowest XINTF timing available. That is:

1. XTIMCLK = ½ SYSCLKOUT
2. XCLKOUT = ½ XTIMCLK
3. XRDLEAD = XWRLEAD = 3
4. XRDACTIVE = XWRACTIVE = 7
5. XRDTRAIL = XWRACTIVE = 3
6. XSIZE = 3 for 16-bit wide
7. X2TIMING = 1. Timing values are 2:1.
8. USERREADY = 1, READYMODE = 1 (XREADY sampled synchronous mode)

The first 7 words of the data stream are read at this slow timing. Words 2 – 7 include configuration information that will be used to adjust the PLLCR/PLLSTS and XINTF XTIMING6. The rest of the data stream is read using the new configuration.

The 16-bit data stream is shown in [Table 2-10](#) and the 8-bit data stream is shown in [Table 2-11](#).

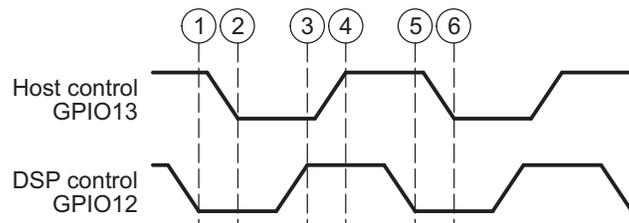
Table 2-10. XINTF Parallel Boot 16-Bit Data Stream

Word	XD[15:0]	Description
1	10AA	10AA (KeyValue for memory width = 16bits)
2	AABB	PLLCR register = 0xAABB
3	000B	PLLSTS[DIVSEL] bits = 0xB
4	AABB	XTIMING6[31:16]
5	CCDD	XTIMING6[15:0] (XTIMING6 = 0xAABBCCDD)
6	EEFF	XINTCNF2[31:16]
7	GGHH	XINTCNF2[15:0] (XINTCNF2 = 0xEEFFGGHH)
8	0000	reserved
9	0000	reserved
10	AABB	Entry point PC[22:16]
11	CCDD	Entry point PC[15:0] (PC = 0xAABBCCDD)
12	MMNN	Block size (number of words) of the first block of data to load = 0xMMNN words
13	AABB	Destination address of first block Addr[31:16]
14	CCDD	Destination address of first block Addr[15:0] (Addr = 0xAABBCCDD)
15	XXXX	First word of the first block in the source being loaded
...		...
...		Data for this section.
...		...
.	XXXX	Last word of the first block of the source being loaded
.	MMNN	Block size of the 2nd block to load = 0xMMNN words
.	AABB	Destination address of second block Addr[31:16]
.	CCDD	Destination address of second block Addr[15:0]
.	XXXX	First word of the second block in the source being loaded
.		...
n	XXXX	Last word of the last block of the source being loaded (More sections if required)
n+1	0000	Block size of 0000h - indicates end of the source program

Table 2-11. XINTF Parallel Boot 8-Bit Data Stream

Bytes	XD[7:0] (Byte 1 of 2)	XD[7:0] (Byte 2 of 2)	Description
1 2	AA	08	0x08AA (KeyValue for memory width = 16bits)
3 4	BB	AA	PLLCR register = 0xAABB
5 6	0B	00	PLLSTS[DIVSEL] bits = 0xB
7 8	BB	AA	XTIMING6[31:16]
9 10	DD	CC	XTIMING6[15:0] (XTIMING6 = 0xAABBCCDD)
11 12	FF	EE	XINTCNF2[31:16]
13 14	HH	GG	XINTCNF2[15:0] (XINTCNF2 = 0xEEFFGGHH)
15 16	00	00	reserved
17 18	00	00	reserved
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

Figure 2-23 shows an overview of the Parallel XINTF bootloader flow.

Figure 2-23. XINTF_Parallel Bootloader Handshake Protocol


1. The 28x device indicates it is ready to start receiving data by pulling the GPIO12 pin low.
2. The bootloader waits until the host puts data on XD[15:0]. The host signals to the 28x device that data is ready by pulling the GPIO13 pin low.
3. The 28x device reads the data and signals the host that the read is complete by pulling GPIO12 high.
4. The bootloader waits until the host acknowledges the 28x by pulling GPIO13 high.
5. The 28x device again indicates it is ready for more data by pulling the GPIO12 pin low.

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

Figure 2-18 shows an overview of the XINTF Parallel bootloader flow.

Figure 2-24. XINTF Parallel Mode Overview

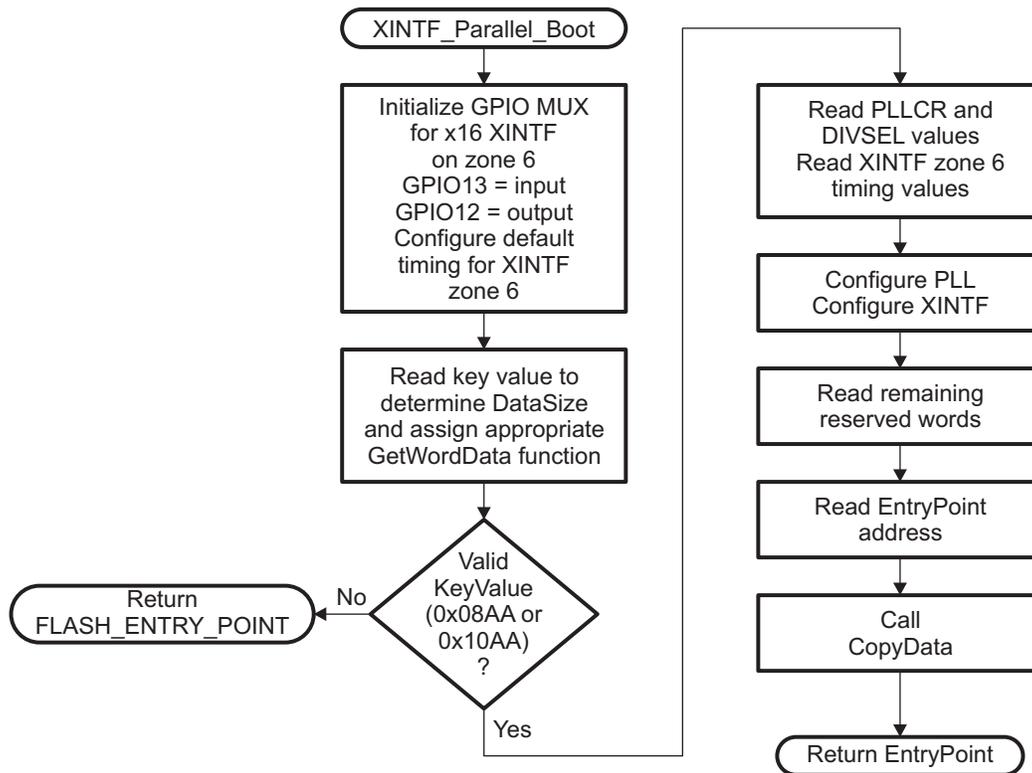


Figure 2-19 shows the transfer flow from the host side. The operating speed of the CPU and host are not critical in this mode as the host will wait for the 28x and the 28x will in turn wait for the host. In this manner the protocol will work with both a host running faster and a host running slower than the 28x device.

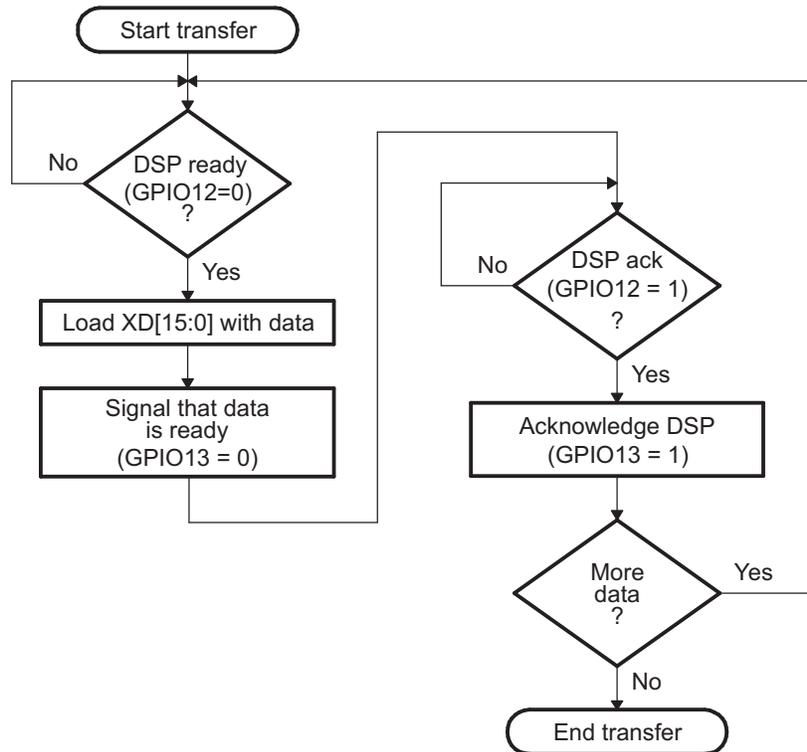
Figure 2-25. XINTF Parallel Mode - Host Transfer Flow


Figure 2-20 and Figure 2-21 show the flow used to read a single word of data from the parallel port. The loader uses the method shown in Figure 2-9 to read the key value and to determine if the incoming data stream width is 8-bit or 16-bit. A different GetWordData function is used by the parallel loader depending on the data size of the incoming data stream.

- **16-bit data stream**

For an 16-bit data stream, the function `XINTF_Parallel_GetWordData16bit` is used. This function reads all 16-bits at a time. The flow of this function is shown in Figure 2-20.

- **8-bit data stream**

For an 8-bit data stream, the function `XINTF_Parallel_GetWordData8bit` is used. The 8-bit routine, shown in Figure 2-21, 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 2-26. 16-Bit Parallel GetWord Function

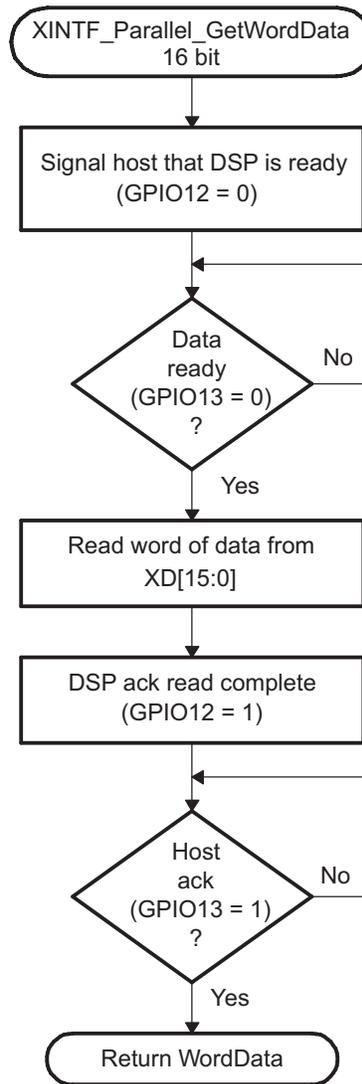
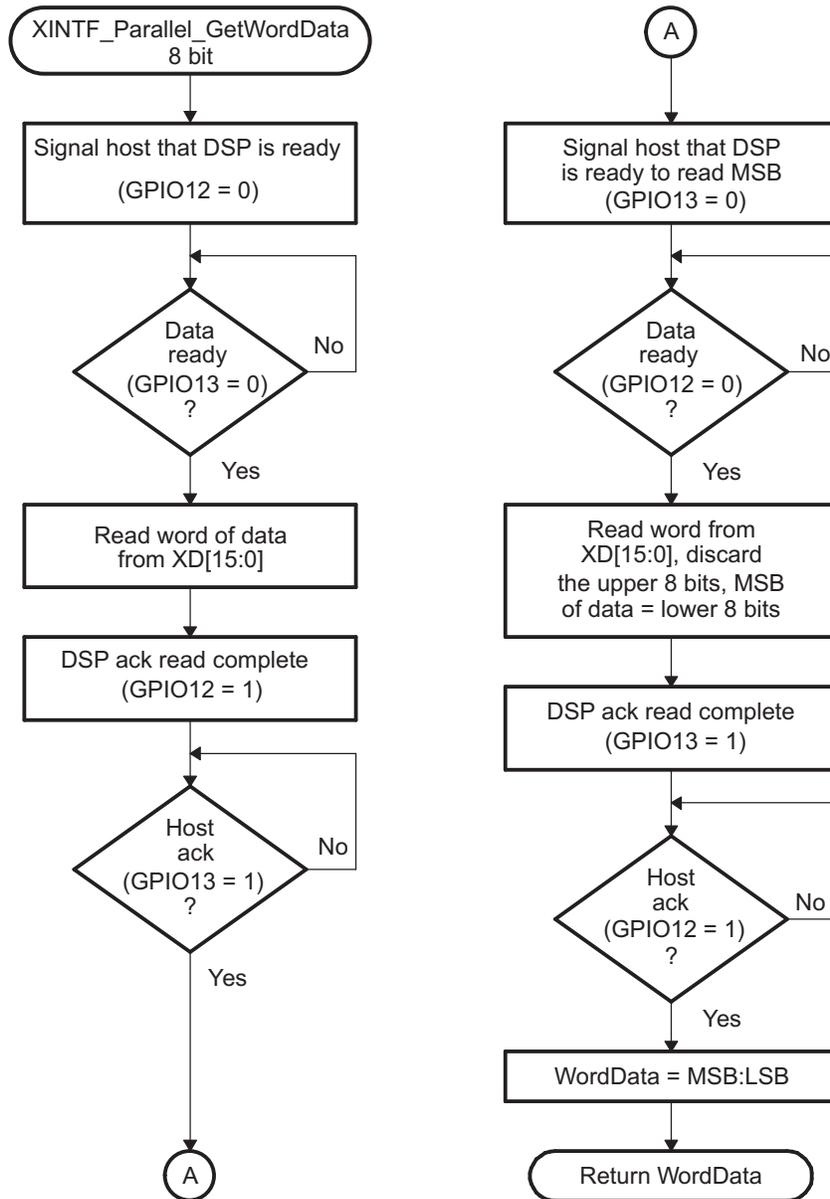


Figure 2-27. 8-Bit Parallel GetWord Function



2.20 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 2-28](#). The SPI bootloader supports an 8-bit data stream. It does not support a 16-bit data stream.

Figure 2-28. SPI Loader



The SPI-A loader uses following pins:

- SPISIMOA on GPIO16
- SPISOMIA on GPIO17
- SPICLKA on GPIO18
- SPISTEAA 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 2-12. 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 2-12. 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 the entry point for the flash (0x33 7FF6) is returned to the calling routine.
- 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 2-29. Data Transfer From EEPROM Flow

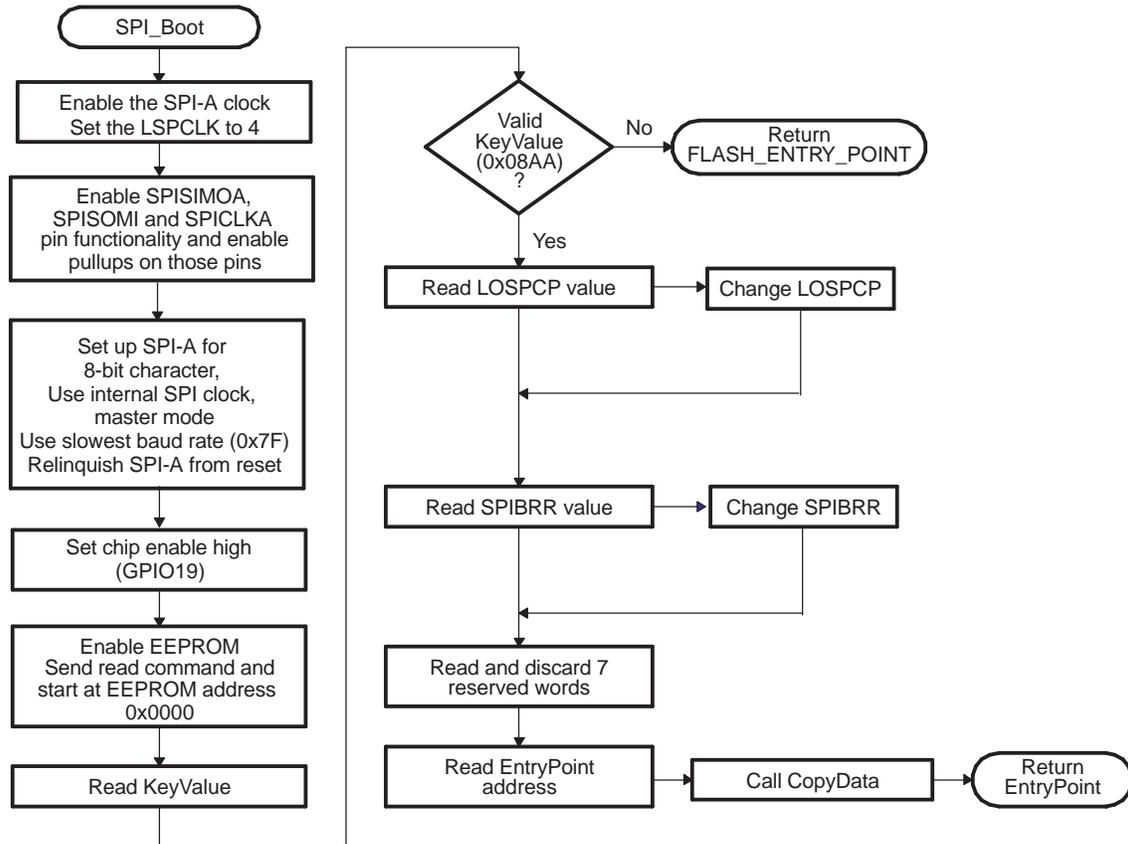
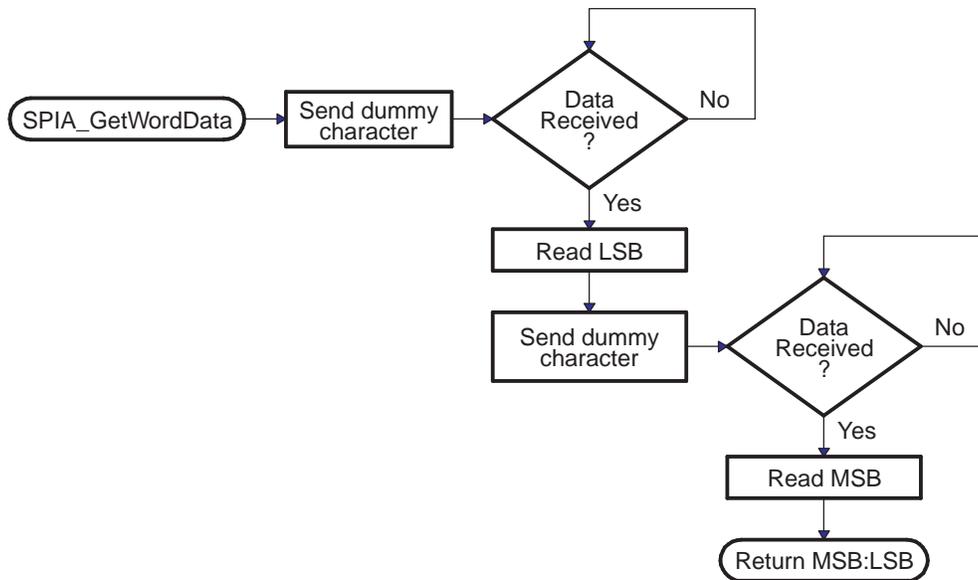


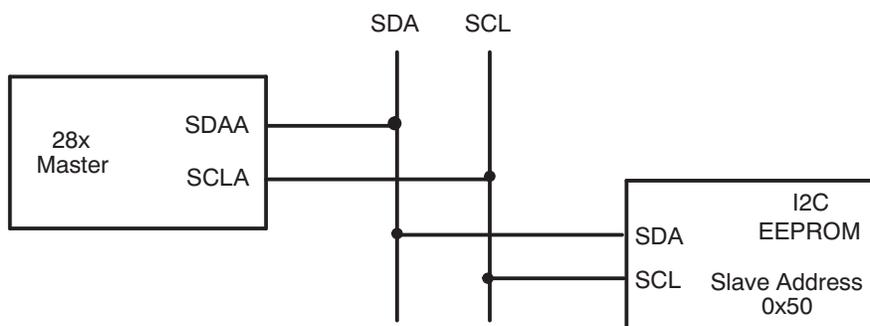
Figure 2-30. Overview of SPIA_GetWordData Function



2.21 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 2-31](#). The EEPROM must adhere to conventional I2C EEPROM protocol, as described in this section, with a 16-bit base address architecture.

Figure 2-31. EEPROM Device at Address 0x50



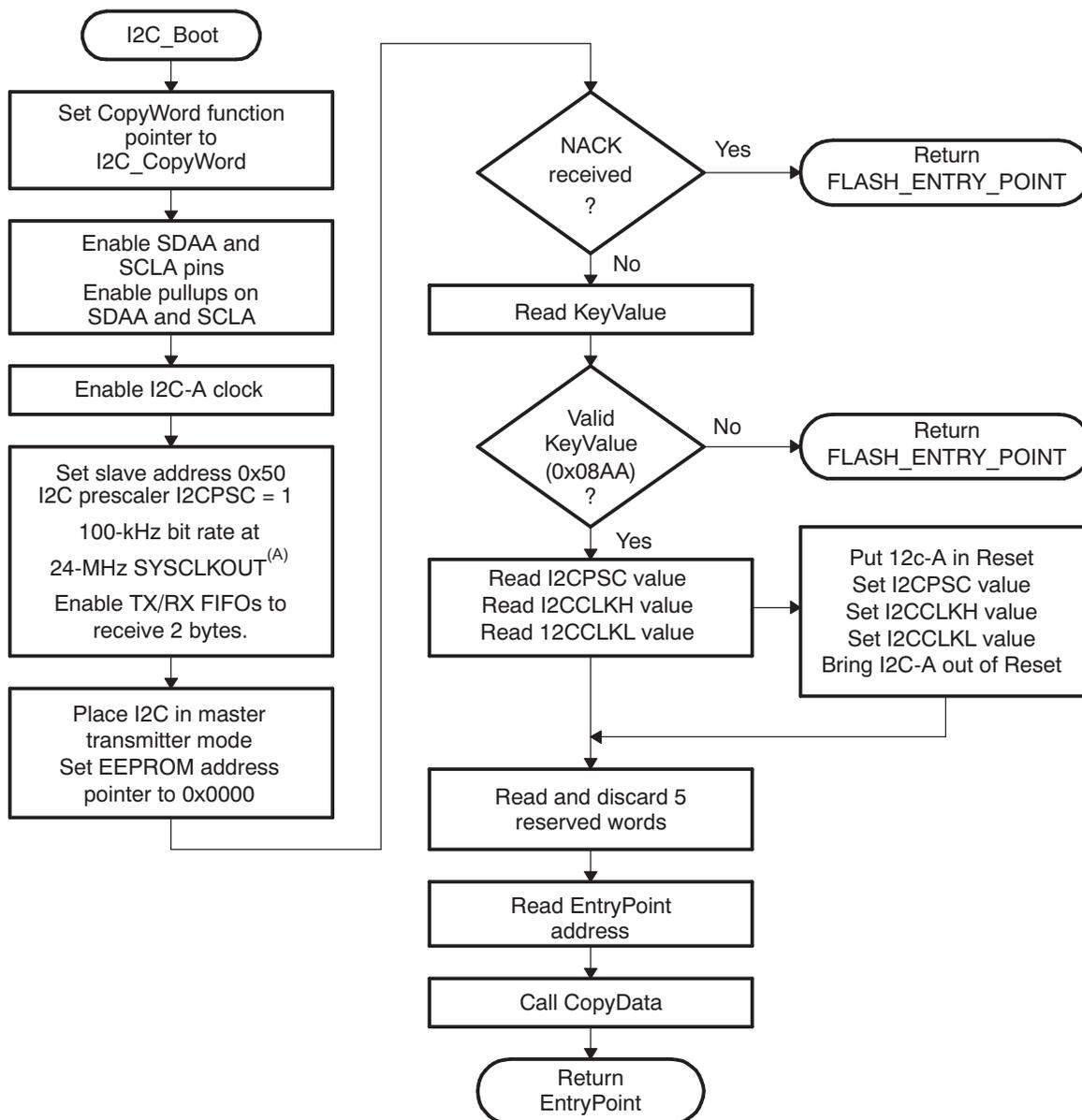
The I2C loader uses following pins:

- 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 2-32. Overview of I2C_Boot Function



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

To use the I2C-A bootloader, the input clock frequency to the device must be between 28 MHz and 48 MHz. This input clock frequency will result in a default 14 MHz to 24 MHz system clock (SYSCLKOUT). By default, the bootloader sets the I2CPSC prescale value to 1 so that the I2C clock will be divided down from SYSCLKOUT. This results in an I2C clock between 7 MHz and 12 MHz, which meets the 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 bootloader 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 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 jump to the flash entry point. 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 2-13](#) shows the 8-bit data stream used by the I2C.

Table 2-13. 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 2-33](#) and [Figure 2-34](#). The first communication, which sets the EEPROM address pointer to 0x0000 and reads the KeyValue (0x08AA) from it, is shown in [Figure 2-33](#). All subsequent reads are shown in [Figure 2-34](#) and are read two bytes at a time.

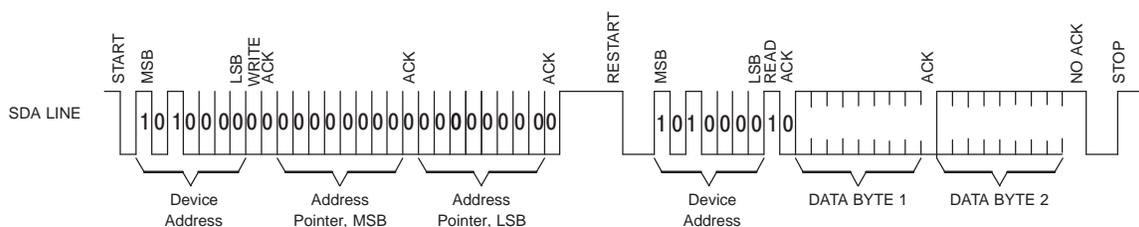
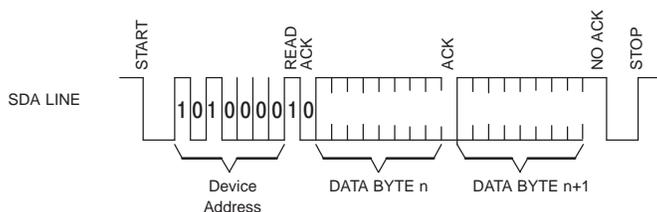
Figure 2-33. Random Read


Figure 2-34. Sequential Read



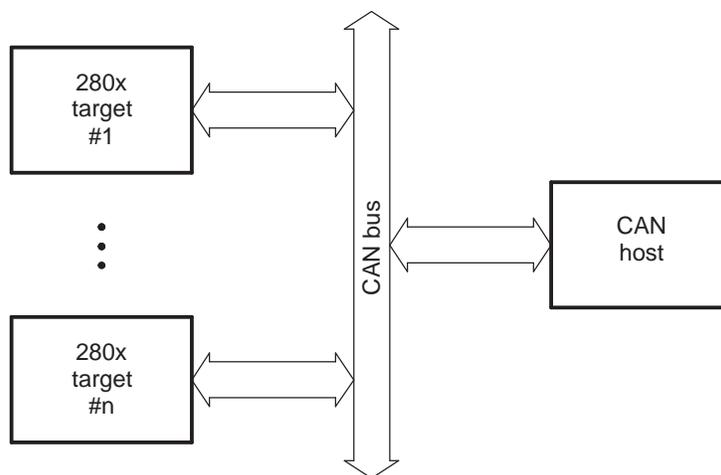
2.22 eCAN Boot Function

The eCAN bootloader 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 can download a kernel to reconfigure the eCAN if higher data throughput is desired.

The eCAN-A loader uses following pins:

- CANRXA on GPIO30
- CANTXA on GPIO31

Figure 2-35. Overview of eCAN-A bootloader 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 2-14](#).

Table 2-14. Bit-Rate Values for Different XCLKIN Values

XCLKIN	SYSCLOCKOUT	Bit Rate
30 MHz	15 MHz	500 kbps
15 MHz	7.5 MHz	250 kbps

The SYSCLOCKOUT values shown 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 device, transmit AA first, followed by 08. The program flow of the CAN bootloader is identical to the SCI bootloader. The data sequence for the CAN bootloader is shown in [Table 2-15](#):

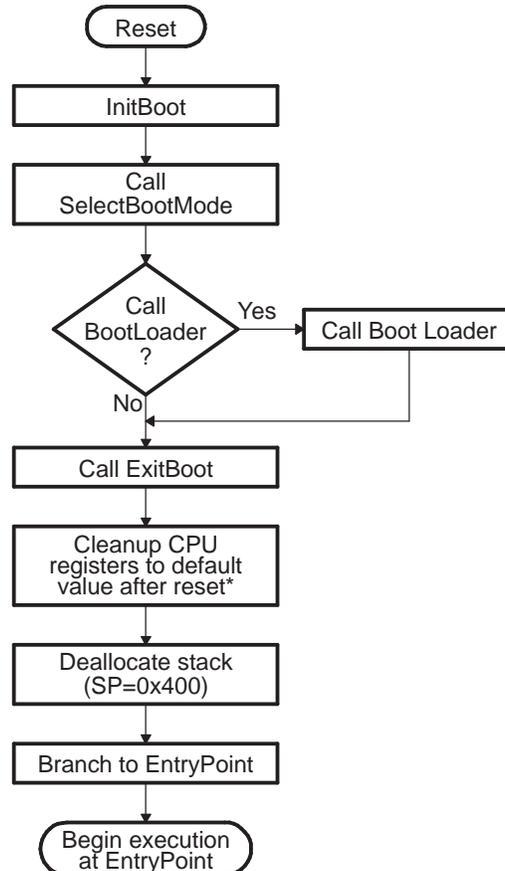
Table 2-15. eCAN 8-Bit Data Stream

Bytes	Byte 1 of 2	Byte 2 of 2	Description
1 2	AA	08	0x08AA (KeyValue for memory width = 16bits)
3 4	00	00	reserved
5 6	00	00	reserved
7 8	00	00	reserved
9 10	00	00	reserved
11 12	00	00	reserved
13 14	00	00	reserved
15 16	00	00	reserved
17 18	00	00	reserved
19 20	BB	00	Entry point PC[22:16]
21 22	DD	CC	Entry point PC[15:0] (PC = 0xAABBCCDD)
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

2.23 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 2-36. 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 2-16. 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	ST1	0x0A0B
	15:10 OVC = 0		15:13 ARP = 0
	9:7 PM = 0		12 XF = 0
	6 V = 0		11 M0M1MAP = 1
	5 N = 0		10 reserved
	4 Z = 0		9 OBJMODE = 1
	3 C = 0		8 AMODE = 0
	2 TC = 0		7 IDLESTAT = 0
	1 OVM = 0		6 EALLOW = 0
	0 SXM = 0		5 LOOP = 0
			4 SPA = 0
			3 VMAP = 1
			2 PAGE0 = 0
			1 DBGM = 1
			0 INTM = 1

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	62
3.2 Example: Preparing a COFF File For eCAN Bootloading	63

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 2.10](#). 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.
-spibr 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 <code>_c_int00</code> unless defined otherwise by the -e linker option.
-i2c8	Specify the source of the bootloader table as the I2C-A port, 8-bit
-i2cpssc value	Specify the value for the I2CPSSC 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.
-i2cckl value	Specify the value for the I2CCKL 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 section	page	origin	length	attributes/ 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.....	68
4.2 Bootloader Code Revision History	68
4.3 Bootloader Code Listing (V2.0)	69

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 FF B9	Flash API silicon compatibility check. This location is read by some versions of the flash API to make sure it is running on a compatible silicon version.
0x3F FF BA	Boot ROM Version Number
0x3F FF BB	MM/YY of release (in decimal)
0x3F FF BC	Least significant word of checksum
0x3F FF BD	...
0x3F FF BE	...
0x3F FF BF	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 <http://www-s.ti.com/sc/techlit/spru963.zip>

Table 4-2. Bootloader Revision Per Device

Device(s)	Silicon REVID (Address 0x883)	Boot ROM Revision
F2833x/F2823x	0 (First silicon)	Version 1
F2833x/F2823x	1	Version 2

4.2 Bootloader Code Revision History

- **Version: 2, Released: March 2008:**

The following changes were made:

- Corrected the GPIO configuration for boot to XINTF x16, x32 and parallel XINTF modes.
- Updated the McBSP boot loader to echo back data received. Version 1 did not echo back the data.
- The eCAN loader leaves the SAM bit in its default state (0). Version 1 changed SAM to 1.

- **Version: 1, Released: June 2007:**

The initial release of the boot ROM.

4.3 Bootloader Code Listing (V2.0)

The following code listing is for the boot ROM code V2.0. To determine the version of the bootloader code check the contents of memory address 0x3F FFBA in the boot ROM. See [Section 4.1](#) for more information.

```
// TI File $Revision: /main/8 $
// Checkin $Date: July 2, 2007 16:41:57 $
//#####
//
// FILE: F2833x_Boot.h
//
// TITLE: F2833x Boot ROM Definitions.
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#ifndef TMS320X2833X_BOOT_H
#define TMS320X2833X_BOOT_H

//-----
// Fixed boot entry points:
//
#define FLASH_ENTRY_POINT 0x33FFF6
#define OTP_ENTRY_POINT 0x380400
#define RAM_ENTRY_POINT 0x000000
#define XINTF_ENTRY_POINT 0x100000
#define PASSWORD_LOCATION 0x33FFF6

#define XTIMING_X16_VAL 0x0043FFFF
#define XTIMING_X32_VAL 0x0041FFFF
#define XINTCNF2_VAL 0x00010D14
#define DIVSEL_BY_4 0
#define DIVSEL_BY_2 2
#define DIVSEL_BY_1 3

#define ERROR 1
#define NO_ERROR 0
#define EIGHT_BIT 8
#define SIXTEEN_BIT 16
#define EIGHT_BIT_HEADER 0x08AA
#define SIXTEEN_BIT_HEADER 0x10AA

#define TI_TEST_EN ((*(unsigned int *)0x09c0) & 0x0001)

typedef Uint16 (* uint16fptr)();
extern uint16fptr GetWordData;
extern Uint16 BootMode;

#endif // end of TMS320x2833x_BOOT_H definition
```

```

;; TI File $Revision: /main/10 $
;; Checkin $Date: March 4, 2008 15:50:32 $
;#####
;;
;; FILE:    Init_Boot.asm
;;
;; TITLE:   2833x Boot Rom Initialization and Exit routines.
;;
;; Functions:
;;
;;     _InitBoot
;;     _ExitBoot
;;
;; Notes:
;;
;#####
;; $TI Release: 2833x/2823x Boot ROM V2 $
;; $Release Date: March 4, 2008 $
;#####

.global _InitBoot
.ref _SelectBootMode

.sect ".Flash"    ; Flash API checks this for
.word 0xFFFFE    ; silicon compatability

.sect ".Version"
.word 0x0002      ; 2833x Boot ROM Version 1
.word 0x0308      ; Month/Year: (3/08 = Mar 2008)

.sect ".Checksum"; 64-bit Checksum
.long 0xAA58557D ; least significant 32-bits
.long 0x000008D9 ; most significant 32-bits

.sect ".InitBoot"

;-----
; _InitBoot
;-----
; This function performs the initial boot routine
; for the boot ROM.
;
; This module performs the following actions:
;
;     1) Initializes the stack pointer
;     2) Sets the device for C28x operating mode
;     3) Calls the main boot functions
;     4) Calls an exit routine
;-----

_InitBoot:
; Initialize the stack pointer.

__stack:    .usect ".stack",0
MOV SP, #__stack ; Initialize the stack pointer

; Initialize the device for running in C28x mode.

C28OBJ      ; Select C28x object mode
C28ADDR     ; Select C27x/C28x addressing
C28MAP      ; Set blocks M0/M1 for C28x mode
CLRC PAGE0  ; Always use stack addressing mode
MOVW DP,#0  ; Initialize DP to point to the low 64 K
CLRC OVM

; Set PM shift of 0

SPM 0

; Decide which boot mode to use

```

```

    LCR  _SelectBootMode

; Cleanup and exit.  At this point the EntryAddr
; is located in the ACC register
    BF  _ExitBoot,UNC

;-----
; _ExitBoot
;-----
;-----
;This module cleans up after the boot loader
;
; 1) Make sure the stack is deallocated.
;    SP = 0x400 after exiting the boot
;    loader
; 2) Push 0 onto the stack so RPC will be
;    0 after using LRETR to jump to the
;    entry point
; 2) Load RPC with the entry point
; 3) Clear all XARn registers
; 4) Clear ACC, P and XT registers
; 5) LRETR - this will also clear the RPC
;    register since 0 was on the stack
;-----

_ExitBoot:

;-----
;    Insure that the stack is deallocated
;-----

    MOV SP,#__stack

;-----
; Clear the bottom of the stack.  This will end up
; in RPC when we are finished
;-----

    MOV  *SP++,#0
    MOV  *SP++,#0

;-----
; Load RPC with the entry point as determined
; by the boot mode.  This address will be returned
; in the ACC register.
;-----

    PUSH ACC
    POP  RPC

;-----
; Put registers back in their reset state.
;
; Clear all the XARn, ACC, XT, and P and DP
; registers
;
; NOTE: Leave the device in C28x operating mode
;       (OBJMODE = 1, AMODE = 0)
;-----

    ZAPA
    MOVL XT,ACC
    MOVZ ARO,AL
    MOVZ AR1,AL
    MOVZ AR2,AL
    MOVZ AR3,AL
    MOVZ AR4,AL
    MOVZ AR5,AL
    MOVZ AR6,AL
    MOVZ AR7,AL
    MOVW DP, #0

;-----

```

```

; Restore ST0 and ST1. Note OBJMODE is
; the only bit not restored to its reset state.
; OBJMODE is left set for C28x object operating
; mode.
;
; ST0 = 0x0000      ST1 = 0x0A0B
; 15:10 OVC = 0    15:13      ARP = 0
; 9: 7  PM = 0    12          XF = 0
; 6    V = 0     11  MOM1MAP = 1
; 5    N = 0     10  reserved
; 4    Z = 0     9   OBJMODE = 1
; 3    C = 0     8   AMODE = 0
; 2    TC = 0    7   IDLESTAT = 0
; 1    OVM = 0   6   EALLOW = 0
; 0    SXM = 0   5   LOOP = 0
;                4   SPA = 0
;                3   VMAP = 1
;                2   PAGE0 = 0
;                1   DBGM = 1
;                0   INTM = 1
;-----
MOV  *SP++,#0
MOV  *SP++,#0x0A0B
POP  ST1
POP  ST0
;-----
; Jump to the EntryAddr as defined by the
; boot mode selected and continue execution
;-----

LRETR

;eof -----

```

```

// TI File $Revision: /main/10 $
// Checkin $Date: January 9, 2008 12:59:56 $
//#####
//
// FILE:      SelectMode_Boot.c
//
// TITLE:     2833x Boot Mode selection routines
//
// Functions:
//
//     Uint32 SelectBootMode(void)
//     inline void SelectMode_GPOISelect(void)
//
// Notes:
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"
#include "TMS320x2833x_Boot.h"

// External functions referenced by this file
extern Uint32 SCI_Boot(void);
extern Uint32 SPI_Boot(void);
extern Uint32 Parallel_Boot(void);
extern Uint32 XINTF_Parallel_Boot(void);
extern Uint32 I2C_Boot(void);
extern Uint32 CAN_Boot();
extern Uint32 MCBSP_Boot();
extern Uint32 XINTF_Boot(Uint16 size);
extern void WatchDogEnable(void);
extern void WatchDogDisable(void);
extern void WatchDogService(void);
extern void ADC_cal(void);

// Functions in this file
Uint32 SelectBootMode(void);

//
//      GPIO87   GPIO86   GPIO85   GPIO84
//      XA15     XA14     XA13     XA12
//      PU       PU       PU       PU
//
//=====
//Mode F      1         1         1         1   Jump to Flash
//Mode E      1         1         1         0   SCI-A boot
//Mode D      1         1         0         1   SPI-A boot
//Mode C      1         1         0         0   I2C-A boot
//Mode B      1         0         1         1   eCAN-A boot
//Mode A      1         0         1         0   McBSP-A boot
//Mode 9      1         0         0         1   Jump to XINTF x16
//Mode 8      1         0         0         0   Jump to XINTF x32
//Mode 7      0         1         1         1   Jump to OTP
//Mode 6      0         1         1         0   Parallel GPIO I/O boot
//Mode 5      0         1         0         1   Parallel XINTF boot
//Mode 4      0         1         0         0   Jump to SARAM
//Mode 3      0         0         1         1   Branch to check boot mode
//Mode 2      0         0         1         0   Boot to flash, bypass ADC cal
//Mode 1      0         0         0         1   Boot to SARAM, bypass ADC cal
//Mode 0      0         0         0         0   Boot to SCI-A, bypass ADC cal

#define FLASH_BOOT           0xF
#define SCI_BOOT             0xE
#define SPI_BOOT             0xD
#define I2C_BOOT             0xC
#define CAN_BOOT             0xB
#define MCBSP_BOOT          0xA
#define XINTF_16_BOOT        0x9
#define XINTF_32_BOOT        0x8
#define OTP_BOOT             0x7
#define PARALLEL_BOOT        0x6
#define XINTF_PARALLEL_BOOT  0x5
    
```

```

#define RAM_BOOT          0x4
#define LOOP_BOOT        0x3
#define FLASH_BOOT_NOCAL 0x2
#define RAM_BOOT_NOCAL   0x1
#define SCI_BOOT_NOCAL   0x0

Uint32 SelectBootMode()
{
    Uint32 EntryAddr;

#ifdef _DEBUGSELECT
    // To debug without having to select
    // the boot mode via jumpers, define
    // _DEBUGSELECT and comment out the
    // appropriate boot mode to test

    EALLOW;
    SysCtrlRegs.WDCR = 0x0068;    // Disable watchdog module
    EDIS;

    // EntryAddr = SCI_Boot();
    // EntryAddr = SPI_Boot();
    // EntryAddr = Parallel_Boot();
    // EntryAddr = XINTF_Parallel_Boot();
    EntryAddr = XINTF_Boot(16);
    // EntryAddr = XINTF_Boot(32);
    // EntryAddr = MCBSP_Boot();
    // EntryAddr = I2C_Boot();
    // EntryAddr = CAN_Boot();

    EALLOW;
    SysCtrlRegs.WDCR = 0x0028; // Enable watchdog module
    SysCtrlRegs.WDKEY = 0x55; // Clear the WD counter
    SysCtrlRegs.WDKEY = 0xAA;
    EDIS;
#endif

#ifdef _DEBUGSELECT
    EALLOW;

    // At reset we are in /4 mode.  Change to /2
    SysCtrlRegs.PLLSTS.bit.DIVSEL = DIVSEL_BY_2;

    // Set MUX for BOOT Select
    GpioCtrlRegs.GPCMUX2.bit.GPIO87 = 0;
    GpioCtrlRegs.GPCMUX2.bit.GPIO86 = 0;
    GpioCtrlRegs.GPCMUX2.bit.GPIO85 = 0;
    GpioCtrlRegs.GPCMUX2.bit.GPIO84 = 0;

    // Set DIR for BOOT Select
    GpioCtrlRegs.GPCDIR.bit.GPIO87 = 0;
    GpioCtrlRegs.GPCDIR.bit.GPIO86 = 0;
    GpioCtrlRegs.GPCDIR.bit.GPIO85 = 0;
    GpioCtrlRegs.GPCDIR.bit.GPIO84 = 0;
    EDIS;

    WatchDogService();

    if(TI_TEST_EN == 0)
    {
        do
        {
            // Read the BootMode from the pins.  If the mode is
            // "LOOP_BOOT" then keep checking with the watchdog enabled.
            // "LOOP_BOOT" will typically only used for debug
            BootMode = GpioDataRegs.GPCDAT.bit.GPIO87 << 3;
            BootMode |= GpioDataRegs.GPCDAT.bit.GPIO86 << 2;
            BootMode |= GpioDataRegs.GPCDAT.bit.GPIO85 << 1;
            BootMode |= GpioDataRegs.GPCDAT.bit.GPIO84;
            if (BootMode == LOOP_BOOT) asm("    ESTOP0");
        } while (BootMode == LOOP_BOOT);
    }
}

```

```

WatchDogService();
// Read the password locations - this will unlock the
// CSM only if the passwords are erased. Otherwise it
// will not have an effect.
CsmPwl.PSWD0;
CsmPwl.PSWD1;
CsmPwl.PSWD2;
CsmPwl.PSWD3;
CsmPwl.PSWD4;
CsmPwl.PSWD5;
CsmPwl.PSWD6;
CsmPwl.PSWD7;

WatchDogService();

// First check for modes which bypass ADC calibration
if(BootMode == FLASH_BOOT_NOCAL) return FLASH_ENTRY_POINT;
if(BootMode == RAM_BOOT_NOCAL) return RAM_ENTRY_POINT;
if(BootMode == SCI_BOOT_NOCAL)
{
    WatchDogDisable();
    EntryAddr = SCI_Boot();
    goto DONE;
}

WatchDogService();

// Call ADC Cal.
// This function is programmed into the OTP by the factory
EALLOW;
SysCtrlRegs.PCLKCR0.bit.ADCENCLK = 1;
ADC_cal();
SysCtrlRegs.PCLKCR0.bit.ADCENCLK = 0;
EDIS;

// Check for modes which do not require a boot loader (Flash/RAM/OTP)
if(BootMode == FLASH_BOOT) return FLASH_ENTRY_POINT;
else if(BootMode == RAM_BOOT) return RAM_ENTRY_POINT;
else if(BootMode == OTP_BOOT) return OTP_ENTRY_POINT;
else if(BootMode == XINTF_16_BOOT) return EntryAddr = XINTF_Boot(16);
else if(BootMode == XINTF_32_BOOT) return EntryAddr = XINTF_Boot(32);

// Disable the watchdog and check for the other boot modes

WatchDogDisable();

if(BootMode == SCI_BOOT) EntryAddr = SCI_Boot();
else if(BootMode == SPI_BOOT) EntryAddr = SPI_Boot();
else if(BootMode == I2C_BOOT) EntryAddr = I2C_Boot();
else if(BootMode == CAN_BOOT) EntryAddr = CAN_Boot();
else if(BootMode == MCBSP_BOOT) EntryAddr = MCBSP_Boot();
else if(BootMode == PARALLEL_BOOT) EntryAddr = Parallel_Boot();
else if(BootMode == XINTF_PARALLEL_BOOT) EntryAddr = XINTF_Parallel_Boot();
else return FLASH_ENTRY_POINT;

DONE:
    WatchDogEnable();

#endif // end of not _DEBUGSELECT

    return EntryAddr;
}

```

```

// TI File $Revision: /main/5 $
// Checkin $Date: May 30, 2007 13:36:47 $
//#####
//
// FILE:      SysCtrl_Boot.c
//
// TITLE:     F2810/12 Boot Rom System Control Routines
//
// Functions:
//
//     void WatchDogDisable(void)
//     void WatchDogEnable(void)
//
// Notes:
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"
#include "TMS320x2833x_Boot.h"

//-----
// This module disables the watchdog timer.
//-----

void WatchDogDisable()
{
    EALLOW;
    SysCtrlRegs.WDCR = 0x0068;           // Disable watchdog module
    EDIS;
}

//-----
// This module enables the watchdog timer.
//-----

void WatchDogEnable()
{
    EALLOW;
    SysCtrlRegs.WDCR = 0x0028;           // Enable watchdog module
    SysCtrlRegs.WDKEY = 0x55;           // Clear the WD counter
    SysCtrlRegs.WDKEY = 0xAA;
    EDIS;
}

//-----
// This module services the watchdog timer.
//-----
void WatchDogService(void)
{
    EALLOW;
    SysCtrlRegs.WDKEY = 0x0055;
    SysCtrlRegs.WDKEY = 0x00AA;
    EDIS;
}

//-----
// This module sets up the PLL.
//-----
void InitPll(Uint16 val, Uint16 divsel)
{
    // Make sure the PLL is not running in limp mode
    // If it is, there is nothing we can do here.
    // The user must check for this condition in the main application
    if (SysCtrlRegs.PLLSTS.bit.MCLKSTS != 0) return;

    EALLOW;

    // Change the PLLCR

```

```
if (SysCtrlRegs.PLLCR.bit.DIV != val)
{
    // CLKINDIV MUST be 0 before PLLCR can be changed
    SysCtrlRegs.PLLSTS.bit.DIVSEL = DIVSEL_BY_4;

    // Before setting PLLCR turn off missing clock detect logic
    SysCtrlRegs.PLLSTS.bit.MCLKOFF = 1;

    SysCtrlRegs.PLLCR.bit.DIV = val;
    while(SysCtrlRegs.PLLSTS.bit.PLLLOCKS != 1) {}

    // Turn missing clock detect back on
    SysCtrlRegs.PLLSTS.bit.MCLKOFF = 0;
}

if (SysCtrlRegs.PLLSTS.bit.DIVSEL != divsel)
{
    if((divsel == DIVSEL_BY_4) || (divsel == DIVSEL_BY_2))
    {
        SysCtrlRegs.PLLSTS.bit.DIVSEL = divsel;
    }
    else
    {
        if(SysCtrlRegs.PLLSTS.bit.DIVSEL == DIVSEL_BY_4)
        {
            // If switching to 1/1 from 1/4
            // * First go to 1/2 and let the power settle some
            // * Then switch to 1/1
            SysCtrlRegs.PLLSTS.bit.DIVSEL != DIVSEL_BY_2;
            asm(" RPT #200 || NOP");
            asm(" RPT #200 || NOP");
        }
        SysCtrlRegs.PLLSTS.bit.DIVSEL = DIVSEL_BY_1;
    }
}

EDIS;
}

// EOF -----
```

```

// TI File $Revision: /main/4 $
// Checkin $Date: July 2, 2007 16:41:42 $
//#####
//
// FILE:    Shared_Boot.c
//
// TITLE:   2833x Boot loader shared functions
//
// Functions:
//
//     void CopyData(void)
//     Uint32 GetLongData(void)
//     void ReadReservedFn(void)
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"
#include "TMS320x2833x_Boot.h"

// GetWordData is a pointer to the function that interfaces to the peripheral.
// Each loader assigns this pointer to it's particular GetWordData function.
uint16fptr GetWordData;
Uint16 BootMode;

// Function prototypes
Uint32 GetLongData();
void CopyData(void);
void ReadReservedFn(void);

//#####
// void CopyData(void)
//-----
// This routine copies multiple blocks of data from the host
// to the specified RAM locations. There is no error
// checking on any of the destination addresses.
// That is it is assumed all addresses and block size
// values are correct.
//
// Multiple blocks of data are copied until a block
// size of 00 00 is encountered.
//
//-----

void CopyData()
{
    struct HEADER {
        Uint16 BlockSize;
        Uint32 DestAddr;
    } BlockHeader;

    Uint16 wordData;
    Uint16 i;

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

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

    while(BlockHeader.BlockSize != (Uint16)0x0000)
    {
        BlockHeader.DestAddr = GetLongData();
        for(i = 1; i <= BlockHeader.BlockSize; i++)
        {
            wordData = (*GetWordData)();
            *(Uint16 *)BlockHeader.DestAddr++ = wordData;
        }
    }
}

```

```

    }

    // Get the size of the next block
    BlockHeader.BlockSize = (*GetWordData());
  }
  return;
}

//#####
// Uint32 GetLongData(void)
//-----
// This routine fetches a 32-bit value from the peripheral
// input stream.
//-----

Uint32 GetLongData()
{
    Uint32 longData;

    // Fetch the upper 1/2 of the 32-bit value
    longData = ( (Uint32)(*GetWordData()) << 16);

    // Fetch the lower 1/2 of the 32-bit value
    longData |= (Uint32)(*GetWordData());

    return longData;
}

//#####
// void Read_ReservedFn(void)
//-----
// This function reads 8 reserved words in the header.
// None of these reserved words are used by the
// this boot loader at this time, they may be used in
// future devices for enhancements. Loaders that use
// these words use their own read function.
//-----

void ReadReservedFn()
{
    Uint16 i;
    // Read and discard the 8 reserved words.
    for(i = 1; i <= 8; i++)
    {
        GetWordData();
    }
    return;
}

// TI File $Revision: /main/5 $
// Checkin $Date: May 8, 2007 12:28:35 $
//#####
//
// FILE:     SPI_Boot.c
//
// TITLE:    2833x SPI Boot mode routines
//
// Functions:
//
//     Uint32 SPI_Boot(void)
//     inline void SPIA_Init(void)
//     Uint16 SPIA_SetAddress_KeyChk(void)
//     inline void SPIA_Transmit(ul6 cmdData)
//     inline void SPIA_ReservedFn(void);
//     Uint32 SPIA_GetWordData(void)
//
// Notes:
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"

```

```

#include "TMS320x2833x_Boot.h"

// Private functions
inline void SPIA_Init(void);
inline Uint16 SPIA_Transmit(Uint16 cmdData);
inline void SPIA_ReservedFn(void);
Uint16 SPIA_GetWordData(void);
Uint16 SPIA_SetAddress_KeyChk(void);

// External functions
extern void CopyData(void);
Uint32 GetLongData(void);

//#####
// Uint32 SPI_Boot(void)
//-----
// This module is the main SPI boot routine.
// It will load code via the SPI-A port.
//
// It will return a entry point address back
// to the ExitBoot routine.
//-----

Uint32 SPI_Boot()
{
    Uint32 EntryAddr;

    // Assign GetWordData to the SPI-A version of the
    // function. GetWordData is a pointer to a function.
    GetWordData = SPIA_GetWordData;
    // 1. Init SPI-A and set
    //     EEPROM chip enable - low
    SPIA_Init();
    // 2. Enable EEPROM and send EEPROM Read Command
    SPIA_Transmit(0x0300);
    // 3. Send Starting Address for Serial EEPROM (16-bit - 0x0000,0000)
    //     or Serial Flash (24-bit - 0x0000,0000,0000)
    //     Then check for 0x08AA data header, else go to flash
    if(SPIA_SetAddress_KeyChk() != 0x08AA) return FLASH_ENTRY_POINT;
    // 4. Check for Clock speed change and reserved words
    SPIA_ReservedFn();
    // 5. Get point of entry address after load
    EntryAddr = GetLongData();
    // 6. Receive and copy one or more code sections to destination addresses
    CopyData();
    // 7. Disable EEPROM chip enable - high
    //     Chip enable - high
    GpioDataRegs.GPASET.bit.GPIO19 = 1;

    return EntryAddr;
}

//#####
// Uint16 SPIA_SetAddress_KeyChk(void)
//-----
// This routine sends either a 16-bit address to
// Serial EEPROM or a 24-bit address to Serial
// FLASH. It then fetches the 2 bytes that make
// up the key value from the SPI-A port and
// puts them together to form a single
// 16-bit value. It is assumed that the host is
// sending the data in the form MSB:LSB.
//-----

Uint16 SPIA_SetAddress_KeyChk()
{
    Uint16 keyValue;
    // Transmit first byte of Serial Memory address
    SPIA_Transmit(0x0000);
    // Transmit second byte of Serial Memory address
    SPIA_Transmit(0x0000);
    // Transmit third byte of Serial Memory address (0x00) if using Serial Flash
    // or receive first byte of key value if using Serial EEPROM.
    keyValue = SPIA_Transmit(0x0000);
}

```

```

// If previously received LSB of key value (Serial EEPROM), then fetch MSB of key value
if (keyValue == 0x00AA)
{
    keyValue |= (SPIA_Transmit(0x0000)<<8);
}
else
{
    // Serial Flash is being used - keyValue will be received after 24-bit address in the next 2
bytes
    // Fetch Key Value LSB (after 24-bit addressing)
    keyValue = SPIA_Transmit(0x0000);
    // Fetch Key Value MSB (after 24-bit addressing)
    keyValue |= (SPIA_Transmit(0x0000)<<8);
}
return keyValue;
}

//#####
// void SPIA_Init(void)
//-----
// Initialize the SPI-A port for communications
// with the host.
//-----

inline void SPIA_Init()
{
    // Enable SPI-A clocks
    EALLOW;
    SysCtrlRegs.PCLKCR0.bit.SPIAENCLK = 1;
    SysCtrlRegs.LOSPCP.all = 0x0002;
    // Enable FIFO reset bit only
    SpiaRegs.SPIFFTX.all=0x8000;
    // 8-bit character
    SpiaRegs.SPICCR.all = 0x0007;
    // Use internal SPICLK master mode and Talk mode
    SpiaRegs.SPICTL.all = 0x000E;
    // Use the slowest baud rate
    SpiaRegs.SPIBRR = 0x007f;
    // Relinquish SPI-A from reset
    SpiaRegs.SPICCR.all = 0x0087;
    // Enable SPISIMO/SPISOMI/SPICLK pins
    // Enable pull-ups on SPISIMO/SPISOMI/SPICLK/SPISTE pins
    // GpioCtrlRegs.GPAPUD.bit.GPIO16 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO17 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO18 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO19 = 0;
    GpioCtrlRegs.GPAPUD.all &= 0xFFFF0FFF;
    // GpioCtrlRegs.GPAMUX2.bit.GPIO16 = 1;
    // GpioCtrlRegs.GPAMUX2.bit.GPIO17 = 1;
    // GpioCtrlRegs.GPAMUX2.bit.GPIO18 = 1;
    GpioCtrlRegs.GPAMUX2.all |= 0x00000015;
    // SPI-A pins are asynch
    // GpioCtrlRegs.GPAQSEL2.bit.GPIO16 = 3;
    // GpioCtrlRegs.GPAQSEL2.bit.GPIO17 = 3;
    // GpioCtrlRegs.GPAQSEL2.bit.GPIO18 = 3;
    GpioCtrlRegs.GPAQSEL2.all |= 0x0000003F;
    // IOPORT as output pin instead of SPISTE
    GpioCtrlRegs.GPAMUX2.bit.GPIO19 = 0;
    GpioCtrlRegs.GPADIR.bit.GPIO19 = 1;
    // Chip enable - low
    GpioDataRegs.GPACLEAR.bit.GPIO19 = 1;
    EDIS;
    return;
}

//#####
// Uint16 SPIA_Transmit(Uint16 cmdData)
//-----
// Send a byte/words through SPI transmit channel
//-----

inline Uint16 SPIA_Transmit(Uint16 cmdData)
{
    Uint16 recvData;

```

```

    // Send Read command/dummy word to EEPROM to fetch a byte
    SpiaRegs.SPITXBUF = cmdData;
    while( (SpiaRegs.SPISTS.bit.INT_FLAG) !=1);
    // Clear SPIINT flag and capture received byte
    recvData = SpiaRegs.SPIRXBUF;
    return recvData;
}

//#####
// void SPIA_ReservedFn(void)
//-----
// This function reads 8 reserved words in the header.
// The first word has parameters for LOSPCP
// and SPIBRR register 0xMSB:LSB, LSB = is a three
// bit field for LOSPCP change MSB = is a 6bit field
// for SPIBRR register update
//
// If either byte is the default value of the register
// then no speed change occurs. The default values
// are LOSPCP = 0x02 and SPIBRR = 0x7F
// The remaining reserved words are read and discarded
// and then returns to the main routine.
//-----

inline void SPIA_ReservedFn()
{
    Uint16 speedData;
    Uint16 I;

    // update LOSPCP register
    speedData = SPIA_Transmit((Uint16)0x0000);
    EALLOW;
    SysCtrlRegs.LOSPCP.all = speedData;
    EDIS;
    asm(" RPT #0x0F ||NOP");

    // update SPIBRR register
    speedData = SPIA_Transmit((Uint16)0x0000);
    SpiaRegs.SPIBRR = speedData;
    asm(" RPT #0x0F ||NOP");

    // Read and discard the next 7 reserved words.
    for(I = 1; I <= 7; I++)
    {
        SPIA_GetWordData();
    }
    return;
}

//#####
// Uint16 SPIA_GetWordData(void)
//-----
// This routine fetches two bytes from the SPI-A
// port and puts them together to form a single
// 16-bit value. It is assumed that the host is
// sending the data in the form MSB:LSB.
//-----

Uint16 SPIA_GetWordData()
{
    Uint16 wordData;
    // Fetch the LSB
    wordData = SPIA_Transmit(0x0000);
    // Fetch the MSB
    wordData |= (SPIA_Transmit(0x0000) << 8);
    return wordData;
}

// TI File $Revision: /main/4 $
// Checkin $Date: June 4, 2007 14:35:14 $
//#####
//
// FILE: SCI_Boot.c
//

```

```

// TITLE:   2833x SCI Boot mode routines
//
// Functions:
//
//   Uint32 SCI_Boot(void)
//   inline void SCIA_Init(void)
//   inline void SCIA_AutobaudLock(void)
//   Uint32 SCIA_GetWordData(void)
//
// Notes:
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"
#include "TMS320x2833x_Boot.h"

// Private functions
inline void SCIA_Init(void);
inline void SCIA_AutobaudLock(void);
Uint16 SCIA_GetWordData(void);

// External functions
extern void CopyData(void);
Uint32 GetLongData(void);
extern void ReadReservedFn(void);

//#####
// Uint32 SCI_Boot(void)
//-----
// This module is the main SCI boot routine.
// It will load code via the SCI-A port.
//
// It will return a entry point address back
// to the InitBoot routine which in turn calls
// the ExitBoot routine.
//-----

Uint32 SCI_Boot()
{
    Uint32 EntryAddr;

    // Assign GetWordData to the SCI-A version of the
    // function.  GetWordData is a pointer to a function.
    GetWordData = SCIA_GetWordData;

    SCIA_Init();
    SCIA_AutobaudLock();

    // If the KeyValue was invalid, abort the load
    // and return the flash entry point.
    if (SCIA_GetWordData() != 0x08AA) return FLASH_ENTRY_POINT;

    ReadReservedFn();

    EntryAddr = GetLongData();

    CopyData();

    return EntryAddr;
}

//#####
// void SCIA_Init(void)
//-----
// Initialize the SCI-A port for communications
// with the host.
//-----

inline void SCIA_Init()

```

```

{
    // Enable the SCI-A clocks
    EALLOW;
    SysCtrlRegs.PCLKCR0.bit.SCIAENCLK=1;
    SysCtrlRegs.LOSPCP.all = 0x0002;
    SciaRegs.SCIFFTX.all=0x8000;
    // 1 stop bit, No parity, 8-bit character
    // No loopback
    SciaRegs.SCICCR.all = 0x0007;
    // Enable TX, RX, Use internal SCICLK
    SciaRegs.SCICTL1.all = 0x0003;
    // Disable RxErr, Sleep, TX Wake,
    // Disable Rx Interrupt, Tx Interrupt
    SciaRegs.SCICTL2.all = 0x0000;
    // Relinquish SCI-A from reset
    SciaRegs.SCICTL1.all = 0x0023;
    // Enable pull-ups on SCI-A pins
    // GpioCtrlRegs.GPAPUD.bit.GPIO28 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO29 = 0;
    GpioCtrlRegs.GPAPUD.all &= 0xCFFFFFFF;
    // Enable the SCI-A pins
    // GpioCtrlRegs.GPAMUX2.bit.GPIO28 = 1;
    // GpioCtrlRegs.GPAMUX2.bit.GPIO29 = 1;
    GpioCtrlRegs.GPAMUX2.all |= 0x05000000;
    // Input qual for SCI-A RX is asynch
    GpioCtrlRegs.GPAQSEL2.bit.GPIO28 = 3;
    EDIS;
    return;
}

//#####
// void SCIA_AutobaudLock(void)
//-----
// Perform autobaud lock with the host.
// Note that if autobaud never occurs
// the program will hang in this routine as there
// is no timeout mechanism included.
//-----

inline void SCIA_AutobaudLock()
{
    Uint16 byteData;

    // Must prime baud register with >= 1
    SciaRegs.SCILBAUD = 1;
    // Prepare for autobaud detection
    // Set the CDC bit to enable autobaud detection
    // and clear the ABD bit
    SciaRegs.SCIFFCT.bit.CDC = 1;
    SciaRegs.SCIFFCT.bit.ABDCLR = 1;
    // Wait until we correctly read an
    // 'A' or 'a' and lock
    while(SciaRegs.SCIFFCT.bit.ABD != 1) {
        // After autobaud lock, clear the ABD and CDC bits
        SciaRegs.SCIFFCT.bit.ABDCLR = 1;
        SciaRegs.SCIFFCT.bit.CDC = 0;
        while(SciaRegs.SCIRXST.bit.RXRDY != 1) { }
        byteData = SciaRegs.SCIRXBUF.bit.RXDT;
        SciaRegs.SCITXBUF = byteData;
    }

    return;
}

//#####
// Uint16 SCIA_GetWordData(void)
//-----
// This routine fetches two bytes from the SCI-A
// port and puts them together to form a single
// 16-bit value. It is assumed that the host is
// sending the data in the order LSB followed by MSB.
//-----

```

```
Uint16 SCIA_GetWordData()
{
    Uint16 wordData;
    Uint16 byteData;

    wordData = 0x0000;
    byteData = 0x0000;

    // Fetch the LSB and verify back to the host
    while(SciaRegs.SCIRXST.bit.RXRDY != 1) { }
    wordData = (Uint16)SciaRegs.SCIRXBUF.bit.RXDT;
    SciaRegs.SCITXBUF = wordData;

    // Fetch the MSB and verify back to the host
    while(SciaRegs.SCIRXST.bit.RXRDY != 1) { }
    byteData = (Uint16)SciaRegs.SCIRXBUF.bit.RXDT;
    SciaRegs.SCITXBUF = byteData;

    // form the wordData from the MSB:LSB
    wordData |= (byteData << 8);

    return wordData;
}

// EOF-----
```

```

// TI File $Revision: /main/3 $
// Checkin $Date: May 8, 2007 12:28:30 $
//#####
//
// FILE:    Parallel_Boot.c
//
// TITLE:   2833x Parallel Port I/O boot routines
//
// Functions:
//
//     Uint32 Parallel_Boot(void)
//     inline void Parallel_GPIOSelect(void)
//     inline Uint16 Parallel_CheckKeyVal(void)
//     Uint16 Parallel_GetWordData_8bit()
//     Uint16 Parallel_GetWordData_16bit()
//     void Parallel_WaitHostRdy(void)
//     void Parallel_HostHandshake(void)
// Notes:
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"
#include "TMS320x2833x_Boot.h"

// Private function definitions
inline void Parallel_GPIOSelect(void);
inline Uint16 Parallel_CheckKeyVal(void);
Uint16 Parallel_GetWordData_8bit(void);
Uint16 Parallel_GetWordData_16bit(void);
void Parallel_WaitHostRdy(void);
void Parallel_HostHandshake(void);

// External function definitions
extern void CopyData(void);
extern Uint32 GetLongData(void);
extern void ReadReservedFn(void);

#define HOST_CTRL        GPIO27 // GPIO27 is the host control signal
#define DSP_CTRL         GPIO26 // GPIO26 is the DSP's control signal

#define HOST_DATA_NOT_RDY GpioDataRegs.GPADAT.bit.HOST_CTRL!=0
#define WAIT_HOST_ACK    GpioDataRegs.GPADAT.bit.HOST_CTRL!=1

// Set (DSP_ACK) or Clear (DSP_RDY) GPIO 17
#define DSP_ACK          GpioDataRegs.GPASET.bit.DSP_CTRL = 1;
#define DSP_RDY          GpioDataRegs.GPACLEAR.bit.DSP_CTRL = 1;
#define DATA            GpioDataRegs.GPADAT.all

//#####
// Uint32 Parallel_Boot(void)
//-----
// This module is the main Parallel boot routine.
// It will load code via GP I/O port B.
//
// This boot mode accepts 8-bit or 16-bit data.
// 8-bit data is expected to be the order LSB
// followed by MSB.
//
// This function returns a entry point address back
// to the InitBoot routine which in turn calls
// the ExitBoot routine.
//-----

Uint32 Parallel_Boot()
{
    Uint32 EntryAddr;
    // Setup for Parallel boot
    Parallel_GPIOSelect();

```

```

    // Check for the key value. Based on this the data will
    // be read as 8-bit or 16-bit values.
    if (Parallel_CheckKeyVal() == ERROR) return FLASH_ENTRY_POINT;
    // Read and discard the reserved words
    ReadReservedFn();
    // Get the entry point address
    EntryAddr = GetLongData();
    // Load the data
    CopyData();

    return EntryAddr;
}

//#####
// void Parallel_GPIOSelect(void)
//-----
// Enable I/O pins for input GPIO 15:0. Also
// enable the control pins for HOST_CTRL and
// DSP_CTRL.
//-----

inline void Parallel_GPIOSelect()
{
    EALLOW;
    // Enable pull-ups for GPIO Port A 15:0
    // GPIO Port 15:0 are all I/O pins
    // and DSP_CTRL/HOST_CTRL
    // GpioCtrlRegs.GPAPUD.bit.GPIO15 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO14 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO13 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO12 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO11 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO10 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO9 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO8 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO7 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO6 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO5 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO4 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO3 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO2 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO1 = 0;
    // GpioCtrlRegs.GPAPUD.bit.GPIO0 = 0;
    // GpioCtrlRegs.GPAPUD.bit.DSP_CTRL = 0;
    // GpioCtrlRegs.GPAPUD.bit.HOST_CTRL = 0;
    GpioCtrlRegs.GPAPUD.all &= 0xF3FF0000;

    // 0 = I/O pin 1 = Peripheral pin
    GpioCtrlRegs.GPAMUX1.all = 0x0000;
    GpioCtrlRegs.GPAMUX2.bit.DSP_CTRL = 0;
    GpioCtrlRegs.GPAMUX2.bit.HOST_CTRL = 0;

    // HOST_CTRL is an input control
    // from the Host
    // to the DSP Ack/Rdy
    // DSP_CTRL is an output from the DSP Ack/Rdy
    // 0 = input 1 = output
    GpioCtrlRegs.GPADIR.bit.DSP_CTRL = 1;
    GpioCtrlRegs.GPADIR.bit.HOST_CTRL = 0;

    EDIS;
}

//#####
// void Parallel_CheckKeyVal(void)
//-----
// Determine if the data we are loading is in
// 8-bit or 16-bit format.
// If neither, return an error.
//
// Note that if the host never responds then
// the code will be stuck here. That is there

```

```

// is no timeout mechanism.
//-----
inline Uint16 Parallel_CheckKeyVal()
{
    Uint16 wordData;

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

    wordData = Parallel_GetWordData_16bit();

    if(wordData == SIXTEEN_BIT_HEADER)
    {
        // Assign GetWordData to the parallel 16bit version of the
        // function. GetWordData is a pointer to a function.
        GetWordData = Parallel_GetWordData_16bit();
        return SIXTEEN_BIT;
    }
    // If not 16-bit mode, check for 8-bit mode
    // Call Parallel_GetWordData with 16-bit mode
    // so we only fetch the MSB of the KeyValue and not
    // two bytes. We will ignore the upper 8-bits and combine
    // the result with the previous byte to form the
    // header KeyValue.

    wordData = wordData & 0x00FF;
    wordData |= Parallel_GetWordData_16bit() << 8;
    if(wordData == EIGHT_BIT_HEADER)
    {
        // Assign GetWordData to the parallel 8bit version of the
        // function. GetWordData is a pointer to a function.
        GetWordData = Parallel_GetWordData_8bit();
        return EIGHT_BIT;
    }
    // Didn't find a 16-bit or an 8-bit KeyVal header so return an error.
    else return ERROR;
}

//#####
// Uint16 Parallel_GetWordData_16bit()
// Uint16 Parallel_GetWordData_8bit()
//-----
// This routine fetches a 16-bit word from the
// GP I/O port. The 16bit function is used if the
// input 16-bits and the function fetches a
// single word and returns it to the host.
//
// The _8bit function is used if the input stream is
// an 8-bit input stream and the upper 8-bits of the
// GP I/O port are ignored. In the 8-bit case the
// first fetches the LSB and then the MSB from the
// GPIO port. These two bytes are then put together to
// form a single 16-bit word that is then passed back
// to the host. Note that in this case, the input stream
// from the host is in the order LSB followed by MSB
//-----
Uint16 Parallel_GetWordData_8bit()
{
    Uint16 wordData;

    // Get LSB.

    Parallel_WaitHostRdy();
    wordData = DATA;
    Parallel_HostHandshake();

    // Fetch the MSB.

    wordData = wordData & 0x00FF;
    Parallel_WaitHostRdy();
    wordData |= (DATA << 8);
    Parallel_HostHandshake();
}

```

```

    return wordData;
}

Uint16 Parallel_GetWordData_16bit()
{
    Uint16 wordData;

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

    Parallel_WaitHostRdy();
    wordData = DATA;
    Parallel_HostHandshake();
    return wordData;
}

//#####
// void Parallel_WaitHostRdy(void)
//-----
// This routine tells the host that the DSP is ready to
// receive data.  The DSP then waits for the host to
// signal that data is ready on the GP I/O port.e
//-----
void Parallel_WaitHostRdy()
{
    DSP_RDY;
    while(HOST_DATA_NOT_RDY) { }
}

//#####
// void Parallel_HostHandshake(void)
//-----
// This routine tells the host that the DSP has received
// the data.  The DSP then waits for the host to acknowledge
// the receipt before continuing.
//-----
void Parallel_HostHandshake()
{
    DSP_ACK;
    while(WAIT_HOST_ACK) { }
}
// EOF -----

```

```

// TI File $Revision: /main/5 $
// Checkin $Date: May 8, 2008 09:07:49 $
//#####
//
// FILE:    I2C_Boot.c
//
// TITLE:   2833x I2C Boot mode routines
//
// Functions:
//
//     Uint32 I2C_Boot(void)
//     inline void I2C_Init(void)
//     inline Uint16 I2C_CheckKeyVal(void)
//     inline void I2C_ReservedFn(void)
//     Uint16 I2C_GetWord(void)
//
// Notes:
//     The I2C code contained here is specifically steamlined for the
//     bootloader. It can be used to load code via the I2C port into the
//     RAM and jump to an entry point within that code.
//
// Features/Limitations:
//     - The I2C boot loader code is written to communicate with an EEPROM
//     device at address 0x50. The EEPROM must adhere to conventional I2C
//     EEPROM protocol (see the boot rom documentation) with a 16-bit
//     base address architecture (as opposed to 8-bits). The base address
//     of the code should be contained at address 0x0000 in the EEPROM.
//     - The input frequency to the device must be between 28Mhz and
//     48Mhz, creating a 7Mhz to 12Mhz system clock. This is due to a
//     requirement that the I2C clock be between 7Mhz and 12Mhz to meet all
//     of the I2C specification timing requirements. The I2CPSC default value
//     is hardcoded to 0 so that the I2C clock will not be divided down from
//     the system clock. The I2CPSC value can be modified after receiving
//     the first few bytes from the EEPROM (see the boot rom documentation),
//     but it is advisable not to, as this can cause the I2C to operate out
//     of specification with a system clock between 7Mhz and 12Mhz.
//     - The bit period prescalers (I2CCLKH and I2CCLKL) are configured to
//     run the I2C at 50% duty cycle at 100kHz bit rate (standard I2C mode)
//     when the system clock is 12Mhz. These registers can be modified after
//     receiving the first few bytes from the EEPROM (see the boot rom
//     documentation). This allows the communication to be increased up to
//     a 400kHz 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 C28x application software
//     signals that it is past the bootloader portion of initialization.
//     - The non-acknowledgment bit is only checked during the first message
//     sent to initialize the EEPROM base address. This ensures that an
//     EEPROM is present at address 0x50 before continuing on. If an EEPROM
//     is not present, code will jump to the Flash entry point. The
//     non-acknowledgment bit is not checked during the address phase of
//     the data read messages (I2C_GetWord). If a non-acknowledge is
//     received during the data read messages, the I2C bus will hang.
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"
#include "TMS320x2833x_Boot.h"

// Private functions
inline void I2C_Init(void);
inline Uint16 I2C_CheckKeyVal(void);
inline void I2C_ReservedFn(void);
    Uint16 I2C_GetWord(void);

// External functions
extern void CopyData(void);

```

```

extern Uint32 GetLongData(void);

//#####
// Uint32 I2C_Boot(void)
//-----
// This module is the main I2C boot routine.
// It will load code via the I2C-A port.
//
// It will return an entry point address back
// to the ExitBoot routine.
//-----

Uint32 I2C_Boot(void)
{
    Uint32 EntryAddr;

    // Assign GetWordData to the I2C-A version of the
    // function. GetWordData is a pointer to a function.
    GetWordData = I2C_GetWord;

    // Init I2C pins, clock, and registers
    I2C_Init();

    // Check for 0x08AA data header, else go to flash
    if (I2C_CheckKeyVal() == ERROR) { return FLASH_ENTRY_POINT; }

    // Check for clock and prescaler speed changes and reserved words
    I2C_ReservedFn();

    // Get point of entry address after load
    EntryAddr = GetLongData();

    // Receive and copy one or more code sections to destination addresses
    CopyData();

    return EntryAddr;
}

//#####
// void I2C_Init(void)
//-----
// Initialize the I2C-A port for communications
// with the host.
//-----

inline void I2C_Init(void)
{
    // Configure I2C pins and turn on I2C clock
    EALLOW;
    GpioCtrlRegs.GPBMUX1.bit.GPIO32 = 1;    // Configure as SDA pin
    GpioCtrlRegs.GPBMUX1.bit.GPIO33 = 1;    // Configure as SCL pin
    GpioCtrlRegs.GBPUD.bit.GPIO32 = 0;     // Turn SDA pullup on
    GpioCtrlRegs.GBPUD.bit.GPIO33 = 0;     // Turn SCL pullup on
    GpioCtrlRegs.GPBQSEL1.bit.GPIO32 = 3;  // Asynch
    GpioCtrlRegs.GPBQSEL1.bit.GPIO33 = 3;  // Asynch
    SysCtrlRegs.PCLKCR0.bit.I2CAENCLK = 1; // Turn I2C module clock on
    EDIS;

    // Initialize I2C in master transmitter mode
    I2caRegs.I2CSAR = 0x0050;    // Slave address - EEPROM control code
    I2caRegs.I2CPSC.all = 1;    // I2C clock should be between 7Mhz-12Mhz: valid for CLKIN=28-
48Mhz
    I2caRegs.I2CCLKL = 54;      // Prescalers set for 100kHz bit rate
    I2caRegs.I2CCLKH = 54;      // at a 12Mhz I2C clock

    I2caRegs.I2CMDR.all = 0x0620; // Master transmitter
    // Take I2C out of reset
    // Stop when suspended

    I2caRegs.I2CFFTX.all = 0x6000; // Enable FIFO mode and TXFIFO
    I2caRegs.I2CFFRX.all = 0x2000; // Enable RXFIFO

```

```

    return;
}

//#####
// Uint16 I2C_CheckKeyVal(void)
//-----
// This routine sets up the starting address in the
// EEPROM by writing two bytes (0x0000) via the
// I2C-A port to slave address 0x50. Without
// sending a stop bit, the communication is then
// restarted and two bytes are read from the EEPROM.
// If these two bytes read do not equal 0x08AA
// (little endian), an error is returned.
//-----

inline Uint16 I2C_CheckKeyVal(void)
{
    // To read a word from the EEPROM, an address must be given first in
    // master transmitter mode. Then a restart is performed and data can
    // be read back in master receiver mode.
    I2caRegs.I2CCNT = 0x02;           // Setup how many bytes to send
    I2caRegs.I2CDXR = 0x00;          // Configure fifo data for byte
    I2caRegs.I2CDXR = 0x00;          // address of 0x0000

    I2caRegs.I2CMDR.all = 0x2620;    // Send data to setup EEPROM address

    while (I2caRegs.I2CSTR.bit.ARDY == 0) // Wait until communication
    {                                     // complete and registers ready
    }

    if (I2caRegs.I2CSTR.bit.NACK == 1) // Set stop bit & return error if
    {                                     // NACK received
        I2caRegs.I2CMDR.bit.STP = 1;
        return ERROR;
    }

    // Check to make sure key value received is correct
    if (I2C_GetWord() != 0x08AA) {return ERROR;}

    return NO_ERROR;
}

//#####
// void I2C_ReservedFn(void)
//-----
// This function reads 8 reserved words in the header.
// 1st word - parameters for I2CPSC register
// 2nd word - parameters for I2CCLKH register
// 3rd word - parameters for I2CCLKL register
//
// The remaining reserved words are read and discarded
// and then program execution returns to the main routine.
//-----

inline void I2C_ReservedFn(void)
{
    Uint16 I2CPrescaler;
    Uint16 I2cClkHData;
    Uint16 I2cClkLData;
    Uint16 i;

    // Get I2CPSC, I2CCLKH, and I2CCLKL values
    I2CPrescaler = I2C_GetWord();
    I2cClkHData = I2C_GetWord();
    I2cClkLData = I2C_GetWord();

    // Store I2C clock prescalers
    I2caRegs.I2CMDR.bit.IRS = 0;
    I2caRegs.I2CCLKL = I2cClkLData;
    I2caRegs.I2CCLKH = I2cClkHData;
    I2caRegs.I2CPSC.all = I2CPrescaler;
    I2caRegs.I2CMDR.bit.IRS = 1;
}

```

```

    // Read and discard the next 5 reserved words
    for (i=1; i<=5; i++)
    {
        I2cClkHData = I2C_GetWord();
    }

    return;
}

//#####
// Uint16 I2C_GetWord(void)
//-----
// This routine fetches two bytes from the I2C-A
// port and puts them together little endian style
// to form a single 16-bit value.
//-----

Uint16 I2C_GetWord(void)
{
    Uint16 LowByte;

    I2caRegs.I2CCNT = 2;           // Setup how many bytes to expect
    I2caRegs.I2CMDR.all = 0x2C20; // Send start as master receiver

    // Wait until communication done
    while (I2caRegs.I2CMDR.bit.STP == 1) {}

    // Combine two bytes to one word & return
    LowByte = I2caRegs.I2CDRR;
    return (LowByte | (I2caRegs.I2CDRR<<8));
}

//=====
// No more.
//=====

```

```

// TI File $Revision: /main/6 $
// Checkin $Date: February 4, 2008 16:39:39 $
//#####
//
// FILE:    CAN_Boot.c
//
// TITLE:   2833x CAN Boot mode routines
//
// Functions:
//
//     Uint32 CAN_Boot(void)
//     void CAN_Init(void)
//     Uint32 CAN_GetWordData(void)
//
// Notes:
// BRP = 1, Bit time = 15. This would yield the following bit rates with the
// default PLL setting:
//
// XCLKIN = 30 MHz  SYSCLKOUT = 15 MHz  CAN module clock = 7.5 MHz Bit rate = 500 kbits/s
// XCLKIN = 15 MHz  SYSCLKOUT = 7.5 MHz  CAN module clock = 3.75 MHz  Bit rate = 250 kbits/s
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"
#include "TMS320x2833x_Boot.h"

// Private functions
void CAN_Init(void);
Uint16 CAN_GetWordData(void);

// External functions
extern void CopyData(void);
extern Uint32 GetLongData(void);
extern void ReadReservedFn(void);

//#####
// Uint32 CAN_Boot(void)
//-----
// This module is the main CAN boot routine.
// It will load code via the CAN-A port.
//
// It will return a entry point address back
// to the InitBoot routine which in turn calls
// the ExitBoot routine.
//-----

Uint32 CAN_Boot()
{
    Uint32 EntryAddr;

    // If the missing clock detect bit is set, just
    // loop here.
    if(SysCtrlRegs.PLLSTS.bit.MCLKSTS == 1)
    {
        for(;;);
    }

    // Assign GetWordData to the CAN-A version of the
    // function.  GetWordData is a pointer to a function.
    GetWordData = CAN_GetWordData;

    CAN_Init();

    // If the KeyValue was invalid, abort the load
    // and return the flash entry point.
    if (CAN_GetWordData() != 0x08AA) return FLASH_ENTRY_POINT;

    ReadReservedFn();

    EntryAddr = GetLongData();
}

```

```

    CopyData();

    return EntryAddr;
}

//#####
// void CAN_Init(void)
//-----
// Initialize the CAN-A port for communications
// with the host.
//-----

void CAN_Init()
{
    /* Create a shadow register structure for the CAN control registers. This is
    needed, since only 32-bit access is allowed to these registers. 16-bit access
    to these registers could potentially corrupt the register contents or return
    false data. This is especially true while writing to/reading from a bit
    (or group of bits) among bits 16 - 31 */

    struct ECAN_REGS ECanaShadow;

    EALLOW;

    /* Enable CAN clock */

    SysCtrlRegs.PCLKCR0.bit.ECANAENCLK=1;

    /* Configure eCAN-A pins using GPIO regs*/

    GpioCtrlRegs.GPAMUX2.bit.GPIO30 = 1; // GPIO30 is CANRXA
    GpioCtrlRegs.GPAMUX2.bit.GPIO31 = 1; // GPIO31 is CANTXA

    /* Enable internal pullups for the CAN pins */

    GpioCtrlRegs.GPAPUD.bit.GPIO30 = 0;
    GpioCtrlRegs.GPAPUD.bit.GPIO31 = 0;

    /* Asynch Qual */

    GpioCtrlRegs.GPAQSEL2.bit.GPIO30 = 3;

    /* Configure eCAN RX and TX pins for CAN operation using eCAN regs*/

    ECanaShadow.CANTIIOC.all = ECanaRegs.CANTIIOC.all;
    ECanaShadow.CANTIIOC.bit.TXFUNC = 1;
    ECanaRegs.CANTIIOC.all = ECanaShadow.CANTIIOC.all;

    ECanaShadow.CANRIOCI.all = ECanaRegs.CANRIOCI.all;
    ECanaShadow.CANRIOCI.bit.RXFUNC = 1;
    ECanaRegs.CANRIOCI.all = ECanaShadow.CANRIOCI.all;

    /* Initialize all bits of 'Message Control Register' to zero */
    // Some bits of MSGCTRL register come up in an unknown state. For proper operation,
    // all bits (including reserved bits) of MSGCTRL must be initialized to zero

    ECanaMboxes.MBOX1.MSGCTRL.all = 0x00000000;

    /* Clear all RMPn, GIFn bits */
    // RMPn, GIFn bits are zero upon reset and are cleared again as a precaution.

    ECanaRegs.CANRMP.all = 0xFFFFFFFF;

    /* Clear all interrupt flag bits */

    ECanaRegs.CANGIF0.all = 0xFFFFFFFF;
    ECanaRegs.CANGIF1.all = 0xFFFFFFFF;

    /* Configure bit timing parameters for eCAN*/

    ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;

```

```

ECanaShadow.CANMC.bit.CCR = 1 ;          // Set CCR = 1
ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;

ECanaShadow.CANES.all = ECanaRegs.CANES.all;

do
{
    ECanaShadow.CANES.all = ECanaRegs.CANES.all;
} while(ECanaShadow.CANES.bit.CCE != 1 );    // Wait for CCE bit to be set..

ECanaShadow.CANBTC.all = 0;

ECanaShadow.CANBTC.bit.BRPREG = 0;
ECanaShadow.CANBTC.bit.TSEG2REG = 2;
ECanaShadow.CANBTC.bit.TSEG1REG = 10;

ECanaRegs.CANBTC.all = ECanaShadow.CANBTC.all;

ECanaShadow.CANMC.all = ECanaRegs.CANMC.all;
ECanaShadow.CANMC.bit.CCR = 0 ;          // Set CCR = 0
ECanaRegs.CANMC.all = ECanaShadow.CANMC.all;

ECanaShadow.CANES.all = ECanaRegs.CANES.all;

do
{
    ECanaShadow.CANES.all = ECanaRegs.CANES.all;
} while(ECanaShadow.CANES.bit.CCE != 0 );    // Wait for CCE bit to be cleared..

/* Disable all Mailboxes */

ECanaRegs.CANME.all = 0;    // Required before writing the MSGIDs

/* Assign MSGID to MBOX1 */

ECanaMboxes.MBOX1.MSGID.all = 0x00040000;    // Standard ID of 1, Acceptance mask disabled

/* Configure MBOX1 to be a receive MBOX */

ECanaRegs.CANMD.all = 0x0002;

/* Enable MBOX1 */

ECanaRegs.CANME.all = 0x0002;

EDIS;

return;
}

//#####
// Uint16 CAN_GetWordData(void)
//-----
// This routine fetches two bytes from the CAN-A
// port and puts them together to form a single
// 16-bit value. It is assumed that the host is
// sending the data in the order LSB followed by MSB.
//-----

Uint16 CAN_GetWordData()
{
    Uint16 wordData;
    Uint16 byteData;

    wordData = 0x0000;
    byteData = 0x0000;

    // Fetch the LSB
    while(ECanaRegs.CANRMP.all == 0) { }
    wordData = (Uint16) ECanaMboxes.MBOX1.MDL.byte.BYTE0;    // LS byte

    // Fetch the MSB

```

```

byteData = (Uint16)ECanaMboxes.MBOX1.MDL.byte.BYTE1;    // MS byte

// form the wordData from the MSB:LSB
wordData |= (byteData << 8);

/* Clear all RMPn bits */

ECanaRegs.CANRMP.all = 0xFFFFFFFF;

return wordData;
}

/*
Data frames with a Standard MSGID of 0x1 should be transmitted to the ECAN-A bootloader.
This data will be received in Mailbox1, whose MSGID is 0x1. No message filtering is employed.

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. Following is the
order in which data should be transmitted:
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 - MS part of 32-bit address (aabb)
dd cc - 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) - Entry-point address of first section = 0xeeffgghh
xx xx - First word of first section
xx xx - Second word.....
...
...
...
xxx - 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) - Entry-point address of second section = 0xeeffgghh
xx xx - First word of second section
xx xx - Second word.....
...
...
...
xxx - Last word of second section
(more sections, if need be)
00 00 - Section length of zero for next section indicates end of data.
*/

/*
Notes:
-----

Rev history:
-----
Changes from rev 0 to Rev A:-
1. SAM is left at its default value of 0.

*/
// EOF-----

```

```

// TI File $Revision: /main/8 $
// Checkin $Date: January 10, 2008 11:08:20 $
//#####
//
// FILE: XINTF_Boot.c
//
// TITLE: 2833x XINTF boot routine
//
// Functions:
//
// Uint32 XINTF_Boot(Uint16 size)
//
// Notes:
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"
#include "TMS320x2833x_Boot.h"

// Private function definitions

// External function definitions

//#####
// Uint32 XINTF_Boot(Uint16 size)
//-----
// This module is the main Parallel boot routine.
// It will load code via GP I/O port B.
//
// This function configures XINTF zone 6 and returns
// the first address in XINTF zone 6 to the InitBoot
// routine. InitBoot then calls ExitBoot
//-----
Uint32 XINTF_Boot(Uint16 size)
{
    EALLOW;

    SysCtrlRegs.PCLKCR3.bit.XINTFENCLK = 1;

    // GPIO64-GPIO79 (XD0-XD15)
    GpioCtrlRegs.GPCMUX1.all = 0xAAAAAAAA;

    // GPIO80-GPIO87 (XA8-XA15)
    // (top half of this register is reserved)
    GpioCtrlRegs.GPCMUX2.all = 0x0000AAAA;

    // XZCS6n, XA17-XA19
    GpioCtrlRegs.GPAMUX2.all |= 0xFF000000;

    // XREADYn, XRNW, XWE0n, XA16, XA0-XA7
    GpioCtrlRegs.GPBMUX1.all |= 0xFFFFF0F0;

    if (size == 16)
    {
        XintfRegs.XTIMING6.all = XTIMING_X16_VAL;
    }
    else if (size == 32)
    {
        // GPIO48-GPIO63 (XD16-XD31)
        GpioCtrlRegs.GPBMUX2.all = 0xFFFFFFFF; // XINTF functionality
        GpioCtrlRegs.GPBQSEL2.all = 0xFFFFFFFF; // Asynchronous inputs
        XintfRegs.XTIMING6.all = XTIMING_X32_VAL;
    }
    XintfRegs.XINTCNF2.all = XINTCNF2_VAL;
}

```

```
    EDIS;  
    return XINTF_ENTRY_POINT;  
  
}
```

// EOF -----

```

// TI File $Revision: /main/7 $
// Checkin $Date: January 10, 2008 11:08:11 $
//#####
//
// FILE:      XINTF_Parallel_Boot.c
//
// TITLE:     2833x XINTF Parallel Port I/O boot routines
//
// Functions:
//
//      Uint32 XINTF_Parallel_Boot(void)
//      inline void XINTF_Parallel_GPIOSelect(void)
//      inline Uint16 XINTF_Parallel_CheckKeyVal(void)
//      Uint16 XINTF_Parallel_GetWordData_8bit()
//      Uint16 XINTF_Parallel_GetWordData_16bit()
//      void XINTF_Parallel_WaitHostRdy(void)
//      void XINTF_Parallel_HostHandshake(void)
// Notes:
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"
#include "TMS320x2833x_Boot.h"

// Private function definitions
inline void XINTF_Parallel_GPIOSelect(void);
inline Uint16 XINTF_Parallel_CheckKeyVal(void);
Uint16 XINTF_Parallel_GetWordData_8bit(void);
Uint16 XINTF_Parallel_GetWordData_16bit(void);
void XINTF_Parallel_WaitHostRdy(void);
void XINTF_Parallel_HostHandshake(void);
void XINTF_Parallel_ReservedFn(void);

// External function definitions
extern void CopyData(void);
extern Uint32 GetLongData(void);
extern void InitPll(Uint16 val, Uint16 divsel);

#define HOST_CTRL      GPIO13 // GPIO13 is the host control signal
#define DSP_CTRL      GPIO12 // GPIO12 is the DSP's control signal

#define HOST_DATA_NOT_RDY  GpioDataRegs.GPADAT.bit.HOST_CTRL!=0
#define WAIT_HOST_ACK    GpioDataRegs.GPADAT.bit.HOST_CTRL!=1

// Set (DSP_ACK) or Clear (DSP_RDY) GPIO 12
#define DSP_ACK        GpioDataRegs.GPASET.bit.DSP_CTRL = 1;
#define DSP_RDY        GpioDataRegs.GPACLEAR.bit.DSP_CTRL = 1;
#define DATA          (Uint16 *)0x100000

//#####
// Uint32 XINTF_Parallel_Boot(void)
//-----
// This module is the main XINTF Parallel boot routine.
// It will load code via XD[15:0].
//
// This boot mode accepts 8-bit or 16-bit data.
// 8-bit data is expected to be the order LSB
// followed by MSB.
//
// This function returns a entry point address back
// to the InitBoot routine which in turn calls
// the ExitBoot routine.
//-----

Uint32 XINTF_Parallel_Boot()
{
    Uint32 EntryAddr;
    // Setup for Parallel boot

```

```

XINTF_Parallel_GPIOSelect();
// Check for the key value. Based on this the data will
// be read as 8-bit or 16-bit values.
if (XINTF_Parallel_CheckKeyVal() == ERROR) return FLASH_ENTRY_POINT;
// Read and discard the reserved words
XINTF_Parallel_ReservedFn();
// Get the entry point address
EntryAddr = GetLongData();
// Load the data
CopyData();

return EntryAddr;

}

//#####
// void Parallel_GPIOSelect(void)
//-----
// Enable I/O pins for input GPIO 15:0. Also
// enable the control pins for HOST_CTRL and
// DSP_CTRL.
//-----

inline void XINTF_Parallel_GPIOSelect()
{
    EALLOW;

    // enable clock to XINTF module
    SysCtrlRegs.PCLKCR3.bit.XINTFENCLK = 1;

    // GPIO64-GPIO79 (XD0-XD15)
    GpioCtrlRegs.GPCMUX1.all = 0xAAAAAAAA;

    // GPIO80-GPIO87 (XA8-XA15)
    // (top half of this register is reserved)
    GpioCtrlRegs.GPCMUX2.all = 0x0000AAAA;

    // XZCS6n, XA17-XA19
    GpioCtrlRegs.GPAMUX2.all |= 0xFF000000;

    // XREADYn, XRNW, XWE0n, XA16, XA0-XA7
    GpioCtrlRegs.GPBMUX1.all |= 0xFFFFF0F0;

    // Use the default XINTF timing
    XintfRegs.XTIMING6.all = XTIMING_X16_VAL;
    XintfRegs.XINTCNF2.all = XINTCNF2_VAL;

    GpioCtrlRegs.GPAMUX1.bit.DSP_CTRL = 0; // GPIO
    GpioCtrlRegs.GPAPUD.bit.DSP_CTRL = 0; // Pullup enabled
    GpioDataRegs.GPASET.bit.DSP_CTRL = 1; // Set the pin high to start
    GpioCtrlRegs.GPADIR.bit.DSP_CTRL = 1; // Output

    GpioCtrlRegs.GPAMUX1.bit.HOST_CTRL = 0; // GPIO
    GpioCtrlRegs.GPAPUD.bit.HOST_CTRL = 0; // Pullup enabled
    GpioCtrlRegs.GPADIR.bit.HOST_CTRL = 0; // Input

    EDIS;
}

//#####
// void XINTF_Parallel_CheckKeyVal(void)
//-----
// Determine if the data we are loading is in
// 8-bit or 16-bit format.
// If neither, return an error.
//
// Note that if the host never responds then
// the code will be stuck here. That is there
// is no timeout mechanism.
//-----
inline Uint16 XINTF_Parallel_CheckKeyVal()
{
    Uint16 wordData;

```

```

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

wordData = XINTF_Parallel_GetWordData_16bit();

if(wordData == SIXTEEN_BIT_HEADER)
{
// Assign GetWordData to the parallel 16bit version of the
// function. GetWordData is a pointer to a function.
GetWordData = XINTF_Parallel_GetWordData_16bit;
return SIXTEEN_BIT;
}
// If not 16-bit mode, check for 8-bit mode
// Call Parallel_GetWordData with 16-bit mode
// so we only fetch the MSB of the KeyValue and not
// two bytes. We will ignore the upper 8-bits and combine
// the result with the previous byte to form the
// header KeyValue.

wordData = wordData & 0x00FF;
wordData |= XINTF_Parallel_GetWordData_16bit() << 8;
if(wordData == EIGHT_BIT_HEADER)
{
// Align GetWordData to the parallel 8bit version of the
// function. GetWordData is a pointer to a function.
GetWordData = XINTF_Parallel_GetWordData_8bit;
return EIGHT_BIT;
}
// Didn't find a 16-bit or an 8-bit KeyVal header so return an error.
else return ERROR;
}

//#####
// Uint16 XINTF_Parallel_GetWordData_16bit()
// Uint16 XINTF_Parallel_GetWordData_8bit()
//-----
// This routine fetches a 16-bit word from the
// first address in XINTF zone 6. The 16bit
// function is used if the input 16-bits and the
// function fetches a single word and returns it to the host.
//
// The _8bit function is used if the input stream is
// an 8-bit input stream and the upper 8-bits of the
// XD lines are ignored. In the 8-bit case the
// first fetches the LSB and then the MSB from the
// XINTF address. These two bytes are then put together to
// form a single 16-bit word that is then passed back
// to the host. Note that in this case, the input stream
// from the host is in the order LSB followed by MSB
//-----
Uint16 XINTF_Parallel_GetWordData_8bit()
{
    Uint16 wordData;

    // Get LSB.

    XINTF_Parallel_WaitHostRdy();
    wordData = *DATA;
    XINTF_Parallel_HostHandshake();

    // Fetch the MSB.

    wordData = wordData & 0x00FF;
    XINTF_Parallel_WaitHostRdy();
    wordData |= (*DATA << 8);
    XINTF_Parallel_HostHandshake();
    return wordData;
}

Uint16 XINTF_Parallel_GetWordData_16bit()
{

```

```

    Uint16 wordData;

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

    XINTF_Parallel_WaitHostRdy();
    wordData = *DATA;
    XINTF_Parallel_HostHandshake();
    return wordData;
}

//#####
// void XINTF_Parallel_WaitHostRdy(void)
//-----
// This routine tells the host that the DSP is ready to
// receive data.  The DSP then waits for the host to
// signal that data is ready on the GP I/O port.e
//-----
void XINTF_Parallel_WaitHostRdy()
{
    DSP_RDY;
    while(HOST_DATA_NOT_RDY) { }
}

//#####
// void XINTF_Parallel_HostHandshake(void)
//-----
// This routine tells the host that the DSP has received
// the data.  The DSP then waits for the host to acknowledge
// the receipt before continuing.
//-----
void XINTF_Parallel_HostHandshake()
{
    DSP_ACK;
    while(WAIT_HOST_ACK) { }
}

//#####
// void XINTF_Parallel_ReservedFn(void)
//-----
// This function reads 8 reserved words in the header.
// The first 6 words are as follows:
//
// PLLCR
// PLLSTS
// XTIMING6[31:16]
// XTIMING6[15:0]
// XINTCNF2[31:16]
// XINTCNF2[15:0]
//
// The remaining reserved words are read and discarded
// and then returns to the main routine.
//-----

void XINTF_Parallel_ReservedFn(void)
{
    Uint16 pllcr, divsel;
    Uint32 xtiming6, xintcnf2;
    Uint16 i;

    pllcr    = GetWordData(); // word 1
    divsel   = GetWordData(); // word 2
    xtiming6 = GetLongData(); // word 3 & 4
    xintcnf2 = GetLongData(); // word 5 & 6

    InitPll(pllcr,divsel);

    EALLOW;
    if (xtiming6 != 0) XintfRegs.XTIMING6.all = xtiming6;
    if (xintcnf2 != 0) XintfRegs.XINTCNF2.all = xintcnf2;
    EDIS;

    // Read and discard the next 2 reserved words.
    for(i = 1; i <= 2; i++)

```

```
    {  
        GetWordData();  
    }  
    return;  
}  
  
// EOF -----
```

```

// TI File $Revision: /main/10 $
// Checkin $Date: February 4, 2008 16:39:21 $
//#####
//
// FILE:      MCBSP_Boot.c
//
// TITLE:     2833x MCBSP-A Boot mode routines
//
// Functions:
//
//      Uint32 MCBSP_Boot()
//      void MCBSP_Init(void)
//      Uint32 MCBSP_GetWordData(void)
//
// Notes:
// XCLKIN = 30 MHz  SYSCLKOUT = 15 MHz  LSPCLK = 3.75 MHz
// XCLKIN = 15 MHz  SYSCLKOUT = 7.5 MHz  LSPCLK = 1.875 MHz
//
// Hardware connections:
//
//      Master          2833x Slave
//      -----
//      MDX      ->      MDRA
//      MDR      <-      MDXA
//      FSX      ->      FSRA
//      FSR      <-      FSXA
//      CLKX     ->      CLKXA (shorted to CLKRA*)
//      CLKR (internal*)
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####

#include "DSP2833x_Device.h"
#include "TMS320x2833x_Boot.h"

// Private functions
void MCBSP_Init(void);
Uint16 MCBSP_GetWordData(void);

// External functions
extern void CopyData(void);
extern Uint32 GetLongData(void);
extern void ReadReservedFn(void);

//#####
// Uint32 MCBSP_Boot(void)
//-----
// This module is the main MCBSP boot routine.
// It will load code via the MCBSP-A port.
//
// It will return a entry point address back
// to the InitBoot routine which in turn calls
// the ExitBoot routine.
//-----

Uint32 MCBSP_Boot()
{
    Uint32 EntryAddr;

    // If the missing clock detect bit is set, just
    // loop here.
    if(SysCtrlRegs.PLLSTS.bit.MCLKSTS == 1)
    {
        for(;;);
    }

    // Assign GetWordData to the MCBSP-A version of the
    // function. GetWordData is a pointer to a function.
    GetWordData = MCBSP_GetWordData;
}

```

```

MCBSP_Init();

// If the KeyValue was invalid, abort the load
// and return the flash entry point.
if (MCBSP_GetWordData() != 0x10AA) return FLASH_ENTRY_POINT;

ReadReservedFn();

EntryAddr = GetLongData();

CopyData();

return EntryAddr;
}

//#####
// void MCBSP_Init(void)
//-----
// Initialize the MCBSP-A port for communications
// with the host.
//-----

void MCBSP_Init()
{
    Uint16 k;

    EALLOW;

/* Enable MCBSP clock */

    SysCtrlRegs.PCLKCR0.bit.MCBSPAENCLK=1;

/* Configure MCBSP-A pins using GPIO regs*/

    GpioCtrlRegs.GPAMUX2.bit.GPIO20 = 2;        // GPIO20 is MDXA pin
    GpioCtrlRegs.GPAMUX2.bit.GPIO21 = 2;        // GPIO21 is MDRA pin
    GpioCtrlRegs.GPAMUX2.bit.GPIO22 = 2;        // GPIO22 is MCLKXA pin
    GpioCtrlRegs.GPAMUX2.bit.GPIO23 = 2;        // GPIO23 is MFSXA pin
    GpioCtrlRegs.GPAMUX1.bit.GPIO7 = 2;         // GPIO7 is MCLKRA pin
    GpioCtrlRegs.GPAMUX1.bit.GPIO5 = 2;         // GPIO5 is MFSRA pin

/* Enable internal pullups for the MCBSP pins */

    GpioCtrlRegs.GPAPUD.bit.GPIO20 = 0;
    GpioCtrlRegs.GPAPUD.bit.GPIO21 = 0;
    GpioCtrlRegs.GPAPUD.bit.GPIO22 = 0;
    GpioCtrlRegs.GPAPUD.bit.GPIO23 = 0;
    GpioCtrlRegs.GPAPUD.bit.GPIO7 = 0;
    GpioCtrlRegs.GPAPUD.bit.GPIO5 = 0;

/* Asynch Qual */

    GpioCtrlRegs.GPAQSEL2.bit.GPIO21 = 3;
    GpioCtrlRegs.GPAQSEL1.bit.GPIO7 = 3;
    GpioCtrlRegs.GPAQSEL1.bit.GPIO5 = 3;

// McBSP-A register settings

    McbspaRegs.SPCR2.all=0x0000;                // Reset FS generator, sample rate generator & transmitter
    McbspaRegs.SPCR1.all=0x0000;                // Reset Receiver, Right justify word

    McbspaRegs.MFFINT.all=0x0;                  // Disable all interrupts

    McbspaRegs.RCR2.all=0x0;                    // Single-phase frame, 1 word/frame, No companding
    (Receive)
    McbspaRegs.RCR1.all=0x0;

    McbspaRegs.XCR2.all=0x0;                    // Single-phase frame, 1 word/frame, No companding
    (Transmit)
    McbspaRegs.XCR1.all=0x0;

    McbspaRegs.RCR1.bit.RWDLEN1=2;              // 16-bit word
    McbspaRegs.XCR1.bit.XWDLEN1=2;              // 16-bit word

```

```

McbspaRegs.RCR2.bit.RDATDLY = 1; // 1-bit data delay on the receive side
McbspaRegs.XCR2.bit.XDATDLY = 1; // 1-bit data delay on the transmit side

McbspaRegs.SRGR2.bit.CLKSM = 1; // CLKSM=1 (If SCLKME=0, i/p clock to SRG is LSPCLK)
McbspaRegs.SRGR2.bit.FPER = 31; // FPER = 32 CLKG periods

McbspaRegs.SRGR1.bit.FWID = 1; // Frame Width = 1 CLKG period
McbspaRegs.SRGR1.bit.CLKGDV = 1; // CLKG frequency = LSPCLK/(CLKGDV+1) = LSPCLK/2

McbspaRegs.PCR.all = 0;
McbspaRegs.PCR.bit.FSXM = 1; // TX FSX to host

McbspaRegs.SPCR2.bit.XRST=1; // Release TX from Reset
McbspaRegs.SPCR1.bit.RRST=1; // Release RX from Reset
for(k=0; k<50000; k++){ // delay_loop()
McbspaRegs.SPCR2.all |=0x00C0; // Frame sync & sample rate generators pulled out of reset
for(k=0; k<50000; k++){ // delay_loop()

EDIS;

return;
}

//#####
// Uint16 MCBSP_GetWordData(void)
//-----
// This routine fetches the 16-bit word from MCBSP-A
//-----

Uint16 MCBSP_GetWordData()
{
    Uint16 wordData;
    wordData = 0x0000;

// Fetch word
while( McbspaRegs.SPCR1.bit.RRDY == 0){

wordData = (Uint16) McbspaRegs.DRR1.all;
McbspaRegs.DXR1.all = wordData;
asm (" nop");
return wordData;
}

/*
Revision history:
-----
Written for Rev A silicon. Received characters from host are now echoed by the 2833x device.
This may be used for handshaking.

*/

// EOF-----

```

```

;; TI File $Revision: /main/3 $
;; Checkin $Date: May 8, 2007 12:28:28 $
;#####
;;
;; FILE:      ITRAPIsr.asm
;;
;; TITLE:    2833x Boot Rom ITRAP ISR.
;;
;; Functions:
;;
;;      _ITRAPIsr
;;
;; Notes:
;;
;#####
;; $TI Release: 2833x/2823x Boot ROM V2 $
;; $Release Date: March 4, 2008 $
;#####

      .def _ITRAPIsr

;-----
; _ITRAPIsr
;-----
;-----
; This is the ITRAP interrupt service routine for
; the boot ROM CPU vector table. This routine
; would be called should an ITRAP be encountered
; before the PIE module was initialized and enabled.
;
; This module performs the following actions:
;
;      1) enables the watchdog
;      2) loops forever
;-----

      .sect ".Isr"

_ITRAPIsr:
      SETC OBJMODE      ;Set OBJMODE for 28x object code
      EALLOW           ;Enable EALLOW protected register access
      MOVZ DP, #7029h>>6 ;Set data page for WDCR register
      MOV @7029h, #0028h ;Clear WDDIS bit in WDCR to enable Watchdog
      EDIS             ;Disable EALLOW protected register access
      SB 0,UNC         ;Loop forever

;eof -----

```

```

/*
// TI File $Revision: /main/11 $
// Checkin $Date: March 4, 2008 15:51:17 $
//#####
//
// FILE:      F2833x_boot_rom_lnk.cmd
//
// TITLE:     F2833x boot rom linker command file
//
//
//#####
// $TI Release: 2833x/2823x Boot ROM V2 $
// $Release Date: March 4, 2008 $
//#####
*/

```

MEMORY

```

{
PAGE 0 :
    IQTABLES   : origin = 0x3FE000, length = 0x000b50
    IQTABLES2  : origin = 0x3FEB50, length = 0x00008c
    FPUTABLES  : origin = 0x3FEBDC, length = 0x0006A0
    TI_PRPG    : origin = 0x3FF27C, length = 0x000090
    TI_MISR    : origin = 0x3FF30C, length = 0x000040
    BOOT       : origin = 0x3FF34C, length = 0x0006AF
    RSVD1      : origin = 0x3FF9FB, length = 0x0005BE
    FLASH_API  : origin = 0x3FFFB9, length = 0x000001
    VERSION    : origin = 0x3FFFB9, length = 0x000002
    CHECKSUM   : origin = 0x3FFFCB, length = 0x000004
    VECS       : origin = 0x3FFFC0, length = 0x000040

    ADC_CAL    : origin = 0x380080, length = 0x000009

PAGE 1 :
    EBSS       : origin = 0x002, length = 0x004
    STACK      : origin = 0x006, length = 0x200
}

```

SECTIONS

```

{
    IQmathTables : load = IQTABLES, PAGE = 0
    IQmathTables2: load = IQTABLES2, PAGE = 0
    FPUMathTables: load = FPUTABLES, PAGE = 0
    .InitBoot    : load = BOOT, PAGE = 0
    .text        : load = BOOT, PAGE = 0
    .Isr         : load = BOOT, PAGE = 0
    .Flash       : load = FLASH_API PAGE = 0
    .BootVecs    : load = VECS, PAGE = 0
    .Checksum    : load = CHECKSUM, PAGE = 0
    .Version     : load = VERSION, PAGE = 0
    .stack       : load = STACK, PAGE = 1
    .ebss        : load = EBSS, PAGE = 1
    rsvd1        : load = RSVD1, PAGE = 0
    ti_prpg_sect: load = TI_PRPG, PAGE = 0
    ti_misr_sect: load = TI_MISR, PAGE = 0
    .adc_cal     : load = ADC_CAL, PAGE = 0, TYPE = NOLOAD
}

```


Revision History

A.1 Changes Made in Revision A

Technical changes made in this revision are shown in [Table A-1](#).

Table A-1. Additions, Deletions, and Changes

Location	Description
Global	Added the 2823x devices
Section 1.1	Corrected address range of boot ROM
Section 1.2	Added text to the On-Chip Boot ROM Math Tables section. Added two linker command examples and associated text.
Figure 2-3	Changed a location in the Jump to branch instruction in flash memory bullet following Figure 2-3 . Updated the 2nd note regarding the input divider.
Figure 2-4	Changed jump to location in Figure 2-4
Section 2.16	Changed McBSP_Boot Function section
Section 2.19	Updated the XINTF Parallel boot section.
Example 3-3	Changed the GPIO34TOG Data Stream example
Table 4-2	Changed the Bootloader Revision Per Device table
Section 4.2	Changed the Bootloader Code Revision History
Section 4.3	Changed the Bootloader Code Listing (V1.0) section

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

Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
RF/IF and ZigBee® Solutions	www.ti.com/lprf

Applications

Audio	www.ti.com/audio
Automotive	www.ti.com/automotive
Broadband	www.ti.com/broadband
Digital Control	www.ti.com/digitalcontrol
Medical	www.ti.com/medical
Military	www.ti.com/military
Optical Networking	www.ti.com/opticalnetwork
Security	www.ti.com/security
Telephony	www.ti.com/telephony
Video & Imaging	www.ti.com/video
Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2008, Texas Instruments Incorporated