Application Report
SPRACO7–October 2019

USB Flash Programming of C2000™ Microcontrollers

Rajaravi Krishna Katta, Harshmeet Singh, and Salvatore Pezzino

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 the USB flash programming implementation for C2000 devices and provides PC utilities to evaluate the solution with.

Contents

List of Figures

Trademarks

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 performing a firmware update by including some basic loading utilities in ROM. Depending on the device and the communications peripherals present, the 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 C2000Ware. While this implementation is targeted at C2000 devices using the Universal Serial Bus (USB) 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'. One of the main limitations of flash is that it can only be erased a sector at a time.

The underlying principle for how a flash memory functions are 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 the 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 and all flash operations. 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, namely 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.

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: Please refer the *Boot ROM* section of the device-specific technical reference manual (TRM) to configure the device in USB Boot mode.

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).

In the implementation described in this application report, the USB loader is used. If this is the case when the device boots, the USB bootloader in ROM begins executing and waits for the USB enumeration to complete. 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}" "-boot -b ${BuildArtifactFileName}" -o "${BuildArtifactFileBaseName}.dat"

Alternatively, you can use the TI hex2000 utility to convert .out files into the correct boot hex format.

hex2000.exe -boot -b <input_file_name.out> -o <output_file_name.dat>

3.2 Code Load

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 word-by-word and sent out of the USB port without any modification.

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

Data control flow of the ROM USB bootloader:

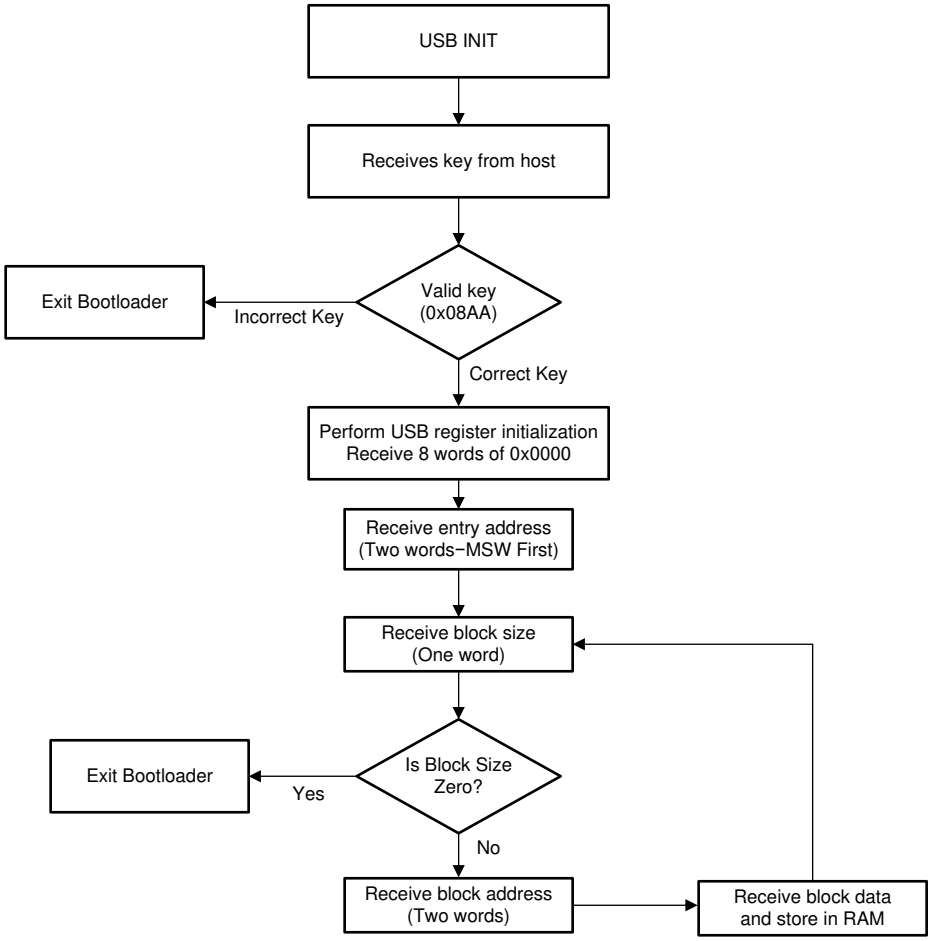

Figure 1. Bootloader Control Flow

4 Flash Kernel

4.1 Implementation

This flash kernel is similar to the USB loader in ROM. To enable the kernel to erase and program flash, the kernel is equipped with the flash API. The flash kernel essentially performs the same operation as the ROM bootloader, however, because it is equipped with the flash API it is able to erase and program flash to perform the firmware update.

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 USB boot mode. At this point, the device is waiting for the USB Connection. After USB Driver is detected the flash kernel can be transferred in bulk. 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. USB is configured to work in Bulk Transfer Mode. After getting the data, wait for any ongoing transfer to complete and then disconnect from the bus, disable USB interrupts, and reset the USB module. Then Bypass the AUX and CPU Clock.

At the beginning of the download process some preliminary data is transferred before the actual flash application code, including a key, a few reserved fields, and the application entry point. 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.

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.

4.1.1 Process of loading CPU02 kernel (and application) through CPU01 kernel

The CPU01 kernel implementation sets up a USB connection with a host and receives a binary application for CPU01 in SCI 8 format to run on the device and program it into Flash. Then, CPU01 receives a CPU02 kernel and loads that into shared RAM.

NOTE: For F2837xD, only CPU01 has access to the USB module. So, CPU02 depends on CPU01 to receive the data.

CPU01 then boots CPU02 with an IPC message and CPU01 continues to receive another binary application to be run in CPU02 flash and it transmits the binary application to CPU02 through IPC. CPU02 reads the application from IPC and programs it into flash. After CPU01 and CPU02 completes execution, they both branch to their respective applications programmed in their respective Flash Banks. Refer below image for understanding.

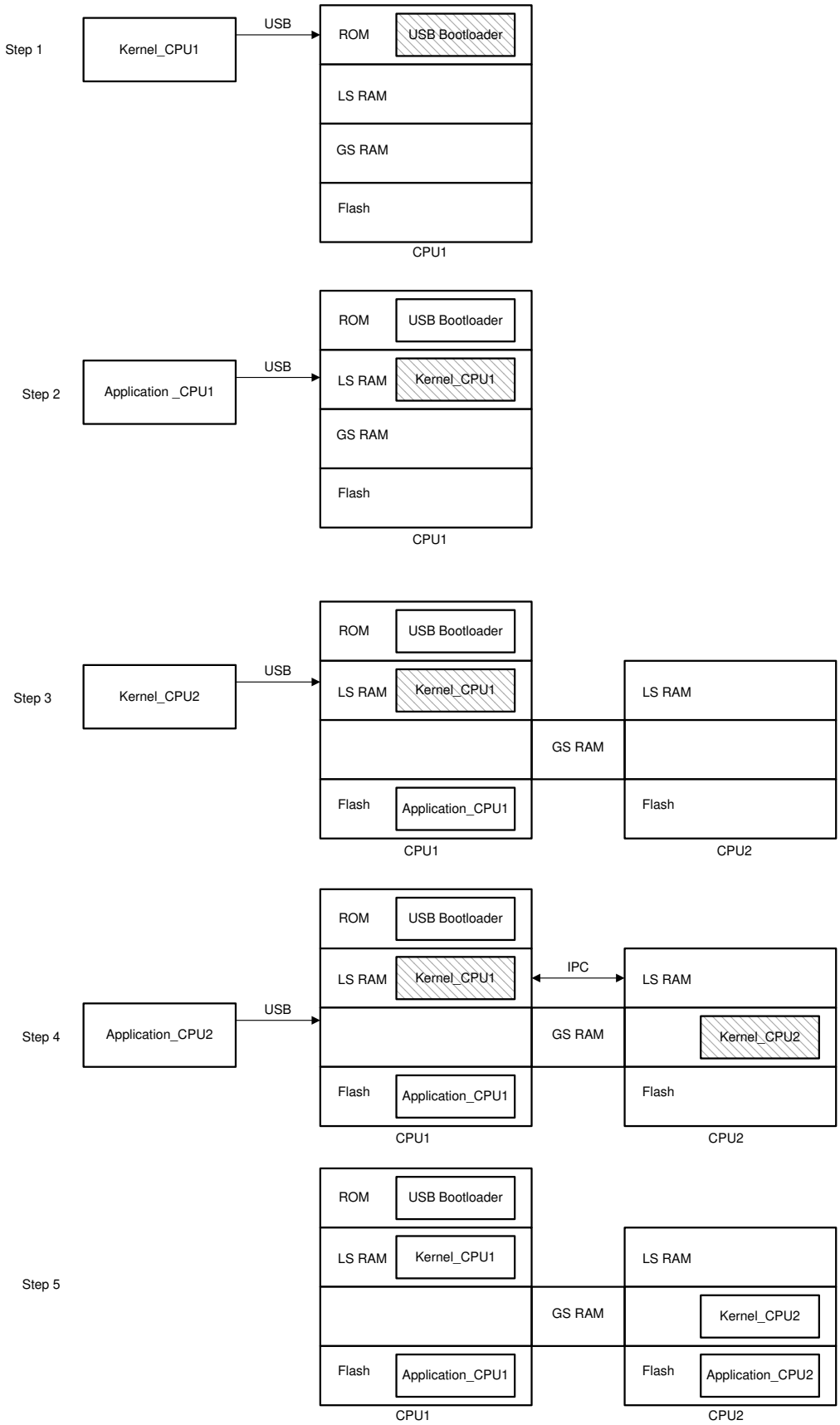

Figure 2. Steps Involved in Loading Kernel and Application Images

1. Step#1: Load kernel to CPU1 LSRAM using USB Bootloader which resides in ROM.
2. Step#2: Execute the CPU1 Kernel to receive the application data over USB and program the flash application to CPU1 Flash.
3. Step#3: CPU1 Kernel receives CPU2 Kernel over USB and programs to shared RAM.
4. Step#4: CPU1 Kernel receives Application_CPU2 over USB, sends data to CPU2 Kernel using IPC. CPU2 Kernel Application_CPU2 programs to CPU2 Flash.
5. Final image shows where the kernels and applications reside at the end.

5 Example Implementation

The kernel described above is available in {C2000Ware_Directory}/device_support/f2837xD/dual/F2837xD_usb_flash_kernels. The host application is found in the C2000Ware (C2000Ware_x_xx_xx_xx/utilities/flash_programmers/usb_flash_programmer). The source and executable are found in the *usb_flash_programmer* folder. This section details the usb_flash_programmer: how to build, run and use it with Flash Kernel.

5.1 Device Setup

5.1.1 Kernels

The source files 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. 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 usb_flash_prorgammer and saves it as the example name with a .dat file extension.

5.1.2 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 usb_flash_programmer.

1. Make sure the boot mode pins are configured properly to boot the device to USB boot mode (see Section 3.1).
2. Connect the Micro USB Cable to the Control card from the PC and then Open Device Manager.
3. Click on the Unrecognized device and update the device driver from the path : {C2000Ware_Directory}/utilities/flash_programmers/usb_flash_programmer/windows_driver. F28x7x USB Bootloader should display on the device manager.

This should boot the device to USB boot mode.

5.2 PC Application: usb_flash_programmer

5.2.1 Overview

The command line PC utility is a lightweight (~64KB 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/usb_flash_programmer/src/VS2010_USBLoader2000).

To use this tool to program the C2000 device, ensure that the target board has been reset and is currently in the USB boot mode and connected to the PC. Below describes the command line usage of the tool:

```
usb_flash_programmer.exe [-l] [-q] [-h] <input file names>
```

Option	Description
-q	- Quiet mode. Disable output to stdout.
-l	- List attached devices without programming.
-h	- Print this usage information

NOTE: Input filenames are loaded in the ascending order on the command line

5.2.2 Building usb_flash_programmer

The USB loader client performs the following actions:

1. Parse the command line options.
2. Read data from the input file.
3. Attempt to open the USB device.
4. Read and print the USB string descriptors.
5. Send the input file data via a bulk OUT transfer.
6. Clean up the USB library environment.
7. Return a pass/fail to the calling environment.
8. If multiple arguments: Go to 1.

Command line options and file IO can be done through the C standard library, but USB operations can only be done through the operating system's device driver framework. There are two widely-used libraries that provide this capability. The first is libusb, an open-source (LGPL) library that features a Unix-style API. The second is WinUSB, which is part of the Windows Driver Development Kit. Both libraries run in user mode and provide generic access to USB devices without the need for a customer driver. Libusb is very easy to use and is also available on Linux, but it's somewhat slower and any distribution is complicated by the license. WinUSB is harder to use, but is faster and the resulting software is simpler to distribute. The precompiled version of usb_flash_programmer.exe included with this package uses WinUSB, but source code is provided for both libraries.

Similar to the libraries, there are two common compilers available for Windows development. One is MinGW, a port of the GNU Compiler Collection (GCC). The other is Microsoft's Visual Studio. These are as different as night and day, but describing them is beyond the scope of this document. Both are supported in this package.

The source code is divided into several files. Command line options and file IO are handled by the generic main file (usb_flash_programmer.c). The main() function works with the chosen USB library through generic wrapper functions. These are defined in usb_flash_programmer.h along with the device driver GUIDs and USB vendor and product IDs. Library-specific functionality is implemented in libusb_wrapper.c and winusb_wrapper.c. These #include library-specific header files and link with a static library file.

To use WinUSB, compile and link together the following files:

- usb_flash_programmer.c
- winusb_wrapper.c
- setupapi.lib (from the DDK)
- winusb.lib (from the DDK)

WinUSB header files are found in:

- C:\WinDDK\7600.16385.1\inc\api
- C:\WinDDK\7600.16385.1\inc\ddk

WinUSB libraries are around in:

- C:\WinDDK\7600.16385.1\lib\win7\i386

To use libusb, compile and link together the following files:

- usb_flash_programmer.c
- libusb_wrapper.c
- MinGW: libusb-1.0.a (from libusb for MINGW32)
- VS: libusb-1.0.lib (from libusb for MS32)

Libusb is not included in this package. Windows binaries can be downloaded from their web site at http://libusb.info. To use the makefile or Visual Studio projects included in this package, put libusb-1.0.a, libusb-1.0.lib, and libusb.h in the libusb subdirectory below the main C files.

To use Visual Studio, open the solution in the VS2010_USBLoader2000 subdirectory. There will be two projects: one for WinUSB and one for libusb. Build the one for your chosen library. The resulting executable will be in the build_libusb or build_winusb subdirectory.

To use MinGW, use the provided makefile. To build a libusb executable, run 'make loader_libusb'. To build a WinUSB executable, run 'make loader_winusb'. Alternatively, run 'mingw32-make loader_libusb' and 'mingw32-make loader_winusb'.

To use MinGW to build a winusb executable, check winusb_wrapper.c. You may need to manually change NTDDI_VERSION for your system.

5.2.3 Running usb_flash_programmer for F2837xD

1. Navigate to the folder containing the compiled usb_flash_programmer executable.
2. Run the executable usb_flash_programmer.exe with the following command
   a. >.\usb_flash_programmer.exe <~\F2837xD_usb_flash_kernels_cpu01.dat> <\blinky_cpu01.dat> <~\F2837xD_usb_flash_kernels_cpu02.dat> <~\blinky_cpu02.dat>

For more information, see Figure 2.

6 References

1. Texas Instruments: *ROM Code and Peripheral Booting* section from the *TMS320F2837xD Dual-Core Delfino Microcontrollers Technical Reference Manual* (SPRUHM8)
2. Texas Instruments: *C2000 F021 Flash API Reference Guide* (SPNU595)
3. Texas Instruments: *Serial Flash Programming of C2000 Microcontrollers*
4. C2000Ware Installer

IMPORTANT NOTICE AND DISCLAIMER

TI PROVIDES TECHNICAL AND RELIABILITY DATA (INCLUDING DATA SHEETS), 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, regulatory 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 or other applicable terms available either on 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.

TI objects to and rejects any additional or different terms you may have proposed.

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265