



Anushree Biradar, Sira Rao, and Vamsikrishna Gudivada

## ABSTRACT

Embedded processors often need to be programmed in situations where JTAG cannot be used to program the target device. In these cases, the engineer needs to rely on serial programming solutions. C2000™ devices aid in this endeavor through the inclusion of several program loading utilities in ROM. These utilities are useful, but only solve half of the programming problem because they only allow loading application code into RAM. This application report builds on these ROM loaders by using a flash kernel. A flash kernel is loaded to RAM using a ROM loader - it is then executed and used to program the target device's on-chip Flash memory with the end application. This document details one possible implementation for C2000 devices and provides PC utilities to evaluate the solution with.

---

## Table of Contents

<b>1 Introduction</b> .....	2
<b>2 Programming Fundamentals</b> .....	3
<b>3 ROM Bootloader</b> .....	3
<b>4 Flash Kernel A</b> .....	4
4.1 Implementation.....	4
<b>5 Flash Kernel B</b> .....	5
5.1 Implementation.....	5
<b>6 Example Implementation</b> .....	9
6.1 Device Setup.....	9
6.2 Host Application: serial_flash_programmer.....	10
6.3 Host Application: Firmware Updates on F28004x With SCI Flash Kernel.....	12
<b>7 Troubleshooting</b> .....	13
7.1 General.....	13
7.2 SCI Boot.....	14
7.3 F2837x.....	14
<b>8 References</b> .....	15
<b>9 Revision History</b> .....	16

## List of Figures

Figure 6-1. Serial Flash Programmer Prompt for Next Command After Downloading Flash Kernel to RAM.....	11
Figure 6-2. Serial Flash Programmer After Downloading Application to Flash.....	12

## List of Tables

Table 3-1. Default Boot Modes for F28004x devices.....	3
Table 5-1. Packet Format.....	6
Table 5-2. ACK/NAK Values.....	6
Table 5-3. CPU1 Kernel Commands.....	6
Table 5-4. CPU2 Kernel Commands.....	7
Table 5-5. Erase Packet.....	8
Table 5-6. Unlock Packet.....	8
Table 5-7. Run Packet.....	8
Table 5-8. Status Codes.....	8

## Trademarks

C2000™ and Code Composer Studio™ are trademarks of Texas Instruments.

Microsoft Visual Studio® is a registered trademark of Microsoft Corporation in the United States and/or other countries.

All trademarks are the property of their respective owners.

## 1 Introduction

As applications grow in complexity, the need to fix bugs, add features, and otherwise modify embedded firmware is increasingly critical in end applications. Enabling functionality like this can be easily and inexpensively accomplished through the use of bootloaders.

A bootloader, also referred to as a ROM loader or simply loader, is a small piece of code that resides in the target device's boot-ROM memory that allows the loading and execution of code from an external host. In most cases, a communication peripheral such as Universal Asynchronous Receiver/Transmitter (UART) or Controller Area Network (CAN) is used to load code into the device rather than JTAG, which requires an expensive specialized tool.

Boot Pins are used to configure different boot modes using various peripherals that determine which ROM loader is invoked. In this report, the peripheral used is Serial Communications Interface (SCI, generally referred to as UART). The boot modes that are associated with the boot pins refer to the first instance of the peripheral - for SCI, the boot mode would be associated with SCIA.

C2000 devices partially solve the problem of firmware updates by including some basic loading utilities in ROM. Depending on the device and the communications peripherals present, code can be loaded into on-chip RAM using UART, Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), Ethernet, CAN, and a parallel mode using General Purpose Input/Outputs (GPIOs). A subset of these loaders is present in every C2000 device and they are very easy to use, but they can only load code into RAM. How does one bridge the gap and program their application code into non-volatile memory?

This application report aims to solve this problem by using a flash kernel. Flash kernels have been around for some time, but this document discusses the specifics of the kernels and the host application tool found in C2000Ware. While this implementation is targeted at C2000 devices using the SCI peripheral, the same principles apply to all devices in the C2000 product line and all communications options supported by the ROM loaders. A command line tool is provided to parse and transmit the application image from the host PC (Windows only) to the embedded device.

In summary, application programming to non-volatile memory like flash requires two steps:

1. Use the SCI ROM bootloader to download a flash kernel to RAM.
2. Run the flash kernel in RAM to download the application to flash.

## 2 Programming Fundamentals

Before programming a device, it is necessary to understand how the non-volatile memory of C2000 devices works. Flash is a non-volatile memory that allows users to easily erase and re-program it. Erase operations set all the bits in a sector to '1' while programming operations selectively clear bits to '0'. Flash on certain devices can only be erased one sector at a time, but others have bank erase options. The term serial flash used in this report - serial flash kernels, serial flash programmer, and so forth only refers to the serial communication interface (SCI).

Flash operations on all C2000 devices are performed using the CPU. Algorithms are loaded into RAM and executed by the CPU to perform any flash operation. For example, erasing or programming the flash of a C2000 device with Code Composer Studio™ entails loading flash algorithms into RAM and letting the processor execute them. There are no special JTAG commands that are used. All flash operations are performed using the flash application programming interface (API). Because all flash operations are done using the CPU, there are many possibilities for device programming. Irrespective of how the kernels and application are brought into the device, flash is programmed using the CPU.

## 3 ROM Bootloader

At the beginning, the device boots and, based on the boot mode, decides if it should execute code already programmed into the Flash memory or load in code using one of the ROM loaders. This application report focuses on the boot execution path when the emulator is not connected.

### Note

This section is based on the TMS320F28004x device. Specific information for a particular device can be found in the *Boot ROM* section of the device-specific technical reference manual (TRM).

**Table 3-1. Default Boot Modes for F28004x devices**

Boot Mode	GPIO24 (default boot mode select pin 1)	GPIO32 (default boot mode select pin 0)
Parallel I/O	0	0
SCI/Wait boot	0	1
CAN	1	0
Flash	1	1

After the boot ROM readies the device for use, it decides where it should start executing. In the case of a standalone boot, it does this by examining the state of two GPIOs (as seen in [Table 3-1](#), the default choices are GPIO 24 and 32). In some cases, two values programmed into one time programmable (OTP) memory can be examined. In the implementation described in this application report, the SCI loader is used, so at power up GPIO 32 must be forced high and GPIO 24 must be forced low. If this is the case when the device boots, the SCI loader in ROM begins executing and waits to autobaud lock with the host (for a character to be received in order to determine the baud rate at which the communications will occur). At this point, the device is ready to receive code from the host.

The ROM loader requires data to be presented to it in a specific structure. The structure is common to all ROM loaders and is described in detail in the *Bootloader Data Stream Structure* section of [1]. You can easily generate your application in this format by using the hex2000 utility included with the TI C2000 compiler. This file format can even be generated as part of the Code Composer Studio build process by adding a post-build step with the following options:

```
"${CG_TOOL_HEX}" "${BuildArtifactFileName}" -boot -sci8 -a -o "${BuildArtifactFileName}.txt"
```

Alternatively, you can use the TI hex2000 utility to convert COFF and EABI .out files into the correct boot hex format. To do this, you need to enable the C2000 Hex Utility under Project Properties. The command is below:

```
hex2000.exe -boot -sci8 -a -o <file.txt> <file.out>
```

As stated before, ROM loaders can only load code into RAM, which is why they are used to load in flash kernels, which will be described in [Section 4](#) and [Section 5](#).

## 4 Flash Kernel A

Flash Kernel A runs on:

- TMS320F2802x
- TMS320F2803x
- TMS320F2805x
- TMS320F2806x
- TMS320F2833x

To find the location of the flash kernel projects for these devices, see [Section 7](#).

### 4.1 Implementation

Flash Kernel A is based off SCI ROM loader sources. To enable this code to erase and program flash, flash APIs must be incorporated, which is done by linking the flash APIs. Before any application data is received, the flash kernel erases the flash of the device readying it for programming. A buffer is used to hold the received contiguous blocks of application code. When the buffer is full or a new block of noncontiguous data is detected, the code in the buffer is programmed. This continues until the entire application is received.

The protocol used to transfer the application data has been slightly modified from the SCI ROM loader protocol. This was done to improve the speed of programming while also ensuring robust communications. With the original SCI ROM loader protocol, most of the time is spent not transferring data, but waiting for the data to propagate through the different layers of the operating system. This problem is compounded by the fact that data must be sent a single byte at a time with the SCI ROM loader (due to the echo based flow control), so every byte incurs the OS transport delay. The flash kernel uses the same protocol but calculates a checksum that is sent after every block of data. This allows the PC side application to send many bytes at a time through the different layers of the operating system, substantially decreasing the latency of communications.

This flash kernel can be used with a CSM locked device. If the device is locked, the serial flash programmer can still be used by loading the flash kernel into unsecure RAM and modifying the kernel to unlock the device before it erases and programs the flash. In that case, *CsmUnlock()* (present in <device\_name>\_SysCtrl.c) should be modified to write the correct CSM passwords to the CSM registers. This will unlock the device. If the user does not want to unlock CSM while programming the device, flash APIs will have to be in flash (not ROM) and be copied to Secure RAM from where they can program/erase secure flash sectors without unlocking the device.

#### 4.1.1 Application Load

This section walks through the entire flow of programming an application into flash using the SCI boot mode.

Ensure the device is ready for SCI communications by resetting the device while ensuring the boot mode pins are in the proper state to select SCI Boot mode. These are the steps that follow:

1. The device receives the autobaud character that determines the baud rate at which the load will take place. This happens soon after the host initiates a transfer command.
2. The flash kernel is transferred to the device, waiting for the character to be echoed before sending the next. Make sure the flash kernel is built and linked to RAM alone.
3. The ROM transfers control and the flash kernel begins to execute. There is a small delay in which the kernel must prepare the device for flash programming before it is ready to begin communications, and in this time the kernel configures the PLL and flash wait states.
4. The kernel enters an autobaud mode and waits for the autobaud character to be received. This potentially allows the kernel to communicate at a higher speed than was used for the ROM loader because the PLL is configured for a higher speed by the kernel. SCI has an autobaud lock that allows the host to send any baud rate they want - a single baud rate does not need to be agreed on for both sides.
5. Once the baud rate is locked, the application can be downloaded using the same format as the flash kernel. At the beginning of the download process, a key, a few reserved fields, and the application entry point are transferred before the actual application code.
6. After the entry point is received, the kernel begins to erase the flash. Erasing flash can take a few seconds, so it is important to note that while it looks like the application load may have failed, it is likely that the flash is just being erased.

7. Once the flash is erased, the application load continues by transferring each block of application code and programming it to flash.
8. After a block of data is programmed into flash, a checksum is sent back to the host PC to ensure that all of the data was correctly received by the embedded device. This process continues until the entire application has been programmed into flash.

Now that the application is programmed into flash, the flash kernel attempts to run the application by branching to the entry point that was transferred to it at the start of the application load process. A device reset is needed for this.

## 5 Flash Kernel B

Flash Kernel B runs on:

- TMS320F2807x
- TMS320F2837xD
- TMS320F2837xS
- TMS320F28004x

To find the location of the flash kernel projects for these devices, see [Section 7](#).

### 5.1 Implementation

Flash Kernel B is more robust than Kernel A. It communicates with the host PC application provided in C2000Ware (*C2000Ware\_x\_xx\_xx\_xx > utilities > flash\_programmers > serial\_flash\_programmer*) and provides feedback to the host on the receiving of packets and completion of commands given to it.

After loading the kernel into RAM and executing it via the SCI ROM bootloader, the kernel first initializes the PLLs of the device, initializes SCIA, and takes control of the flash pump, if necessary. It then waits for an 'a' or 'A' from the host in order to perform an auto baud lock with the host. After this, the kernel begins a while loop, which waits on commands from the host, executes the commands, and sends a status packet back to the host. This while loop breaks when a Run or Reset command is sent.

Commands are sent in a packet described in [Table 5-1](#) and each packet is either acknowledged or not-acknowledged. All commands, except for Run and Reset, send a packet after completion with the status of the operation. The status packet sends a 16-bit status code and 32-bit address. In case of an error, the address in the data specifies the address of the first error. In case of NO\_ERROR, the address is 0x12345678.

In the case of a Device Firmware Upgrade (DFU) command, the following steps take place:

1. The kernel receives a file in the hex boot format byte-by-byte from the SCI module and calculates a checksum over the block size. After receiving the block of data, it sends back the checksum.
2. After receiving a block of data and storing it in a buffer, the kernel erases the sector if it has not been previously erased, and programs the data into flash at the correct address along with ECC using flash API library.

---

#### Note

To get more information on the blocks of data and how block size is determined, see the *Bootloader Data Stream Structure* section of the device-specific TRM. Note that the block and block size mentioned here are not to be confused with the command packet mentioned earlier.

---

3. Afterwards, the kernel verifies that the data and ECC were programmed correctly into flash.

---

#### Note

This kernel only erases sectors that are needed to program the application and data into flash. This is different from Flash Kernel A that erases the entire flash at the start of kernel execution. However, Flash Kernel B also supports an erase command independent of the DFU command, which gives the user the ability to erase specific sectors or the entire flash of the device.

---

Similarly, the verify command receives a file in the hex boot format and in place of erasing and programming the flash, it only verifies the contents of the flash.

### 5.1.1 Packet Format

Packets are sent in a standard format between the host and device. The packet allows for a variable amount of data to be sent while ensuring correct transmission and reception of the packet. The header, footer and checksum fields help to ensure that the data was not corrupted during transmission. The checksum is the summation of the bytes in the command and data fields.

**Table 5-1. Packet Format**

Header	Data Length	Command	Data	Checksum	Footer
2 Bytes	2 Bytes	2 Bytes	Length Bytes	2 Bytes	2 Bytes
0x1BE4	Length of Data in Bytes	Command	Data	Checksum of Command and Data	0xE41B

The host and the device both send packets one word at a time (16-bits), the LSB followed by the MSB. Both the host and device respond to a packet with an ACK or NAK.

**Table 5-2. ACK/NAK Values**

ACK	NAK
0x2D	0xA5

### 5.1.2 CPU1 Kernel Commands

CPU1 commands for the dual core F2837xD are acceptable for the F2807x, F28004x, and F2837xS single core device kernels excluding the *Run CPU1 Boot CPU2* and *Reset CPU1 Boot CPU2*. A brief description of the command codes are provided in [Table 5-3](#).

**Table 5-3. CPU1 Kernel Commands**

Kernel Commands	Command Code	Description
DFU CPU1	0x0100	<ol style="list-style-type: none"> <li>1. Receive the packet with no data</li> <li>2. Receive the flash application byte-by-byte in boot hex format</li> <li>3. Selective Erase, Program, and Verify</li> <li>4. Send status packet</li> </ol> <p>If successful, the address sent in the status packet is the entry point address of the programmed flash application</p>
Erase CPU1	0x0300	<ol style="list-style-type: none"> <li>1. Receive the packet with 32-bit data (described in <a href="#">Section 5.1.4</a>)</li> <li>2. Erase the sectors specified in the data</li> <li>3. Send status packet</li> </ol>
Verify CPU1	0x0500	<ol style="list-style-type: none"> <li>1. Receive the packet with no data</li> <li>2. Receive the flash application byte-by-byte in boot hex format</li> <li>3. Verify flash contents</li> <li>4. Send status packet</li> </ol>
Unlock CPU1 – Zone 1	0x000A	<ol style="list-style-type: none"> <li>1. Receive the packet with 128-bit data (described in <a href="#">Section 5.1.4</a>)</li> <li>2. Write the password to the DCSM Key Registers</li> <li>3. Check to see if Zone 1 is unlocked</li> <li>4. Send status packet</li> </ol>
Unlock CPU1 – Zone 2	0x000B	<ol style="list-style-type: none"> <li>1. Receive the packet with 128-bit data (described in <a href="#">Section 5.1.4</a>)</li> <li>2. Write the password to the DCSM Key Registers</li> <li>3. Check to see if Zone 2 is unlocked</li> <li>4. Send status packet</li> </ol>
Run CPU1	0x000E	<ol style="list-style-type: none"> <li>1. Receive the packet with a 32-bit address (described in <a href="#">Section 5.1.4</a>)</li> <li>2. Branch to the 32-bit address</li> </ol>
Reset CPU1	0x000F	<ol style="list-style-type: none"> <li>1. Receive the packet with no data</li> <li>2. Break the while loop and enable WatchDog Timer to time-out and reset</li> </ol>

**Table 5-3. CPU1 Kernel Commands (continued)**

Kernel Commands	Command Code	Description
Run CPU1 Boot CPU2 <sup>(1)</sup>	0x0004	<ol style="list-style-type: none"> <li>1. Receive the packet with 32-bit address</li> <li>2. Release the flash pump, boot CPU2 by IPC to SCI boot mode, give CPU2 control of SCI and shared RAM, and then wait for CPU2 to signal.</li> <li>3. Branch to the address</li> </ol>
Reset CPU1 Boot CPU2 <sup>(1)</sup>	0x0007	<ol style="list-style-type: none"> <li>1. Receive the packet with no data</li> <li>2. Release the flash pump, boot CPU2 by IPC to SCI boot mode, give CPU2 control of SCI and shared RAM, and then wait for CPU2 to signal.</li> <li>3. Break the while loop and enable WatchDog Timer to time-out and reset.</li> </ol>

(1) This command is not available to F2807x, F2837xS, and F28004x single core device kernels.

### 5.1.3 CPU2 Kernel Commands

Table 5-4 shows the functions, command codes, and descriptions for the CPU2 commands used on dual core F2837xD device kernel.

**Table 5-4. CPU2 Kernel Commands**

Kernel Commands	Command Code	Description
DFU CPU2	0x0200	<ol style="list-style-type: none"> <li>1. Receive the packet with no data</li> <li>2. Receive the flash application byte-by-byte in boot hex format</li> <li>3. Selective Erase, Program, and Verify</li> <li>4. Send status packet</li> </ol> <p>If successful, the address sent in the status packet is the entry point address of the programmed flash application</p>
Erase CPU2	0x0400	<ol style="list-style-type: none"> <li>1. Receive the packet with 32-bit data (described in <a href="#">Section 5.1.4</a>)</li> <li>2. Selective erase the sectors specified in the data</li> <li>3. Send status packet</li> </ol>
Verify CPU2	0x0600	<ol style="list-style-type: none"> <li>1. Receive the packet with no data</li> <li>2. Receive the flash application byte-by-byte in boot hex format</li> <li>3. Verify flash contents</li> <li>4. Send status packet</li> </ol>
Unlock CPU2 – Zone 1	0x000C	<ol style="list-style-type: none"> <li>1. Receive the packet with 128-bit data (described in <a href="#">Section 5.1.4</a>)</li> <li>2. Write the password to the DCSM Key Registers</li> <li>3. Check to see if Zone 1 is unlocked</li> <li>4. Send status packet</li> </ol>
Unlock CPU2 – Zone 2	0x000D	<ol style="list-style-type: none"> <li>1. Receive the packet with 128-bit data (described in <a href="#">Section 5.1.4</a>)</li> <li>2. Write the password to the DCSM Key Registers</li> <li>3. Check to see if Zone 2 is unlocked</li> <li>4. Send status packet</li> </ol>
Run CPU2	0x0010	<ol style="list-style-type: none"> <li>1. Receive the packet with a 32-bit address (described in <a href="#">Section 5.1.4</a>)</li> <li>2. Branch to the 32-bit address</li> </ol>
Reset CPU2	0x000F	<ol style="list-style-type: none"> <li>1. Receive the packet with no data</li> <li>2. Break the while loop and enable WatchDog Timer to time-out and reset</li> </ol>

### 5.1.4 Packet Data

This section describes the data expected for the commands that require data to be sent to the device.

- **Erase**

Each bit of the 32-bit data sent with the erase command corresponds to a sector.

- Data Bit 0 – Sector A
- Data Bit 1 – Sector B
- And, so forth

**Table 5-5. Erase Packet**

Header	Data Length	Command	Data	Checksum	Footer
0x1BE4	6 (bytes)	0x0300 CPU1 0x0400 CPU2	32-bit Data (1 means erase, 0 means do not erase)	Checksum of Command and Data	0xE41B

- **Unlock**

- 1st 32 bits is Key 1
- 2nd 32 bits is Key 2
- 3rd 32 bits is Key 3
- 4th 32 bits is Key 4

**Table 5-6. Unlock Packet**

Header	Data Length	Command	Data	Checksum	Footer
0x1BE4	4 (bytes)	0x000A CPU1 Z1 0x000B CPU1 Z2 0x000C CPU2 Z1 0x000D CPU2 Z2	128-bit Data	Checksum of Command and Data	0xE41B

- **Run**

- 32-bit address

**Table 5-7. Run Packet**

Header	Data Length	Command	Data	Checksum	Footer
0x1BE4	4 (bytes)	0x000E CPU1 0x0020 CPU2	32-bit Data	Checksum of Command and Data	0xE41B

### 5.1.5 Status Codes

After a command is completed, the kernel sends a status packet to the host with the same format as the host-to-device packet. This lets the host know if an error occurred, what type of error, and where the error occurred. The command field is the command last completed. The data field consists of a 16-bit status code followed by a 32-bit address where the error occurs. If there is no error the address is 0x12345678 unless it is responding to a DFU command in which case the address is the entry point address of the hex boot format file of the application just programmed into flash. This address could then be used for the RUN command, which tells the CPU which address to branch to and begin executing code.

Table 5-8 displays the status codes.

**Table 5-8. Status Codes**

Status Code	Value	Description
NO_ERROR	0x1000	Return on Success
BLANK_ERROR	0x2000	Return on Erase Error
VERIFY_ERROR	0x3000	Return on Verify Error
PROGRAM_ERROR	0x4000	Return on Programming Error
COMMAND_ERROR	0x5000	Return on Invalid Command Error
UNLOCK_ERROR	0x6000	Return on Unsuccessful Unlock



## 6 Example Implementation

The kernels described above are available in C2000Ware under examples folder for the specific device within the examples directory. For example, the flash kernel for F2837x is found at `C2000Ware_x_x_xx_xx > device_support > f2837xd > examples > dual > F2837xD_sci_flash_kernels > cpu01`. The host application is found in C2000Ware (`C2000Ware_x_xx_xx_xx > utilities > flash_programmers > serial_flash_programmer`). The source and executable are found in the `serial_flash_programmer` folder. This section details the `serial_flash_programmer`: how to build, run and use it with Flash Kernel A and B.

---

### Note

The flash kernel of the appropriate device must be supplied to the host application tool being used to program the flash. The `serial_flash_programmer` starts the same way independent of the kernel or device. It first loads the kernel to the device through the SCI ROM bootloader. After this, the tool's functionality differs depending on the device and kernel being used.

---

## 6.1 Device Setup

### 6.1.1 Flash Kernels

Flash kernel source and project files for Code Composer Studio (CCS) are provided in C2000Ware, in the corresponding device's examples directory. Load the project into CCS and build the project. These projects have a post-build step in which the compiled and linked .out file is converted into the correct boot hex format needed by the SCI ROM bootloader and is saved as the example name with a .txt extension.

### 6.1.2 Hardware

After building the kernels in CCS, it is important to setup the device correctly to be able to communicate with the host PC running the `serial_flash_programmer`. The first thing to do is make sure the boot mode select pins are configured properly to boot the device to SCI boot mode. Next, connect the appropriate SCI boot loader GPIO pins to the Rx and Tx pins that are connected to the host PC COM port. A transceiver is often needed in order to convert a Virtual COM port from the PC to GPIO pins that can connect to the device. On some systems, like the controlCARD, an FTDI chip is used to interface the GPIO pins used for SCI communication to a USB Virtual COM port. Refer to the user guide for the controlCARD to get information on the switch configuration needed to enable SCI communication. In this case, the PC must connect to the mini-USB on the controlCARD and use channel B of the FTDI chip to connect to the GPIO pins on the device. After the hardware is setup correctly to communicate with the host, reset the device. This should boot the device to SCI boot mode.

## 6.2 Host Application: serial\_flash\_programmer

### 6.2.1 Overview

The command line PC utility is a lightweight (~292KB executable) programming solution that can easily be incorporated into scripting environments for applications like production line programming. It was written using Microsoft Visual Studio® in C++. The project and its source can be found in C2000Ware (C2000Ware\_x\_xx\_xx\_xx > utilities > flash\_programmers > serial\_flash\_programmer).

To use this tool to program the C2000 device, ensure that the target board has been reset and is currently in the SCI boot mode and connected to the PC COM port. The command line usage of the tool is described below:

```
serial_flash_programmer.exe -d <device> -k <kernel file> -a <app file> -p COM <num>
[-m] <kernel2 name> [-n] <app2 name> [-b] <baudrate> [-q] [-w] [-v]
```

-d <device>	- The name of the device to connect and load to: F2802x, F2803x, F2805x, F2806x, F2807x, F2837xD, F2837xS, or F28004x
-k <file>	- The file name for the CPU1 flash kernel. This file must be in the ASCII SCI boot format.
-a <file>	- The application file name to download or verify to CPU1 This file must be in the ASCII SCI boot format
-m <file>	- The file name for the CPU2 flash kernel. This file must be in the ASCII SCI boot format.
-n <file>	- The application file name to download or verify to CPU2. This file must be in the ASCII SCI boot format.
-p COM<num>	- Set the COM port to be used for communications
-b <num>	- Set the baud rate for the COM port.
-? or -h	- Show help.
-q	- Quiet mode. Disable output to stdout
-w	- Wait for a key press before exiting.
-v	- Enable verbose output.

-d, -k, -a, -p are mandatory parameters. If the baud rate is omitted, it will be set to 9600 by default.

#### Note

Both the flash kernels and flash application MUST be in the SCI8 boot format. This was discussed earlier in [Section 3](#) and can be generated from the OUT file using the hex2000 utility.

### 6.2.2 Building and Running serial\_flash\_programmer Using Visual Studio

Serial\_flash\_programmer.cpp can be compiled using Visual Studio.

1. Navigate to the serial\_flash\_programmerdirectory.
2. Double click the serial\_flash\_programmer.sln to open the Visual Studioproject.
3. When Visual Studio opens, select Build → BuildSolution.
4. After Visual Studio completes the build, select Debug → serial\_flash\_programmerproperties.
5. Select Configuration Properties →Debugging.
6. Select the input box next to the CommandArguments.
7. Type the arguments in the following format. The arguments are described in [Section 6.2.1](#).
  - Format:

```
-d <device> -k <file> -a <file> -p COM<num> -b <baudrate>
```

- Example:

```
-d f2807x -k C:\Documents\flash_kernel.txt -a C:\Documents\Test.txt -p COM7 -b 9600
```

8. Click Apply and OK.
9. Select Debug → Start Debugging to begin running the project.

### 6.2.3 Running serial\_flash\_programmer for F2806x (Flash Kernel A)

#### Note

It is recommended to reset the device before running serial\_flash\_programmer so that Autobaud will complete correctly.

1. Navigate to the folder containing the compiled serial\_flash\_programmer executable.
2. Run the executable serial\_flash\_programmer.exe with the following command:

```
:> serial_flash_programmer.exe -d f2806x -k <~\f28069_flash_kernel.txt> -a <file> -p COM<num>
```

This first loads the f28069\_flash\_kernel into RAM of the device using the bootloader. Then, the kernel executes and loads and program flashes with the file specified by the '-a' command line argument.

### 6.2.4 Running serial\_flash\_programmer for F2837xD (Flash Kernel B)

#### Note

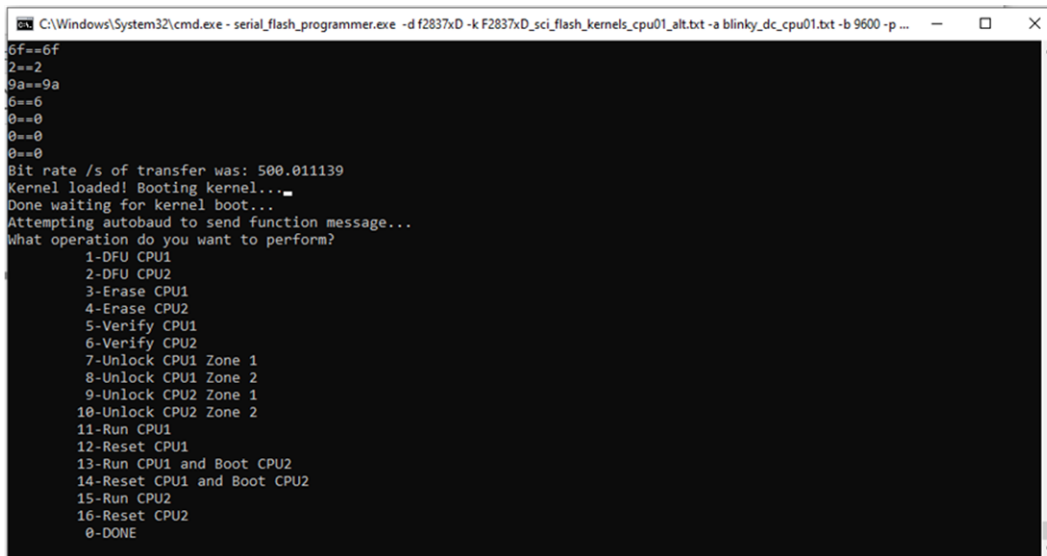
It is recommended to reset the device before running serial\_flash\_programmer so that auto baud will complete correctly.

1. Navigate to the folder containing the compiled serial\_flash\_programmer executable.
2. Run the executable serial\_flash\_programmer.exe with the following command:

```
:> serial_flash_programmer.exe -d f2837xD -k <~\F2837xD_sci_flash_kernels_cpu01.txt>  
-a <file> -m <~\F2837xD_sci_flash_kernels_cpu02.txt> -n <file> -p COM<num> -v
```

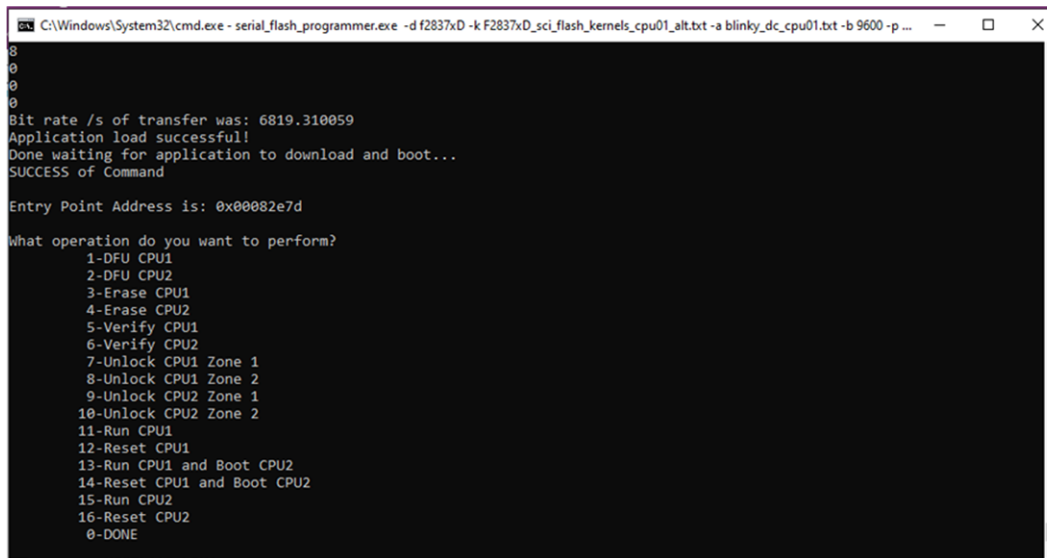
This will automatically connect to the device, perform an auto baud lock, and download the CPU1 kernel into RAM and execute it. Now, the CPU1 kernel is running and waiting for a packet from the host.

3. The serial\_flash\_programmer prints the options to the screen to choose from that will be sent to the device kernel (see [Figure 6-1](#)). Select the appropriate number and then provide any necessary information when asked for that command (described in [Section 5.1](#)). In the case of DFU or Verify, the original command already specified the Application file, so no additional information will be required at this point. [Figure 6-2](#) shows what the window will look like after the execution of command 1- DFU CPU1.



```
C:\Windows\System32\cmd.exe - serial_flash_programmer.exe -d f2837xD -k F2837xD_sci_flash_kernels_cpu01_alt.txt -a blinky_dc_cpu01.txt -b 9600 -p ...
6f==6f
2==2
9a==9a
6==6
0==0
0==0
0==0
Bit rate /s of transfer was: 500.011139
Kernel loaded! Booting kernel...
Done waiting for kernel boot...
Attempting autobaud to send function message...
What operation do you want to perform?
 1-DFU CPU1
 2-DFU CPU2
 3-Erase CPU1
 4-Erase CPU2
 5-Verify CPU1
 6-Verify CPU2
 7-Unlock CPU1 Zone 1
 8-Unlock CPU1 Zone 2
 9-Unlock CPU2 Zone 1
10-Unlock CPU2 Zone 2
11-Run CPU1
12-Reset CPU1
13-Run CPU1 and Boot CPU2
14-Reset CPU1 and Boot CPU2
15-Run CPU2
16-Reset CPU2
0-DONE
```

Figure 6-1. Serial Flash Programmer Prompt for Next Command After Downloading Flash Kernel to RAM



```

C:\Windows\System32\cmd.exe - serial_flash_programmer.exe -d f2837xD -k F2837xD_sci_flash_kernels_cpu01_alt.txt -a blinky_dc_cpu01.txt -b 9600 -p ...
8
0
0
0
0
Bit rate /s of transfer was: 6819.310059
Application load successful!
Done waiting for application to download and boot...
SUCCESS of Command

Entry Point Address is: 0x00002e7d

what operation do you want to perform?
1-DFU CPU1
2-DFU CPU2
3-Erase CPU1
4-Erase CPU2
5-Verify CPU1
6-Verify CPU2
7-Unlock CPU1 Zone 1
8-Unlock CPU1 Zone 2
9-Unlock CPU2 Zone 1
10-Unlock CPU2 Zone 2
11-Run CPU1
12-Reset CPU1
13-Run CPU1 and Boot CPU2
14-Reset CPU1 and Boot CPU2
15-Run CPU2
16-Reset CPU2
0-DONE
  
```

**Figure 6-2. Serial Flash Programmer After Downloading Application to Flash**

## 6.3 Host Application: Firmware Updates on F28004x With SCI Flash Kernel

### 6.3.1 Overview

As explained, the SCI Flash Kernel enables firmware updates. This section will walk through one possible method of doing so using SCI boot and flash boot modes. For this scenario, the device will boot from flash by default, and use SCI boot for firmware updates.

One thing worth noting is what exactly booting from flash entails. As noted earlier, a bootloader is used by a device to load and run code from an external source. The SCI flash kernel is loaded to RAM and then the application is loaded to flash. In the case of booting from flash, the device is already equipped with the application, either through Uniflash, CCS or a custom bootloader.

In this scenario, once the device boots from flash, through the flash entry point, the application will run. For firmware updates, the device can boot up in SCI boot mode, where the serial flash programmer will download the SCI flash kernel to RAM which allows the download of the new application to flash - this allows the firmware upgrade to happen without putting the SCI flash kernel in flash, thus saving flash space.

### 6.3.2 Boot Pin Configurations

In order to have both of the boot modes for flash Boot and SCI boot, the boot mode needs to be configured. The boot control GPIO Pins determine the boot mode configuration - for F28004x devices, the Boot control Pins are GPIO 24 and GPIO 32. In order to boot from flash, both GPIO 24 and GPIO 32 have to be pulled up to 1, and for SCI Boot, GPIO 24 needs to be pulled down to 0.

How a user pulls down the GPIO pin depends on the implementation - one possible example is that the device itself will set the GPIO pin low prior to a firmware upgrade - more details can be found in [10] .

### 6.3.3 Using Three Boot Modes

Suppose that a user wants to switch between 3 boot modes - SCI, flash and I2C. Three boot mode pins will need to be used, since they do not fall into the 4 device default boot modes.

In order to have further customization of boot modes as desired here, the user needs to use the BOOTPIN\_CONFIG location of the user configurable DCSM OTP. This allows the user to select up to 3 GPIO pins to have up to 8 total boot modes.

The BOOTDEF register of the user configurable DCSM OTP will also allow the user to configure boot modes beyond the default boot modes.

To switch between SCI, flash, and I2C Boot, the following steps need to be taken:

1. Set BOOTPIN\_CONFIG appropriately – bits 7-0, 15-8, 23-16 all to indicate which GPIOs will be used to represent the boot mode. Bits 31-24 will have the key 0x5A to validate the earlier bits.
2. Set BOOTDEF1 to 0x01 to indicate SCI boot with boot mode 1 (when boot mode select GPIOs = 001) and the SCI GPIOs are GPIO29 and GPIO28.
3. Set BOOTDEF3 to 0x03 to indicate Flash boot with boot mode 3 (when boot mode select GPIOs = 011). The Flash entry point is 0x80000 from Flash Bank 0, sector 0.
4. Set BOOTDEF7 to 0x07 to indicate I2C boot with boot mode 7 (when boot mode select GPIOs = 111), and the I2C GPIOs are GPIO32 and GPIO33.

For experimentation with boot mode options, users can use the EMU\_BOOTPIN\_CONFIG register before writing to OTP\_BOOTPIN\_CONFIG. OTP\_BOOTDEF also has a EMU\_BOOTDEF for experimentation.

The equivalent of the BOOTPIN\_CONFIG registers on the F2837x is the BOOTCTRL register.

### 6.3.4 Performing Live Firmware Updates

We have seen how the flash kernel is downloaded to RAM, and facilitates downloading the application to flash. By storing the flash kernel in flash, the user does not need to download the flash kernel to RAM, thus saving time. It is available in flash to directly download the application to flash. On the other hand, it takes up some flash space. This is a tradeoff that certain users may decide to use. It is especially useful to enable certain features, such as Live Firmware Updates (LFU). In LFU, the application only uses flash boot mode, and when the application is running, a user can choose to update firmware. The application will pass control to the flash kernel in flash, which will facilitate updating the application in Flash. On a single-bank flash, this typically requires a device reset. But if the device contains multiple flash banks, a device reset can be avoided with LFU, allowing a seamless transition to new firmware.

For further details on how to perform Live Firmware Updates, see [6] and [7].

## 7 Troubleshooting

Below are solutions to some common issues encountered by users when utilizing the SCI flash kernel.

### 7.1 General

**Question:** I cannot find the SCI flash kernel projects, where are they?

**Answer:**

Device	Build Configurations	Location
F2802x	RAM	C2000Ware_x_xx_xx_xx > device_support > f2802x > examples > structs > f28027_flash_kernel
F2803x	RAM	C2000Ware_x_xx_xx_xx > device_support > f2803x > examples > c28 > f2803x_flash_kernel
F2805x	RAM	C2000Ware_x_xx_xx_xx > device_support > f2805x > examples > c28 > f28055_flash_kernel
F2806x	RAM	C2000Ware_x_xx_xx_xx > device_support > f2806x > examples > c28 > f28069_sci_flash_kernel
F2807x	RAM	C2000Ware_x_xx_xx_xx > device_support > f2807x > examples > cpu1 > F2807x_sci_flash_kernel
F2833x	RAM	C2000Ware_x_xx_xx_xx > device_support > f2833x > examples > f28335_flash_kernel
F2837xS	RAM	C2000Ware_x_xx_xx_xx > device_support > f2837xs > examples > cpu1 > F2837xS_sci_flash_kernel > cpu01
F2837xD	RAM	C2000Ware_x_xx_xx_xx > device_support > f2837xd > examples > dual > F2837xD_sci_flash_kernels
F28004x	RAM, Flash with LDFU, Flash without LDFU	C2000Ware_x_xx_xx_xx > driverlib > f28004x > examples > flash, select flashapi_ex2_sci_kernel

**Question:** What is the difference between Flash Kernel A and Flash Kernel B?

**Answer:** Type A flash kernels erase the entirety of flash before streaming any data in, whereas Type B flash kernels erase only the required sectors as data is being streamed in. Type B kernels also provide users the choice to erase all or chosen sectors as they want, as a standalone operation. In addition, Type B kernels support a variety of user commands as opposed to Type A kernels which only support an application load command.

**Question:** What are the first things I should check if the SCI flash kernel does not download?

**Answer:**

- One area of the program to check would be the linker command file - make sure all flash sections are aligned to 128-bit boundaries. In SECTIONS, add a comma and "ALIGN(8)" after each line where a section is allocated to flash.
- One other common issue users encounter is that they are not using the correct boot pins for SCI boot mode. For example, on the F28004x devices, SCI boot has 4 options for GPIO pins to use. Make sure that the pins for the default option are not being used for something else. If it is, make sure that another SCI boot option is used, so that it can be connected to another set of pins. Make sure that the SCI kernel project uses this SCI boot GPIO option as the parameter for SCI\_GetFunction() as well.
- When using long cables, use a lower baud rate to get rid of noise.
- For baud rate and connection issues, try running SCI loopback and echoback examples for the device (you should be able to find these in C2000Ware in the device\_support or driverlib folders for your device).

## 7.2 SCI Boot

**Question:** I cannot download the SCI kernel to RAM in SCI boot mode, what should I do?

**Answer:** Make sure to use the correct GPIO pins to use SCI boot mode. Refer to the previous question for a detailed explanation.

## 7.3 F2837x

### F2837xS

**Question:** The SCI Kernel is downloaded to RAM and Application is downloaded to Flash, but no command window appears, and once that is done, Flash seems to be unwritten to?

**Answer:** To address the issue where no command window appears after the Flash kernel is downloaded to RAM - SCI Flash Kernel project contains 2 possible SCIA Boot options:

- SCIA Boot option 2 (GPIO 28, 29)
- SCIA Boot option 1 (GPIO 84, 85)

Use the right pins for your hardware configuration. To address the issue where Flash does not get written properly - Align all Flash sections to 128-bit boundaries in the linker command file.

### F2837xD

**Question:** I want to boot from Flash and SCI, how do I switch between the two? Flash Boot is used for normal operation, SCI Boot for firmware update.

**Answer:**

There are a few options here. A few are mentioned in [Section 6.3](#) - the device can boot up in Flash boot mode for normal operation, and in SCI Boot mode for firmware upgrades. The user will need to design a scheme that allows changing the boot mode select pins settings. Alternately, the user can decide to only use the Flash boot mode, and then in software, branch to the SCI Boot function. This can be through an IO trigger or host command, that serves as a cue to call the SCI\_Boot() function. The advantage of this approach is that the user does not need to change the boot mode select pins settings. Note that in this case, the SCI ROM Bootloader is not invoked.

**Question:** Using SCI Boot mode to upgrade firmware works, however using Flash boot and branching to SCI Boot in software by calling SCI\_Boot() does not work. What should I look at to fix this?

**Answer:**

- Make sure the correct GPIO pins are being used for SCI boot mode.
- Also, it is necessary to disable the watchdog and interrupts prior to the SCI\_Boot() call in your program, since Flash is being updated.

**Question:** With the ControlCARD, I cannot download the SCI Flash Kernel to RAM in SCI Boot, what steps should I take to do so?

**Answer:**

1. The controlCARD requires that option 2 of SCI Boot be used in order for it to work on the device. For more details on this, refer to Section 2.1 of [12]. In order to do this, make sure the SCI Flash Kernel project builds with SCI\_BOOT\_ALTERNATE as the parameter for SCI\_GetFunction.
2. Next, set the SW1:A position 1 to ON on the controlCARD. This allows the emulator to be connected. Set the Boot mode to SCI by setting SW1 Position 1 to 0, and SW1 Position 2 to 1.
3. After that, make sure that SW1:A position 2 is also ON. GPIO28 (and pin 76 of the 180-pin controlCARD connector) will be coupled to the FTDI's USB-to-Serial adapter. This allows UART communication to a computer via the FTDI chip.
4. After this, connect the emulator via CCS, and look up the location 0xD00 in memory. Change it to 0x815A. In order to use Alternate GPIOs (28, 29) for SCI Boot mode, we need to set the BMODE field (bits 15:8) of the corresponding BOOTCTRL register (either EMUBOOTCTRL (0xD00) or Z1-BOOTCTRL (0x7801E which is in User configurable DCSM OTP) to 0x81).
5. If you wish to run the flash kernel on CPU2 as well, connect to it in CCS. Click "Reset CPU", followed by "Resume". The program will hit ESTOP. Click "Resume" again.
6. In CCS, for CPU1, click "Reset CPU", followed by "Resume" (F8).
7. Now run the command serial\_flash\_programmer.exe with the appropriate command. The kernel will be downloaded to RAM. When the menu pops up, select the desired operation to proceed. To see what the command window will look like at this stage, see [Figure 6-1](#).

### **F2837xD LaunchPad**

**Question:** I cannot get the SCI Flash Kernel to load into RAM, is there a reason why?

**Answer:** The F2837xD LaunchPad does not support SCI Boot Mode, so this board cannot be used with the SCI Flash Kernel (see Section 2.1 of [11]).

## **8 References**

1. Texas Instruments: [TMS32028004x Piccolo Microcontrollers Technical Reference Manual](#)
2. Texas Instruments: [ROM Code and Peripheral Booting](#) section from the [TMS320F2837xD Dual-Core Delfino Microcontrollers Technical Reference Manual](#)
3. Piccolo Flash API User's Guide - located within controlSUITE at: (/controlSUITE/libs/utilities/flash\_api/DEVICE/VERSION/doc)
4. Texas Instruments: [TMS320F28M35x and TMS320F28M36x Flash API Reference Guide](#)
5. Texas Instruments: [TMS320C28x Assembly Language Tools User's Guide](#)
6. Texas Instruments: [Live Firmware Update With Device Reset on C2000™ MCUs](#)
7. Texas Instruments: [Live Firmware Update Without Device Reset on C2000™ MCUs](#)
8. Texas Instruments: [USB Flash Programming of C2000™ Microcontrollers](#)
9. Texas Instruments: [TMS320F28004x Boot Features and Configurations](#)
10. Texas Instruments: [C2000™ Software Controlled Firmware Update Process](#)
11. Texas Instruments: [LAUNCHXL-F28379D Overview User's Guide](#)
12. Texas Instruments: [TMS320F28379D controlCARD User's Guide](#)

## 9 Revision History

NOTE: Page numbers for previous revisions may differ from page numbers in the current version.

<b>Changes from Revision C (March 2019) to Revision D (October 2020)</b>	<b>Page</b>
• Updated the numbering format for tables, figures and cross-references throughout the document.....	2
• Updates were made in <a href="#">Section 1</a> .....	2
• Updates were made in <a href="#">Section 2</a> .....	3
• Updates were made in <a href="#">Section 3</a> .....	3
• Updates were made in <a href="#">Section 4.1.1</a> .....	4
• Updates were made in <a href="#">Section 5.1</a> .....	5
• Update was made in <a href="#">Section 6</a> .....	9
• Updated information in <a href="#">Section 6.1.2</a> .....	9
• Updates were made in <a href="#">Section 6.2.1</a> .....	10
• Update was made in <a href="#">Section 6.2.3</a> .....	11
• Updates were made in <a href="#">Section 6.2.4</a> .....	11
• Added new <a href="#">Section 6.3</a> .....	12
• Added new <a href="#">Section 7</a> .....	13

---



## IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATASHEETS), DESIGN RESOURCES (INCLUDING REFERENCE DESIGNS), APPLICATION OR OTHER DESIGN ADVICE, WEB TOOLS, SAFETY INFORMATION, AND OTHER RESOURCES "AS IS" AND WITH ALL FAULTS, AND DISCLAIMS ALL WARRANTIES, EXPRESS AND IMPLIED, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT OF THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, or other requirements. These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI's products are provided subject to TI's Terms of Sale ([www.ti.com/legal/termsofsale.html](http://www.ti.com/legal/termsofsale.html)) or other applicable terms available either on [ti.com](http://ti.com) or provided in conjunction with such TI products. TI's provision of these resources does not expand or otherwise alter TI's applicable warranties or warranty disclaimers for TI products.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265  
Copyright © 2020, Texas Instruments Incorporated