

Serial Flash Programming of C2000™ Microcontrollers

Trey German, Salvatore Pezzino, Shashank Kulkarni, Strong Zhang and Terry Lin

ABSTRACT

Often times, embedded processors must be programmed in situations where JTAG is not a viable option for programming the target device. When this is the case, the engineer must rely on some type of serial programming solution. C2000 devices aid in this endeavor through their inclusion of several program loading utilities included in ROM. These utilities are useful, but only solve half of the programming problem because they only allow loading program code to RAM. This application report builds on these ROM loaders by introducing the idea of a Flash kernel. A Flash kernel is loaded using one of the ROM loaders and is then executed and used to program the target device's Flash with the end application. This document details one possible implementation for C2000 devices and provides PC utilities to evaluate the solution with.

Contents

1	Introduction	2
2	Programming Fundamentals	2
3	ROM Bootloader	3
4	Flash Kernel A	5
5	Flash Kernel B	6
6	Example Implementation	11
7	References	14

List of Figures

1	F2802x Boot Flowchart	3
2	Bootloader Data Structure	5
3	serial_flash_programmer Prompting for Next Command	14

List of Tables

1	Device Standalone Boot Modes F2802x Example	3
2	Packet Format	8
3	ACK/NAK Values	8
4	CPU1 Kernel Commands	8
5	CPU2 Kernel Commands	9
6	Erase Packet	10
7	Unlock Packet	10
8	Run Packet	10
9	Status Codes	10

C2000, controlSUITE, 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 other trademarks are the property of their respective owners.

1 Introduction

As applications become more and more complex, the need to fix bugs, add features, and otherwise modify embedded firmware is increasingly critical in end applications. Often times, end equipment customers are asked to do these firmware upgrades themselves in order to save the manufacturer maintenance costs. Enabling functionality like this can easily and cheaply be accomplished through the use of bootloaders.

A bootloader is a small piece of code that resides in the target device's memory that allows it to load and execute code from an external source. 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. This allows the end customer to use a more common communications channel to upgrade their embedded device's firmware rather than JTAG, which requires an expensive specialized tool.

C2000 devices partially solve the problem of boot-loading by including some basic loading utilities in ROM. Depending on the device and the communications peripherals present, code can be loaded into RAM on C2000 devices using: UART, Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), Ethernet, CAN, and even 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 introducing the idea of a Flash kernel. The concept of a Flash kernel is not new or unique. This technique has been used time and time again, but this document discusses the specifics of the kernels and the host application tool found in controlSUITE™. While this implementation is targeted at C2000 devices using the Serial Communications Interface (SCI) UART peripheral, the same principles apply to all devices in the C2000 product line and all communications options supported in the ROM loaders. A command line tool is provided to parse and transmit the application from the host PC (Windows and Linux) to the embedded device.

2 Programming Fundamentals

Before programming a device, you need to understand how the non-volatile memory of C2000 devices works. For the most part, all C2000 devices use Flash as their non-volatile memory technology. Flash is a non-volatile memory technology that allows you to easily erase and program your memory. Erase operations set all of the bits in a sector to '1' while programming operations selectively clear bits to '0'. This is one of the main limitations of Flash, it can only be erased a sector at a time.

The underlying principle for how a Flash memory functions is the same between different devices, families, and even different companies, but the implementation varies quite a bit. Flash memory comes in many variants, each with its own design tradeoffs. For example, some Flash may operate faster but may be larger and more expensive to manufacture. There are also differences in terms of programming interface. Some Flash memories have dedicated hardware that is used to program and erase Flash via a set of registers, while others use algorithms, which run on the CPU in order to perform Flash operations.

In all cases, Flash operations on 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™ software is actually loading Flash algorithms into RAM and letting the processor execute them. There are no special JTAG commands that are used. Flash operations are always performed using the same underlying software, the Flash API. Because Flash operations are always done using the CPU, this opens a world of possibilities for device programming. Any way that information can enter the chip can be used to load code into the device for Flash programming.

C2000, controlSUITE, 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 other trademarks are the property of their respective owners.

3 ROM Bootloader

3.1 Functionality

To begin, the device boots and decides if it should execute code already programmed into the device or load in code using one of the loaders in ROM.

NOTE: The material in this section is based on the 2802x, including boot flow, pin numbers, boot modes, and so forth. Specific information for a particular device can be found in the *Boot ROM* section of the device-specific technical reference manual (TRM).

Figure 1 describes the sequence of events that take place just after the controller is reset.

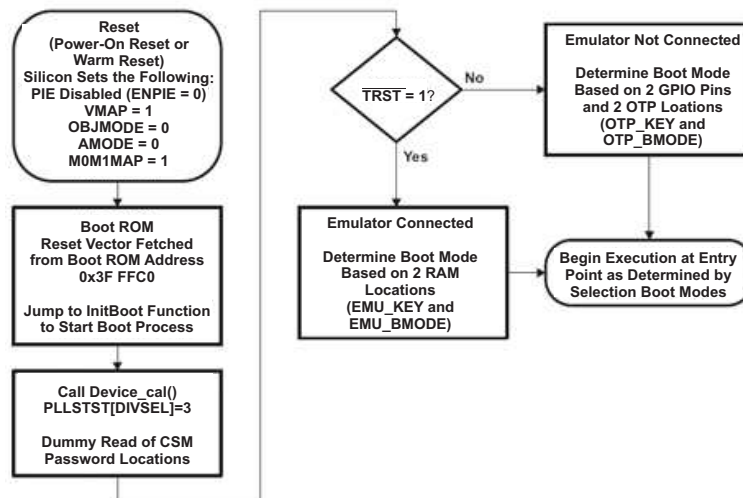


Figure 1. F2802x Boot Flowchart

The ultimate goal is to be able to program the Flash on a blank device without any external hardware, so this application report focuses on the boot execution path of when the emulator is not connected (TRST == 0 as standalone boot).

Table 1. Device Standalone Boot Modes F2802x Example

TRST	GPIO37 TDO	GPIO34	EMU Key Read From 0x0D00	EMU BMODE Read From 0x0D01	OTP KEY Read From 0x3D7BFB	OTP BMODE Read From 0x3D7BFE	Boot Mode Selected ⁽¹⁾	EMU Key Written to 0x0D00 ⁽²⁾	EMU BMODE Written to 0x0D01 ⁽²⁾
0	0	0	X ⁽³⁾	X	X	X	Parallel I/O	0x55AA	0x0000
0	0	1	X	X	X	X	SCI	0x55AA	0x0001
0	1	0	X	X	X	X	Wait	0x55AA	0x0002
0	1	1	X	X	!=0x005A	X	GetMode: Flash	0x55AA	0x0003
					0x005A	0x001	GetMode: SCI		
						0x00B	GetMode: Flash		
						0x004	GetMode: SPI		
						0x005	GetMode: I2C		
						0x006	GetMode: OTP		
						0x007	GetMode: CAN		
						Other	GetMode: Flash		

⁽¹⁾ Get Mode indicates the boot mode was derived from the values programmed in the OTP_KEY and OTP_BMODE locations.

⁽²⁾ The boot ROM writes this value to EMU_KEY and EMU_BMODE. This value can be used or overwritten if a debugger is connected.

⁽³⁾ x = don't care.

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 (for example, GPIO 34 and 37) and in some cases two values programmed into one-time programmable (OTP). In the implementation described in this application report, the SCI (UART) loader is used, so at power up GPIO34 must be forced high and GPIO37 must be forced low. If this is the case when the device boots, the SCI loader in ROM begins executing and waits for a character to be received in order to determine the baud rate where the communications will occur at. At this point, the device is ready to receive the code from the host.

The bootloader requires data to be presented to it in a specific structure. This structure is common to all bootloaders and it 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 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 .out files into the correct boot hex format.

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

3.2 Code Load

As stated previously, after the SCI boot mode is entered, the device waits for a character ('a' or 'A' to be specific) in order to determine the baud rate where communications will occur at. After the baud rate has been determined, the loading follows the flow described in the *BootROM* section of the device-specific TRM. If the code was properly formatted with the hex2000 utility, the file may be read character-by-character and sent out of the serial port without any modification. Flow control is implemented via an echo (host does not send the next character until it receives an echo of the previous character).

This process can only load code into RAM, which is why it is used to load in the Flash kernels described in [Section 4](#) and [Section 5](#). The ROM cannot access RAM protected by the Code Security Module (CSM). Therefore, the device needs to be unlocked, or the load must be to unsecure RAM.

Data structure expected by the ROM SCI bootloader:

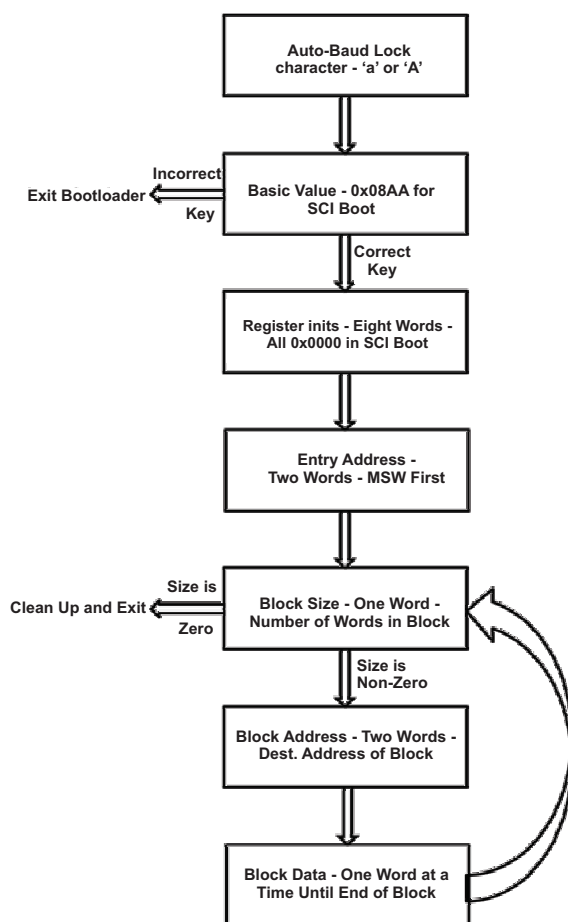


Figure 2. Bootloader Data Structure

4 Flash Kernel A

This Flash kernel runs on:

- F2802x
- F2803x
- F2805x
- F2806x
- F2833x ⁽⁴⁾

4.1 Implementation

This Flash kernel is actually surprisingly similar to the SCI loader in ROM. In fact, the Flash kernel was based off the SCI loader sources. To enable this code to erase and program Flash, the Flash API must be incorporated into the SCI loader, which was accomplished by linking against the Flash API contained in ROM. Before any application data is received, the Flash kernel erases the Flash of the device readying it for programming. Instead of using pointers to copy received data to the proper location in RAM, a buffer was added that holds contiguous pieces of application code. When the buffer is full or a new block of non-contiguous data is detected, the code in the buffer is programmed. This continues until the entire application is received.

⁽⁴⁾ Does not have Flash API in ROM.

The protocol used to communicate the application data has been slightly modified from the ROM SCI loader protocol. This was done to improve the speed of programming while also ensuring robust communications. When writing PC side loader applications it was found that 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 stock SCI loader (due to the echo based flow control), so every byte incurs the OS transport delay. The Flash kernel uses the same protocol but drops the echo flow control for 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.

The Flash kernel currently does not have the ability to unlock the code security module (CSM) to enable Flash programming. This functionality should be easy to add if an application has a need for this. One possible implementation would be to send the CSM password from the host PC to the target device, and have the target device unlock itself before performing any erase operations.

4.1.1 Application Load

Now that each of the pieces of this Flash programming puzzle are understood, you can walk through the entire flow of programming an application into Flash using the SCI boot mode.

Before communicating with the device, ensure that it is ready to receive communications. To do this, reset the device while ensuring the GPIOs are in the proper state to select the SCI boot mode. At this point, the device is waiting to receive the autobaud character in order to determine the baud rate where the load will take place at. After sending the autobaud character, the Flash kernel can be transferred to the device one byte at a time, waiting for the character to be echoed before sending the next. Make sure the Flash kernel is built and linked to RAM alone.

When the Flash kernel is loaded, the ROM transfers control and the kernel begins to execute. The Flash kernel must prepare the device for Flash programming before it is ready to begin communications, so a small delay is needed. During this time, the Flash kernel configures the PLL and Flash wait states. After the kernel has finished configuring the device, it once again 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. Once the baud rate is locked, the application can be downloaded using the same format as the ROM loader. 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. It is after the entry point is received that the kernel begins to erase the Flash. Erasing the 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. Once the Flash is erased, the application load continues by transferring each block of application code and programming it to Flash. Remember the communications protocol of the Flash kernel is slightly different from that of the ROM loaders. 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.

5 Flash Kernel B

This Flash Kernel B runs on:

- F2807x
- F2837xD
- F2837xS

5.1 Implementation

This flash kernel is an enhanced version of Flash Kernel A. It has increased functionality and is more suitable for a broader flash programming solution.

Flash Kernel B gives the user flexibility with a variety of functions to perform on the device including device firmware upgrade (DFU) (that erases the flash and loads and programs an application into flash), a strict erase operation, verifies flash contents, unlocks dual code security module (DCSM), runs the device and resets the device. For the F2837xD device, two kernels are provided: one for each core and some additional functionality for CPU1 that is used to boot CPU2 to SCI boot mode in order to load its kernel in the same way CPU1 does. More details and an example using the F2837xD are provided in [Section 6.2.4](#).

Functions of Flash Kernel B:

- Device Firmware Upgrade (DFU)
- Erase
- Verify
- Unlock Zone 1
- Unlock Zone 2
- Run
- Reset

Flash Kernel B is a more robust kernel than Kernel A. It communicates with the host PC application provided in controSUITE (controlSUITE/device_support/~Utilities/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 bootloader, the kernel first initializes the PLLs of the device, initializes SCIA, and seizes the flash pump, if necessary. It then waits for an 'a' or 'A' from the host in order to perform an autobaud 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 2](#) 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 case of an DFU, the kernel receives a file in the hex boot format byte-by-byte from the SCI module and echoes the byte back to the host. This is different from Flash Kernel A, which sends back a checksum of a block of data. After receiving 64 bits 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 with ECC enabled using the F021_api_f2837xD_C28x.lib. Afterwards, it verifies that the data and ECC were programmed correctly into flash. 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 the kernel. However, Flash Kernel B provides an erase function independent of the DFU, which gives the user the ability to erase specific sectors or the entire flash of the device.

Similarly, the verify operation 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.

[Section 5.1.1](#) details the packet format, commands, and protocols. All command packets except for DFU and Reset require data to be sent to the kernel, which is used for that command. The details discussed in [Section 5.1.2](#) and [Section 5.1.3](#) are for the F2837xD flash kernels for CPU1 and CPU2, respectively. They have identical functionality except for two additional commands that CPU1 can process in order to boot CPU2. However, the commands have different values for CPU1 and CPU2. This helps to ensure correctness when using the flash kernels for flash solutions. Single core devices (F2807x, F2837xS) accept CPU1 commands minus the two boot CPU2 commands.

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 2. 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 a 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 3. 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 and F2837xS single core device kernels excluding *Run CPU1 Boot CPU2* and *Reset CPU1 Boot CPU2*. A brief description of the command codes are provided in [Table 4](#).

Table 4. CPU1 Kernel Commands

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

Table 4. CPU1 Kernel Commands (continued)

Kernel Commands	Command Code	Description
Run CPU1 Boot CPU2 ⁽¹⁾	0x0004	<ol style="list-style-type: none"> 1. Receive the packet with 32-bit address 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. 3. Branch to the address
Reset CPU1 Boot CPU2 ⁽¹⁾	0x0007	<ol style="list-style-type: none"> 1. Receive the packet with no data 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. 3. Break the while loop and enable WatchDog Timer to time-out and reset.

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

5.1.3 CPU2 Kernel Commands

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

Table 5. CPU2 Kernel Commands

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

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 6. Erase Packet

Header	Data Length	Command	Data	Checksum	Footer
0x1BE4	6 (bytes)	0x0300 CPU1 0x0400 CPU2	32-bit Data	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 7. Unlock Packet

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

- **Run**

- 32-bit address

Table 8. 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. 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 and 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 9 displays the status codes.

Table 9. 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

Table 9. Status Codes (continued)

Status Code	Value	Description
UNLOCK_ERROR	0x6000	Return on Unsuccessful Unlock

6 Example Implementation

The kernels described above are available in controlSUITE under examples folder for the specific device within the `\device_support` folder. The host application is found in the `\device_support\Utilities` folder in controlSUITE. 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 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 which is using the SCI bootloader. After this, the tool's functionality differs depending on the device and kernel being used.

6.1 Device Setup

6.1.1 Kernels

The source files and project files for Code Composer Studio (CCS) are provided in controlSUITE (www.ti.com/controlsuite), in the corresponding device's *device support* directory. Load the project into CCS and build the project. In these projects is a post-build step which converts the compiled and linked .out file to the correct boot hex format needed for the serial_flash_programmer and saves it as the example name with a .txt file 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 pins are configured properly to boot the device to SCI boot mode (see [Section 3.1](#)). 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 two GPIO pins, which can connect to the device. On some systems, like the controlCARD, an FTDI chip can be used to connect the GPIO pins used for SCI communication via a USB Virtual COM port. In this case, the PC must connect to the mini-usb on the device and use channel B of the FTDI to connect to the GPIO pins. 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 PC Application: serial_flash_programmer

6.2.1 Overview

The command line PC utility is a lightweight (~128KB 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 controlSUITE (www.ti.com/controlsuite) in the `device_support\Utilities\serial_flash_programmer` folder.

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. Below describes the command line usage of the tool:

```
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, f2837xD, f2837xS, or f2807x.
-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 baudrate is omitted, the communication will occur at 9600 baud.

NOTE: Both the flash kernels and flash application MUST be in the SCI8 boot format. This was discussed earlier in [Section 3.1](#) and can be generated from the OUT file using the hex2000 utility.

6.2.2 Building serial_flash_programmer in Visual Studio

Serial_flash_programmer.cpp can be compiled using Visual Studio.

NOTE: If Microsoft Visual Studio is not installed, a free version of Microsoft Visual Studio express can be found [here](#).

1. Navigate to the serial_flash_programmer directory.
2. Double click the serial_flash_programmer.sln to open the Visual Studio project.
3. When Visual Studio opens, select Build → Build Solution.
4. After Visual Studio completes the build, select Debug → serial_flash_programmer properties.
5. Select Configuration Properties → Debugging.

6. Select the input box next to the Command Arguments.
7. Type the arguments in the following format. The arguments are described in [Section 6.2.1](#).
(a) Format:

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

- (b) 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 will first load the f28069_flash_kernel into RAM of the device using the bootloader. Then, the kernel will execute and load and program flash with the file specified by the '-a' command line argument.

6.2.4 Running serial_flash_programmer for F2827xD (Flash Kernel B)

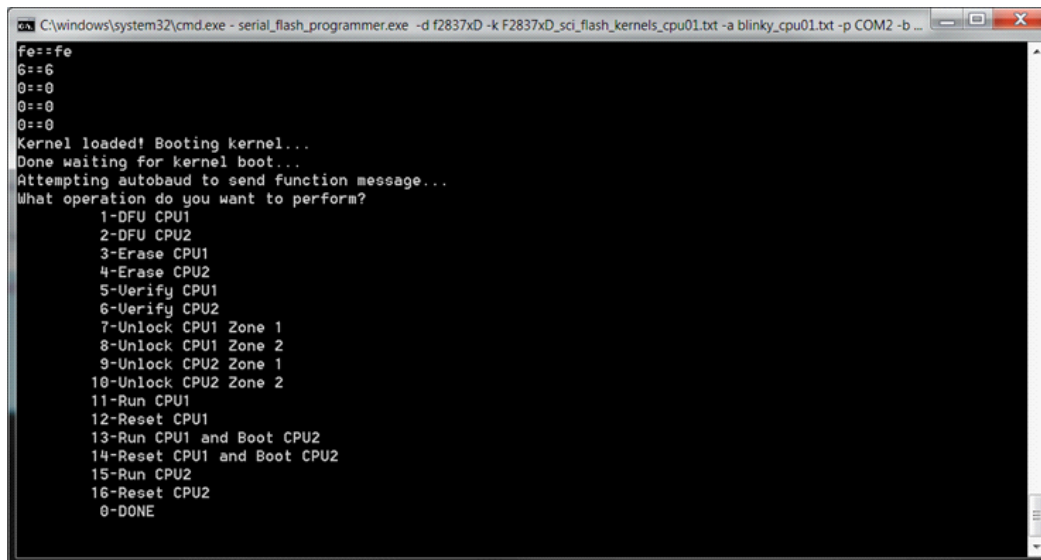
NOTE: It is recommended to reset the device before running serial_flash_programmer so that Autobaud will complete correctly.

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 f2837xD -k <~\F2837xD_sci_flash_kernels_cpu01.txt>  
-a <file> -m <~\F2837xD_sci_flash_kernels_cpu02.txt> -n <file> -p COM<num>
```

This will automatically connect to the device, perform an autobaud 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 3](#)). Select the appropriate number and then provide any necessary information when asked for that command (described in [Section 5.1](#)).



```

C:\windows\system32\cmd.exe - serial_flash_programmer.exe -d f2837xD -k F2837xD_sci_flash_kernels_cpu01.txt -a blinky_cpu01.txt -p COM2 -b ...
Fe==fe
6==6
0==0
0==0
0==0
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 3. serial_flash_programmer Prompting for Next Command

7 References

1. *TMS320x2802x Piccolo Boot ROM Reference Guide* ([SPRUFN6](#))
2. *ROM Code and Peripheral Booting* section from the *TMS320F2837xD Dual-Core Delfino Microcontrollers Technical Reference Manual* ([SPRUHM8](#))
3. Piccolo Flash API User's Guide - located within controlSUITE at: (/controlSUITE/libs/utilities/flash_api/DEVICE/VERSION/doc)
4. *C2000 F021 Flash API Reference Guide* ([SPNU595](#))
5. *TMS320C28x Assembly Language Tools User's Guide* ([SPRU513](#))

Revision History

Changes from Original (March 2014) to A Revision	Page
• Updated information in the Abstract.	1
• Updated information in Section 1	2
• Updated information in Section 2	2
• Updated information in Section 3.1	3
• Updated information in Section 3.2	4
• Updated information in Section 4	5
• Updated information in Section 4.1	5
• Updated information in Section 4.1.1	6
• Added new Section 5	6
• Updated information in Section 5.1	6
• Updated information in Section 5.1.1	7
• Updated information in Section 5.1.2	8
• Updated information in Section 5.1.3	9
• Updated information in Section 5.1.4	10
• Updated information in Section 5.1.5	10
• Updated information for Section 6	11
• Updated information in Section 6.1.2	11
• Updated information in Section 6.2.1	12
• Updated information in Section 7	14

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

IMPORTANT NOTICE

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

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

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

TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI components or services are used. Information published by TI regarding third-party products or services does not constitute a license to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of significant portions of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

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

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

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

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

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

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

Products

Audio	www.ti.com/audio
Amplifiers	amplifier.ti.com
Data Converters	dataconverter.ti.com
DLP® Products	www.dlp.com
DSP	dsp.ti.com
Clocks and Timers	www.ti.com/clocks
Interface	interface.ti.com
Logic	logic.ti.com
Power Mgmt	power.ti.com
Microcontrollers	microcontroller.ti.com
RFID	www.ti-rfid.com
OMAP Applications Processors	www.ti.com/omap
Wireless Connectivity	www.ti.com/wirelessconnectivity

Applications

Automotive and Transportation	www.ti.com/automotive
Communications and Telecom	www.ti.com/communications
Computers and Peripherals	www.ti.com/computers
Consumer Electronics	www.ti.com/consumer-apps
Energy and Lighting	www.ti.com/energy
Industrial	www.ti.com/industrial
Medical	www.ti.com/medical
Security	www.ti.com/security
Space, Avionics and Defense	www.ti.com/space-avionics-defense
Video and Imaging	www.ti.com/video

TI E2E Community

e2e.ti.com