ROM

Texas Instruments CC2538™ Family of Products

User's Guide

Document License

This work is licensed under the Creative Commons Attribution-NoDerivs 3.0 Unported License (CC BY-ND 3.0). To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/3.0/legalcode or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

Copyright

Copyright © 2013 Texas Instruments Incorporated. All rights reserved. CC2538 and SmartRF are registered trademarks of Texas Instruments. ARM and Thumb are registered trademarks and Cortex is a trademark of ARM Limited. Other names and brands may be claimed as the property of others.

Please be aware that an important notice concerning availability, standard warranty, and use in critical applications of Texas Instruments semiconductor products and disclaimers thereto appears at the end of this document.

Texas Instruments
Post Office Box 655303
Dallas, TX 75265
http://www.ti.com

Revision Information

User guide literature numbers from Texas Instruments RF Products start with *SWRU*. The document revision is indicated by a letter suffix after the literature number. The initial version of a document does not have a letter suffix (for example SWRU321). The first revision is suffixed *A*, the second *B*, and so on (for example SWRU321**B**). The literature number for this document is in the document footer.

This document was updated on April 16, 2013 (build 38531).

Table of Contents

1   Introduction

The CC2538 device family ROM from Texas Instruments contains a utility function library and a ROM-based serial boot loader. Applications can use the utility function library to reduce their flash footprint, thus allowing the flash to be used for other purposes (such as additional features in the application). The boot loader is used as an initial program loader when the flash is empty.

A table at the beginning of the ROM points to the entry points for the utility function APIs provided in the ROM. Accessing the API through this table provides scalability; while the API locations may change in future versions of the ROM, the API table will not. The table is located immediately after the Cortex-M3® vector table in the ROM.

The CC2538 peripheral driver library contains a file called `driverlib/source/rom.h` that assists with calling the utility library functions in the ROM. The naming conventions for the table and APIs used in this document match those used in that file.

The following is an example of calling the ROM_ResetDevice() function:

```
#include "hw_memmap.h"
#include "hw_types.h"
#include "rom.h"

int
main(void)
{
    // ...

    ROM_ResetDevice();

    // ....
}
```

Any compiler can use the API provided by the ROM so long as the compiler complies with the embedded applications binary interface (EABI), which includes all recent compilers for the CC2538 family of devices.

Documentation Overview

Chapter 2 describes the ROM-based CC2538 boot loader.

Chapter 3 describes the utility function library.

2   Boot Loader

2.1   Introduction

The ROM-based boot loader executes each time the device is reset when no valid image exists in the flash. The flash is assumed to be empty or without a valid image, if no flash vector table is configured in the customer configuration area (CCA) in the top flash page. If a valid flash image exists, the boot loader can still be entered on reset using the boot loader backdoor mechanism, as long as the mechanism is not disabled by configuration in the CCA. In any case, when run, the ROM-based boot loader uses one of the following interfaces to update the flash:

- UART0 using a custom serial protocol
- SSI0 using a custom serial protocol

Unable to detect the presence of an attached crystal, the boot loader initially operates from the internal oscillator, a 16MHz clock. For greater speed, an external 32MHz crystal oscillator can eventually be selected using a dedicated boot loader command.

2.2   Entering the Boot Loader

On reset, the Cortex-M3 executes the following steps as part of the ROM-based start-up sequence:

1. Checks that a valid flash image exists in the flash. If not, jump into a ROM-based boot loader that allows the flash to be programmed and stays there until a system reset executes.
2. Updates the Vector Location NVIC register with the address of the flash vector table found in the application entry point field in the CCA. The CCA is in the top flash page.
3. Loads the stack pointer with the initial stack pointer value found in the flash-based vector table.
4. Branches to the address of the flash reset interrupt service routine (ISR) found in the second entry of the flash vector table.

Note:
While the ROM-based start-up sequence executes, all interrupts are handled by a minimal vector table in the ROM. As part of transferring control to the flash application (in steps 2 through 4), interrupt responsibility is transferred to the image in flash. A vector table must thus be part of the image in flash.

2.2.1   Configuration of a Valid Flash Image

To perform a validity check of the flash image, check the Image Validity field and the Application Entry Point field within the CCA. As the CCA resides in the flash top page, the location of the page

depends on the device flash size. When a valid image is programmed into the flash, the CCA must configured as follows:

- The 32-bit Image Valid field must have the value of 0x00000000 to indicate a valid image in flash.
- The 32-bit Application Entry Point field must contain the start address of the vector table residing in flash.

If the Image Validity field indicates a valid image in flash and the Application Entry Point field has a value within the address range of the flash with an alignment of 0x80 bytes minimum, it is determined that the flash contains a valid image.

Note:
> Once the boot loader is disabled, it can be re-enabled by changing the CCA area that determines if a valid image is present such that the boot loader is entered after a reset.

2.2.2 Boot Loader Backdoor

A boot loader backdoor can be enabled to force entry of the ROM boot loader on reset even if a valid image in flash is present. This action is practical during a software development cycle where programming of a faulty image would potentially lock the device from further programming, unless a chip erase is executed through JTAG® . If the functionality is enabled the boot loader is entered on power up or reset if a specified level is applied to a specified pin on pad A no later than 10 us after reset. An 8-bit field in the CCA area configures the backdoor functionality in the flash top page. The following table describes the layout of the backdoor configuration byte.

Bit	Field element	Description
7:5	RESERVED	Reserved for future use. Should be all ones.
4	Enable	Enables or disables the backdoor and boot loader functions: 1: Backdoor and boot loader enabled 0: Backdoor and boot loader disabled
3	Level	Sets the active level for the selected pin on pad A that is used when backdoor is enabled: 1: Active high 0: Active low
2:0	Pin Number	The pin number (7:0) on pad A that is used when the backdoor is enabled

Table 2.1: Boot loader backdoor configuration

To enter the boot loader after power up or reset using the backdoor, the selected pin on pad A must externally be set to the configured active level no later than 10us after reset. The pin level is read only once. Toggling of the pin level will have no effect after the pin has been read.

Note:
> If the Enable bit is set to 0 in the CCA area of the flash top page, the CC2538 ROM boot loader ignores any received boot loader commands, even if no application image is present in the flash memory. This security feature enhancement makes it impossible for an attacker that has gained access to the ROM boot loader, to reveal any flash image contents by using the boot loader commands. If the boot loader is disabled, it ignores any received boot loader command.

2.2.3   Customer Configuration Area

The user can disable erasing or writing to certain pages with a page lock bit that is available to each individual 2k flash page. The lock bits are placed in the uppermost available page (the CCA) together with the boot loader configuration bytes. The uppermost available page number depends on the flash size option. For example, for the 512 KB option, the CCA is page number 255. When a page n is locked (for example LOCK_PAGE_N[n] = 0) it is impossible to erase it or write to it. A write or erase operation to the page shall be aborted. The debug lock bit (LOCK_DEBUG_N) is used to lock the debug interface through ICEPick.

The following table shows how the page lock bits and the debug lock bit are laid out in the upper 32 bytes of the CCA page. The table also shows the layout of the boot loader configuration options. Note that some of the page lock bits are unused for certain flash size options.

Byte	Bit	Name	Description
2047	7	LOCK_DEBUG_N	Debug lock bit: 1: Debug access is allowed 0: Debug access is blocked
2047	6:0	LOCK_PAGE_N[254:248]	Page 254 - 248 lock bits: 1: Write and Erase allowed 0: Write and Erase not allowed
2046	7:0	LOCK_PAGE_N[247:240]	Page 247 - 240 lock bits: 1: Write and Erase allowed 0: Write and Erase not allowed
2045	7:0	LOCK_PAGE_N[239:232]	Page 239 - 232 lock bits: 1: Write and Erase allowed 0: Write and Erase not allowed
..	..	..	..
2016	7:0	LOCK_PAGE_N[7:0]	Page 7 - 0 lock bits: 1: Write and Erase allowed 0: Write and Erase not allowed
2015-2012	-	Application Entry Point	The address of the Vector Table present in the flash memory. Note: The address must be in little-endian format
2011-2008	-	Image Valid	Must have the value of 0x00000000 to indicate valid image in flash
2007	7:0	Boot loader backdoor	Back door configuration
2006	7:0	RESERVED	Reserved for future use
2005	7:0	RESERVED	Reserved for future use
2004	7:0	RESERVED	Reserved for future use

Table 2.2: Customer Configuration Area Layout

2.3   Serial Interfaces

The serial interfaces used to communicate with the boot loader share a common protocol and differ only in the physical connections and signaling used to transfer the bytes of the protocol.

2.3.1   UART Interface

The UART pins **UART0 Tx** and **UART0 Rx** are used to communicate with the boot loader. The device communicating with the boot loader drives the **UART0 Rx** pin on the CC2538 device, while the CC2538 device drives the **UART0 Tx** pin.

The serial data format is fixed at 8 data bits, no parity, and one stop-bit. An auto-baud feature is used to determine the baud rate at which data is transmitted. Because the system clock must be at least 32 times the baud rate, the maximum baud rate that can be used is 500 Kbaud (16MHz divided by 32). This data rate number can be doubled if an external 32MHz crystal oscillator is in use and selected using the **COMMAND_SET_XOSC** boot loader command.

2.3.1.1   Auto-Baud

The boot loader provides a method that automatically detects the UART baud rate being used to communicate with the device. To synchronize with the host the boot loader requires, that two bytes with the value of 0x55 are received. If synchronization succeeds the Boot Loader will return an acknowledge consisting of two bytes with the values of 0x00, 0xCC. If synchronization fails the boot loader will be waiting for another synchronization attempt.

2.3.2   SSI Interface

The SSI pins **SSI0 Fss**, **SSI0 Clk**, **SSI0 Tx**, and **SSI0 Rx** are used to communicate with the boot loader. The device communicating with the boot loader drives the **SSI0 Rx**, **SSI0 Clk**, and **SSI0 Fss** pins, while the CC2538 device drives the **SSI0 Tx** pin.

The serial data format is fixed to the Motorola format with SPH and SPO set to 1 (see the applicable CC2538 family data sheet for more information on this format). Because the system clock must be at least 12 times the serial clock rate, the maximum serial clock rate that can be used is 1.3 MHz (16MHz divided by 12). This data rate number can be doubled if an external 32MHz crystal oscillator is in use, and if this is selected using the **COMMAND_SET_XOSC** boot loader command.

2.3.3   Pin Configuration

The boot loader supports updating through the UART0 and SSI0 ports, which are available on the CC2538 device. The SSI port has the advantage of supporting higher and more flexible data rates, but it also requires more connections to the device. The UART has the disadvantage of having slightly lower and possibly less flexible rates. However, the UART requires fewer pins and can be easily implemented with any standard UART connection. The boot loader configures port A to include all pins used by the supported serial communication on the UART0 and SSI0 peripherals. This configuration is shown in the following table.

The boot loader selects the first interface accessed by the external device. Once selected, the module clock for the inactive interface (UART0 or SSI0) is disabled. To switch over to the other interface, the CC2538 must be reset.

Function	Port	Pin	Direction
UART0 Rx	A	0	Input
UART0 Tx	A	1	Output
SSI0 Clk	A	2	Input
SSI0 Fss	A	3	Input
SSI0 Rx	A	4	Input
SSI0 Tx	A	5	Output

Table 2.3: Pin Configuration of Serial Interfaces for Boot Loader

2.3.4   Serial Protocol

The boot loader uses well-defined packets on the serial interfaces to ensure reliable communications with the update program. The packets are always acknowledged or not acknowledged by the communicating devices. The packets use the same format for receiving and sending packets. This format includes the method used to acknowledge successful or unsuccessful reception of a packet. While the actual signaling on the serial ports is different, the packet format remains independent of the method of transporting the data.

Byte	1	2	3	...	N+2
Field	Size	Check Sum	Data 1	...	Data N

Table 2.4: Packet Format

Perform the following steps to successfully send a packet:

1. Send the size of the packet that will be sent to the device. The size is always the number of bytes of data + 2 bytes.
2. Send the checksum of the data buffer to help ensure proper transmission of the command. The checksum is simply a sum of the data bytes.
3. Send the actual data bytes.
4. Wait for a single-byte acknowledgment from the device indicating either that the data is properly received or that a transmission error is detected.

Perform the following steps to successfully receive a packet:

1. Waiting for the device to return non-zero data is important because the device can send zero bytes between a sent and received data packet. The first nonzero byte received is the size of the packet being received.
2. Read the next byte (the checksum for the packet).
3. Read the data bytes from the device. Packet size minus 2 bytes of data are sent during the data phase. For example, if the packet size is 3, only 1 byte of data is to be received.
4. Calculate the checksum of the data bytes and ensure that it matches the checksum received in the packet.
5. Send an acknowledge (ACK) or not-acknowledge (NAK) to the device to indicate the successful or unsuccessful reception of the packet.

An acknowledge packet is sent whenever the boot loader successfully receives and verifies a packet. A not-acknowledge packet is sent whenever a sent packet is detected to have an error,

usually as a result of a checksum error or malformed data in the packet. This notifies the sender to re-transmit the previous packet.

The ACK response consists of 2 bytes with the values of 0x00, 0xCC.

An NACK response consists of 2 bytes with the values of 0x00, 0x33 and is sent if an invalid packet is received.

Note:
> If the SSI interface is used and the host has sent a packet containing a boot loader command that requires a time consuming code execution, the host must include a delay before clocking the response out of the device. The delay should be included after step 3 in the send sequence above. The time consuming commands are:
>
> - COMMAND_SEND_DATA
> - COMMAND_ERASE
> - COMMAND_CRC32

2.4   Boot Loader Commands

The following commands are used by the custom protocol.

COMMAND_PING
= 0x20

This command is used to receive an ACK from the boot loader, thus indicating that communication has been established. This command is a single byte.

The format of the command follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_PING;
```

COMMAND_DOWNLOAD
= 0x21

This command is sent to the boot loader to indicate where to store data and how many bytes will be sent by the COMMAND_SEND_DATA commands that follow. The command consists of two 32-bit values that are both transferred most-significant bit (MSB) first. The first 32-bit value is the address into which to start programming data, while the second is the 32-bit size of the data that will be sent. The Program Size parameter must be a multiple of 4. This command should be followed by a COMMAND_GET_STATUS command to ensure that the program address and program size were valid for the boot loader. On the CC2538 device the flash starts at address 0x00200000.

The format of the command follows:

```
unsigned char ucCommand[9];

ucCommand[0] = COMMAND_DOWNLOAD;
ucCommand[1] = Program Address [31:24];
ucCommand[2] = Program Address [23:16];
ucCommand[3] = Program Address [15:8];
ucCommand[4] = Program Address [7:0];
ucCommand[5] = Program Size [31:24];
ucCommand[6] = Program Size [23:16];
ucCommand[7] = Program Size [15:8];
ucCommand[8] = Program Size [7:0];
```

COMMAND_RUN
= 0x22

This command is sent to the boot loader to transfer execution control to the specified address. The command is followed by a 32-bit value, transferred MSB first, that is the address to which execution control is transferred. Note, this command will not re-turn to the boot loader command receive loop.

The format of the command follows:

```
unsigned char ucCommand[5];

ucCommand[0] = COMMAND_RUN;
ucCommand[1] = Run Address [31:24];
ucCommand[2] = Run Address [23:16];
ucCommand[3] = Run Address [15:8];
ucCommand[4] = Run Address [7:0];
```

COMMAND_GET_STATUS
= 0x23

This command returns the status of the last command issued. Typically, this command should be received after every command is sent to ensure that the previous command was successful or, if unsuccessful, to properly respond to a failure. The command requires 1 byte in the data of the packet, and the boot loader should respond by sending a packet with 1 byte of data that contains the current status code.

The format of the command follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_GET_STATUS;
```

The following are the definitions for the possible status values that can be returned from the boot loader when COMMAND_GET_STATUS is sent to the the CC2538 device.

```
COMMAND_RET_SUCCESS
COMMAND_RET_UNKNOWN_CMD
COMMAND_RET_INVALID_CMD
COMMAND_RET_INVALID_ADD
COMMAND_RET_FLASH_FAIL
```

COMMAND_SEND_DATA
= 0x24

This command should only follow a COMMAND_DOWNLOAD command or another COMMAND_SEND_DATA command, if more data is needed. Consecutive send data commands automatically increment the address and continue programming from the previous location. The transfer size is limited by the maximum size of a packet, which allows simultaneous transfer of up to 252 data bytes. The command terminates programming once the number of bytes indicated by the COMMAND_DOWNLOAD command is received. Each time this function is called, it should be followed by a COMMAND_GET_STATUS command to ensure that the data is successfully programmed into the flash. If the boot loader sends a NAK to this command, the boot loader does not increment the current address, which allows for retransmission of the previous data.

The format of the command follows:

```
unsigned char ucCommand[9];

ucCommand[0] = COMMAND_SEND_DATA;
ucCommand[1] = Data[0];
ucCommand[2] = Data[1];
ucCommand[3] = Data[2];
ucCommand[4] = Data[3];
ucCommand[5] = Data[4];
ucCommand[6] = Data[5];
ucCommand[7] = Data[6];
ucCommand[8] = Data[7];
```

COMMAND_RESET
= 0x25

This command tells the boot loader to reset. This is used after downloading a new image to the CC2538 device to cause the new application to start from a reset. The normal boot sequence occurs and the image runs as if from a hardware reset. This command can also be used to reset the boot loader if a critical error occurs and the host device wants to restart communication with the boot loader.

The boot loader responds with an ACK signal to the host device before executing the software reset on the CC2538 device running the boot loader. The ACK informs the updater application that the command is received successfully and the part will be reset.

The format of the command follows:

```
unsigned char ucCommand[1];

ucCommand[0] = COMMAND_RESET;
```

COMMAND_ERASE
= 0x26

This command erases the required flash pages. A single flash page has the size of 2KB, which is the minimum size that can be erased. The command consists of two 32-bit values that are both transferred MSB first. The first 32-bit value is the address in a flash page from where the erase starts, and the second 32-bit value is the number of bytes comprised by the erase. The boot loader responds with an ACK signal to the host device after the actual erase operation is performed. The page erase operation always starts erasing at the start address of the page that incorporates the input Data Address. The erase ends at the end of the page, which is determined from the input parameters. On the CC2538 device the flash starts at address 0x00200000.

The format of the command follows:

```
unsigned char ucCommand[9];

ucCommand[0] = COMMAND_ERASE;
ucCommand[1] = Data Address [31:24];
ucCommand[2] = Data Address [23:16];
ucCommand[3] = Data Address [15: 8];
ucCommand[4] = Data Address [ 7: 0];
ucCommand[5] = Data Size [31:24];
ucCommand[6] = Data Size [23:16];
ucCommand[7] = Data Size [15: 8];
ucCommand[8] = Data Size [ 7: 0];
```

COMMAND_CRC32
= 0x27

This command is used to check a flash area using the CRC32 calculation. The command consists of two 32-bit values that are both transferred MSB first. The first 32-bit value is the address in flash from where the CRC32 calculation starts, and the second 32-bit value is the number of bytes comprised by the CRC32 calculation. The command sends the ACK in response to the command after the actual CRC32 calculation. The result is finally returned as 4 bytes in a packet. The boot loader then waits for an ACK from the host as a confirmation that the packet is received. On CC2538 the flash starts at address 0x00200000.

The format of the command follows:

```
unsigned char ucCommand[9];
ucCommand[0]= COMMAND_CRC32;
ucCommand[1]= Data Address [31:24];
ucCommand[2]= Data Address [23:16];
ucCommand[3]= Data Address [15: 8];
ucCommand[4]= Data Address [ 7: 0];
ucCommand[5]= Data Size [31:24];
ucCommand[6]= Data Size [23:16];
ucCommand[7]= Data Size [15: 8];
ucCommand[8]= Data Size [7: 0];
```

COMMAND_GET_CHIP_ID
= 0x28

This command makes the boot loader return the value of the Chip ID. The boot loader first respond by sending the ACK in response to the command and then send a packet with 4 bytes of data that contains the Chip ID value. The 2 LSB bytes hold the chip part number while the remaining bytes are reserved for future use and have the value of 0x00. The boot loader then waits for an ACK from the host as a confirmation that the packet received.

The format of the command follows:

```
unsigned char ucCommand[1];
ucCommand[0]= COMMAND_GET_CHIP_ID;
```

COMMAND_SET_XOSC
= 0x29

This command is used to switch system clock source from the internal 16MHz RC oscillator to the external 32MHz crystal oscillator. This can be used to increase transfer speed on the boot loader serial interfaces. The device responds with an ACK on the command packet before the switch to the external 32MHz crystal oscillator. If the command is transferred on the UART interface, the host must execute a UART baud rate auto-detection sequence to establish the baud rate to be used while the device is running from the external 32MHz crystal oscillator. The device must be reset to switch back to running from the internal 16MHz RC oscillator.

The format of the command follows:

```
unsigned char ucCommand[1];
ucCommand[0]= COMMAND_SET_XOSC;
```

COMMAND_MEMORY_READ
= 0x2A

This command is used to read either an 8-bit or 32-bit word at a specified address within the device memory map. The command consists of a 32-bit value that is transferred, MSB first, followed by an 8-bit value specifying the read access width. The 32-bit value is the address within the device memory map to be read and the 8-bit value specifies the access width of the read operation. Allowed values for the access width are 1 and 4 (1 = 1 byte / 4 = 4 bytes). The device sends an ACK in response to the command after the specified data is read from memory. The read data is finally returned as 4 bytes (MSB first) in a packet. The boot loader then waits for an ACK from the host as a confirmation that the packet is received. For an access width value of 1, the read data is located in the LSB of the 4 bytes returned while the MSB bytes have the value of 0x00. If an invalid access width is specified within the command, the boot loader will acknowledge the command by sending an ACK and then respond with 4 bytes with the value 0x00 in a packet. In this case, any following COMMAND_GET_STATUS commands report COMMAND_RET_INVALID_CMD.

The format of the command follows:

```
unsigned char ucCommand[6];
ucCommand[0]= COMMAND_MEMORY_READ;
ucCommand[1]= Data Address [31:24];
ucCommand[2]= Data Address [23:16];
ucCommand[3]= Data Address [15: 8];
ucCommand[4]= Data Address [ 7: 0];
ucCommand[5]= Read access width [7:0];
```

COMMAND_MEMORY_WRITE
= 0x2B

This command is used to write either an 8-bit or 32-bit word to a specified address within the device memory map. The command consists of two 32-bit values that are transferred, MSB first, followed by an 8-bit value specifying the write access width. The first 32-bit value is the address within the device memory map to be written. The second 32-bit value is the data to be written. The 8-bit value specifies the access width of the write operation. Allowed values for the access width are 1 and 4 (1 = 1 byte / 4 = 4 bytes). The device sends an ACK in response to the command after the specified data is written to memory. For an access width value of 1, the data to be written must be located in the LSB of the 4 data bytes in the command. If an invalid access width is specified within the command, the boot loader acknowledges the command by sending an ACK but does not perform any memory write operation. In this case, any following COMMAND_GET_STATUS commands report COMMAND_RET_INVALID_CMD.

The format of the command follows:

```
unsigned char ucCommand[10];
ucCommand[0]= COMMAND_MEMORY_WRITE;
ucCommand[1]= Data Address [31:24];
ucCommand[2]= Data Address [23:16];
ucCommand[3]= Data Address [15: 8];
ucCommand[4]= Data Address [ 7: 0];
ucCommand[5]= Data [31:24]
ucCommand[6]= Data [23:16]
ucCommand[7]= Data [15: 8]
ucCommand[8]= Data [7: 0]
ucCommand[9]= Read access width [7:0];
```

2.4.1 Boot Loader Return Values

The boot loader can return the following values in response to the COMMAND_GET_STATUS command.

COMMAND_RET_SUCCESS
= 0x40

This value is returned in response to a COMMAND_GET_STATUS command and indicates that the previous command completed successful.

The format of the status value follows:

```
unsigned char ucStatus[1];

ucCommand[0] = COMMAND_RET_SUCCESS;
```

`COMMAND_RET_UNKNOWN_CMD` = 0x41	This is returned in response to a `COMMAND_GET_STATUS` command and indicates that the command sent is an unknown command.

The format of the status value follows:

```
unsigned char ucStatus[1];

ucCommand[0] = COMMAND_RET_UNKNOWN_CMD;
```

`COMMAND_RET_INVALID_CMD` = 0x42	This is returned in response to a `COMMAND_GET_STATUS` command and indicates that the previous command is formatted incorrectly.

The format of the status value follows:

```
unsigned char ucStatus[1];

ucCommand[0] = COMMAND_RET_INVALID_CMD;
```

`COMMAND_RET_INVALID_ADR` = 0x43	This is returned in response to a `COMMAND_GET_STATUS` command and indicates that the previous download command contained an invalid address value.

The format of the status value follows:

```
unsigned char ucStatus[1];

ucCommand[0] = COMMAND_RET_INVALID_ADR;
```

`COMMAND_RET_FLASH_FAIL` = 0x44	This is returned in response to a `COMMAND_GET_STATUS` command and indicates that an attempt to program or erase the flash has failed.

The format of the status value is as follows:

```
unsigned char ucStatus[1];

ucCommand[0] = COMMAND_RET_FLASH_FAIL;
```

The definitions for these commands are provided as part of the CC2538 peripheral driver library, in `source/bl_commands.h`.

3   ROM Utility Function Library

3.1   Introduction

The CC2538 ROM contains a utility function library. Applications can use the utility function library to reduce their flash footprint, thus allowing the flash to be used for other purposes (such as additional features in the application).

Any compiler can use the API provided by the ROM as long as the compiler complies with the embedded applications binary interface (EABI), which includes all recent compilers for the CC2538 family of devices.

This implementation is located as object code in the ROM with `source/rom.h` containing the API definitions for use by applications.

3.2   API Functions

Functions

- unsigned long ROM_Crc32 (unsigned char *pucData, unsigned long ulByteCount)
- unsigned long ROM_GetChipId (void)
- unsigned long ROM_GetFlashSize (void)
- int ROM_memcmp (const void *S1, const void *S2, size_t Length)
- void * ROM_memcpy (void *Dst, const void *Src, size_t Length)
- void * ROM_memmove (void *Dst, const void *Src, size_t Length)
- void * ROM_memset (void *Dst, int C, size_t Length)
- long ROM_PageErase (unsigned long ulStartAddress, unsigned long ulEraseSize)
- long ROM_ProgramFlash (unsigned long *pulData, unsigned long ulAddress, unsigned long ulCount)
- void ROM_ResetDevice (void)

3.2.1   Detailed Description

The Utility API is broken into three groups of functions:

- Those that deal with flash programming
- Those that deal with security and chip configuration
- Those that handle data movement

The flash programming is handled by the ROM_PageErase() function and the ROM_ProgramFlash() function.

Device configuration can be accessed using the ROM_GetFlashSize() function. A reset equivalent to a pin reset can be executed using the ROM_ResetDevice() function.

Easy data memory handling can be implemented using the following functions: ROM_memset(), ROM_memcpy(), ROM_memcmp(), ROM_memmove() and ROM_Crc32().

3.2.2   Function Documentation

3.2.2.1   ROM_Crc32

This function calculates CRC32 over a given address range.

Prototype:
```
unsigned long
ROM_Crc32(unsigned char *pucData,
          unsigned long ulByteCount)
```

Parameters:
   pucData is a pointer to the image data.
   ulByteCount is the size of the image in bytes.

Description:
   This function calculates the CRC32 over a given address range. The CRC value is calculated using CRC32 IEEE 802.3 with polynomial $x32 + x26 + x23 + x22 + x16 + x12 + x11 + x10 + x8 + x7 + x5 + x4 + x2 + x + 1$

Returns:
   Calculated CRC-32 value

3.2.2.2   ROM_GetChipId

This function returns the chip ID.

Prototype:
```
unsigned long
ROM_GetChipId(void)
```

Description:
   This function returns the chip ID.

Returns:
   Chip ID

3.2.2.3   ROM_GetFlashSize

This function returns the flash size.

Prototype:
```
unsigned long
ROM_GetFlashSize(void)
```

Description:
    This function returns the flash size in number of bytes.

Returns:
    None

3.2.2.4   ROM_memcmp

This function compares two memory areas.

Prototype:
```
int
ROM_memcmp(const void *S1,
           const void *S2,
           size_t Length)
```

Parameters:
    S1 is a pointer to the start of a memory area.
    S2 is a pointer to the start of a second memory area.
    Length is the number of characters to compare.

Description:
    This function compares not more than **Length** characters of the object pointed to by **S1** with the object pointed to by **S2**.

Returns:
    This function returns an integer greater than, equal to or less than 0 according to whether the object pointed to by **S1** is greater than, equal to, or less than the object pointed to by **S2**.

3.2.2.5   ROM_memcpy

This function copies data from one memory area to another area.

Prototype:
```
void *
ROM_memcpy(void *Dst,
           const void *Src,
           size_t Length)
```

Parameters:
    Dst is a pointer to the start of the destination memory area.
    Src is a pointer to the start of the source memory area.
    Length is the number of characters to copy.

Description:
    This function copies **Length** bytes from the memory region pointed to by **Src** to the memory region pointed to by **Dst**. If the regions overlap, the behavior is undefined.

Returns:
    The function returns a pointer to the first byte of the **Dst** region.

3.2.2.6   ROM_memmove

This function moves a block of data from one memory area to another.

Prototype:
```
void *
ROM_memmove(void *Dst,
            const void *Src,
            size_t Length)
```

Parameters:
Dst is a pointer to the start of the destination memory area.
Src is a pointer to the start of the source memory area.
Length is the number of characters to copy.

Description:
This function moves **Length** characters from the block memory pointed to by **Src** to the memory region pointed to by **Dst**. ROM_memmove reproduces the characters correctly at the **Dst** region even if the two areas overlap.

Returns:
Returns **Dst** as passed.

3.2.2.7   ROM_memset

Set a memory area to a specified value

Prototype:
```
void *
ROM_memset(void *Dst,
           int C,
           size_t Length)
```

Parameters:
Dst is a pointer to start of the memory to be set.
C is the value to set.
Length is the number of characters to set.

Description:
This function converts the argument **C** into an unsigned char and fills the first **Length** characters of the array pointed to by **Dst** to the value.

Returns:
Returns the value of **Dst**.

3.2.2.8   ROM_PageErase

This function erases flash pages.

Prototype:
```
long
ROM_PageErase(unsigned long ulStartAddress,
              unsigned long ulEraseSize)
```

Parameters:
> ***ulStartAddress*** is the starting address in flash of the first page to be erased. Must be a multiple of 4 and within the flash memory map area.
> ***ulEraseSize*** is the number of bytes that must be erased.

Description:
> This function erases the required number of sequential flash pages starting at the page derived from ulStartAddress and up to the page required by the number of bytes in ulEraseSize. It is up to the caller to verify the contents after the erase, if such verification is required.
>
> This function does not return until the required flash pages are erased or an error condition occurs.

Returns:
> Returns 0 on success, or -1 if an error is encountered or -2 if invalid input parameters.

3.2.2.9 ROM_ProgramFlash

This function programs the flash.

Prototype:
```
long
ROM_ProgramFlash(unsigned long *pulData,
                 unsigned long ulAddress,
                 unsigned long ulCount)
```

Parameters:
> ***pulData*** is a pointer to the data to be programmed.
> ***ulAddress*** is the starting address in flash to be programmed. Must be a multiple of 4 and within the flash memory map area.
> ***ulCount*** is the number of bytes to be programmed. Must be a multiple of 4.

Description:
> This function programs a sequence of 32-bits words into the on-chip flash. Programming each location consists of the result of an AND operation of the new data and the existing data; in other words bits that contain 1 can remain 1 or be changed to 0, but bits that are 0 cannot be changed to 1. Therefore, a word can be programmed multiple times as long as these rules are followed; if a program operation tries to change a 0 bit to a 1 bit, the value of that bit is not changed.
>
> Because the flash is programmed one word at a time, the starting address and byte count must both be multiples of 4. It is up to the caller to verify the programmed contents, if such verification is required.
>
> This function does not return until the data has been programmed or an access violation has occurred.

Returns:
> Returns 0 on success, or -1 if a programming error is encountered or -2 if invalid input parameters.

3.2.2.10 ROM_ResetDevice

This function performs a system reset of the CC2538 SoC.

Prototype:
```
void
ROM_ResetDevice(void)
```

Description:
This function performs a system reset equivalent to an external pin reset.

Returns:
None

3.3 Programming Example

The following example shows how to use the ROM utility API to determine the size of the flash.

```
unsigned long ulSize;

ulSize = ROM_GetFlashSize();
```

IMPORTANT NOTICE

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		