Application Report
SPRAAS0D– March 2011

Using the TMS320DM646x DMSoC Bootloader

Device Applications

ABSTRACT

This document describes the functionality of the TMS320DM6467 Digital Media System-on-Chip (DMSoC) ARM ROM bootloader (RBL) software. The ARM ROM bootloader resides in the ROM of the device beginning at address 0x00008000. The RBL implements methods for booting in the listed modes and uses boot configuration pins to determine boot mode. If an improper boot mode is chosen, or an error is detected during boot from a slave device, the RBL communicates this through the universal asynchronous receiver/transmitter (UART), as the default boot device.

When booting in master mode, the RBL reads boot information from the slave device, as and when required. When booting in slave mode, the RBL depends on the master device to feed boot information, as and when required. Please note that for all boot modes, the RBL disables the watchdog timer for the duration of boot. All applications MUST avoid configuring the watchdog timer during the boot process.

- HPI
- PCI
- EMIFA Direct Boot
- NAND
- I2C (master)
- SPI (master)
- UART

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www.ti.com/lit/zip/SPRAAS0.

Contents

List of Figures

List of Tables

1 Boot Mode Description

The TMS320DM6467 DMSoC has BTMODE and PCIEN pins for boot configuration (see Table 1). The RBL reads these pins and branches to the appropriate code to implement the selected boot.

Table 1. Boot Mode Description

BTMODE[3:0]	PCIEN	ARM-Boot Mode	Hardware Boot	Notes
0000	0 or 1	Emulation Boot	ROM	
0001	0 or 1	Reserved	ROM	Default to UART0 Boot
0010	0	HPI Boot 16-Bit	ROM	
	1	PCI Boot Without Auto-Initialization	ROM	
0011	0	HPI Boot 32-Bit	ROM	
	1	PCI Boot With Auto-Initialization	ROM	
0100	0	EMIFA Direct Boot	EMIFA	
	1	Error	ROM	Default to UART0 Boot
0101	0	Reserved	ROM	Default to UART0 Boot
	1			
0110	0	I2C Boot	ROM	TMS320DM6467 DMSoC is the I2C master
	0			
	1			
	1			
0111	0	NAND Flash Boot	ROM	
	1	Error	ROM	Default to UART0 Boot
1000	0	UART0 Boot	ROM	
	1			
1001	0 or 1	Reserved	ROM	
1010 1011 1100 1101	0 1	Reserved	ROM	Default to UART0 Boot
1110	0 or 1	SPI Boot	ROM	TMS320DM6467 DMSoC is the SPI master
1111	0	Reserved	ROM	Default to UART0 Boot
	1			

Basically, the PCIEN state does not affect the bootmode. The bootmode does not use EMIFA/PCI such as emulation boot. However, the $\overline{\text{PCI\_RST}}$ is multiplexed with EM_A22/ATA_H2/GP13. It may not tie high, especially the EM_A22/ATA_H2, which is output when you use the EMIFA/ATA. In this situation, if you set PCIEN = 1, the $\overline{\text{PCI\_RST}}$ is not driven high; the TMS320DM6467 DMSoC is in reset state. As a result, the TMS320DM6467 DMSoC can not boot. It is recommended to set PCIEN = 0 when you do not use PCI.

1.1 Terms and Abbreviations

AEMIF	Asynchronous External Memory Interface
AIS	Application Image Script
BL	Boot Loader (referring to the bootloader in this text)
Bootloader	SW/Code for TMS320DM6467 DMSoC ROM Bootloader
DSP	Digital Signal Processor
EEPROM	Electrically Erasable Programmable Read-Only Memory
GPIO	eneral-Purpose Input/Output
HINT	Host Interrupt
HPI	Host Port Interface
I2C	Inter-Integrated Circuit
PCI	Peripheral Component Interconnect
RBL	ROM Boot Loader
ROM	Read-Only Memory
SPI	Serial Peripheral Interface
UBL	User Boot Loader

2 Memory Map

Place the UBL in the range of 0x00000020 to 0x000074FF, as shown in Figure 1. 0x00000000 to 0x0000001F and 0x00007500 to 0x00007FFF are reserved for the ROM bootloader. ARM internal memory in the TMS320DM6467 DMSoC is physically the same for both ITCM and DTCM. This restriction is also applied for the DTCM area. (The address offset to the top of DTCM start address is 0x00010000).

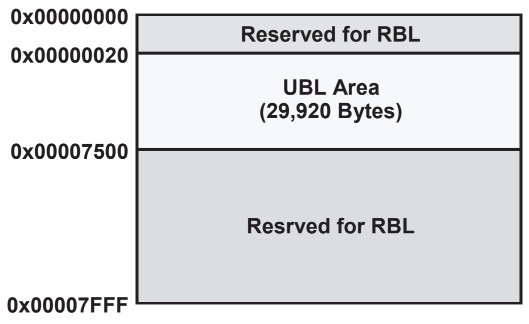

Figure 1. RBL Memory Map

3 Boot Modes

This section discusses various boot modes.

3.1 Emulation-Boot Mode

In this boot mode, the RBL executes the infinity loop. The debugger, such as Code Controller Studio, is responsible for performing any code download and controlling the device.

3.2 HPI Boot 16/32-Bit Mode

In HPI boot 16/32-bit mode, the RBL performs the sequence shown in Figure 2:

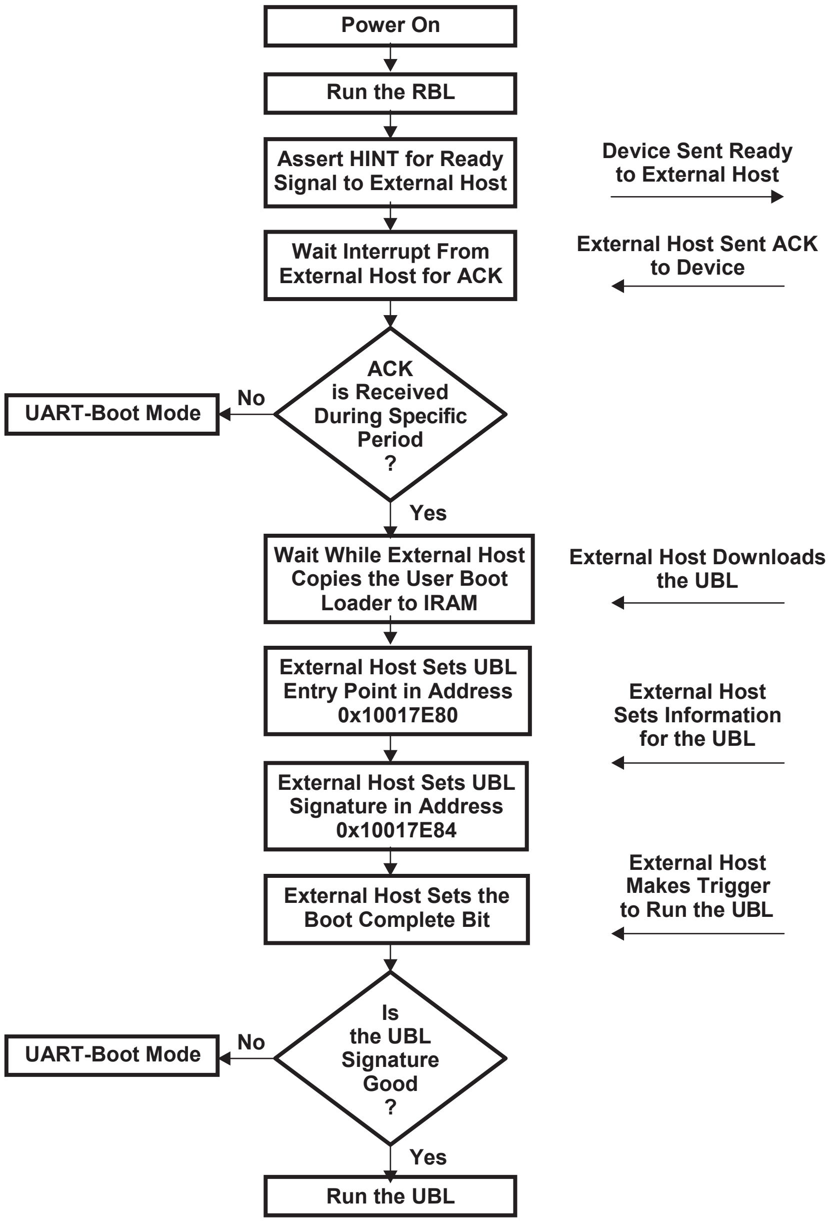
Power On
Run the RBL
Assert HINT for Ready Signal to External Host
Device Sent Ready to External Host
Wait Interrupt From External Host for ACK
External Host Sent ACK to Device
ACK is Received During Specific Period ?
No
UART-Boot Mode
Yes
Wait While External Host Copies the User Boot Loader to IRAM
External Host Downloads the UBL
External Host Sets UBL Entry Point in Address 0x10017E80
External Host Sets Information for the UBL
External Host Sets UBL Signature in Address 0x10017E84
External Host Makes Trigger to Run the UBL
External Host Sets the Boot Complete Bit
Is the UBL Signature Good ?
No
UART-Boot Mode
Yes
Run the UBL

Figure 2. HPI-Boot Sequence

The RBL signals to the Host that the TMS320DM6467 DMSoC is ready via the HINT signal. The RBL waits 50 seconds for the HINT to ACK. There is no timeout once the RBL drops into a polling loop for the external Host to complete the code download. The boot complete (BC) bit is located at bit 0 of the Boot Status Register (BOOTSTAT). The BOOTSTAT register is located at 0x01C40010. The entry point must be specified in terms of the ARM Instruction TCM area and should be in the range of 0x000000020–0x000074FF. The external Host should download the UBL between 0x10010020 and 0x10017500, as shown in Figure 3.

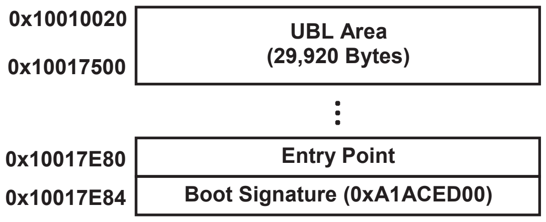

Figure 3. Memory Map

3.3 PCI-Boot Mode

In PCI-boot mode, the RBL performs the sequence shown in Figure 4:

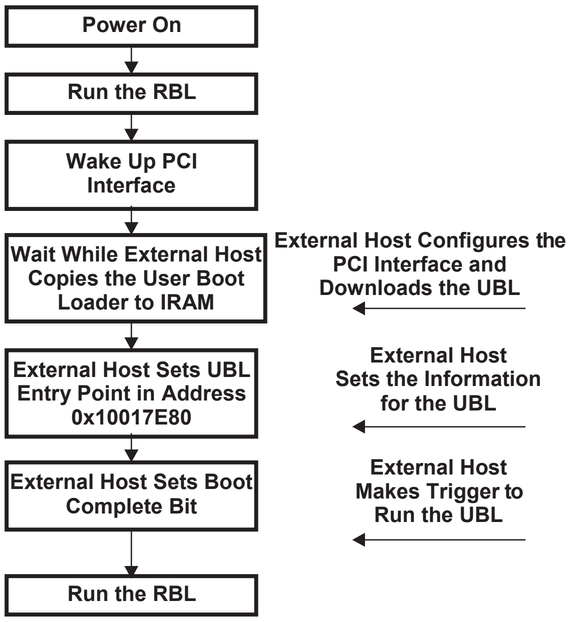
Power On
Run the RBL
Wake Up PCI Interface
Wait While External Host Copies the User Boot Loader to IRAM
External Host Configures the PCI Interface and Downloads the UBL
External Host Sets UBL Entry Point in Address 0x10017E80
External Host Sets the Information for the UBL
External Host Sets Boot Complete Bit
External Host Makes Trigger to Run the UBL
Run the RBL

Figure 4. PCI-Boot Sequence

The boot complete (BC) bit is located at bit 0 of the BOOTSTAT register. The BOOTSTAT register is located at 0x01C40010. The entry point must be specified in terms of the ARM Instruction TCM area and should be in the range of 0x00000020–0x000074FF. The external Host should download the UBL between 0x10010020 and 0x10017500, as shown in Figure 5.

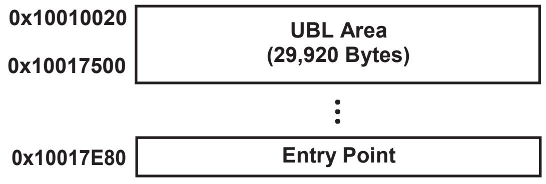

Figure 5. Memory Map

The PCI module provides full visibility for a PCI host into the DM6467 DMSoC memory through six sets of PCI Slave Base Address Translation Registers (PCIBAR0TRL-PCIBAR5TRL) and PCI Base Address Mask Registers (PCIBAR0MSK-PCIBAR5MSK). The ARM can use any of these sets of registers to map any memory region or memory-mapped registers (MMRs) to the PCI memory-map. These registers can be configured by software at any time. The default values of these registers provide the mapping shown in Table 2.

Table 2. PCI Base Address

Base Address	Window Size	Memory Space
0	32KB [(1)]	ARM Internal Memory
1	16KB	DDR2 Control Registers
2	4MB	Chip-Level MMRs
3	128KB	GEM L2 RAM
4	8MB	DDR2 Memory
5	8MB	DDR2 Memory

(1) The RBL changes Window Size for Base Address 0. The PCIIF IP's default value of Window Size for Base Address 0 is 16KB.

3.4 PCI Boot With Auto-Initialization Mode

In PCI boot with auto-initialization mode, the RBL performs the sequence shown in Figure 6:

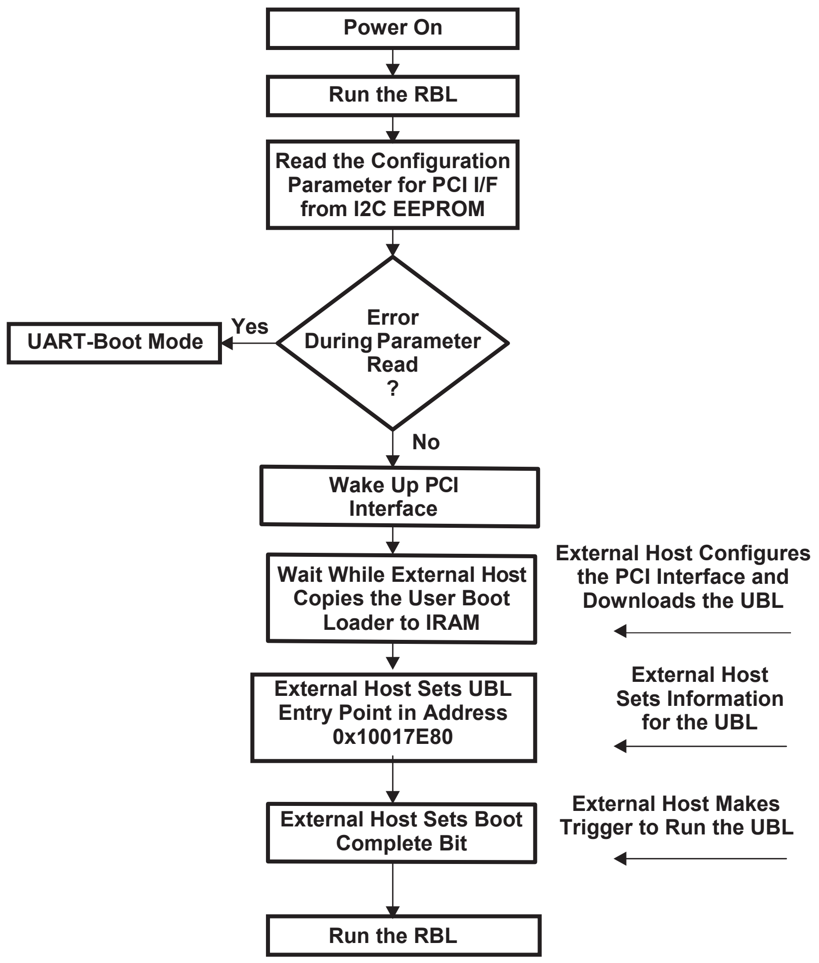
Power On
Run the RBL
Read the Configuration Parameter for PCI I/F from I2C EEPROM
Error During Parameter Read ?
Yes
UART-Boot Mode
No
Wake Up PCI Interface
Wait While External Host Copies the User Boot Loader to IRAM
External Host Configures the PCI Interface and Downloads the UBL
External Host Sets UBL Entry Point in Address 0x10017E80
External Host Sets Information for the UBL
External Host Sets Boot Complete Bit
External Host Makes Trigger to Run the UBL
Run the RBL

Figure 6. PCI With I2C-Boot Sequence

The boot complete (BC) bit is located at bit 0 of the BOOTSTAT register. The BOOTSTAT register is located at 0x01C40010. The entry point must be specified in terms of the ARM Instruction TCM area and should be in the range of 0x00000020–0x000074FF. The external Host should download the UBL between 0x10010020 and 0x10017500, as shown in Figure 7.

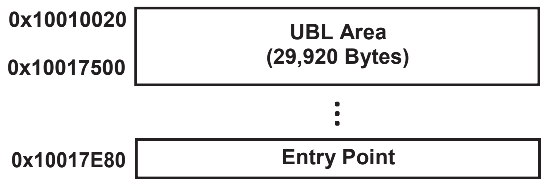

Figure 7. Memory Map

3.4.1 I2C EEPROM Memory Layout for PCI Configuration Parameters

The RBL requires big-endian format for the data stored in the I2C EEPROM. Byte addresses 0x400 through 0x41B of the I2C EEPROM are reserved for auto-initialization of PCI configuration registers. The remaining locations are not used for auto-initialization and can be used for storing other data. Table 3 summarizes the I2C EEPROM memory layout, as required for PCI auto-initialization.

Table 3. I2C EEPROM Memory Layout for PCI Configuration Parameters

Byte Address	Contents
0x400	Vender ID [15:8]
0x401	Vender ID [7:0]
0x402	Device ID [15:8]
0x403	Device ID [7:0]
0x404	Class code [7:0]
0x405	Revision ID [7:0]
0x406	Class code [23:16]
0x407	Class code [15:8]
0x408	Subsystem vender ID [15:8]
0x409	Subsystem vender ID [7:0]
0x40A	Subsystem ID [15:8]
0x40B	Subsystem ID [7:0]
0x40C	Max_Latency
0x40D	Min_Grant
0x40E-0x418	Reserved(use 00h)
0x419	Checksum [15:8]
0x41A	Checksum [7:0]

3.5 EMIFA-Boot Mode

Normally, the asynchronous EMIF (EMIFA) boot is automatically handled without the RBL.

If the EMIFA boot is selected, the boot controller in the system module drives the ARM926's INITRAM input low. This causes the ARM to attempt an instruction fetch over its BIU from address 0x00000000. The Boot Branch Injector logic, within the ARM_SS, intercepts this fetch and injects a *Branch to 0x02000000* instruction into the pipeline. Once outside the ARM_SS, chip-level ARM Instruction Address Modification logic inserts a 1 on bit 30 of the VBUSP address bus to modify the access to address 0x42000000, which is the start of the EMIFA CS2 memory region.

Code within the EMIFA memory should execute a branch to the actual EMIFA address, and then disable the Instruction Address Modification logic by clearing the ADDRMOD bit in the ARM Boot Configuration Register (ARMBOOT) of the System Module.

Example code for branch to the actual EMIFA address is as shown below:

```
MOV R1, #0x42000000 ADD R1, R1, #0x10 MOV PC, R1 MOV R0, R0
```

3.6 NAND-Boot Mode

The outline of operations followed in the NAND mode is depicted in Figure 8.

The NAND-boot mode assumes the NAND is located on the EM_CS2 interface, whose bus width at reset is controlled by the CS2BW pin. The RBL reads the state of the CS2BW pin to determine which bus width it will use when reading data from the NAND.

The Device ID is read from the NAND device; any necessary access information, such as the block and page sizes, etc., are obtained from the device information table in the RBL. Then, the RBL searches for the UBL descriptor in page 0 of the block, after the CIS/IDI block (block 1). A specific command sent to the NAND device retrieves the device ID and information about the device from the table in the RBL.

If a valid UBL, as determined by reading a proper UBL signature, is not found here, the next block is searched. This search continues for up to five blocks. The provision for additional searching is made in case the first few consecutive blocks are marked as bad, i.e., they have errors. Feedback from customers indicates that searching five blocks is sufficient to handle the errors found in virtually all NAND devices. If no valid UBL signature is found in the search, the RBL reverts to the UART-boot mode.

The RBL copies the UBL into ARM internal RAM, starting at 0x0000:0020. Note that the actual copy is made of the lower 30KB of the TCM Data area: 0x0020–0x74FF.

The NAND RBL uses the hardware error detection capability and checksums embedded within the NAND to determine if a read error occurs when reading the UBL. If a read error occurs, the UBL immediately halts the copy from NAND, and the RBL continues to search the block following that block in which the magic number was found for another instance of a magic number. When a magic number is found, the process is repeated. Using this retry process, the magic number and UBL can be duplicated up to five times, giving significant redundancy and error resilience to NAND read errors.

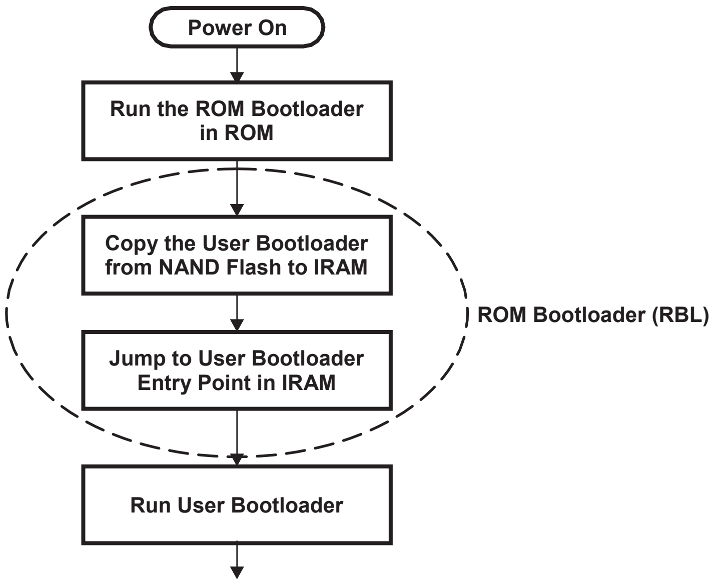
Power On
Run the ROM Bootloader in ROM
Copy the User Bootloader from NAND Flash to IRAM
Jump to User Bootloader Entry Point in IRAM
ROM Bootloader (RBL)
Run User Bootloader

Figure 8. NAND-Boot Sequence

3.6.1 UBL Descriptor

Parameters to be stored about the UBL in the block after the CIS/IDI block. All parameters are 32-bit value:

0x00000000	Boot Signature (0xA1ACED00)
0x00000004	UBL Entry Point (4 Bytes)
0x00000008	Code Size (Pages) (4 Bytes)
0x0000000C	UBL Starting Block # (4 Bytes)
0x00000010	UBL Starting Page # (4 Bytes)

Figure 9. Boot Parameters

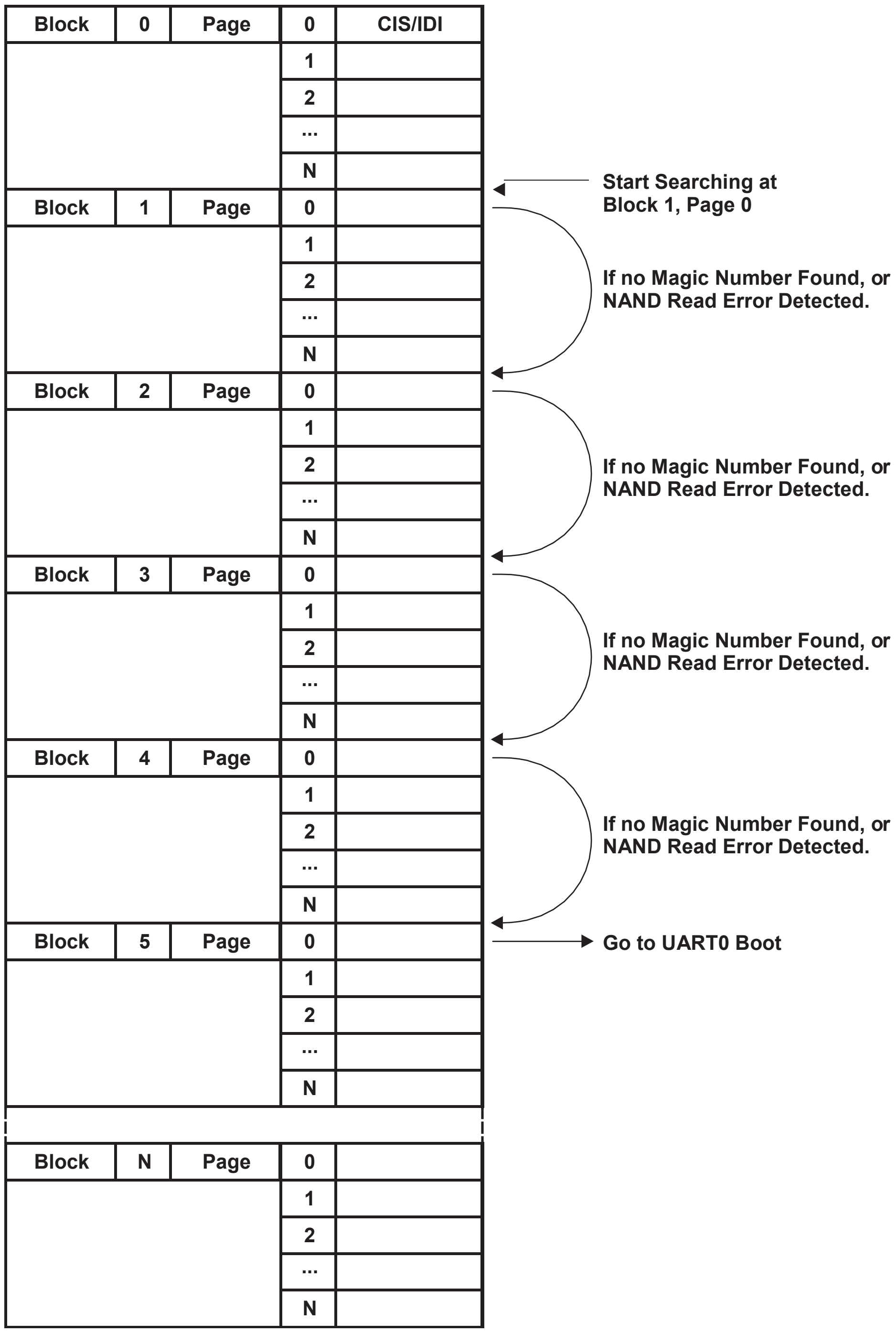
Block 0 Page 0 CIS/IDI
1
2
...
N
Block 1 Page 0
Block 2 Page 0
Block 3 Page 0
Block 4 Page 0
Block 5 Page 0
Block N Page 0
Start Searching at Block 1, Page 0
If no Magic Number Found, or NAND Read Error Detected.
If no Magic Number Found, or NAND Read Error Detected.
If no Magic Number Found, or NAND Read Error Detected.
If no Magic Number Found, or NAND Read Error Detected.
Go to UART0 Boot

Figure 10. Boot Parameter Search

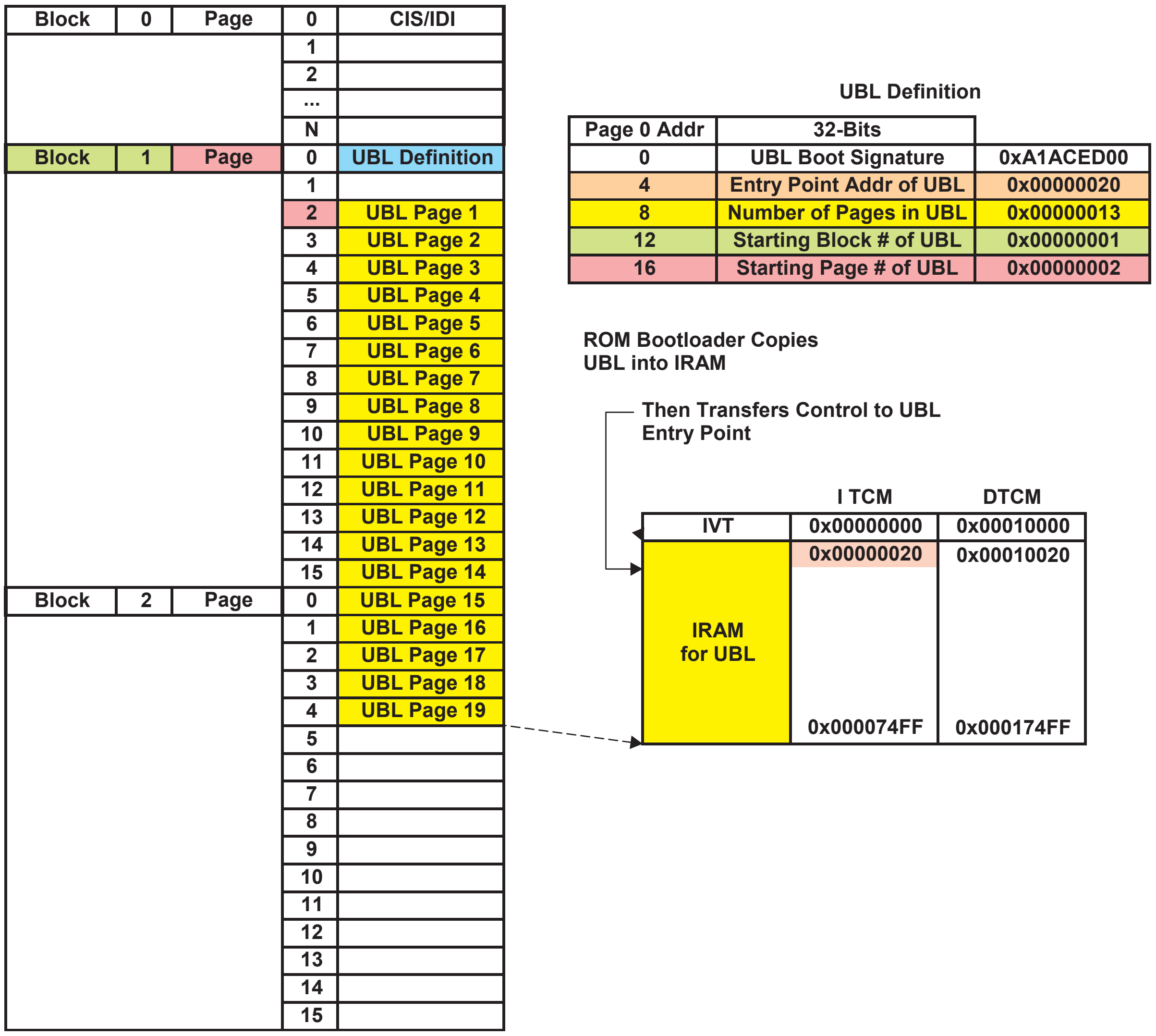

Figure 11. Example

3.6.2 NAND Device IDs Supported

The list of NAND IDs and characteristics supported by the RBL is given in Table 4.

Table 4. Support Device ID List

Device ID	Number of Pages Per Block	Bytes Per Page (including extra data)	Block Shift Value (for address)	Number of Address Cycles
0x39	16	512+16	12	3
0x6B	16	512+16	12	3
0xE3	16	512+16	12	3
0xE5	16	512+16	12	3
0xE6	16	512+16	12	3
0x33	32	512+16	13	3
0x35	32	512+16	13	3
0x73	32	512+16	13	3
0x75	32	512+16	13	3
0x36	32	512+16	13	4

Table 4. Support Device ID List (continued)

Device ID	Number of Pages Per Block	Bytes Per Page (including extra data)	Block Shift Value (for address)	Number of Address Cycles
0x46	32	512+16	13	4
0x56	32	512+16	13	4
0x71	32	512+16	13	4
0x74	32	512+16	13	4
0x76	32	512+16	13	4
0x79	32	512+16	13	4
0xA1	64	2048+64	22	4
0xB1	64	2048+64	22	4
0xC1	64	2048+64	22	4
0xF1	64	2048+64	22	4
0xAA	64	2048+64	22	5
0xAC	64	2048+64	22	5
0xDA	64	2048+64	22	5
0xDC	64	2048+64	22	5

3.6.3 NAND Flash Connection

The NAND Flash should connect at the CS2 space, and for work with ATA. The RBL uses EM_A16/17 for ALE/CLE as shown in Figure 12.

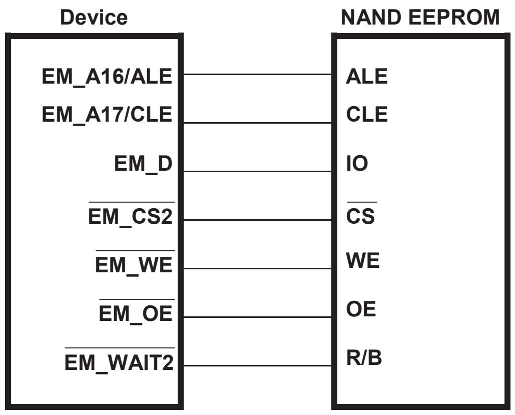
Device
NAND EEPROM
EM_A16/ALE
EM_A17/CLE
EM_D
$\overline{\text{EM\_CS2}}$
$\overline{\text{EM\_WE}}$
$\overline{\text{EM\_OE}}$
$\overline{\text{EM\_WAIT2}}$
ALE
CLE
IO
$\overline{\text{CS}}$
WE
OE
R/B

Figure 12. NAND Flash Connection

NOTE: The TMS320DM6467 DMSoC does not support NAND Flashes that require the chip select to stay low during the t_R time for a read.

3.6.4 Limitation of NAND Flash Device Selection for NAND Boot

The RBL has the following limitations in selecting the NAND Flash device for NAND boot.

- TMS320DM6467 doesn't support non-$\overline{\text{CE}}$ don't-care NAND Flash devices.
- Limitation of driving BUSY signal only at t_R time for read

The NAND Flash should meet the above two limitations to use it as a boot device. If the NAND Flash is not used as a boot device, then any type of NAND Flash can be used with GPIO control of the $\overline{\text{CE}}$ to overcome the above limitations.

3.6.4.1 TMS320DM6467 Doesn't Support Non-$\overline{\text{CE}}$ Don't-Care NAND Flash Devices

According to the *TMS320DM646x DMSoC Asynchronous External Memory Interface (EMIF) User's Guide* (SPRUEQ7), the DM6467 AEMIF does not support NAND Flash devices that require the chip select signal to remain low during the t_R time for a read (non-$\overline{\text{CE}}$ don't-care mode).

On the other hand, non-$\overline{\text{CE}}$ don't-care type NAND Flash devices exist in the DM6467 boot supported NAND Flash device ID list (Table 4). This is because some non-$\overline{\text{CE}}$ don't-care NAND Flash devices and $\overline{\text{CE}}$ don't-care Flash devices have the same device ID. For example, both SAMSUNG K9F1208U0C and STMicro NAND512W3A2C have the same device ID (0x76). The K9F1208U0C requires the chip select signal to remain low during t_R time for a read (non-$\overline{\text{CE}}$ don't-care), but NAND512W3A2C does not.

Also note that the same NAND Flash device with different package types behave differently. For example, the SAMSUNG K9F1208U has two types of package: TSOP and FBGA. The TSOP package requires the chip select signal to remain low (non-$\overline{\text{CE}}$ don't-care) during the t_R time for a read, but the FBGA package does not. This means that the device supports K9F1208U of FBGA package type.

Before choosing a NAND Flash as a boot device, look at the Device Operation Section of the device-specific data sheet to see if it requires the chip select signal to remain low during the t_R time for a read or not. Figure 13 is a read operation of the K9F1208X0C Flash device and is from its datasheet. For K9F1208X0C-P, $\overline{\text{CE}}$ must be held low during t_R. Figure 14 is for STMicro NAND512W3A2C and the $\overline{\text{E}}$ (chip select signal) during t_R time is don't care. DM6467 supports NAND512W3A2C, but not K9F128X0C-P.

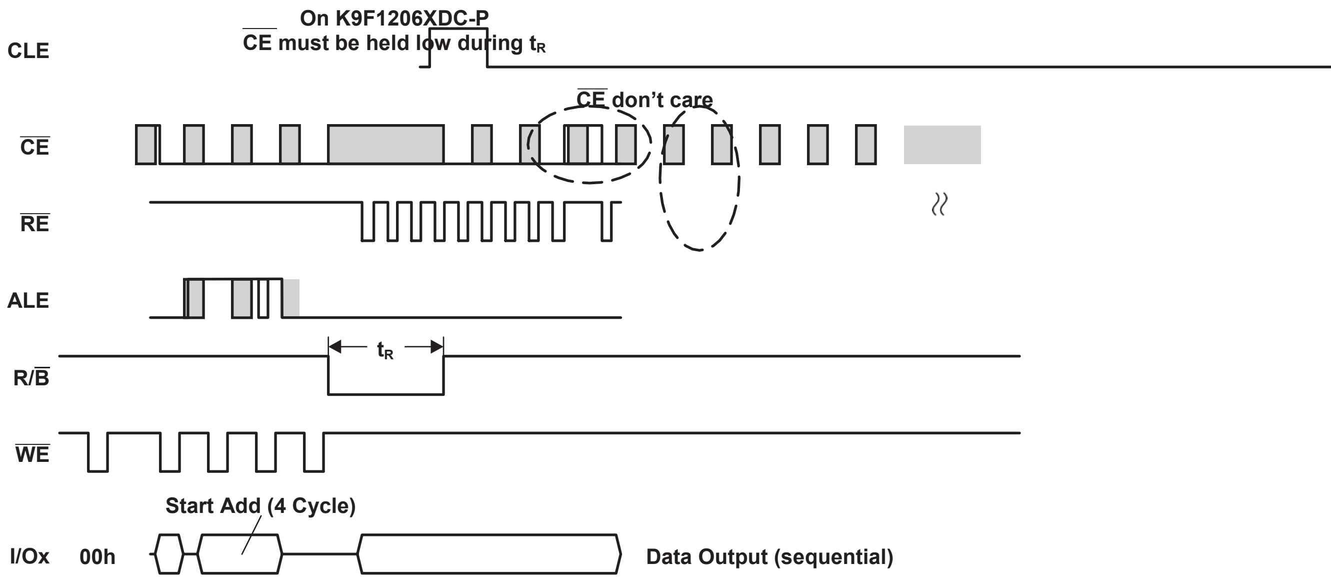

Figure 13. SAMSUNG K9F1208X0C Read Operation

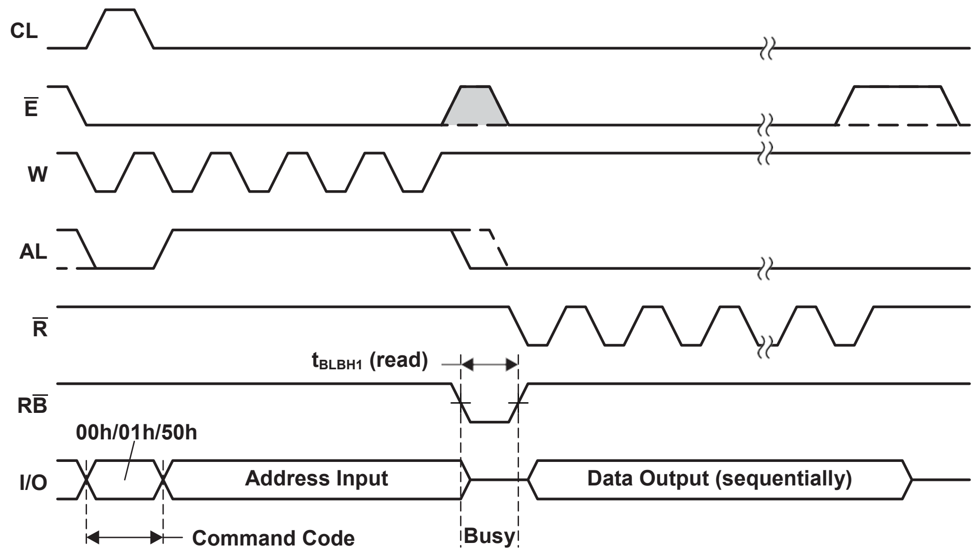

Figure 14. STMicro NAND512W3A2C Read Operation

3.6.4.2 *Limitation of Driving BUSY Signal Only at t_R Time for a Read*

The RBL requires that the NAND Flash does not drive the BUSY signal expect at t_R time for a read. See the NAND Flash device-specific data sheet before using it as a NAND boot device to make sure that it meets this limitation.

The NAND Flash device, that does not require $\overline{CE}$ to remain low during the t_R time for read, also meets this requirement. If you want to use a non-$\overline{CE}$ don't-care Flash device by adding a glue logic to fake the $\overline{CE}$ signal to stay low during the t_R time for a read, then its also needed to fake the BUSY signal if that NAND Flash drives the BUSY signal expect at t_R time.

3.6.4.3 *NAND Boot Advisories*

After selecting the proper NAND Flash, make sure that the following advisories are met for NAND boot to work properly:

- Write Protect Should be Enabled for NAND Flash Boot
- EM_WAIT[5:3] Signals Should Not Toggle During NAND Boot

For more details on these advisories, see the *TMS320DM6467 Digital Media System-on-Chip (DMSoC) Silicon Revisions 1.1 and 1.0 Silicon Errata* (SPRZ251).

3.7 UART0-Boot Mode

This section discusses the UART-boot mode.

3.7.1 UART Setting for External Host

The TMS320DM6467 DMSoC UART-boot mode uses the following settings:

Speed: 115.2Kbps,
Bit length: 8-bit,
Parity: Non-Parity,
Stop: One Stop Bit

3.7.2 UART0 Boot Process

Figure 15 shows how the RBL implements the serial boot process. After initialization, there are three main receive sequences: ACK, 1KB CRC32 table, and UBL. For each receive sequence, the time-out check in RBL is not shown for simplicity. If during the sequence, the timeout value is reached, the serial-boot mode restarts from the beginning which sends out the BOOTME message. The error checking behavior for the UART receive mode is the same. For each byte received, if there's an error, the RBL restarts from the beginning.

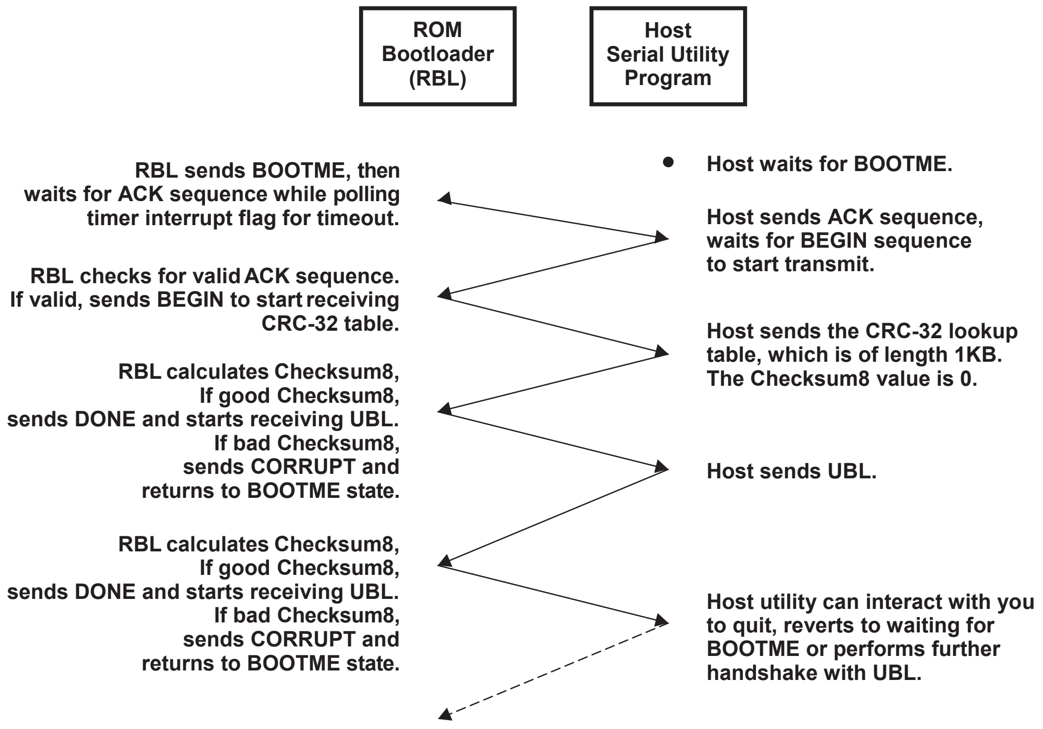
ROM Bootloader (RBL)
Host Serial Utility Program
Host waits for BOOTME.
RBL sends BOOTME, then waits for ACK sequence while polling timer interrupt flag for timeout.
Host sends ACK sequence, waits for BEGIN sequence to start transmit.
RBL checks for valid ACK sequence. If valid, sends BEGIN to start receiving CRC-32 table.
Host sends the CRC-32 lookup table, which is of length 1KB. The Checksum8 value is 0.
RBL calculates Checksum8, If good Checksum8, sends DONE and starts receiving UBL. If bad Checksum8, sends CORRUPT and returns to BOOTME state.
Host sends UBL.
RBL calculates Checksum8, If good Checksum8, sends DONE and starts receiving UBL. If bad Checksum8, sends CORRUPT and returns to BOOTME state.
Host utility can interact with you to quit, reverts to waiting for BOOTME or performs further handshake with UBL.

Figure 15. UART-Boot Sequence

3.7.3 UART0 Bootloader Data Sequence

The UART0 bootloader data sequences consist of handshake messages, the UBL header, and the UBL payload itself. The message uses a fixed 8-byte ASCII string; also included is the null string terminator. Short messages have leading spaces besides the null.

Table 5 lists the values for the handshake sequences and the header for the UBL.

Table 5. Value for the Handshake Sequence

Sequence	Value	Usage
BOOTME	^BOOTME/0	Notify host utility serial boot mode begins. This is an 8-byte ASCII value. ^ is a space
ACK	^^^^ACK/0 UBL 8-Byte Checksum UBL 4-Byte Count UBO 4-Byte ARM Physical Start Address TBD 4-Byte Zeros	For host utility to respond within time-out period by sending a 28-byte header to prepare for reception of UBL. The checksum is a 32-bit checksum. Note that the start address is where the RBL jumps to after the downloading process (i.e., UBL entry point).
BEGIN	^^BEGIN/0	RBL signals host utility to begin transmission of UBL.
DONE	^^^DONE/0	RBL signals host utility that data was received OK and the transfer can be terminated.
BAD ADDR	BADADDR/0	Bad start address received.
BAD COUNT	^BADCNT/0	Bad count received.
CORRUPT	CORRUPT/0	RBL signals host utility that there is an error with the transmission. Normally, the host utility should ask you to reset the board.
UBL	variable	The format for the UBL is the same as NAND boot.

3.7.4 UBL Image Generation

The CRC32 checksum value is calculated for the UBL data and passed by the Host serial utility. The polynomial used for CRC32 is:

X^32+X^26+X^23+X^22+X^16+X^12+X^11+X^10+X^8+X^7+X^5+X^4+X^2+X^1+X^0.

The CRC32 results in a 32-bit value (4bytes), through which the host serial utility transmits eight characters (bytes); this is shown in the following example.

For a given UBL data, let the calculated checksum (CRC32) value be 0xffaa10a1. Then, instead of the host utility transmitting *ascii(0xff) ascii(0xaa) ascii(0x10) ascii(0xa1)*, it will transmit *ffaa10a1*. The RBL will appropriately interpret the eight characters (bytes).

ARM code generation tools can generate the UBL, but the final format is expected in binary memory image format with no headers, etc.

The starting address of the UBL is at 0x0020, and you can use between 0x0020 and 0x74ff.

3.8 I2C Master Boot Mode

The I2C-boot mode requires use of Application Image Script (AIS) as the primary data format for loading code/data. For more detailed information regarding AIS, see *Using the TMS320C642x Bootloader* (SPRAAK5).

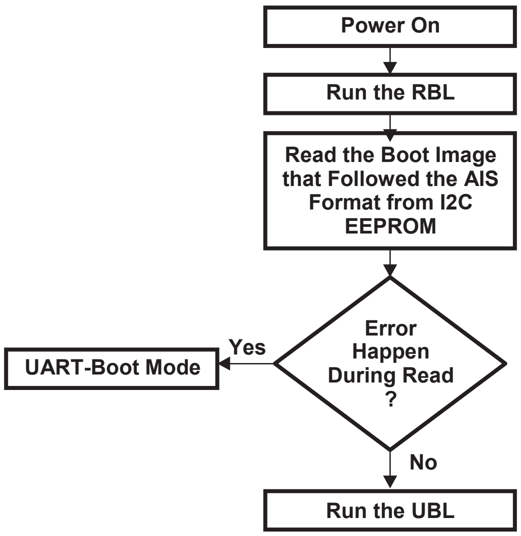
Power On
Run the RBL
Read the Boot Image that Followed the AIS Format from I2C EEPROM
Error Happen During Read ?
Yes
UART-Boot Mode
No
Run the UBL

Figure 16. I2C-Boot Sequence

The I2C bootloader only supports the following AIS commands:

- Section Load
- JUMP_CLOSE Command
- Enable CRC
- Disable CRC
- Request CRC

The I2C bootloader assumes I2C EEPROM specification shown in Table 6.

Table 6. I2C EEPROM Specification

Slave Address	0x50
Clock Frequency that EEPROM supported	90 kHz

Table 7. Example I2C Boot Image

Address	Data	Comment
0	0x00000002	Dummy
4	0x41504954	Magic Number
8	0x58535903	Enable CRC
C	0x58535901	Section Load
10	0x0000200C	Section Start Address
14	0x00000008	Section Length
18	0x4700A000	Data
1C	0xEAFFFFFE	Data
20	0x58535902	Request CRC
24	0xD1AE239C	CRC Value
28	0xFFFFFFE0	Offset Address
2C	0x58535901	Section Load
30	0x00002000	Section Start Address

Table 7. Example I2C Boot Image (continued)

Address	Data	Comment
34	0x0000000C	Section Length
38	0x0000000A	Data
3C	0x0000000B	Data
40	0x0000000C	Data
44	0x58535902	Request CRC
48	0x6B4ABA9D	CRC Value
4C	0xFFFFFFDC	Offset Address
50	0x58535906	Jump_Close
54	0x0000200C	Jump Address

The I2C boot mode supports 7-bit peripheral device address and 16-bit data word address.

3.9 SPI Master Boot

Like the I2C-boot mode, the SPI-boot mode also uses the Application Image Script (AIS) as the primary data format for loading code/data.

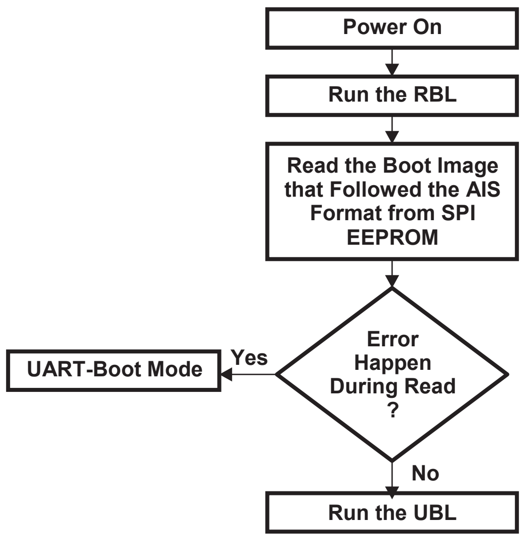
Power On
Run the RBL
Read the Boot Image that Followed the AIS Format from SPI EEPROM
Error Happen During Read ?
Yes
UART-Boot Mode
No
Run the UBL

Figure 17. SPI-Boot Sequence

The SPI bootloader only supports the following AIS commands:

- Section Load
- JUMP_CLOSE Command
- Enable CRC
- Disable CRC
- Request CRC

The SPI bootloader communicates with the SPI EEPROM using the following information:

- SPI_CLK, SPI_SIMO, SPI_SOMI, $\overline{\text{SPI\_CS0}}$
- The SPI bootloader does not use SPI_EN.
- The SPI bootloader drives SPI_CLK at 990 kHz.
- The SPI bootloader uses only 16-bit address support.

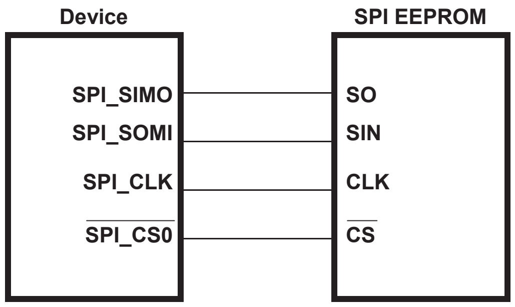

Figure 18. SPI Connection

Table 8. Example Boot Image

Address	Data	Comment
0	0x05040302	Dummy
4h	0x41504954	Magic Number
8h	0x58535903	Enable CRC
Ch	0x58535901	Section Load
10h	0x0000200C	Section Start Address
14h	0x00000004	Section Length
18h	0xEAFFFFFE	Data
1Ch	0x58535906	Jump_Close
20h	0x0000200C	Jump Address

4 References

- *TMS320DM646x DMSoC Asynchronous External Memory Interface (EMIF) User's Guide* (SPRUEQ7)
- *TMS320DM6467 Digital Media System-on-Chip (DMSoC) Silicon Revisions 1.1 and 1.0 Silicon Errata* (SPRZ251).
- *Using the TMS320C642x Bootloader* (SPRAAK5)

Appendix A AIS Use Cases for I2C and SPI Master-Boot Modes

A.1 AIS Generation Environment

In order to generate an AIS script, you are required to make use of a Perl program and an accompanying executable supplied by TI.

If you need to generate an AIS script on your system for the first time, you are required to have the Perl program installed. The following are recommended steps for ensuring a correct AIS generation environment.

1. Download and install the Perl program.
2. Make sure the following Perl directories exist within the path variable of the operating system (O/S).
   (a) Perl\site\bin
   (b) Perl\bin
3. Add the Perl program extension, pl, to the PATHEXT variable. This is optional and only required if invoking the Perl program without the pl extension.
4. Reboot your system.
5. Required TI utilities (you can place these utilities anywhere where the directory exists, within the path statement)
   (a) genAPI.pl
   (b) ofd6x.exe

A.1.1 I2C and SPI Command Line Entries for Generating AIS File

The following steps demonstrate how to generate an AIS script for the I2C EEPROM.

1. Generate the application (boot) code.
   (a) The internal ARM memory map reserved for the user application is between 0x00000020 to 0x00007500.
   (b) If you have an application that requires larger memory, then you are required to complete the boot in two steps. The first step allows the RBL to boot the UBL. The second step enables the necessary UBL resource and boots the actual application similar to the RBL.

NOTE: The RBL does not enable any peripheral, other than the one used for the selected boot option; it is the responsibility of the application code to eventually configure the device appropriately. This requires the user application to fit within the ARM internal RAM memory since other resources, including DDR, are not enabled.

2. Generate the AIS script for the I2C peripheral in ASCII format.
   (a) The following generates an AIS image script for the I2C peripheral in ASCII format. The I2C writer should take an input file in .txt format and burn the output onto the I2C EEPROM.

```
genAIS.pl –I inputUserFile.out -o outputFile.ais -otype ascii -bootmode i2cmaster -addrsz 16
```

3. Generate the AIS script for the SPI peripheral in binary format.
   (a) The following generates an AIS image script for the SPI peripheral in binary format. The SPI writer should take an input file in binary format and burn the output onto the SPI EEPROM.

   ```
   genAIS.pl –I inputUserFile.out -o outputFile.ais -otype bin -bootmode spimaster -addrsz 16
   ```

NOTE: The I2C and SPI writer tools are Code Composer Studio™ executables (.out files), available as part of the DM6467 Digital Video Software Development Kit (DVSDK) software release at www.ti.com\dvevmupdates. You need Code Composer Studio (version 3.3 or later) and a compatible JTAG emulator to perform this bootloader programming operation. Use the instructions in the *TMS320DM6467 DVEVM Getting Started Guide* (SPRUF88) once the DVSDK is installed. The I2C and SPI writers, in the form of CCS *.out files, can be found under the folder PSP_0x_xx_xx_xxx/bin/dm646x. To program the I2C or SPI bootloader, you need to copy the i2c_eeprom_writer.out or spi_eeprom_writer.out file to a Windows™ PC where Code Composer Studio is installed. For detailed instructions on how to perform these steps using Code Composer Studio, see the *DM646x_Linux_PSP_UserGuide.pdf* under the *docs* subfolder.