KeyStone Architecture
DSP Bootloader

User Guide

Literature Number: SPRUGY5C
July 2013

Release History

Release	Date	Description/Comments
SPRUGY5C	August 2013	• Changed document title to include ***DSP***, corrected literature number. No content changes.
SPRUGY5C	July 2013	• Added boot examples for various boot scenarios. (Page 3-21) • Modified the boot process for various boot modes by dividing them into three sub processes. (Page 3-2) • Moved the Boot Parameter table and Devstat Register values into the device data manual. (Page 3-1)
SPRUGY5B	June 2012	• Added and corrected details describing the Ethernet boot process. (Page 3-8) • Added and corrected details describing the I2C boot process. (Page 3-13) • Added and corrected details describing the PCIe boot process. (Page 3-12) • Added or modified statements describing the boot process. (Page 2-4) • Added some new terms and abbreviations. (Page 1-2) • Added the abbreviation RBL to differentiate from Intermediate Boot Loader (IBL). (Page 1-1)
SPRUGY5A	September 2011	• Added the I2C slave mode device configuration table. (Page 3-15) • Added the interrupt control procedure used in the PCIe boot process. (Page 3-12) • Changed device configuration tables for various boot modes to accommodate the structure defined in the ROM code. (Page 2-4) • Corrected the secondary core boot process. (Page 2-4) • Modified the general structure and clarified the description of bootloader initialization after hard reset and soft reset. (Page 2-5) • Moved memory map information to its respective device-specific data manual. (Page 2-4)
SPRUGY5	November 2010	Initial Release

Contents

List of Tables

List of Figures

Preface

About This Manual

This document describes the features of the on-chip bootloader provided with C66x_Digital Signal Processors (DSP).

This document should be used in conjunction with the device-specific data manuals and user guides for peripherals used during the boot. This document supports only non-secure boot mode.

Notational Conventions

This document uses the following conventions:

- Commands and keywords are in **boldface** font.
- Arguments for which you supply values are in *italic* font.
- Terminal sessions and information the system displays are in `screen font`.
- Information you must enter is in **`boldface screen font`**.
- Elements in square brackets ([ ]) are optional.

Notes use the following conventions:

Note—Means reader take note. Notes contain helpful suggestions or references to material not covered in the publication.

The information in a caution or a warning is provided for your protection. Please read each caution and warning carefully.

CAUTION—Indicates the possibility of service interruption if precautions are not taken.

WARNING—Indicates the possibility of damage to equipment if precautions are not taken.

Related Documentation from Texas Instructions

C66x CorePac User Guide	SPRUGW0
DDR3 Memory Controller for KeyStone Devices User Guide	SPRUGV8
External Memory Interface (EMIF16) for KeyStone Devices User Guide	SPRUGZ3
HyperLink for KeyStone Devices User Guide	SPRUGW8
Inter Integrated Circuit (I^2C) for KeyStone Devices User Guide	SPRUGV3
Multicore Shared Memory Controller (MSMC) for KeyStone Devices User Guide	SPRUGW7
Peripheral Component Interconnect Express (PCIe) for KeyStone Devices User Guide	SPRUGS6
Phase Locked Loop (PLL) Controller for KeyStone Devices User Guide	SPRUGV2
Power Sleep Controller (PSC) for KeyStone Devices User Guide	SPRUGV4
Serial Peripheral Interface (SPI) for KeyStone Devices User Guide	SPRUGP2
Serial RapidIO (SRIO) for KeyStone Devices User Guide	SPRUGW1

Trademarks

Chapter 1

Introduction

IMPORTANT NOTE—The information in this document should be used in conjunction with information in the device-specific Keystone Architecture data manual that applies to the part number of your device.

This document describes the features of the on-chip ROM boot loader (RBL) when driven by C66x Digital Signal Processors (DSP). For the KeyStone devices that have a ARM in their architecture, the ROM bootloader can also be driven by the ARM. This document does not discuss about the features of the on-chip ROM boot loader when driven by the ARM. This document does not discuss the features and implementation of intermediate boot loader (IBL) and the application utility for merging multiple core applications into one application called multicore application deployment (MAD). For information on these items, see the applicable user guides.

This document should be used in conjunction with the device-specific data manuals and user guides for peripherals used during the boot. This document applies to non-secure boot mode only.

1.1 Bootloader Features

The ROM Boot Loader (RBL) is software code that resides in the on-chip read only memory (ROM) to assist the customer in transferring and executing their application code. The start address of the RBL is 0x20B00000.

To accommodate different system scenarios, the RBL provides different boot modes. These boot modes can be broadly classified either as a host boot or memory boot mode.

In a host boot mode, the RBL configures the boot device as a slave and waits for an external master (host) to load the application into the device to boot.

In a memory boot mode, the RBL configures the boot device as a master and initiates the loading of the application code from the slave memory. Because different devices support different sets of boot modes, see the device-specific data manual to obtain the list of boot modes supported in that device.

In all boot modes, the entire boot operation can be partitioned into two sections:

- Initialization
- Boot process

In the initialization phase, the RBL configures the device resources to start the boot process. The resources used depend upon the boot mode requirements.

In the boot process phase, the image is loaded into the device and executed. The boot process depends on the following factors:

- The trigger that initiated the boot operation
- The location of the boot image (host or memory).

If the image is in an external host, the boot process varies depending on the host knowledge of the boot device memory map.

This document covers:

- The different triggers that can initiate the boot operation
- The initialization process
- The boot process based on the location of the image
- The specific boot process features for different boot modes

1.2 Terms and Abbreviations

Term	Definition
I²C	Inter-Integrated Circuit
MSMC	Multicore Shared Memory Controller
PCIe	Peripheral Component Interconnect Express
POR	Power on Reset
RBL	ROM Boot Loader
SPI	Serial Peripheral Interface
SRIO	Serial Rapid Input/Output

Chapter 2

Reset Types and Device Initialization

The boot process can be divided in to two steps. The first step is the initialization process, which depends on the type of reset that triggers the boot (this chapter). The second step is the boot mode-specific process (Chapter 3).

2.1 Reset Types

In KeyStone devices, resets are used as the trigger for initiating the boot process and the boot process varies based on the type of the reset. There are four types of reset supported in the KeyStone architecture:

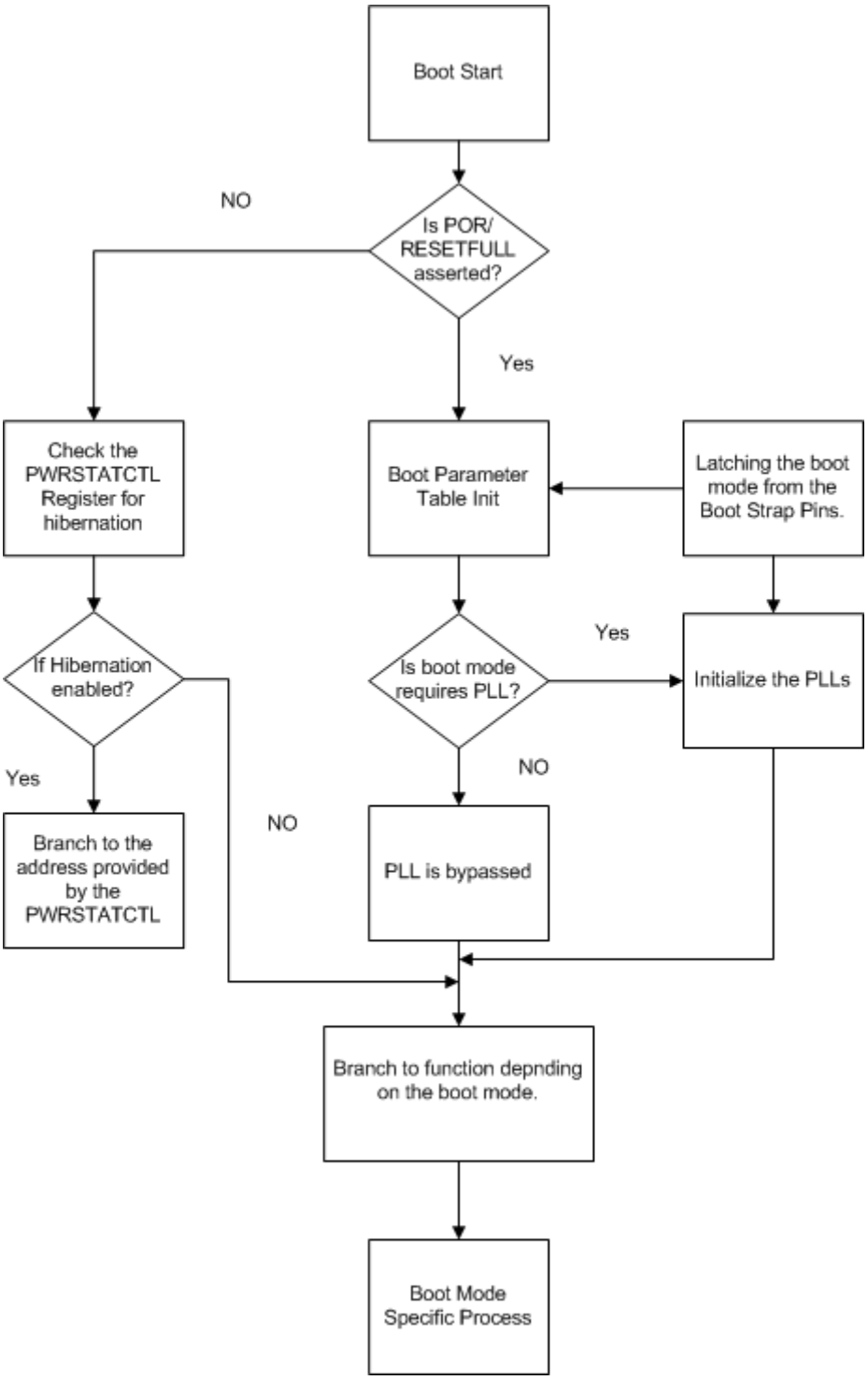
- Power-on reset (POR)
- Hard reset (chip 0 reset + chip 1 reset)
- Soft reset (chip 1 reset)
- Local reset

The first three types of reset are considered global resets because they affect the entire device, while the local reset affects only the CorePac. For local reset, the boot process is not triggered. For further details on the reset types, see the device-specific data manual.

Irrespective of the global reset type, the boot process is executed by C66x CorePac0 when the boot loading process is set to use the C66x as the master to drive the boot.

The boot process flow under the different reset types is shown in Figure 2-1.

Figure 2-1 Boot Process



2.2 Device Initialization

2.2.1 Initialization Process After Power On Reset

Power-on reset resets the entire chip. Everything on the device is reset to its default state and can be initiated by either the POR pin or the RESETFULL pin. While the POR pin is asserted during the power up sequence, the RESETFULL is asserted by a host to reset the entire device. The RESETFULL is also asserted during the power on reset sequence. During the RESETFULL assertion, the general purpose pins used for boot configuration are sampled to latch the boot configuration values into the device status (DEVSTAT) register.

Using the device configuration setup from the device status (DEVSTAT) registers, the boot process executes an initialization code. The initialization settings that are executed by the RBL are listed below:

- The RBL enables the reset isolation in all peripherals that support it. The power state of these peripherals is not changed. The device-specific data manual lists all the peripherals that support reset isolation.
- The RBL also ensures that the power and clock domains are enabled for any peripherals that are required for boot.
- The RBL configures the system PLL to set the device speed. The boot configuration pins provide the RBL with the information about the reference clock used in the system. RBL gets the optimal operating speed of the device from the e-fuse register. The various reference clock frequency and operating clock frequency lists are available in the device-specific data manual.
- The main PLL stays in bypass mode for no-boot, SPI, and I^2C boot. For other boot modes, a PLL initialization sequence executes inside the boot ROM to configure the main PLL in PLL mode.
- The RBL reserves a portion of the L2 in all the cores in the device to perform the boot process. The start address, size, and the definition of the sections reserved are listed in the device-specific data manuals. For EMIF16 boot, no memory is reserved by the RBL; memory usage depends entirely on the image stored in, and executed from, the NOR flash.
- All the interrupts are disabled except IPC interrupts and the host interrupts that are needed for the PCIe, SRIO (DirectIO), and HyperLink boot modes.
- During the boot process, the RBL executes an IDLE command on the secondary CorePacs and keeps the secondary CorePacs waiting for an interrupt. After the application code to be loaded in these secondary CorePacs are loaded and the BOOT_MAGIC_ADDRESS values in individual CorePacs are populated, the application code in the CorePac0 can trigger the IPC interrupt to wake up the secondary cores and branch up to the address specified in the BOOT_MAGIC_ADDRESS.
- All L1D and L1P memory is configured by the boot code as cache memory. L2 memory, however, is configured as addressable memory.
- The RBL also provides an ability for the user to configure the DDR EMIF before loading the image into the external memory during the boot process using a DDR structure. This structure is reserved in the L2. For every section that the RBL reads, it verifies if the DDR enable magic word is set. If the magic word is set, then the DDR structure is used to initialize the DDR. The structure definition of the DDR varies from device to device. See the device-specific data manual for the DDR configuration structure.

- The RBL uses the pin-strapped boot mode pins (available through the DEVSTAT register) to setup the initial configuration structure, which is called the boot parameter table. This table is stored in the reserved section of L2 in CorePac0. Even though the boot parameter table format varies based on the boot mode selected, there are a first few offsets that are common across all boot modes for a specific device. These offsets are listed in the data manual.
- The RBL uses the BOOTCOMPLETE register, which controls the BOOTCOMPLETE pin status, to indicate the completion of the RBL boot process. The BOOTCOMPLETE pin goes high when the boot complete bits in the BOOTCOMPLETE register for all the cores are set. The RBL sets the bits for each CorePac once it completes the boot process in the CorePac and just before it exits the process. Because of legacy implementation, the BOOTCOMPLETE bit in the register corresponding to the CorePac0 is set by the hardware.

2.2.2 Initialization Process After Hard Reset or Soft Reset

The hard and soft resets are initiated by the reset pin and can be configured as a hard reset or a soft reset.

By default, the reset is configured to be a hard reset. If the reset is configured as a hard reset, it will reset all internal modules except the test logic, emulation logic, and the reset-isolated modules. For information on reset isolation and a list of modules that have this feature, see the device-specific data manual.

If the reset is configured as a soft reset, some of the MMRs and the memory are preserved in addition to the modules that are not reset by the hard reset. When the reset is asserted, the RBL first checks if the hibernation is enabled in the power status control register.

If hibernation is enabled, the boot process carries out the hibernation sequence mentioned in section 2.2.3 . If hibernation is not enabled, the RBL will carry out the initialization process mentioned in section 2.2 followed by the boot specific process.

The only difference in this boot process compared to one triggered by the power on reset is that the boot configuration pins are not sampled to update the device status registers. In addition, the power state controller will not reset any peripherals that are reset-isolated in the device. See the device-specific data manual for detailed descriptions of the different reset types.

2.2.3 The Effect of Hibernation on the Initialization Process

To reduce power consumption, hibernation can be used to shutdown CorePacs and peripherals that are not used. The RBL's involvement in the hibernation process is minimal. Before shutting down the CoresPacs, the user needs to set the hibernation enable bit and the mode bit and the branch address offset to which CorePac0 will jump to execute the wake up code sequence in the power state control register.

After a hard or soft reset, the bootloader samples the power state control register to verify that hibernation is enabled. If hibernation is enabled, the bootloader resets a set of peripherals and avoids resets to other peripherals based on the hibernation mode set. In KeyStone devices, there are two hibernation modes: Hibernation1 and Hibernation2.

In Hibernation1 mode, the critical status values and information are stored in the MSMC SRAM. A chip-level register is added to control the reset MSMC parity RAM. Before entering Hibernation1 mode, the user must correctly configure the chip-level register Chip Miscellaneous Control Register so the parity SRAM will not reset when a hard or soft reset is triggered to exit Hibernation1.

During this hibernation mode, only MSMC SRAM content is preserved; the MSMC MMR is not preserved. Therefore, the user should save the MSMC MMR to a known memory location inside the MSMC SRAM before entering this hibernation mode. In general, when Hibernation1 is enabled, CorePac0 will disable DDR self-refresh and branch-to-address specified in PWRSTATECTL and the other CorePacs will be in the IDLE state. After wake-up, the user should make sure that the PWRSTATECTL's standby and hibernation fields have become 0, then branch to the boot magic address. The response time in the case of Hibernation1 is less than 100 ms.

In Hibernation2 mode, the critical status values and data are stored in DDR memory. When the hard or soft reset is triggered to exit from Hibernation2 mode, the MSMC is also reset. The MSMC configuration is set to the default value after reset, so only the lower two gigabytes of four gigabytes of space in the DDR memory will be visible. Therefore, the branching address for exiting Hibernation2 mode should be set at the lower two-gigabyte boundary, between 0x8000_0000 to 0xFFFF_FFFF.

In general, when Hibernation2 is enabled, CorePac0 will also disable DDR self-refresh, reset MSMC parity, and branch-to-address specified in PWRSTATECTL. The other CorePacs will be in the IDLE state and after wake up will verify that PWRSTATECTL's standby and hibernation fields have become 0, then branch to the boot magic address. The response time is less than two seconds.

Before entering hibernation mode, the user can enable or disable reset isolation for the SRIO. When the SRIO has reset-isolation enabled before entering hibernation, the user should also make the LPSC for SRIO active because packet forwarding requires the VBUS clock to function. When the SRIO has reset-isolation disabled before entering hibernation, the SRIO block must be disabled by the user. This includes stopping the VBUS clock and disabling the PHY layer. Stopping the VBUS clock without disabling the PHY layer can cause system congestion and system hang.

When the PCIe is used in EP mode, the user must put the PCIe in the L1 powerdown mode before it enters the hibernation mode; the PCIe power domain should be kept on during hibernation mode. When the device exits hibernation mode, the RC device can issue a reset request to all the EP points to bring the PCIe endpoint alive.

To avoid resetting the DDR during hard reset, reset-isolation is provided for the DDR. The boot code enables the DDR reset-isolation by default and the user has the option to turn off the reset isolation-feature if it is not needed. (See the *Power Sleep Controller (PSC) for KeyStone Devices User Guide* in "Related Documentation from Texas Instruments" on page ø-viii for information on disabling the reset-isolation for DDR3.) Because the DDR contents are preserved, the PLL for the DDR3 EMIF must stay locked and the DDR PHY must be active to preserve the DDR3 content. This avoids full calibration when resuming the normal operation -- full calibration can corrupt the DDR3 content. In summary, the DDR is alive during both hibernation modes and the DDR3 can be put into self-refresh mode to save power.

The MSMC on this device does not support the PSC disable interface. Therefore, the MSMC cannot detect the status of the DDR3 EMIF when the DDR3 EMIF is disabled or enabled by the PSC. Because accesses to a disabled DDR3 EMIF would hang the device, the user can tie off the PSC control to DDR_EMIF to enable it on the chip level (always on). This linking of the PSC control plus the reset-isolation of DDR EMIF make it impossible to reset the EMIF controller independently of the rest of the chip.

Chapter 3

Boot Modes

Because the ROM boot loader (RBL) supports many boot modes, a set of general-purpose pins is used to select a specific boot mode and also provide a minimal configuration for the specified boot mode.

The values on these pins are then latched into a register in boot-configuration space called the Device Status Register (DEVSTAT) as the CorePac comes out of reset. The number of pins used for this configuration and their definitions vary with the devices. See the device-specific data manual for more information on these boot configuration pins.

3.1 Boot Processes

A factor that affects the boot process is the location of the boot image. Even though the boot process is unique to each boot mode, it can be broadly classified based on the location of the boot image. Based on this classification, the boot process falls into three categories:

- The ROM boot loader (RBL) loads the image from a secondary storage (memory boot).
- The image is loaded from a host that knows the memory map of the boot device.
- The image is loaded from a host that does not know the memory map of the boot device.

Within each of these possible boot flows, the boot process can be further divided into three sections for each of the unique boot modes:

- The initialization process, in which the RBL does a specific initialization routine based on the boot mode selected.
- The image loading process, in which the image loading into the device is directed by the peripheral protocol used for the specific boot mode.
- The hand-over process, in which the RBL handles the completion of the boot process and starts the boot image execution.

While rest of this section covers the boot process based on the boot image location, the remainder of this chapter will discuss the individual boot mode operations.

3.1.1 Boot Process in Memory Boot Modes

In the memory boot modes, the RBL controls the image download process for the boot device. The boot image should be translated into the format that the RBL can understand. In case of the memory boot modes, the boot image is converted to a boot table (see Chapter 3 "Boot Table" on page 3-4). The boot table is read from the secondary storage by the RBL and placed in the appropriate memory locations within the device. Once the RBL completes the image download process, it moves the program counter to the c_int00 address captured in the boot table and starts executing the boot image.

3.1.2 Boot Process in Host Boot Mode with Memory Map Knowledge of Boot Device

In the host boot mode, the ROM Boot Loader (RBL) waits for the host to load the image. For the boot modes in which the host knows the memory map of the boot device, the primary core (CorePac0) is in an IDLE wait mode and the host transfers the image into different memory locations of the device. Once the host completes the image loading process, it needs to provide the c_int00 address to the RBL to start the image execution. To do this, the RBL provides a reserved address area in the L2 where the host can write the c_int00 address. This reserved memory is called the boot magic address and the address of this location depends on the device. Check the device-specific data manual for the boot magic address.

Once the host updates the boot magic address, it hands over control to the RBL by waking CorePac0. The RBL starts again, checks the boot magic address, and jumps the program counter to the specified address and starts the image execution.

3.1.3 Boot Process in Host Boot Mode without the Knowledge of Memory Map

In the case of the host boot mode in which the host has no knowledge of the memory map of the boot device, the loading of the image into the device depends the ability of the RBL to decode the data sent by the host. In this case, the image should be translated into the boot table. Transferring this boot table image into the device depends on a specific boot mode. But once this boot image reaches the device, the RBL decodes the boot table and loads the image into the specific memory locations. Once the RBL completes downloading the images into the device, it moves the program counter to the c_int00 address captured in the boot table and starts executing the boot image.

3.2 Boot Configuration Format

The RBL uses a set of tables to carry out the boot process. Before considering the individual boot modes, it is necessary to understand these different types of tables. There are three types of tables used by the RBL:

- Boot parameter table
- Boot table
- Boot configuration table

3.2.1 Boot Parameter Table

The boot parameter table is the most common format the RBL employs to determine the boot flow. Boot parameter tables have a first few parameters in the table common across all the boot modes while the rest of the table format is dependant on the boot mode selected. See the device-specific data manual for the boot parameter table format for different boot modes. The RBL copies a default boot parameter table for each boot mode into the reserved L2 section of CorePac0 and modifies the default values based on the boot configuration selected through the bootstrap pins. This table forms the maps for the RBL to execute the boot process.

3.2.2 Boot Table

The image to be loaded into the device is converted to a format recognizable by the RBL. This format is called the boot table. Code and data sections are inserted into the boot table automatically by the hex conversion utility. The hex conversion utility uses information embedded by the linker in the application file to determine the destination address and length for each section. Adding these sections to the boot table requires no special intervention by the user. The hex conversion utility adds all initialized sections in the application to the boot table. The remaining information included in this section describes the format of the sections in the boot table.

Each section is added to the boot table in the same format. The first entry is a 32-bit count representing the length of the section in bytes. The next entry is a 32-bit destination address, where the first byte of the section is copied.

The RBL continues to read and copy these sections until it encounters a section whose byte count is 0. This indicates the end of the boot table. Then, the bootloader branches to the entry point address (specified at the beginning of the boot table) and begins executing the application.

The boot table format is as follows:

- 32-bit header record indicating where the bootloader should branch after it has completed copying the data
- For each initialized section:
  - 32-bit section byte count
  - 32-bit section address (destination address for the copy)
  - The data to be copied
- A 32-bit termination record (0x00000000)

3.2.3 Boot Configuration Table

A boot configuration table is used if certain peripherals must be programmed with values that differ from their reset values before loading an application. For example, if the application needs to be loaded into DDR memory, the boot configuration table can be used to program the DDR registers and enable the DDR peripheral before loading the application code into DDR.

Each table entry in the boot configuration table has three elements:

- The address to be modified
- The set mask
- The clear mask

The RBL reads the specified address, then sets any bits that are set in the set mask element and clears any bits that are set in the clear mask element. If both the set and clear mask elements are 0, the value in the address field is branched via a standard call with the return address stored in register B3. The boot configuration table is terminated when all three elements are 0.

3.2.4 Utilities Used to Generate Different Tables

Utilities used to generate different table formats are listed below:

- Hex6x is used to convert the application code into a boot table format.
- Romparse is used to append the boot parameter to a boot table or a boot configuration table.
- Bootconvert6x is used to convert the boot table derived from a little endian application code to a big endian format. This is required as the RBL assumes all the images to be in big endian mode.
- B2i2c is used to convert the boot table into a i2c/spi format table. This table can be loaded into an EEPROM that is connected through I^2C to the device.
- Bootpacket is used to break the boot table into packets that can be sent from the host to the device booted in Ethernet boot mode.
- Pcsendpkt is used to help the host send the packets generated by bootpacket to the boot device.

3.3 EMIF16 Bootloader Operation

3.3.1 ROM Bootloader (RBL) Initialization Process

In this mode, the RBL does not do much during the initialization process. The RBL just configures the EMIF16 interface based on the configuration parameters specified in the boot parameter table for the EMIF16 boot mode. The boot parameter structure definition for the EMIF16 boot mode and the parts of this table that can be configured by the bootstrap pins are detailed in the device-specific data manual.

3.3.2 RBL Loading Process

During this process, the RBL just sets the program counter to the base address of the EMIF chip select that is specified through the boot strap pins. See the device-specific data manual to get the base address of the CS2 data memory. No return is expected. The RBL also does not reserve any memory in L2. The image will be stored in the flash and so no image transfer occurs during this boot mode.

3.3.3 RBL Hand Over Process

For this boot mode, there is no additional hand-over process that occurs.

3.4 SRIO Bootloader Operation

3.4.1 RBL Initialization Process

In this boot mode, the RBL configures the SRIO to operate in both DirectIO and messaging mode. Before configuring the SRIO, the SerDes should be configured. The RBL initializes the PLL and the receive and the transmit channel registers. After configuring the SerDes and the SRIO for message and directIO mode, the boot code initializes the QMSS to configure transmit and receive queues. The boot code allocates a 32KB block of local L2 memory on CorePac0 for packet reception. This is divided into seven buffers of 4096 bytes each, with the remainder for the packet descriptors. The RBL also configures the SRIO in promiscuous mode and the queue manager subsystem to route all the messages received in message mode to the internal scratch memory for CorePac0. The RBL also configures the SRIO port and lanes based on the boot parameter values for the SRIO boot mode.

3.4.2 RBL Loading Process

The loading process differs based on which SRIO mode is used by the host. If the directIO mode is used, the loading process follows the steps detailed in section 3.1.2 on page 3-2. In this scenario, the RBL will be constantly polling the boot magic address.

If the host uses the messaging mode, then the loading process follows the steps detailed in section 3.1.3 on page 3-3. The host is responsible for breaking the image into SRIO packets and generating messages destined for the boot device. Because the device is configured to route all the messages received to CorePac0, the packets are sent to CorePac0, where the RBL decodes the boot table content and retrieves the code to be placed in appropriate memory locations within the boot device. The boot messages are simply a segmented boot table prepended with the header shown in Table 3-1. The application that is loading the message from the host device should be responsible for packaging the message packets with the header before sending them to the device. Also, because reset isolation is enabled by default by the RBL, the application code should disable the reset isolation.

Table 3-1 SRIO Message Mode Boot Header

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
block Size = size of packet plus four byte header															
Block Checksum one's complement															

Because there is no provision in messaging mode for handling the messaging output, the host application should provide a delay between message transmissions to avoid an out-of-order message scenario.

3.4.3 RBL Hand-Over Process

The hand-over process also differs based on what SRIO mode is used by the host. If the directIO mode is used, the RBL will keep polling until the boot magic address becomes a non-zero value. Once the boot magic address becomes a non-zero value, the RBL changes the program counter and starts executing the boot image.

In the messaging mode, once the last block of image is received by the RBL, it changes the program counter to the c_int00 address and starts the boot image execution. After the boot table processing is complete, the boot code resets the queue manager, sets the boot complete bit, and branches to the address specified by the boot table. The interrupt maps are restored to their default values.

3.5 Ethernet Bootloader Operation

3.5.1 RBL Initialization Process

When the device is set to boot through the Ethernet mode, the RBL configures SerDes, SGMII, and the switch plus the PASS and the Multicore Navigator (Packet DMA and the QMSS) (if it is available in the device). These initial configurations are determined by querying the boot mode pins of the DEVSTAT register and the boot parameter table for the Ethernet boot. The PHY is not initialized by the RBL and it is the responsibility of the host to initialize the ethernet PHY.

Based on the boot mode selected, the PA subsystem can be driven by either the main PLL reference clock or by the SerDes reference clock. If the main PLL reference clock is used, the PA multiplier configuration can be determined from the three PLL selection bits in the DEVSTAT register and the input clock, which is specified in the device-specific data manual. If the PA reference clock is used, the PA PLL clock configuration is selected based on the boot configuration value set through the bootstrap pins. See the device-specific data manual for the boot configuration definition.

The external connection from the DSP to the host is selected through the device configuration bits in the boot mode pins of the DEVSTAT register. Not all KeyStone devices have a PA subsystem. In this case, only the SerDes reference clock is used to drive the Ethernet subsystem.

3.5.2 RBL Loading Process

Once the RBL initializes the peripherals to be used, it sends an Ethernet-ready packet to notify the host of the MAC address of the device. The structure of the Ethernet-ready packet is explained in section 3.5.2.1 . The host can use this to create packets of the boot table image and send to the boot device. The boot table packet format is explain in section 3.5.2.2 .

3.5.2.1 Ethernet-Ready Announcement Format

The Ethernet-ready announcement frame is made in the form of a BOOTP request so it can use a standard format. No response is processed for this message and it is constructed so that most—if not all—BOOTP and DHCP servers will discard it. The announcement frame is sent every three seconds (the time interval between the packets varies with the clock input); no retransmission is done.

The frame uses the DIX MAC Header format. The MAC header contains:

- Destination MAC address = H-MAC addr (from boot parameters, normally FF:FF:FF:FF:FF:FF)
- Source MAC address = this device's MAC addr (from boot parameters)
- Type = IPV4 (0x800)

The IPV4 header contains:

- Version = 4
- Header length = 0
- TOS = 0
- Len = 328 (300 BOOTP + 8 UDP + 20 IP)
- ID = 0x0001
- Flags + Fragment offset = 0

- TTL = 0x10
- Protocol = UDP (17)
- Header checksum = 0xA9A5
- SRC IP = 0.0.0.0
- DEST IP = 0.0.0.0

The UDP header contains:

- Source port = BOOTP client (68 decimal)
- Destination port = BOOTP server (67 decimal).
- Length = 308 (300 BOOTP + 8 UDP)
- Checksum = 0 (not calculated)

The BOOTP payload contains:

- Opcode = Request (1)
- HW Type = Ethernet (1)
- HW Addr Len = 6
- Hop Count = 0
- Transaction ID = 0x12345678
- Number of seconds = 1
- Client IP = 0.0.0.0
- Your IP = 0.0.0.0
- Server IP = 0.0.0.0
- Gateway IP = 0.0.0.0
- Client HW Addr = this device's MAC address
- Server hostname = ti-boot-table-svr
- Filename = ti-boot-table-XXXX (where XXXX is the four-character device ID from boot parameters)
- Vendor info = all 0s

3.5.2.2 Ethernet Boot Image Packet Format

The boot table format is encapsulated in Ethernet frames with IPV4 and UDP headers. The following paragraphs describe the Ethernet frames that are accepted. Frames not matching the specified criteria are silently discarded and subsequent frames processed.

Frames using both DIX and 802.3 MAC header formats are accepted as are frames with and without VLAN tags. Any source MAC address is acceptable. A destination MAC address of this device (as specified in boot parameters) or the M-MAC specified in the boot parameters are accepted. VLAN fields (other than type/len) are ignored. If 802.3 format MAC format is used, the SNAP/LLC header will be verified and skipped. The type field will select IPV4 type (0x0800).

The IPV4 header validates the Version (4), flag and fragment fields, and protocol (UDP) field. The header length field is parsed to skip header option words. Any source and destination IP addresses are accepted.

The UDP header verifies that the source and destination port numbers match those specified in the boot parameters. If the boot parameter source port field is 0, any source port will be accepted. The UDP header length is sanity-tested against the adjusted frame length. If the UDP length is too long for the frame or is not a multiple of two, the frame is discarded. The UDP checksum is verified and the frame with incorrect UDP checksum is discarded if the UDP checksum field is non-zero.

The following checks are performed on the boot table frame header:

- The magic number field and opcode fields are compared to the expected values.
- The sequence number field is compared to the expected value. The expected value of the sequence number is 0 for the first frame processed and increments by 1 for each processed frame.

The boot table frame payload (which is a multiple of four bytes in length) is processed by the boot-table processing function.

Table 3-2 Ether Boot Packet Format

Ethernet Header, One of The Following Types:
DIX Ethernet (DMAC, SMAC, type: 14 bytes) 802.3 w/ SNAP/LLC (DMAC, SMAC, len, LLC, SNAP: : 22 bytes) DIX Ethernet w/ VLAN (18 bytes) 802.3 w/ VLAN and SNAP/LLC (26 bytes)
IPV4 (20 to 84 bytes) UDP (8 bytes)
Boot Table Frame Header (4 bytes) Boot Table Frame Payload (min 4 bytes, max limited by max Ethernet frame - previous headers)

Table 3-3 Boot Table Frame Header

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Boot Magic Number = 0x544b															
Opcode = 0x01								Sequence number							

3.5.3 RBL Hand Over Process

Once the RBL receives the valid packet, it decodes it to get the image sections and loads them in the appropriate memory location. After the image is loaded, the RBL changes the program counter to the c_int00 address and starts the boot image execution.

3.6 PCI Express (PCIe) Bootloader Operation

3.6.1 RBL Initialization Process

In the PCIe boot mode, the RBL configures the BAR registers, the number of windows, and their sizes to provide memory access to the host. The different BAR configurations that are initialized are shown in the device-specific data manual. The RBL also configures the SerDes from information it obtains from the boot parameter table. The RBL also configures the interrupt subsystem by configuring the chip-level interrupt controller, then executes the IDLE instruction (MSI or legacy interrupts can be used).

3.6.2 RBL Loading Process

Once the host initiates the link with the boot device, it should load all sections of the boot image. The host is also responsible for updating the boot magic address of CorePac0.

3.6.3 RBL Hand-Over Process

In this boot mode, CorePac0 is executing an IDLE until the PCIe interrupt wakes it. This interrupt can be an MSI interrupt or a legacy interrupt. Once CorePac0 comes out of IDLE, the RBL reads the boot magic address and modifies the program counter to start executing the boot image.

3.7 I²C Bootloader Operation

3.7.1 RBL Initialization Process

In the I²C boot mode, the RBL configures the I²C peripheral. Because the data is transferred using I²C from EEPROM, the interrupt subsystem and the EDMA are not enabled. A seven-bit address mode and an eight-bit data mode is the only supported transfer configuration.

I²C can operate either in master mode or in slave mode. In the master mode, the DSP drives the I²C slave device where the image is stored. In the slave mode, the DSP is driven by an I²C master device. The master device is usually another KeyStone device or a FPGA. The boot configuration options in the DEVSTAT register are used to select the master/slave mode. The RBL bypasses the PLL and runs the CorePacs at the reference clock frequency and also configures the I²C bus at the lower rate to start with.

The EEPROM is traditionally partitioned into pages and each page into blocks of 128 bytes, with each block numbered. The user is capable of loading the image at any block and provides the offset of the block through the boot configuration values. In addition, the user also has a provision to provide the speed of the I²C bus in the boot configuration values. A detailed summary of the I²C boot parameter table and the boot configuration definitions are listed in the device-specific data manual.

3.7.2 RBL Loading Process

In this boot mode, the RBL provides the user with the maximum control over the boot process. The loading process in this boot mode is more than just loading the image. The RBL also provides an option for the user to provide an entire boot parameter table, thereby controlling the configuration of the boot process. Based on the different options available to the user, the loading process in this boot mode can vary.

3.7.2.1 Loading a Boot Image From EEPROM Flash

In this mode, the I²C is configured as master. The first block (identified in the boot configuration) will contain the boot parameter table for I²C specifying the correct PLL configurations for the CorePacs. The user is also given a provision in the boot parameter table to provide the starting page of the boot image, block of the boot image, speed of the I²C bus, and the type of table that is available in the next block that the RBL is going to read. The RBL reads the first block, which is a boot parameter table and uses the information to reconfigure the device. Then the RBL gets the block and page information for obtaining the boot image and starts reading block by block. After each block, the RBL decodes the boot table and loads the image in appropriate memory locations.

3.7.2.2 Loading a Boot Image After Register Configuration

In this boot mode, the RBL also provides an option for the user to initialize registers before loading the image. The most common use case for this scenario is configuring the DDR controller before loading the image into DDR memory. In this case, the boot parameter table will have the type as boot config table. The boot parameter table also mentions where the next block to be read by RBL is located. This next block can be either a boot parameter table or can be a boot table. The user will be setting the I²C in master mode.

The boot config table format is explained in the Chapter 2. Each element in the table has three 32-bit fields, as shown in Table 3-4.

Table 3-4 Config Table Layout

Entry 0	Address
	Set Mask
	Clear Mask
Entry 1	Address
	Set Mask
	Clear Mask
…	
Entry N, Table Termination	Address = 0
	Set Mask = 0
	Clear Mask = 0

Table 3-5 shows an example boot configuration. Each of these entries in the table can be a standard entry, branch entry, or termination entry.

Table 3-5 Boot Config Table Format

Offset	Data	Operation
0x0	0x0093001C	Set 16 bits MSBs and clear 16 LSBs at address 0x0093001C
0x4	0xFFFF0000	
0x8	0x0000FFFF	
0xC	0x00000000	Termination
0x10	0x00000000	
0x14	0x00000000	

A standard entry has address ≠ 0, and set mask or clear mask ≠ 0. The ROM code reads the 32-bit value at the address, modifies the value as shown in Table 3-6 on a bit-by-bit basis, and writes the value back.

Table 3-6 Standard Boot Config Table Options

Set Mask Bit	Clear Mask Bit	Operation
0	0	Bit value is unchanged
1	0	Bit value is set
0	1	Bit value is cleared
1	1	Bit value is toggled

A branch entry has address ≠ 0, set mask = 0, clear mask = 0. The boot ROM makes a function call to the address. On return (if there is one), the table processing continues.

The table termination field has address, set mask, and clear mask all set to 0. When this entry is found, the boot ROM modifies the current active boot parameter table. The boot mode is changed to I²C master boot parameter table mode, the address is changed to the current next address value, and the boot is rerun.

3.7.2.3 Loading a Boot Parameter Table

In this boot mode, the RBL provides the option to load only a boot parameter table. This option is useful if the user wants to load a new boot parameter table for booting in a different boot mode. The user will be setting the I²C to master mode.

3.7.2.4 Loading Image Into a Slave Device

The RBL also provides the option to set the I²C to slave mode and connect to a master device, which can then send the image to the boot device. The default slave address is set to the value specified in the boot configuration in the DEVSTAT register plus slave address.

Also, the I²C bus on the master must run the bus at a speed that does not exceed the speed that the boot code is able to handle (100 kHz, minimum). A small delay is required between the blocks. The I²C master also needs to send six data bytes to the slave device before sending the boot table.

The header format should be in the following format:

TTxx xx yy yy zz zz

(for Receive I²C address value in the device configuration of 0x0)

Where:

TT= the slave address (part of the I²C command word not included in the data block) for the DSP in slave I²C boot mode. See the data manual for the slave address.

xx xx = length

yy yy = checksum

zz zz = boot option

3.7.3 RBL Hand-Over Process

In scenarios in which the I²C is set to master mode, the RBL will change the program counter to the c_int00 address and start to execute the image when the RBL completes reading the entire boot image from the EEPROM.

In slave mode scenarios, the master will be sending the blocks that are then read by the slave, and once the slave device encounters the end of image, it changes the program counter to the c_int00 address and starts the execution of the boot image.

3.8 SPI Bootloader Operation

3.8.1 RBL Initialization Process

In the SPI boot mode, the RBL initializes only the SPI peripheral. Similar to the I²C boot mode, in the SPI boot mode, the interrupt subsystem and the EDMA are not enabled. The image is read from the NOR flash. The RBL also bypasses the PLL and runs the CorePacs at the reference clock frequency. The flash connected is also divided in to blocks of 128 bytes and the starting block to be read can be provided by the user through the DEVSTAT register. A detailed summary of the SPI boot parameter table and the boot configuration definitions are listed in the device-specific data manual.

3.8.2 RBL Loading Process

The SPI boot loading process is similar to the loading process explained in section 3.7.2.1 except that the image is read from a NOR flash. All the other loading processes that are detailed for I²C master mode work for the SPI boot mode, too.

3.8.3 RBL Hand-Over Process

In this boot mode, the RBL changes the program counter to the c_int00 address and starts to execute the image when the RBL completes reading the entire boot image from the NOR flash.

3.9 HyperLink Bootloader Operation

3.9.1 RBL Initialization Process

In the HyperLink boot mode, the RBL configures the SerDes lanes to achieve the ultra-short-range (HyperLink) connection. The SerDes clock configurations are provided by the user through the boot configuration fields in the DEVSTAT register. The definition of the boot configuration fields and PCIe boot parameter table are available in the device-specific data manual.The RBL also initializes the chip-level interrupt controller to interrupt the DSP after the boot. After setting up the interrupt configuration, the RBL executes an idle instruction in CorePac0.

3.9.2 RBL Loading Process

Similar to the PCIe boot mode, the RBL has no role in transferring the boot image into the device. The host is responsible to load the code in the appropriate memory location and also populate the boot magic address with the c_int00 address. The host must also interrupt CorePac0 and bring it out of IDLE.

3.9.3 RBL Hand-Over Process

Once CorePac0 comes out of IDLE, the RBL changes the program counter to the address specified in the boot magic address and starts the image execution.

3.10 UART Bootloader Operation

3.10.1 RBL Initialization Process

In the UART boot mode, the UART module is the only peripheral configured. The baud rate, data, parity, and stop bits are configured based on the information in the UART boot parameter table. The boot parameter table definitions and the boot configuration values that can be configured through the bootstrap pins are in the device-specific data manual. Once the RBL configures the UART, it sends the UART pings for few seconds, which can be seen in the host. The UART boot mode supports only the CRC mode of xmodem and does not support checksum mode.

3.10.2 RBL Loading Process

The boot image to be loaded should be in the boot table format. Before the ping from the device stops, load the boot table from the host using the XMODEM protocol. Once the RBL starts receiving the table, it will decode the boot table load the image in appropriate memory locations.

3.10.3 RBL Hand-Over Process

When the entire image is loaded, the RBL loads the program counter with the c_int00 address and starts the execution of the boot image.

3.11 NAND Bootloader Operation

3.11.1 RBL Initialization Process

In this boot mode, the boot image is stored in the NAND flash that is connected to the EMIF16 peripheral. The RBL configures the EMIF16 and then tries to read the geometry of the device. This includes:

- 8-bit or 16-bit data width
- Page size
- Number of pages per block
- Number of address cycles

Because the NAND market moves at such a fast rate, all devices may not be compatible. At run time, the RBL will attempt to check for compatibility with the NAND. The RBL also supports a 4b ECC. The structure of the ECC is stored in the EMIF16 hardware ECC block and the format depends on the EMIF16 hardware design.

The RBL will first check for ONFI compliance. Most ONFI-compliant devices should be supported. A read ID command will be issued to address 0x20 and four bytes will be read. If the four bytes match the word 0x49464E4F (ASCII for ONFI), the RBL will attempt to read the ONFI parameter page for the device geometry.

After which, the RBL will request four bytes of identifier code from the NAND by issuing a read ID command to address 0x00. The first and second bytes are the manufacture ID and device ID, respectively, and the fourth byte has information on the NAND parameters. If the device was found not to be ONFI-compliant, the RBL will try the following to determine geometry:

- Compare device ID with IDs and their parameters stored in ROM. It is impossible to have all devices listed in the table, and some devices may share IDs. If the ID is matched, the configuration is used and the next step is skipped.
- Read fourth byte data. This happens when the RBL finds no match with the device ID. The RBL will assume an 8-bit data bus width and check the device Manufacture ID to indicate *Samsung format* or *Common format* when reading the fourth byte parameters for NAND geometry. If non-Samsung, the RBL will be able to change the width to 16-bit, if necessary. Unfortunately, Samsung devices will remain in 8-bit mode.

The user can also use an I^2C NAND configuration structure to provide the NAND geometry. Each word is read from the EEPROM in big-endian format regardless of the endianess of the device. The I^2C NAND configuration structure definitions are explained in the device-specific data manual. The I^2C NAND mode is supported only in selected devices. See the data manual for details.

3.11.2 RBL loading Process

The boot image to be loaded should be in the boot table format. The RBL will first check the first block set in the device configuration bits for bad block error. The RBL checks the Out Of Band (OoB) region's first six bytes of page 0 and page 1 of each block. If the bytes are all set to 0xFF, then the RBL knows that the block is good and starts reading the block and loading the sections to the appropriate memory locations. It is up to the user to make sure that all the bad blocks are marked for ONFI spec and RBL.

3.11.3 RBL Hand-Over Process

After the entire image is loaded into the device, the RBL will update the program counter with the c_int00 address and starts the boot image execution.

3.12 Boot Scenarios

This section details the some of the most common boot scenarios in KeyStone devices.

3.12.1 Booting a Simple Image Through I²C

In this case, the application to be executed is assumed to reside entirely in the internal memory of the KeyStone device. In this case the user will load the image into the EEPROM flash and the KeyStone device will be set to boot in I²C master mode. As discussed in section 3.7 , during the I²C master boot, the RBL configures the device in the lowest possible configuration.

The first step for the user will be to configure the device in the desired operating condition. So the parameter index offset that will be set during the boot configuration should point to the block that carries the boot parameter table that can configure the device to the desired operating conditions. The boot parameter table should have the information for the PLL configuration, bus speed, core frequency, the table type for the next block to be read, and also the offset for the start of the next block. In this boot scenario, the next table type will be a boot table containing the application to be executed on the device.

Both the boot parameter table and the boot table must be loaded in corresponding blocks in the EEPROM. Before loading, the application to be executed needs to be converted to a boot table format. The hex6x utility, available through the code generation tools for the KeyStone device, will help the user to convert the boot image into a boot table format. The hex6x utility divides the boot image into blocks containing the initialized section and also arranges these blocks in the ascending order of the memory map of these sections. In other words, the section to be stored in L2 memory will be arranged first before a section that is destined for MSMC and so on. For further information on the hex6x utility, see the assembler user guide.

The hex6x utility also provides a way for the user to make the application endian-agnostic. This is particularly important when the device is set in little endian mode. The RBL always assumes that the application to be loaded is in a big endian format. Making the boot table endian-agnostic helps loading the application in the correct byte order. Because all the KeyStone devices currently available are 32-bit devices, hex6x can preserve the byte order only if the sections are 32-bit-aligned. If the user's application contains sections that are not 32-bit-aligned, then the boot table derived should be passed through the bconver64x utility to place these sections in correct byte order.

The bconvert64x utility is provided to the user through the Multi Core Software Development Kit (MCSDK). The output of the bconvert64x utility should then be converted to blocks of 0x80 bytes (with each block is mapped to a parameter index) by the b2i2c utility. The b2i2c utility also adds the length and checksum for each of these blocks.

At this stage, the user should use some I²C writer to load this final boot table into the EEPROM to the appropriate index. Then the user should load the boot parameter table with the correct device configuration and the address offset pointing to the parameter index where the boot table is loaded. If the user uses CCS and loads the EEPROM through the device, then the boot table needs to be converted into a CCS dat file using the b2ccs utility. The romparse utility combines the boot table and the boot parameter table to ease the effort of loading the image into EEPROM.

The b2i2c, b2ccs, and romparse utilities are available through the MCSDK.

The sequence of steps to convert the boot application into the image and combine the image with an initial boot parameter table is listed below:

- The application is converted to boot table using the hex6x utility.
- The boot table is then reformatted to align at a 32 bit boundary using the bconvert64x utility.
- The byte-aligned boot table is then divided into 0x80 byte blocks and appended with length and checksum to adhere to the format required by the RBL by passing through the b2i2c utility.
- Optionally, the I²C boot table image obtained as the output of the b2i2c utility can be converted to a ccs dat file by passing it through the b2ccs utility.
- The boot image can also be combined with the initial boot parameter table using the romparse utility.

After the application boot table and the boot parameter tables are loaded, the user should then set the boot configuration pins to boot the device in I²C master boot mode and also set the parameter index to the block containing the boot parameter table. The boot configuration information is available in the device-specific data manual.

During the boot process, the RBL decodes the boot configuration setup latched in the DEVSTAT register and starts reading the boot parameter table. Then the RBL uses the boot parameter table to initialize the device in the desired operating condition. The RBL also identifies that the next block to be read and also the type of that block to be a boot table. The RBL reads one block at a time and decodes the table to extract the different memory sections and load them appropriately. As explained in Chapter 2, the boot table also contains the information about the start address of the application. After loading all the sections, the RBL extracts the start address and jumps the CorePac0 program counter to that address and starts running the application.

3.12.2 Booting an Image From I²C Into an External DDR Memory Using Boot Config Table

While in the previous boot case scenario, the image is assumed to be residing in the internal memory, in this boot scenario, the boot application is assumed to have some or all sections in the external memory. The RBL does not initialize the DDR EMIF controller as it depends on the DDR RAM connected. However, the RBL provides several methods to configure the controller before loading the application.

One way to do this is to use the boot config table to initialize the controller.

In this case, the boot parameter table that is loaded in the beginning will be pointing to the block that contains the boot config table. The boot parameter table will also be modified to note that the next table type is the boot config table. The boot config table will have the set of registers needed to be configured for initializing the EMIF controller. For the format of the boot config table, see Chapter 2.

While in the previous boot case scenario, the boot ends with the loading of boot table, in this scenario, the boot process needs to continue after configuring the controller registers. The boot parameter table also provides the next block address where the real boot table is located. In this scenario, the boot table and the modified boot parameter table can be loaded into the EEPROM in the same way as in the previous scenario. But the boot config table should be combined to the above image manually.

After the complete image is loaded, the user can set the boot configuration exactly as in the previous boot scenario. After the power on reset, the RBL will load the boot parameter table from the parameter index specified in the boot configuration. After configuring the device to desired operating condition, the RBL will jump to the block containing the boot config table and start decoding the values to be configured in each of the registers.

Once the configurations are complete, the RBL will derive the next block to proceed with the boot process. From this point, the RBL follows the same procedure to load the boot table into different memory locations and start executing the application.

3.12.3 Booting an Image From I²C Into an External DDR Memory Using DDR EMIF Table

Another way to initialize the DDR EMIF controller is to use the DDR EMIF table provided by the RBL. The table varies from one device to another. See the device-specific data manual for the structure of this table for a particular KeyStone part. The RBL uses the fact that the boot tables are arranged in the ascending order of the memory location of the different section. The DDR EMIF table resides in the L2 of the CorePac0. The exact address of this location is shown in the device-specific data manual as well.

The user should embed this table into their application and hard code the location to the specified L2 address for that specific device. The structure of the table contains a set of magic addresses that are monitored by the RBL after every block read. If these magic registers are non-zero, the RBL stops the boot process and uses the structure value in the specified address and configures the DDR EMIF controller. After completing the configuration, the RBL continues its boot process of loading the application section from the EEPROM into different memory locations. Once all the sections are loaded, the RBL will modify the program counter of CorePac0 to the start address and start the execution of the user application.

3.12.4 Booting Multiple CorePacs in The Device From I²C

Up to this point, all the boot scenarios assumed that only CorePac0 is brought out of boot. In all the above cases, the other cores in the device are simply running IDLE and waiting for wakeup by CorePac0. From the RBL point of view, CorePac0 is considered to be the primary boot CorePac and the other CorePacs in the device are considered to be secondary.

This boot scenario assumes that the user application that will be loaded into the device will be executed in all CorePacs. This boot scenario further assumes that the same application is loaded in all the CorePacs and the program resides completely in the shared L2 memory and the external DDR memory. This assumption has one copy of the application for all CorePacs.

The application is created in the same way as in the previous boot scenarios. The process of loading the boot image and also the DDR EMIF controller are similar to the previous boot scenarios. In addition, the user will handle the loading of different sections of the code that need to be executed in different CorePacs. The assumption is that the user designed an application using the Corenum Register, which will be populated with the CorePac number of the executing core.

The user should always provide the logic to populate the boot magic address of the secondary CorePacs and also wake up the secondary CorePacs by triggering an IPC interrupt for each of the secondary CorePacs. These portions will be executed only by CorePac0.

Once the RBL has initialized the DDR controller and loaded the application successfully, the RBL running in CorePac0 will modify the program counter of CorePac0 to the start of the application and start executing the application. CorePac0 will, in turn, run the portion of the code to be run only for CorePac0 and set the boot magic address on the secondary CorePacs and then wake up the secondary CorePacs by setting the IPC interrupts for the other CorePacs.

3.12.5 Booting a Simple Image Through an Ethernet Boot

This boot scenario details the steps to boot an application using the Ethernet. The user sets the boot mode to Ethernet boot and also provides certain configurations for setting up the initial system configuration. The device-specific data manual covers the different boot configurations for Ethernet boot.

Once the device is powered up, after configuring the device, the RBL will start sending a boot packet that contains the MAC address of the device. The application to be loaded must be converted to the boot table as mentioned in the boot scenarios above. The output of the bconvert64x utility should then be converted into a boot packet using the format detailed in the Ethernet boot mode section.

The user can also use the bootpacket utility provided through the MCSDK for generating the boot packet as described in the Ethernet boot section. The user must also develop a host application to send the packet generated by a bootpacket utility to the boot device. The MCSDK provides a utility called pcsendpkt to generate such a packet. This utility takes the MAC address as an argument to generate the packet.

Follow these steps to get a final boot image:

- Convert the application to a boot table using the hex6x utility.
- Reformat the boot table to align at a 32-bit boundary using the bconvert64x utility.
- Slice the output of the bconvert64x utility into the packets that need to be sent from the host to the boot device using a bootpacket utility.
- Use a host application to send the packets across the Ethernet from the host to the boot device.

Once the RBL starts receiving the packet in the expected format, it deconstructs the packet and extracts the boot table. The RBL will then modify the extracted boot table and place the different sections in suitable memory locations. After loading all the sections, the RBL then updates the program counter of CorePac0 and starts executing the user application as detailed in the previous boot scenarios.

3.12.6 Booting a Simple Image Through SRIO DirectIO Mode

This boot scenario details the steps for booting an application through a boot mode that provides the memory map information of the device to the host. In this case, the RBL plays a very minimal role in the boot process. The RBL configures the device to put it into a desired operating condition. Then the RBL keeps polling the boot magic address of CorePac0. The user is responsible for loading the different boot sections into the device and also to populate the boot magic address. The RBL, once it sees the non-zero value in the boot magic address, updates the program counter of CorePac0 to the address mentioned in the boot magic address and starts executing the application. Because the RBL is not responsible for loading the boot application, the image is not converted to the boot table.

In other boot modes in which the device exposes the memory map to the host, the RBL runs an IDLE on CorePac0 and it is up to the user to send an interrupt from the host to wake CorePac0 and let the RBL check the boot magic address.

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, enhancements, improvements and other changes to its semiconductor products and services per JESD46, latest issue, and to discontinue any product or service per JESD48, latest issue. Buyers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All semiconductor products (also referred to herein as “components”) are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its components to the specifications applicable at the time of sale, in accordance with the warranty in TI’s terms and conditions of sale of semiconductor products. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by applicable law, testing of all parameters of each component is not necessarily performed.

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license 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 significant portions 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. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

Buyer acknowledges and agrees that it is solely responsible for compliance with all legal, regulatory and safety-related requirements concerning its products, and any use of TI components in its applications, notwithstanding any applications-related information or support that may be provided by TI. Buyer represents and agrees that it has all the necessary expertise to create and implement safeguards which anticipate dangerous consequences of failures, monitor failures and their consequences, lessen the likelihood of failures that might cause harm and take appropriate remedial actions. Buyer will fully indemnify TI and its representatives against any damages arising out of the use of any TI components in safety-critical applications.

In some cases, TI components may be promoted specifically to facilitate safety-related applications. With such components, TI’s goal is to help enable customers to design and create their own end-product solutions that meet applicable functional safety standards and requirements. Nonetheless, such components are subject to these terms.

No TI components are authorized for use in FDA Class III (or similar life-critical medical equipment) unless authorized officers of the parties have executed a special agreement specifically governing such use.

Only those TI components which TI has specifically designated as military grade or “enhanced plastic” are designed and intended for use in military/aerospace applications or environments. Buyer acknowledges and agrees that any military or aerospace use of TI components which have ***not*** been so designated is solely at the Buyer's risk, and that Buyer is solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI has specifically designated certain components as meeting ISO/TS16949 requirements, mainly for automotive use. In any case of use of non-designated products, TI will not be responsible for any failure to meet ISO/TS16949.

Products		**Applications**	
Audio	www.ti.com/audio	Automotive and Transportation	www.ti.com/automotive
Amplifiers	amplifier.ti.com	Communications and Telecom	www.ti.com/communications
Data Converters	dataconverter.ti.com	Computers and Peripherals	www.ti.com/computers
DLP® Products	www.dlp.com	Consumer Electronics	www.ti.com/consumer-apps
DSP	dsp.ti.com	Energy and Lighting	www.ti.com/energy
Clocks and Timers	www.ti.com/clocks	Industrial	www.ti.com/industrial
Interface	interface.ti.com	Medical	www.ti.com/medical
Logic	logic.ti.com	Security	www.ti.com/security
Power Mgmt	power.ti.com	Space, Avionics and Defense	www.ti.com/space-avionics-defense
Microcontrollers	microcontroller.ti.com	Video and Imaging	www.ti.com/video
RFID	www.ti-rfid.com		
OMAP Applications Processors	www.ti.com/omap	**TI E2E Community**	e2e.ti.com
Wireless Connectivity	www.ti.com/wirelessconnectivity		

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