Application Report

SPRABL7C–September 2011–Revised April 2016

Using the TMS320C5545/35/34/33/32 Bootloader

ABSTRACT

This application report describes the features of the on-chip ROM for the TMS320C5545/35/34/33/32 device. Included is a description of the bootloader and how to interface with it for each of the possible boot devices, as well as instructions for generating a boot image to store on an external device.

Download the related executable file for the C5545/35/34/33/32 bootloader from:

http://www.ti.com/lit/zip/sprabl7.

Contents

List of Figures

List of Tables

1 Introduction

1.1 On-Chip ROM

The on-chip ROM contains several factory-programmed sections:

- Algorithm tables to be referenced by drivers to reduce application data size
- Application programming interface (API) tables for referencing ROM API functions in applications
- Bootloader program

Table 1. ROM Memory Map

Starting Byte Address	Contents
FE_0000h	LCD Table
FE_0860h	WMA Encode Table
FE_9FA0h	WMA Decode Table
FE_D4C4h	MP3 Table
FE_F978h	Equalization Table
FE_FB14h	API Table
FE_FE9Ch	Bootloader Code (and other built-in API functions)
FF_FEFCh	Bootloader ID
FF_FF00h	Bootloader Interrupt Vector Table

1.2 Bootloader Features

The major features of the bootloader are:

- Booting both encrypted and unencrypted images from supported external devices (see Section 1.3.1).
- Support for unencrypted external devices (see Section 1.3.1).

The bootloader also has the following features:

- Port-addressed register configuration during boot
- Programmable delay during register configuration

The bootloader is always invoked after reset. The function of the bootloader is to transfer user code from an external source to RAM. Once the transfer is completed, the bootloader transfers control to this user code.

1.3 Supported External Devices

The bootloader is responsible for bootloading the code from an unencrypted or encrypted external device.

1.3.1 Unencrypted External Devices

All external devices support unencrypted booting.

1.3.2 Encrypted External Devices

128-bit encrypted booting is available using three encrypted boot image formats:

- Not Bound to Device
- Re-Authorization and Bound to Device
- Bound to Device

All external devices support all three encrypted boot image formats, except McSPI, 24-Bit SPI, UART, and USB do not support Re-Authorization and Bound to Device.

After bootloading from an external device, the bootloader decrypts the boot image, if necessary, and writes it into DSP memory (on-chip or off-chip).

NOTE: SARAM31 (byte address 0x4E000 – 0x4FFFF) is reserved for the bootloader.

2 Bootloader Operation

2.1 Bootloader Initialization

When the bootloader begins execution, it performs some initialization before attempting to load code:

- All peripherals are idled (the bootloader un-idles peripherals as it uses them).
- CPU Clock setup
  - If CLK_SEL = 0, the bootloader powers up the PLL and sets its output frequency to 12.288 MHz (multiply 32 768 Hz RTC Clock by 375).
  - If CLK_SEL = 1, the bootloader bypasses the PLL and uses CLKIN. Note that CLKIN is expected to be 11.2896 MHz, 12.0 MHz, or 12.288 MHz.
- The low-voltage detection circuit is disabled to prevent trim setup (next step) from causing an unnecessary reset.
- The bootloader reads the trim values from the e-fuse farm and writes them into the analog trim registers.

2.2 Boot Devices

Each device has a fixed order in which it checks for a valid boot image on each supported boot device. The device order is 16-bit SPI EEPROM, 24-bit SPI serial flash, I2C EEPROM, and SD/SDHC/eMMC/moviNAND. The first device with a valid boot image will be used to load and execute user code.

If none of these devices has a valid boot image, the bootloader will modify the CPU Clock setup as follows:

- If CLK_SEL = 0, the bootloader powers up the PLL and sets its output frequency to 36.864 MHz (multiply 32 768 Hz RTC Clock by 1125).
- If CLK_SEL = 1, the bootloader powers up the PLL and sets it to multiply CLKIN by 3.

This CPU clock setup change is required to meet the minimum frequency needed by the USB module.

Next the bootloader goes into an endless loop checking for data received on either the UART or USB. If a valid boot image is received from either device, it will be used to load and execute user code. If no valid boot image is received, the bootloader will simply continue to monitor these two devices. During this endless loop, if the time since the trim setup exceeds 200 ms, then the bootloader will re-enable the low-voltage detection circuit. This is done to prevent leaving the low-voltage detection disabled for an extended period of time.

See Section 3 for a description of the valid boot image formats.

The following subsections describe details for each supported boot device.

2.2.1 16-Bit SPI EEPROM

The bootloader supports booting from an SPI EEPROM with the following requirements for the external device:

- The device must support at least a 500-kHz SPI clock.
- The device must be connected to SPI CS0 and act as an SPI slave.
- The device uses two bytes (16 bits) for internal addressing (up to 64KB).
- The device must have the capability to auto-increment its internal address counter to allow sequential reads from the device.
- The device may be connected to either of the two available pin-mappings for SPI. The bootloader attempts to communicate on each SPI pin-mapping, one at a time.

2.2.2 24-Bit SPI Serial Flash

The bootloader supports booting from an SPI Flash with the following requirements for the external device:

- The device must support at least a 500-kHz SPI clock.
- The device must be connected to SPI CS0 and act as an SPI slave.
- The device uses three bytes (24 bits) for internal addressing (up to 16MB).
- The device must have the capability to auto-increment its internal address counter to allow sequential reads from the device.
- The device may be connected to either valid pin-mapping for SPI (there are two distinct pin-mappings available). The bootloader attempts to communicate on each SPI pin-mapping, one at a time.
- Any and all write-protect features must be disabled if re-authoring is needed. The Bootloader will not attempt to disable these features.

2.2.3 I2C EEPROM

The bootloader supports booting from an I2C EEPROM with the following requirements for the external device:

- The device must support the *fast* I2C specification (400 kHz).
- The device must respond to slave address 0x50 (7-bit address).
- The device uses two bytes for internal addressing (up to 64KB).
- The device must have the capability to auto-increment its internal address counter to allow sequential reads from the device.

2.2.4 SD

The bootloader supports booting from an SD device with the following requirements for the external device:

- The device must be connected to the eMMC/SD0 interface.
- The SD device must comply with *SD Specifications Part 1 Physical Layer Simplified Specification v1.1* or *v2.0* and formatted in FAT16/32.
- The SD device must use the SD insecure mode (see SD specification). Note that this does not refer to the boot image security. Boot images can be either the encrypted image or unencrypted image for use with SD.
- The boot image must be in the first partition with a filename of “bootimg.bin”.

2.2.5 eMMC/moviNAND

The bootloader supports booting from an eMMC/moviNAND device with the following requirements for the external device:

- The device must be connected to the eMMC/SD0 interface.
- The eMMC/moviNAND device must comply with eMMC Specification v4.3, or later, and formatted in FAT16/32.
- The boot images can be in either the boot partition (with boot-from-partition enabled) or in the user data area formatted in FAT16/32. The boot image must be in the first data partition with filename “bootimg.bin”. Both an encrypted image and unencrypted image are supported for use with eMMC/moviNAND.
- The bootloader will check for user data partition first. If there is no valid boot image, then it will check for the boot from partition option.

2.2.6 UART

The bootloader supports booting from the UART. The bootloader sets up to receive data using the following UART parameters: 8-bit data, odd-parity, 1 stop-bit, and 57 600 baud rate.

To reduce the probability of a receive error, the external transmitter should set up to use two stop-bits.

Note that this setup may result in data receive errors if CLK_SEL is set to use CLKIN when CLKIN is 11.2896 MHz. The error rate may be reduced by the external device by adding additional time between frames (bytes).

2.2.7 USB

The bootloader supports booting from the USB. The bootloader uses bulk-endpoint 1 (OUT), vendor-id 0x0451, and product-id 0x9010.

2.3 Register Configuration

Once the bootloader detects a valid boot image signature (see Section 3), the first data that is used from the boot image is the optional register configuration data. This data allows the user to set up peripheral port-addressed registers during the boot process and before the code sections are copied. This feature provides the capability to change peripheral registers for specific purposes.

NOTE: Using the register configuration feature to reprogram register settings may cause the bootloader to fail.

Since some register configurations may have an associated latency that must be observed before continuing, a delay feature is also available as part of the register configuration data.

For a description of how to insert register configuration data, including delays, into a boot image, see:

- Section 3.1.1, *Creating an Unencrypted Boot Image*
- Section 3.2.4, *Creating an Encrypted Boot Image*

2.4 Code Sections

After the optional register configuration is complete, the bootloader will copy all of the code sections from the boot image to RAM. Each of these code sections may be actual code or just data. These sections are typically defined by the link-control file.

2.5 Bootloader Completion

After all code sections have been copied, the bootloader will wait to ensure that at least 200 ms have elapsed since the trim setup. Re-enable the low-voltage detection circuit, and then branch to the entry-point address specified in the boot image.

At this point, the bootloader’s task is complete and the user application is executing.

3 Boot Images

The bootloader's primary function is to transfer user code into RAM and then transfer control to this user code. The user code must be formatted into a boot image format supported by the bootloader.

There are two distinct formats supported by the bootloader: An unencrypted boot image format, and an encrypted boot image format. These two formats store the same information, with the only difference being the encrypted format uses encryption to protect the user's data or code.

The following sections describe these two boot image formats and how to create boot images.

3.1 *Unencrypted Boot Image Format*

The unencrypted boot image format contains the following information:

- All user code/data sections to be loaded to RAM
- Register configuration data for setting up peripheral registers prior to loading code
- The entry-point of the user's application
- A boot signature to distinguish unencrypted from encrypted boot images. The unencrypted boot image boot signature is 0x09AA.

Table 2 shows the unencrypted boot image format.

Table 2. Unencrypted Boot Image Format

Word	Content	Valid Data Entries
1	Boot Signature (16-bits)	0x09AA
2	Entry Point (32 bits)	Byte address to begin execution (MSW)
3		Byte address to begin execution (LSW)
4	Register Configuration Count (16 bits, N = count)	1 to 2^{16} - 1
5	Register Config #1 Address in I/O space	Repeated according to register configuration count. Register configuration address is any valid register in I/O space. Address 0xFFFF is reserved as a delay indicator to create delay in between register writes or at end of register writes.
6	Register Config #1 Value or delay count	
7	Register Config #2 Address in I/O space	
8	Register Config #2 Value or delay count	
	. . .	
	Register Config #N Address in I/O space	
4+2N	Register Config #N Value or delay count	0 to 2^{16} - 1
5+2N	Section 1 word count (16 bits) Size is the number of valid (non-pad) data words in block M = (size + 2) rounded up to nearest multiple of 64-bit boundary	1 to 2^{16}-1
6+2N	Destination MSW address to load Section 1 (32 bits)	16-bit word address MSW
7+2N	Destination LSW address to load Section 1 (32 bits)	16-bit word address LSW
8+2N	First word of Section 1 (16 bits)	
	. . .	
5+2N+M	Last word of Section 1, often pad data (padded to 64-bit boundary)	
	. . .	
X	Section X word count (16 bits) Size is the number of valid (non-pad) data words in block N' = (size + 2) rounded up to nearest multiple of 64-bit boundary	1 to 2^{16}-1
X+1	Destination MSW address to load Section X (32 bits)	16-bit word address MSW
X+2	Destination LSW address to load Section X (32 bits)	16-bit word address LSW
X+3	First word of Section X (16 bits)	
	. . .	
X+N'	Last word of Section X, often pad data (padded to 64-bit boundary)	
X+N'+1	Zero word. Note that if more than one source block was read, word X+*N'* shown above would be the last word of the last source block. Each block would have the format shown in the shaded entries.	0x0000

3.1.1 Creating an Unencrypted Boot Image

A boot image can be created using the hex conversion utility (hex55) utility. The hex55 utility is intended for creating a unencrypted boot image.

For detailed information on the available hex conversion utility output formats, see the *TMS320C55x DSP Assembly Language Tools User's Guide* (SPRU280).

Use the hex conversion utility (hex55.exe) revision 4.3.5 or later. Earlier versions may not support the boot table features correctly.

3.1.1.1 Creating an Unencrypted Boot Image Using hex55

To create a boot table for the application (*my_app.out*) with the following conditions:

- Desired boot mode is 8-bit standard serial boot
- No registers are configured during the boot
- No programmed delays will occur during the boot
- Desired output is binary format in a file called *my_app.bin*

Use the following options on the hex conversion utility command line or command file:

```
hex55 -boot -v5505 -serial8 -b -o my_app.bin my_app.out
```

```
-boot                        ;option to create a boot table
-v5505                       ;use C55x boot table format
-serial8                     ;boot mode is 8-bit standard serial boot
-b                           ;desired output format is binary format
-o my_app.bin                ;specify the output filename
my_app.out                   ;specify the input file
```

3.1.1.2 Creating an Unencrypted Boot Image with I/O Register Configuration

To create a boot table for the application *my_app.out* with the following conditions:

- Desired boot mode is from 8-bit standard serial boot
- Configure the register address 0x1C8C with the value 0x0001
- After the register is configured, wait 256 cycles before continuing the boot
- Desired output is binary format in file a called *my_app.bin*

Use the following options on the hex conversion utility command line or command file.

Note: If using the reg_config option, ensure there is no space between the address and value.

```
hex55 -boot -v5505 -serial8 -reg_config 0x1c8c,0x0001 -delay 0x100 -b -o my_app.bin my_app.out
```

```
-boot                        ;option to create a boot table
-v5505                       ;use C55x boot table format
-serial8                     ;boot mode is 8-bit standard serial boot
-reg_config 0x1c8c,0x0001    ;write 0x0001 to peripheral register at address 0x1C8C.
                              This can be repeated to program multiple registers.
-delay 0x100                 ;delay for 256 CPU clock cycles
-b                           ;desired output format is binary format
-o my_app.bin                ;specify the output filename
my_app.out                   ;specify the input file
```

NOTE: Using the register configuration feature to reprogram register settings may cause the bootloader to fail.

3.1.1.3 Section Alignment Restrictions When Using Hex55 Utility

All code sections must be aligned on a word boundary. Sections that are not properly aligned will be flagged by the hex55 utility.

To align a code section, use the *align* command in the linker command file as shown below. Note that if any function included in a code output section has an alignment associated with it (in C via CODE_ALIGN pragma) the whole section will inherit that alignment.

```
.text > ROM PAGE 0 align 2
```

3.1.1.4 DOS Command Line for Generating Boot Image Using Hex55

```
hex55 -boot -v5505 -b -serial8 -o USBKey_LED.bin USBKey_LED.out
```

3.2 Encrypted Boot Image Format

Note: The encrypted and unencrypted boot image format is fully supported on normal versions of this device. TI provides a PC-based Encrypted Boot Image Generator tool and a 128-bit development SEED and KEY here. To obtain the Encrypted Boot Image Generator tool, contact your local TI sales representative.

The encrypted boot image format contains the following information:

- All user code and data sections to be loaded to RAM (encrypted).
- Register configuration data for setting up peripheral registers prior to loading code (encrypted).
- The entry-point of the user’s application (encrypted).
- The file-key used for encrypting the remainder of the file (encrypted). The encryption-seed, device-id (if this is a bound image), and a random number are combined to create the file-key.
- The seed-offset for the encryption-seed used. This is an index into the ROM seed table (value = 0 to 127).
- A boot signature to distinguish encrypted boot images from unencrypted boot images, and to indicate the type of encrypted boot image.

Encrypted booting is supported with three types of encrypted boot images:

- Not Bound to Device
- Re-Authorization and Bound to Device
- Bound to Device

Table 3. Encrypted Boot Image Format

Word	Content	Valid Data Entries
1	Boot Signature (16 bits)	0x09A4, 0x09A5, 0x09A6
2	Encryption Seed Offset (16 bits)	0 to 127 (assigned by TI)
3:10	Encrypted File Key (128 bits)	Safer (SHA1(Seed OR Seed + ID),File Key)
11:14	Two pad words (0x00000) and entry point (32 bits) all encrypted	Safer encrypted byte address to begin execution
15	Register configuration count (16 bits, N = count)	1 to 2^{16} – 1 (not encrypted)
16	Register configuration #1 address in I/O space	Repeated according to register configuration count. Register configuration address is any valid register in I/O space. Address 0xFFFF is reserved as a delay indicator to create delay in between register writes or at end of register writes. This section is encrypted and padded to 64-bit boundary.
17	Register configuration #1 value or delay count	
18	Register configuration #2 address in I/O space	
19	Register configuration #2 value or delay count	
	. . .	
	Register configuration #N address in I/O space	
15+2N	Register configuration #N value or delay count	0 to 2^{16} - 1
16+2N	Section 1 word count (16 bits) Size is the number of valid (non-pad) data words in block M = (size + 2) rounded up to nearest multiple of 64-bit boundary	1 to 2^{16}-1 (not encrypted)
17+2N	Destination MSW address to load Section 1 (32-bits)	Safer encrypted C55x 16-bit word address (0x000060 to 0x097FFF) MSW
18+2N	Destination LSW address to load Section 1 (32-bits)	Safer encrypted C55x 16-bit word address (0x000060 to 0x097FFF) LSW
19+2N	First word of Section 1 (16 bits)	Safer encrypted C55x instructions or any 16 bit wide data value
16+2N+M	. . .	Safer encrypted C55x instructions or any 16 bit wide data value
	Last word of Section 1, often pad data (padded to 64-bit boundary)	Safer encrypted C55x instructions or any 16 bit wide data value
	. . .	
X	Section X word count (16-bits) Size is the number of valid (non-pad) data words in block N' = (size + 2) rounded up to nearest multiple of 64-bit boundary	1 to 2^{16}-1 (not encrypted)
X+1	Destination MSW address to load Section X (32 bits)	Safer encrypted C55x 16-bit word address (0x000060 to 0x097FFF) MSW
X+3	Destination LSW address to load Section X (32 bits)	Safer encrypted C55x 16-bit word address (0x000060 to 0x097FFF) LSW
	First word of Section X (16 bits)	Safer encrypted C55x instructions or any 16 bit wide data value
X+N'	. . .	Safer encrypted C55x instructions or any 16 bit wide data value
	Last word of Section X, often pad data (padded to 64-bit boundary)	Safer encrypted C55x instructions or any 16 bit wide data value
X+N'+1	Zero word. Note that if more than one source block was read, word X+*N'* shown above would be the last word of the last source block. Each block would have the format shown in the shaded entries.	0x0000
X+N'+2	Hash (160 bits)	SHA1-HMAC(SHA1(Seed + ID, File Key + data[11: X+N'+1])

3.2.1 Not Bound to Device

The boot signature is 0x09A5.

The bootloader decrypts the boot image based on the SEED and KEY information (128-bit) every time but does not modify the original boot image. The steps to decrypt are:

1. Detect signature from a Boot Source (0x09A5).
2. Receive SEED_OFFSET (0–127) from the boot image.
3. Receive the SEED from ROM.
4. Receive the encrypted file KEY from the boot image.
5. Decrypt the file KEY with the SEED.
6. Decrypt image with the file KEY and store to RAM.
7. Disable ROM only if valid signature is found. If no valid signature is found, ROM remains unlocked.
8. Jump to Entry Point.

3.2.2 Re-Authorization and Bound to Device

The boot signature is 0x09A6.

The encryption is based on the SEED and KEY information. But at first power up, the bootloader decrypts the boot image based on the SEED and KEY information and then re-encrypts the image with SEED + DEVICE-ID from the device ID registers (DIEIDR0–DIEIDR3) which are unique values per device and the KEY. The original boot image is then written to the boot media. Now the boot image has become “bound to device”. From the second time, the bootloader decrypts the boot image based on the SEED + DEVICE-ID (128-bit) and the KEY.

The steps to decrypt are:

1. Detect signature from a Boot Source (0x09A6).
2. Receive SEED_OFFSET (0–127) from the boot image.
3. Receive the SEED from ROM.
4. Receive the encrypted file KEY from the boot image.
5. Decrypt the file KEY with the SEED.
6. Decrypt image with the file KEY and store to RAM.
7. Read the unique Device ID from Die Registers (128-bit).
8. Encrypt the KEY and boot image with the SEED + Die ID.
9. Write back to the boot media with the new signature (0x09A5).
10. Boot from the new boot image.
11. Disable ROM.
12. Jump to Entry Point.

3.2.3 Bound to Device

The boot signature is 0x09A4.

This encryption is used to bind the boot image to a device with the unique DIEID. Each device has a unique DIEID that must be read before encrypting an image.

Use only the first 64 bits (DIEIDR0-DIEIDR3) of the device DIEID registers are used. Enter the values of the DIEID registers in the Encrypted Boot Image Generator tool, C55BootImage.exe, available by contacting your local TI sales respresentative.

For the example in the following section, we will assume the DIEID values are 0x20230502 and 0x26B2068D. Your DIEID values will be different.

3.2.4 Creating an Encrypted Boot Image

TI provides an encrypted Boot Image Generator tool, C55BootImage.exe, which is a graphical user interface (GUI) that uses *.out COFF files created by Code Composer Studio™ and converts them to encrypted binary files used by the bootloader program.

The following steps explain how to use the GUI.

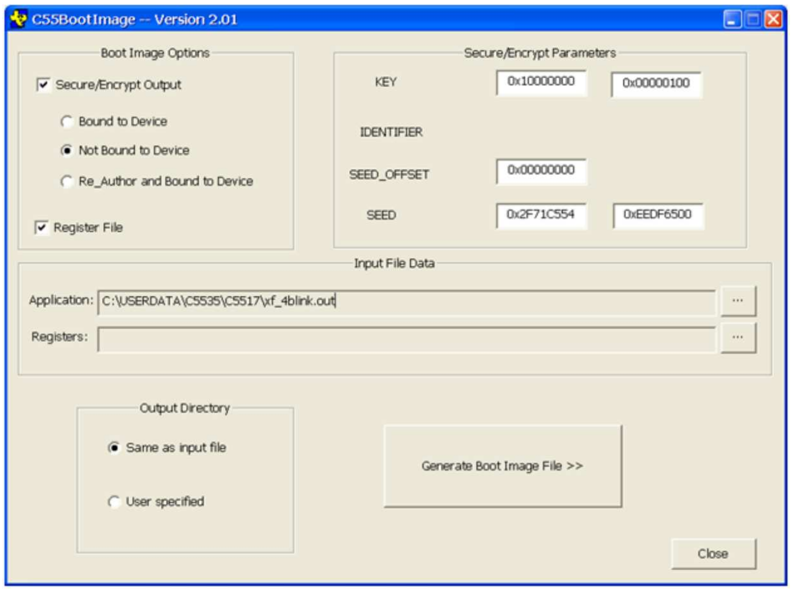
1. Enter Boot Image Option:
   - Encrypt Output
     - Not Bound to Device (Figure 1)
     - Re-Authorization and Bound to Device (Figure 2)
     - Bound to Device (Figure 3)
   - Register File
     - Choose an appropriately formatted register file. The following format is an example of an appropriate format:
       - SIZE: 0x03
       - ADDRESS: 0x1c0c
       - CONTENT: 0x8765
       - ADDRESS: 0xFFFF Note: Address 0xFFFF is to wait the number of cycles before executing next instruction.
       - CONTENT: 0x0100
       - ADDRESS: 0x1c0d
       - CONTENT: 0x4321
2. Enter Encrypt Parameters:
   (a) Type the customer KEY.
   - The key is 64-bit and the key windows are each 32-bit.
   - The customer key for development is 0x1000000000000100.
     - Type 0x10000000 in the left field.
     - Type 0x00000100 in the right field.

   (b) Type the Identifier. The identifier is the unique device ID which the boot image is meant to be bound to (see Figure 3).
   - For Bound to Device only.
   - Type the most significant bit in the left field.
     Assuming the value for DIEID0 is 0x2023 and DIEID1 is 0x0502, type 0x20230502.
   - Type the least significant bit in the right field.
     Assuming the value for DIEID2 is 0x26B2 and DIEID3 is 0x068D, type 0x26B2068D.

   (c) Type the SEED offset. The seed offset for the development key is 0.

   (d) Type the SEED.
   - The seed is 56-bit and the SEED windows are 64-bit.
   - The SEED should be left-aligned: the development SEED is 0x2F71C554EEDF65.
     - Type 0x2F71C554 in the left field.
     - Type 0xEEDF6500 in the right field.
3. Select the Output Directory. The *User specified* option will ask for the directory after pressing *Generate Boot Image File >>*.

C55BootImage can also be used with command-line parameters (e.g., as part of a batch file) instead of the graphical window. To see the command-line parameter formats that are available, with examples, type *C55BootImage /?* from a DOS prompt.


C55BootImage -- Version 2.01

Boot Image Options
- [✓] Secure/Encrypt Output
- ○ Bound to Device
- ● Not Bound to Device
- ○ Re_Author and Bound to Device
- [✓] Register File

Secure/Encrypt Parameters

KEY	0x10000000	0x00000100
IDENTIFIER		
SEED_OFFSET	0x00000000	
SEED	0x2F71C554	0xEEDF6500

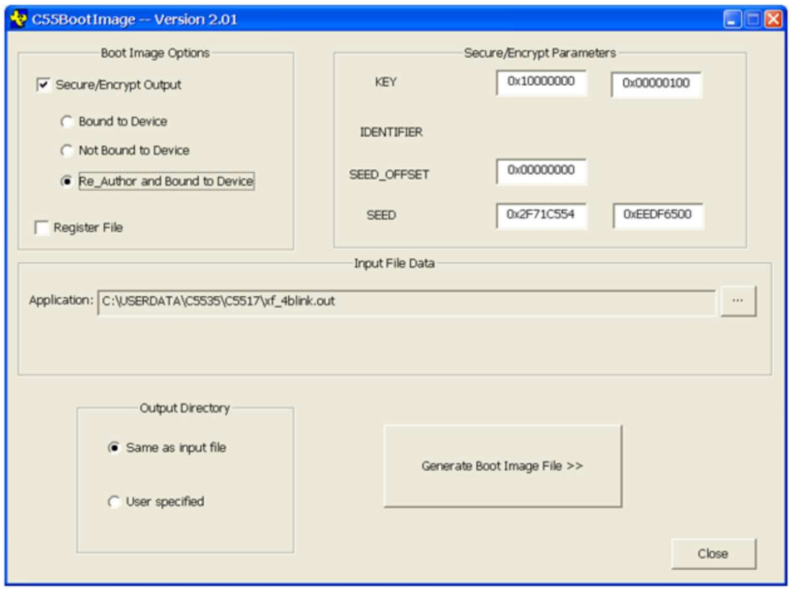
Input File Data
- Application: C:\USERDATA\C5535\C5517\xf_4blink.out
- Registers:

Output Directory
- ● Same as input file
- ○ User specified

Generate Boot Image File >>

Close

Figure 1. Creating an Encrypted Boot Image: Not Bound to Device


C55BootImage -- Version 2.01

Boot Image Options
- [✓] Secure/Encrypt Output
- ○ Bound to Device
- ○ Not Bound to Device
- ● Re_Author and Bound to Device
- [ ] Register File

Secure/Encrypt Parameters

KEY	0x10000000	0x00000100
IDENTIFIER		
SEED_OFFSET	0x00000000	
SEED	0x2F71C554	0xEEDF6500

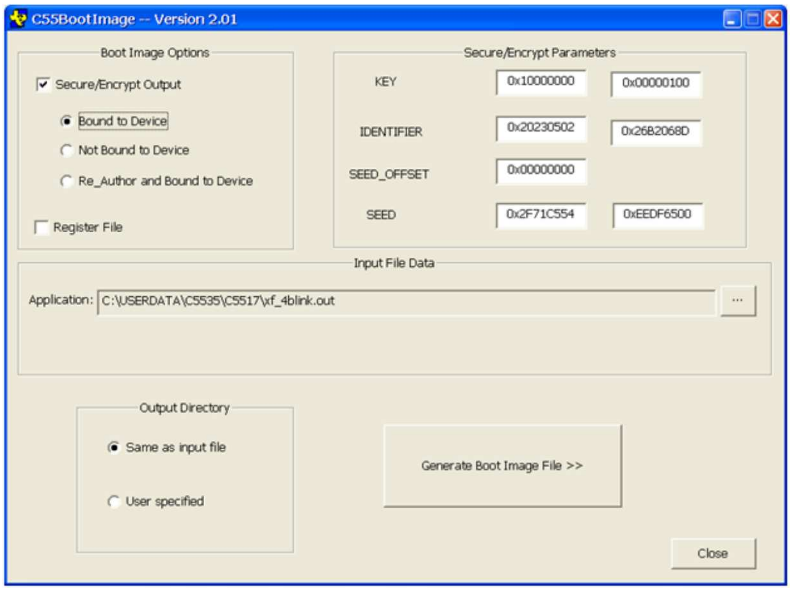
Input File Data
- Application: C:\USERDATA\C5535\C5517\xf_4blink.out

Output Directory
- ● Same as input file
- ○ User specified

Generate Boot Image File >>

Close

Figure 2. Creating an Encrypted Boot Image: Re-Authorization and Bound to Device



Boot Image Options
[✓] Secure/Encrypt Output
[● Bound to Device] ○ Not Bound to Device ○ Re_Author and Bound to Device
[ ] Register File

Secure/Encrypt Parameters

KEY	0x10000000	0x00000100
IDENTIFIER	0x20230502	0x26B2068D
SEED_OFFSET	0x00000000	
SEED	0x2F71C554	0xEEDF6500

Input File Data
Application: C:\USERDATA\C5535\C5517\xf_4blink.out

Output Directory
[● Same as input file] ○ User specified

Figure 3. Creating an Encrypted Boot Image: Bound to Device

3.3 Burn a Boot Image

Once a boot image (*.bin) is generated, customers can burn the boot image into the SPI EEPROM (16-bit or 24-bit), I2C EEPROM, SPI serial flash or SD/SDHC/eMMC/moviNAND. It is done by a utility called programmer which runs on each device using an emulator with CCS. First you will need to load the program, programmer_C5535_eZdsp.out. For writing to SPI serial flash, at the “C5535 eZdsp … input <file_path>” prompt, input the path name for the boot image<enter>. The following display will be shown:

```
SPI Flash…
Erase chip (SPI Flash)…
Open <file-path>
Input file opened
Programming Complete
```

3.4 Boot from Micro SD/SDHC Card

To boot from micro SD/SDHC card:

1. Ensure there is no boot image in the SPI serial flash.
2. Format the micro SD/SDHC to FAT16/32.
3. Copy the boot image into the root directory and rename it to “bootimg.bin”.
4. Insert the micro SD/SDHC in the micro SD slot (J6).
5. Power cycle the C5535 eZdsp.

The bootloader will boot from the micro SD/SDHC card.

3.5 Boot from UART

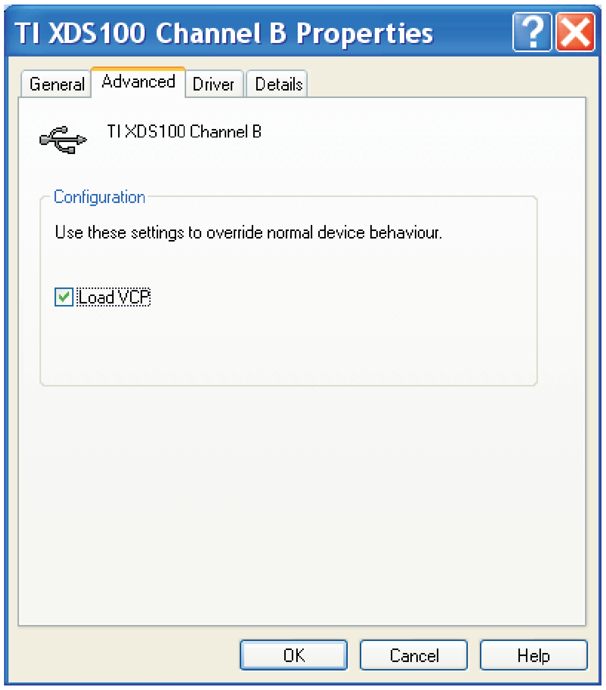
To boot from UART:

1. Ensure there is no boot image in the SPI serial flash or micro SD slot.

   The J2 of the C5535 eZdsp should be connected to a PC USB port via USB cable. It will use the virtual COM port (the COM port number varies by system) through TI XDS100 Channel B.

2. Ensure pin 2 of SW3 is on and the Load VCP is checked for TI XDS100 Channel B (Enable the Virtual COM Port).

   To see the properties for TI XDS100 Channel B, open Control Panel→System→Hardware→Device Manager→USB Controller→TI XDS100 Channel B→Properties→Advanced.



Use these settings to override normal device behaviour.

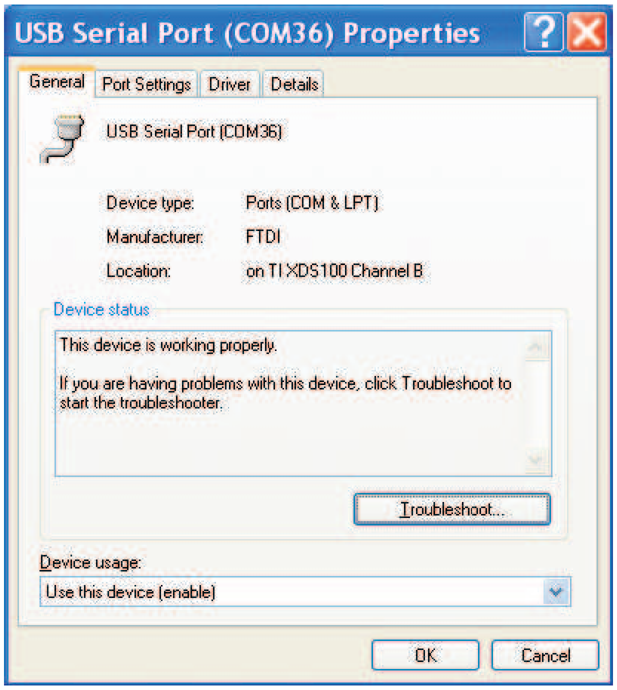
[✓] Load VCP

Enable the Virtual COM Port

To see the properties of a particular COM port, open Control Panel→System→Hardware→Device Manager→Ports (COM & LPT)→USB Serial Port (COMxx).

The properties dialog box is shown in Get the Virtual COM Port Number.

Enable the Virtual COM Port (continued)


USB Serial Port (COM36) Properties

General | Port Settings | Driver | Details

USB Serial Port (COM36)

Device type: Ports (COM & LPT)
Manufacturer: FTDI
Location: on TI XDS100 Channel B

Device status

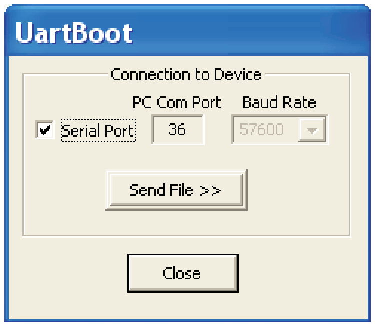
This device is working properly.

If you are having problems with this device, click Troubleshoot to start the troubleshooter.

Device usage: Use this device (enable)

Get the Virtual COM Port Number

3. Run the UartBoot.exe (Running UartBoot.exe) and select the 57600 in “Baud Rate”.
4. Enter the correct COM port number in “PC COM Port” and check the “Serial Port”.
5. Click “Send File >>”.


UartBoot

Connection to Device

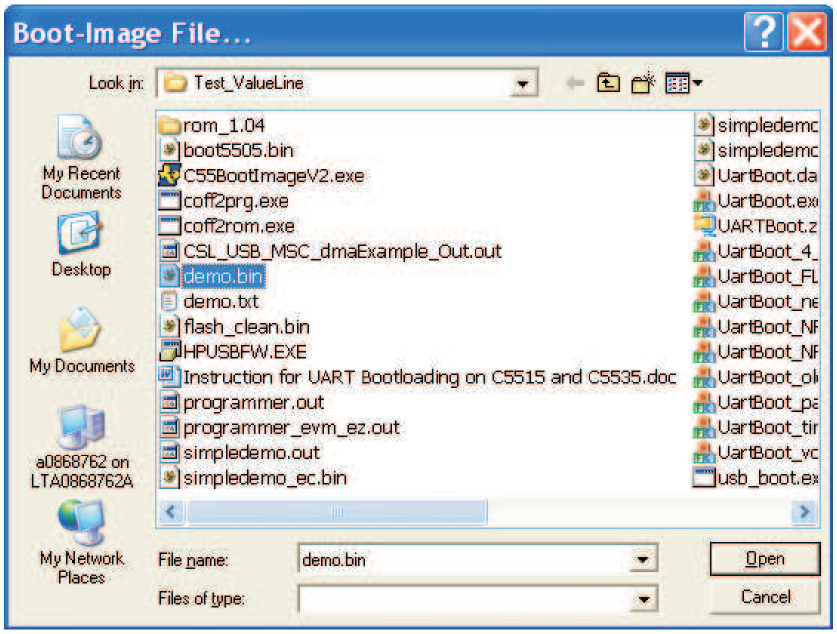
PC Com Port: 36
Baud Rate: 57600
[✓] Serial Port

Running UartBoot.exe

In the file selection dialog box (Select the Boot Image File):

6. Select the file to upload: demo.bin, in this case.
7. Click "Open" to start uploading.

Running UartBoot.exe (continued)

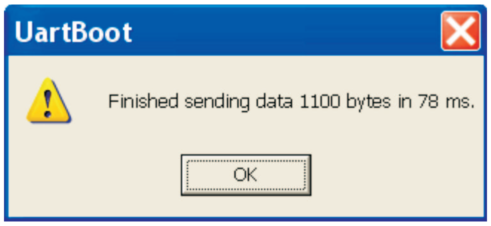


rom_1.04
boot5505.bin
C55BootImageV2.exe
coff2prg.exe
coff2rom.exe
CSL_USB_MSC_dmaExample_Out.out
demo.bin
demo.txt
flash_clean.bin
HPUSBFW.EXE
Instruction for UART Bootloading on C5515 and C5535.doc
programmer.out
programmer_evm_ez.out
simpledemo.out
simpledemo_ec.bin
simpledemc
simpledemc
UartBoot.da
UartBoot.ex
UARTBoot.z
UartBoot_4_
UartBoot_FL
UartBoot_ne
UartBoot_NF
UartBoot_NF
UartBoot_ol
UartBoot_pa
UartBoot_tir
UartBoot_vc
usb_boot.ex

File name: demo.bin
Files of type:

Select the Boot Image File

After the uploading is complete, a message box appears, as in UART Boot is Completed.



Finished sending data 1100 bytes in 78 ms.

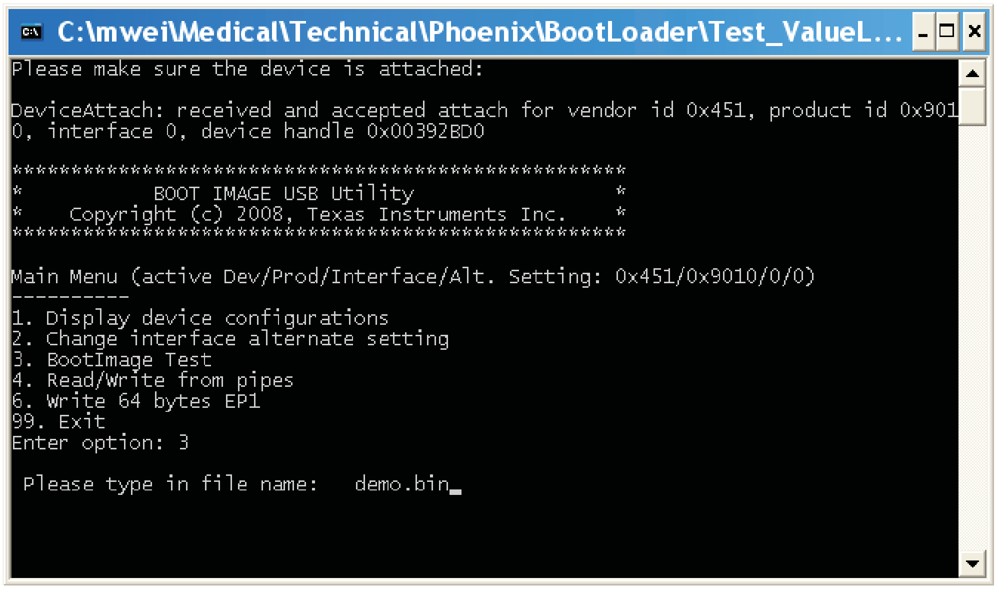
UART Boot is Completed

The demo.bin now runs on the C5535 eZdsp.

3.6 Boot from USB

To boot from USB:

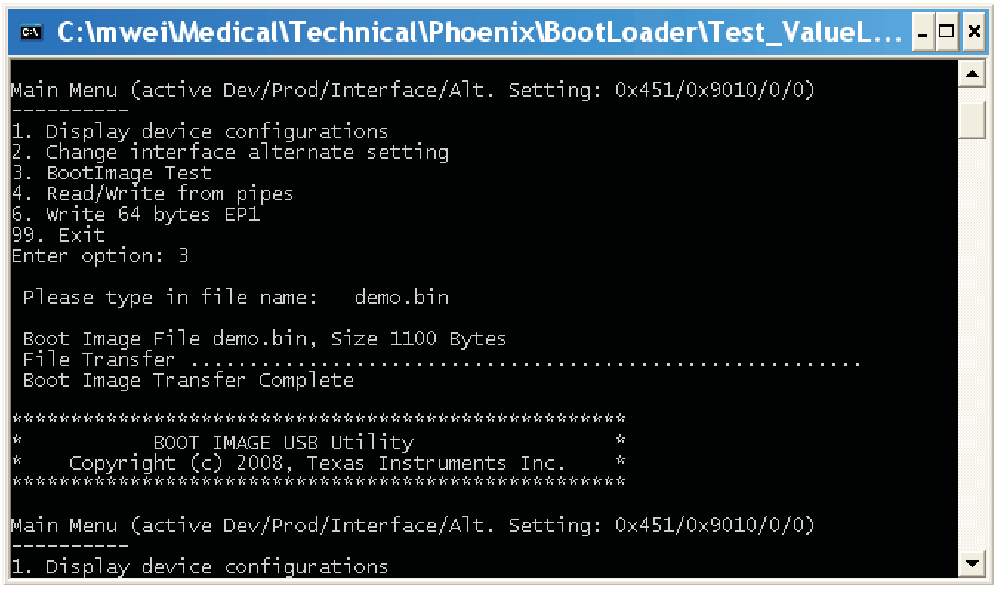
1. Ensure there is no boot image in the SPI serial flash or micro SD slot.
   The J1 of the C5535 eZdsp should be connected to a PC USB port via a USB cable.
2. Execute usb_boot.exe and enter option 3 for BootImage Test.
3. Input the boot image path name: demo.bin, in this case.



```
Please make sure the device is attached:

DeviceAttach: received and accepted attach for vendor id 0x451, product id 0x901
0, interface 0, device handle 0x00392BD0

****************************************************
*           BOOT IMAGE USB Utility                 *
*    Copyright (c) 2008, Texas Instruments Inc.    *
****************************************************

Main Menu (active Dev/Prod/Interface/Alt. Setting: 0x451/0x9010/0/0)
----------
1. Display device configurations
2. Change interface alternate setting
3. BootImage Test
4. Read/Write from pipes
6. Write 64 bytes EP1
99. Exit
Enter option: 3

 Please type in file name:   demo.bin_
```

Running usb_boot.exe

After <enter>, the following dialog box appears.



```
Main Menu (active Dev/Prod/Interface/Alt. Setting: 0x451/0x9010/0/0)
----------
1. Display device configurations
2. Change interface alternate setting
3. BootImage Test
4. Read/Write from pipes
6. Write 64 bytes EP1
99. Exit
Enter option: 3

 Please type in file name:   demo.bin

 Boot Image File demo.bin, Size 1100 Bytes
 File Transfer ....................................................
 Boot Image Transfer Complete

****************************************************
*           BOOT IMAGE USB Utility                 *
*    Copyright (c) 2008, Texas Instruments Inc.    *
****************************************************

Main Menu (active Dev/Prod/Interface/Alt. Setting: 0x451/0x9010/0/0)
----------
1. Display device configurations
```

USB Boot is Completed

The demo.bin now runs on the C5535 eZdsp.

4 References

- *TMS320C55x DSP Assembly Language Tools User's Guide* (SPRU280)

Revision History

Changes from B Revision (January 2014) to C Revision **Page**

- Added C5545 device 1
- TI No longer supports the secure version of the device with customer unique secure SEED,KEY and JTAG disabled 1
- Removed any references to *"Secure ROM"* 1
- Updated intro paragraph of Section 3.2 9

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		