User's Guide Getting Started with Bootloading on C2000[™] Microcontrollers



Matt Kukucka

ABSTRACT

Embedded processors often need to be programmed in situations where a JTAG debug probe cannot be reliably used to program the target device. In these cases, engineers need to rely on programming methods leveraging peripherals such as USB (Universal Serial Bus) or Controller Area Network Flexible Data-Rate (CAN-FD). C2000[™] devices aid in this endeavor through the inclusion of several bootloading utilities in the Boot ROM to load firmware into the on-chip RAM. These utilities are useful, but can be initially confusing to understand and debug in reality. This document introduces the fundamental bootloading configurations and describes how to leverage the most common boot modes to load application code into the on-chip flash.

Table of Contents

1 Introduction	4
2 Configuring the Boot Mode	<mark>6</mark>
2.1 Standalone Boot	<mark>6</mark>
2.1.1 Boot Mode Select Pins (BMSP)	7
2.1.2 Boot Definition Table (BOOTDEF)	8
2.1.3 Boot ROM OTP Configuration Registers	10
2.1.4 CPU2 Boot Flow	10
2.2 Emulation Boot	12
3 Programming the Flash	13
3.1 Flash API	14
3.2 Flash Kernels	14
4 Bootloading Code to Flash	16
4.1 C2000 Hex Utility	16
4.2 Common Boot Modes	17
4.2.1 Boot to Flash	17
4.2.2 SCI Boot	20
4.2.3 CAN Boot	27
4.2.4 CAN-FD Boot	35
4.2.5 USB Boot	42
5 FAQ	48
5.1 Selecting the BMSP GPIOs with a Software-based Implementation	48
5.2 Running a Flash Kernel from the Flash Instead of the RAM	48
5.3 No Symbols Defined When Debugging Boot ROM	51
5.4 Writing Values in the OTP Using the On-Chip Flash Tool	53
5.5 Writing Values in the OTP Using the Flash API Plugin	55
6 Summary	56
7 References	57

List of Figures

Figure 1-1. General Bootloader Design Flow	5
Figure 3-1. On-Chip Flash Tool Location in CCS v12	13
Figure 3-2. On-Chip Flash Tool Location in CCS v20	13
Figure 4-1. Adding Post-Build Steps to Invoke the Hex Utility	16
Figure 4-2. Opening the Target Configuration Menu in CCS.	. 18
Figure 4-3. Launching a Target Configuration in CCS	. 18



Figure 4-4. Connecting to the Target Core in CCS	
Figure 4-5. Navigating to the Memory Browser in CCS	
Figure 4-6. Emulating a Zero-pin Boot to Flash (0x0009 0000).	
Figure 4-7. Finding the Converted SCI Kernel Output File.	
Figure 4-8, F2800157 LaunchPad UART Routing Schematic	22
Figure 4-9. Routing the SCI TX or RX (GPIO28 or GPIO29) Signals to the BoosterPack (BP) Pins	
Figure 4-10 Jumping the XDS TX_BX to the SCI TX_BX GPIOS	23
Figure 4-11 Opening the Target Configuration Menu in CCS	23
Figure 4-12 Launching a Target Configuration in CCS	
Figure 4-12. Connecting to the Target Core in CCS	
Figure 4-14. Navigating to the Memory Browser in CCS	
Figure 4-14, Revulating a Zero pin SCI Boot with SCIPYDA to CPIO28 and SCITYDA to CPIO20	27
Figure 4-16. Ending the VDS COM part in Device Manager	25
Figure 4-10. Finding the ADS COM Fort in Device Manager	
Figure 4-17. Commanding the Senai Flash Flogrammento Download the Application	
Figure 4-10. Ruffling the Application Alter Loading the Application into Flash	20 20
Figure 4-19, F200030 Lautoff ad CAN Halscevel and Connector Sciential	20
Figure 4-20. Adding a Frederined Symbol for the Context Laurici Fad CAN GFIO Assignments	
Figure 4-21. Including the Local CAN Timing Reader File	
Figure 4-22. Comming the CAN timing setungs.	
Figure 4-23. Generaling the DCAN kernel ixt output.	
Figure 4-24. Setting the Cardinal Switch.	
Figure 4-25. Opening the Target Configuration Menu in CCS	
Figure 4-26. Launching a Target Configuration in CCS.	
Figure 4-27. Connecting to the Target Core in CCS	
Figure 4-28. Navigating to the Memory Browser in CCS.	
Figure 4-29. Emulating a Zero-pin CAN Boot with CANRXA=GPI05 and CANTXA=GPI04	
Figure 4-30. DCAN Flash Programmer Kernel Loaded	
Figure 4-31. DCAN Flash Programmer Application Loaded	
Figure 4-32. F280039C LaunchPad CAN Transceiver and Connector Schematic	36
Figure 4-33. Adding a Predefined Symbol for the Correct LaunchPad CAN GPIO Assignments	37
Figure 4-34. Generating the CAN Kernel txt Output.	37
Figure 4-35. Setting the CAN Routing Switch	38
Figure 4-36. Opening the Target Configuration Menu in CCS	39
Figure 4-37. Launching a Target Configuration in CCS	
Figure 4-38. Connecting to the Target Core in CCS	
Figure 4-39. Navigating to the Memory Browser in CCS	40
Figure 4-40. Emulating a Zero-pin MCAN Boot with CANRXA=GPIO5 and CANTXA=GPIO4	40
Figure 4-41. MCAN Flash Programmer Kernel Loaded	42
Figure 4-42. MCAN Flash Programmer Application Loaded	42
Figure 4-43. Generating the Correct USB Kernel Bin Output	43
Figure 4-44. Configure the F2837xD controlCard to Debug Using CCS and the Onboard XDS Emulator	44
Figure 4-45. Opening the Target Configuration Menu in CCS	44
Figure 4-46. Launching a Target Configuration in CCS	45
Figure 4-47. Connecting to the Target Core in CCS	45
Figure 4-48. Navigating to the Memory Browser in CCS	45
Figure 4-49. Configuring the F2837xD controlCard to Emulate USB Boot	46
Figure 4-50. Browsing C2000Ware for the USB Windows Drivers	46
Figure 4-51. The F28x7x USB Boot Loader device appearing in the Device Manager	47
Figure 5-1. Adding a File to a Project into CCS	49
Figure 5-2. Excluding the RAM Linker Command File from the Compiler Build	49
Figure 5-3. Creating a New Build Configuration for the Project	50
Figure 5-4. Creating the CPU1_FLASH Build Configuration	50
Figure 5-5. Add the Predefined Symbol _FLASH to Initialize Flash Functions	51
Figure 5-6. Opening the Target Configuration Menu in CCS	52
Figure 5-7. Launching a Target Configuration in CCS	52
Figure 5-8. Connecting to the Target Core in CCS	
Figure 5-9. CCS View When No Boot ROM Symbols Are Loaded	52
Figure 5-10. Navigating to Load Symbols in CCS.	53
Figure 5-11. Locating the Boot ROM Source Files	
Figure 5-12. Example 1: Flash Plugin Boot Configuration Programmed	
Figure 5-13. Example 2: Flash Plugin Boot Configuration Programmed	



List of Tables

Table 2-1. F280015x Device Default Boot Modes	6
Table 2-2. Device Default Boot Mode Select Pins	6
Table 2-3. Boot Configuration Type Per Device Family	7
Table 2-4. BOOTPIN-CONFIG Bit Fields	8
Table 2-5. BOOTDEF Bit Fields	9
Table 2-6. F280015x Device Boot Modes	9
Table 2-7. F280015x Boot ROM Registers	10
Table 2-8. CPU1TOCPU2IPCBOOTMODE Register Details	11
Table 2-9. Emulation Boot Register Locations.	12
Table 5-1. Resulting Zero Pin Boot Configuration	54
Table 5-2. Two Boot Mode Select Pin Configuration	55
•	

Trademarks

C2000[™], LaunchPad[™], Code Composer Studio[™], and BoosterPack[™] are trademarks of Texas Instruments. Microsoft Visual Studio[®] is a registered trademark of Microsoft Corporation. All trademarks are the property of their respective owners.



1 Introduction

Note

This document only applies to C28x-based microcontrollers. For details on C29x bootloading, please refer to the device-specific Technical Reference Manual.

As applications grow in complexity, the capability for bug squashing, feature additions, and embedded firmware modification is increasingly essential, especially for maintaining device safety and security integrity in the field. C2000 devices accomplish an implementation of firmware updates by offering simple loading utilities in the boot ROM (factory-programmed, Read-Only Memory).

ROM loaders, often referred to as bootloaders, reside in the target device's boot ROM and allow application code to be loaded from an external host through software. Bootloaders serve as a reliable alternative to JTAG debug probes, which require expensive and specialized hardware with direct access to the target device.

Albeit device dependent, users can select from various peripheral bootloaders as a medium to load firmware onto the target device, such as:

- 1. Serial Communications Interface (SCI)
- 2. Serial Peripheral Interface (SPI)
- 3. Inter-Integrated Circuit (I2C)
- 4. Controller Area Network (CAN)
- 5. Controller Area Network Flexible Data-rate (CAN-FD)
- 6. Universal Serial Bus (USB)
- 7. Parallel GPIOs

Every C2000 device has a subset of default boot modes available to be selected. However, if the user requires access to a boot mode not offered in the default boot table or the flexibility for using different GPIO assignments, the One-Time Programmable (OTP) memory must be configured. The OTP registers allow for alternate boot modes to be selected if not offered in the default boot table.

If the user opts to use a peripheral bootloader to load new code onto the device, the application image must be generated in a specific format beforehand, as described in Section 4.1. Once the application is prepared, the data transfer between the boot ROM and host device can proceed over the selected peripheral bootloader. The application code is then loaded into the on-chip RAM by the bootloader and executed.

Peripheral bootloaders are present in every C2000 device's ROM and are simple to use, but are limited to only load code into RAM. Flash kernels bridge the gap between the ROM and Flash by providing an intermediary means to write code into the Flash from the RAM, as described in Section 3.2.

However, users getting started with C2000 bootloading can fall into pitfalls that prove challenging to debug if there is a single misstep along the boot flow (see Figure 1-1 for a general design overview). There are four phases to bootloader design that this report seeks to clarify:

- 1. Choosing and configuring the appropriate boot mode
- 2. Preparing an application to be loaded onto the device through a peripheral bootloader in the boot ROM
- 3. Loading an application into RAM with a bootloader
- 4. Using a flash kernel to program the Flash





Figure 1-1. General Bootloader Design Flow

2 Configuring the Boot Mode

At the end of the ROM boot sequence, the device decides if emulation or standalone boot needs to be entered depending on if there is a JTAG debugger connected. This is achieved by reading the "DCON" bit in the JTAG state machine (except for on F2837xD/F2837xS/F2807x devices, which poll the "TRSTn" pin). Emulation boot mimics the standalone boot flow by sourcing registers located in the RAM that are identical in structure and configuration to the boot registers in the One-Time Programmable (OTP) memory. Consequently, emulation boot allows for the boot registers to be written to as many times as necessary.

Note

TI highly recommends using emulation mode to debug and verify correctness of the device boot configurations before attempting to program the OTP, as you can only program the OTP once.

This chapter details the configuration and usage of standalone boot (in Section 2.1) and emulation boot (in Section 2.2), but note that the emulation boot flow is demonstrated later in Section 4.2.

2.1 Standalone Boot

Note

This chapter is based on the F280015x family of devices, but can be applied any device that employs the BOOTPIN-CONFIG/BOOTDEF registers, as listed in Table 2-3. Device specific information can be found in the Boot ROM chapter of the device's Technical Reference Manual (TRM).

On every CPU reset, the device executes a pre-defined boot sequence in the boot ROM depending on the reset type and boot configuration. After successfully initializing the device (assuming no debugger is connected), the Boot Mode Select Pins (BMSPs) are polled to determine which boot mode to invoke. The BMSPs can be mapped to external GPIO pins, either defined by the device's default configuration (Table 2-1) or the user's custom definition in the OTP memory (see Section 2.1.1 for more details). Referred to as standalone boot, this procedure provides more flexibility for conducting firmware updates through the peripheral modules on the device rather than having a debugger directly connected.

All the latest C2000 devices are pre-programmed with boot modes that can be selected using the default BMSPs, given per device family in Table 2-2. From the selection of factory default boot modes, the device has the option to boot directly to the Flash memory or load new application code into RAM over pre-determined peripheral communication modules, without requiring any boot registers to be programmed. However, what if the user needs to use a peripheral not included in the default boot modes or needs more customizing in the boot options?

Boot Mode	GPIO24 (Default boot mode select pin 1)	GPIO32 (Default boot mode select pin 0)
Parallel IO	0	0
SCI / Wait Boot	0	1
CAN	1	0
Flash	1	1

Table 2-1, F280015x Device Default Boot Modes

Table 2-2. Device Default Boot Mode Select Pins			
	GPIO 24 and GPIO 32	GPIO 72 and GPIO 84	
Device Families	F28002x,	F2837xD,	
	F28003x,	F2837xS,	
	F28004x,	F2807x,	
	F280013x,	F2838x,	
	F280015x,	F28P65x	
	F28P55x		

the O.O. Deside Defended Dest Mede Orders Dis



2.1.1 Boot Mode Select Pins (BMSP)

The Boot Mode Select Pins (BMSPs) are decoded by the device and used to index the boot definition table, determining which boot mode to execute. Besides the default boot configurations available on every device, users can opt for other BMSPs and boot modes by programming the OTP memory. While the memory contents for a ROM are determined at manufacturing time, the OTP can be programmed only once after production allowing for more flexibility in applications where reliable and repeatable reading of data is required.

In the context of standalone boot loading, programming the OTP is required when:

- 1. A boot option is not supported by the default boot options
- 2. Different GPIOs are required for the peripherals or BMSPs
- 3. Different entry point to the application is required
- 4. The flexibility of using multiple boot options is required

Depending on the device families (as listed in Table 2-3), the BMSPs can be modified by writing to the respective BOOTPIN-CONFIG or BOOTCTRL memory locations in the user-configurable Dual Code Security Module (DCSM) OTP [7].

Note

In the DCSM, there are two independent secure zones to which securable resources can be assigned – Zone 1 (Z1) and Zone 2 (Z2). The security module restricts the CPU access to on-chip secure memory and resources without interrupting or stalling CPU execution. When a read occurs to a secure memory location, the read returns a zero value and CPU execution continues with the next instruction. This, in effect, blocks read and write access to secure memories through the JTAG port. Note that insecure resources can still be accessed by the JTAG debugger.

The BOOTPIN-CONFIG register is a 32-bit wide location consisting of four 8-bit wide partitions. Three partitions are designated for the BMSPs and the final partition is a key that designates validity of the OTP configuration. This register can be found in the DCSM OTP as Z1-OTP-BOOTPIN-CONFIG and Z2-OTP-BOOTPIN-CONFIG depending on which zone is being configured.

Note

The configurations programmed in Z2 take priority over the configurations in Z1. Therefore, TI recommends to use the Z1 location first and then Z2 next if the OTP configurations need to be altered.

The BOOTPIN-CONFIG registers in the DCSM OTP can be programmed using the On-Chip Flash tool in CCS or Flash API (see Section 5.4 or Section 5.5, respectively for steps), or graphically with the DCSM tool in SysConfig [8].

BOOTPIN-CONFIG and BOOTDEF Devices	BOOTCTRL Devices
F28002x, F28003x, F28004x, F280013x, F280015x, F2838x, F28P55x, F28P65x	F2807x, F2837xD, F2837xS

 Table 2-3. Boot Configuration Type Per Device Family

The BMSPs can be set to almost any GPIO (see the *Configuring Boot Mode Pins* chapter in the device-specific TRM for exceptions) to be used during boot up, where GPIO0 is 0x0, GPIO1 is 0x01, and so on. Although the BMSPs need to be manually pulled high or low with the external GPIO pins in most cases, a software-controlled firmware update is possible following the method described in Section 5.1.

The BMSPs can also be used in both the boot ROM and the application afterward (with alternate functionality) as long as the hold time for boot-mode pins is not violated (see *Reset - XRSn - Timing Requirements* in the data sheet) [29].



Note

However, if a LaunchPad[™] or controlCard is being used, then the default BMSPs are manually pulled up/down with the external boot switches and cannot be safely used in an application after.

The number of BMSPs used either expands or restricts the potential boot modes selectable in the boot table exponentially. If three BMSPs are used, then up to 8 boot options are selectable. Reducing to two BMSPs means only four boot options are available. Using zero BMSPs means that a single boot option is automatically selected, eliminating the need for external manipulation of the GPIOs as well as freeing up other pins needed to be repurposed for boot pins.

Disabling any particular BMSP can be achieved by writing *0xFF* to the same BOOTPIN-CONFIG memory location as when changing the GPIO number used. When decoding the boot mode, BMSP0 is the least-significant bit and BMSP2 is the most-significant bit of the boot table index value. TI recommends to start with disabling BMSP2 when disabling BMSPs.

In standalone boot, if the Z1 or Z2 OTP loaded registers are not written to with the correct BOOTPIN-CONFIG_KEY (0x5A) designating register validity, then the default BMSPs are decoded to index the default boot table.

Bit	Name	Description
31:24	Key	Write 0x5A to these 8-bits to tell the boot ROM code that the bits in this register are valid.
23:16	Boot Mode Select Pin 2 (BMSP2)	Refer to BMSP0 description.
15:8	Boot Mode Select Pin 1 (BMSP1)	Refer to BMSP0 description.
7:0	Boot Mode Select Pin 0 (BMSP0)	Set to the GPIO pin to be used during boot (up to 255).
		0x0 = GPIO0, 0x01 = GPIO1, and so on.
		Writing 0xFF disables this BMSP and this pin is no longer used to select the boot
		mode.

Table 2-4. BOOTPIN-CONFIG Bit Fields

Note

In the case when BMSP2 is only used (BMSP1 and BMSP0 are disabled), then only the boot table indexes of 0 and 4 are selectable. In the case when BMSP0 is only used, then the selectable boot table indexes are 0 and 1. This is important to keep in mind when programming the boot options in the boot table as each BMSP still maintains positional weight regardless of the condition of the other BMSPs.

Note

Devices that use the BOOTCTRL register (see Table 2-3) have a different boot flow and configuration from the device families that use the BOOTPIN-CONFIG register.

When programmed with a valid key, the BOOTCTRL register allows for different GPIOs to be used as the two-boot mode select pins. The total customizable BMSPs for BOOTCTRL devices is fixed at two, as opposed to the maximum of three for BOOTPIN-CONFIG devices. However, the same GPIO can be assigned to both BMSPs, allowing for a single pin use case on all devices.

Please see the device-specific TRM for more details on the BOOTCTRL register.

2.1.2 Boot Definition Table (BOOTDEF)

When BMSPs are configured in the OTP, a custom boot mode table must also be defined by writing to the boot definition table registers (BOOTDEF) with boot option entries. Replacing the default boot mode selection table, the user-defined BOOTDEF table is indexed using the customized BMSPs in the OTP. For example, instead of parallel boot being tied to boot option 0 in the default configuration, the user can now set the first boot option to any available boot mode, and so on.



The BOOTDEF table is set up by configuring a 64-bit register (see Table 2-5), split into two 32-bit wide locations in the DCSM OTP, called Z1-OTP-BOOTDEF-LOW and Z1-OTP-BOOTDEF-HIGH (or Z2-OTP-BOOTDEF-LOW and Z2-OTP-BOOTDEF-HIGH depending on which zone is configured). These registers are then partitioned into 8-bit wide entries, defining each boot option to be used.

The range of customizable boot modes in the BOOTDEF table depends on how many BMSPs are being used. Recall, zero BMSPs allows for one table entry, one BMSP allows up to two table entries, two BMSPs allows up to four table entries, and three BMSPs allows up to eight table entries.

To configure the BOOTDEF table:

- 1. Select a boot option in the GPIO Assignments section of the data sheet or Technical Reference Manual
- 2. Set the associated BOOTDEF value of the boot option in the intended BOOTDEF-LOW or BOOTDEF-HIGH OTP memory location.

The BOOTDEF registers in the DCSM OTP can be programmed using the On-Chip Flash tool in CCS or Flash API (see Section 5.4 or Section 5.5 respectively for steps), or graphically with the DCSM tool in SysConfig [8].

Once programmed with valid BOOTDEFs, the boot definition table can be indexed with the BMSPs configured in the BOOTPIN-CONFIG register to select which boot option is executed in the boot ROM on reset.

BOOTDEF Name	Byte Position	Name	Description
		[3:0] BOOT_DEF0 Mode	Set the boot mode number from Table 2-6. Any unsupported boot mode causes the device to either go to wait boot (debugger connected) or boot to Flash (standalone).
BOOT_DEF0	7:0	[7:4] BOOT_DEF0 Options	Set alternate/additional boot options. This can include changing the GPIOs for a particular boot peripheral or specifying a different Flash entry point. Refer to <i>GPIO Assignments</i> for valid BOOTDEF values to set in the table.
BOOT_DEF1	15:8	BOOT_DEF1 Mode/ Options	
BOOT_DEF2	23:16	BOOT_DEF2 Mode/ Options	
BOOT_DEF3	31:24	BOOT_DEF3 Mode/ Options	
BOOT_DEF4	39:32	BOOT_DEF4 Mode/ Options	Refer to BOOT_DEF0 description.
BOOT_DEF5	47:40	BOOT_DEF5 Mode/ Options	
BOOT_DEF6	55:48	BOOT_DEF6 Mode/ Options	
BOOT_DEF7	63:56	BOOT_DEF7 Mode/ Options	

Table 2-5. BOOTDEF Bit Fields

Table 2-6. F280015x Device Boot Modes

Boot Number	Boot Mode
0	Parallel
1	SCI/Wait
2	CAN
3	Flash
4	Wait
5	RAM
6	SPI
7	I2C
8	CAN-FD
10	Secure Flash



Note

There are exceptions to the configurability of the boot selection table depending on the device family:

- 1. On F2833x devices, the boot table is not customizable and restricted to the factory default
- On F2802x, F2803x, F2806x, F2837xD, F2837xS, and F2807x, the boot table is semicustomizable, as the 4th entry in the default boot table (GET mode) can be programmed to one additional boot mode

This is different from devices that use the BOOTDEF register (see Table 2-3), allowing for up to eight boot modes to be selected. Please see the device-specific TRM for more details on BOOTCTRL.

2.1.3 Boot ROM OTP Configuration Registers

The boot ROM code involves numerous memory addresses and registers used during execution, supporting boot configurations from DCSM Zone 1 (Z1) and Zone 2 (Z2) registers. The user-configurable DCSM OTP locations used in the standalone boot flow can only be programmed once. The configuration of these registers is detailed in Section 2.1.

In the DCSM context, BOOTPIN-CONFIG maps to GPREG1, and BOOTDEF-LOW/BOOTDEF-HIGH map to GPREG3/GPREG4 respectively. Table 2-7 provides these locations.

Section 5.4 and Section 5.5 detail how to program the DCSM OTP with the On-Chip Flash Tool and Flash API respectively, following example use cases. SysConfig can also be used to program the DCSM OTP with an intuitive graphical user-interface (GUI) [8].

Note

The register addresses in Table 2-7 are for the **F280015x** family of devices. Device specific information can be found in the *Boot ROM* chapter's *Boot ROM Registers* table in the Technical Reference Manual (TRM).

Boot Flow	Register Name	Boot ROM Name	Register Address	User OTP Address
	Z1-GPREG1	Z1-OTP-BOOTPIN-CONFIG	0x0005 F008	0x0007 8008
Standalone	Z1-GPREG2	Z1-OTP-BOOT-GPREG2	0x0005 F00A	0x0007 800A
(Using Z1)	Z1-GPREG3	Z1-OTP-BOOTDEF-LOW	0x0005 F00C	0x0007 800C
	Z1-GPREG4	Z1-OTP-BOOTDEF-HIGH	0x0005 F00E	0x0007 800E
	Z2-GPREG1	Z2-OTP-BOOTPIN-CONFIG	0x0005 F088	0x0007 8208
Standalone	Z2-GPREG2	Z2-OTP-BOOT-GPREG2	0x0005 F08A	0x0007 820A
(Using Z2)	Z2-GPREG3	Z2-OTP-BOOTDEF-LOW	0x0005 F08C	0x0007 820C
	Z2-GPREG4	Z2-OTP-BOOTDEF-HIGH	0x0005 F08E	0x0007 820E

Table 2-7. F280015x Boot ROM Registers

2.1.4 CPU2 Boot Flow

Note

This section is based on the F28P65x family of devices. Device specific information can be found in the *Boot ROM* chapter of the device-specific Technical Reference Manual (TRM).

While CPU1 can be booted in different modes based on the boot pin configuration, CPU2 must be booted by CPU1 using the Inter-Processor Communication (IPC) module [9]. The CPU1 application configures boot mode for CPU2 through the IPCBOOTMODE register and controls when CPU2 is released from reset to boot.

Regardless of the reset source, CPU2 requires the IPC flag to be set by CPU1 on every reset to confirm the contents of the IPCBOOTMODE register are valid and continue the boot process. CPU2 acknowledges and clears the flag during boot up.

At a high-level, the CPU2 boot sequence is as follows:

- 1. CPU2 boots up and is either:
 - a. Held in reset
 - b. In Wait boot mode waiting for the IPC Flag
- 2. CPU1 application configures the CPU1TOCPU2IPCBOOTMODE register
- 3. CPU1 sets the CPU1TOCPU2IPCFLG0 to confirm the contents of CPU1TOCPU2IPCBOOTMODE are valid
- 4. If CPU2 is held in reset, the CPU1 application releases CPU2
- 5. CPU2 acknowledges and clears the IPC flag during boot up
- 6. CPU2 boot ROM runs the specified boot mode in CPU1TOCPU2IPCBOOTMODE

The IPCBOOTMODE register bit-field configurations and requirements for booting CPU2 are shown in the *IPCBOOTMODE Details* section of the F28P65x TRM. Table 2-8 details how this register can be configured for F28P65x devices.

Similar to the CPU1 BOOTPIN-CONFIG and BOOTDEF registers, CPU1TOCPU2IPCBOOTMODE is configured as follows:

- 1. The upper 8-bits contains the key indicating validity (0x5A)
- 2. The lower 8-bits sets CPU2's boot mode
- 3. Bits 16-19 specifies the number of words to be copied from CPU1TOCPU2MSGRAM1 to CPU2 M1RAM if "Copy from IPC Message RAM and Boot to M1RAM" boot mode is invoked

Bit	Name	Valid Values	Description
31:24	Кеу	0x5A	Key must be set for this register to be considered valid.
23:20	Reserved	-	Reserved
19:16	IPC Message RAM Copy Length	0x0 = 0 words (Boot mode not used) 0x1 = 100 words 0x2 = 200 words 0x9 = 900 words 0xA = 1000 words	Sets the data length (in words) for the "Copy from IPC Message RAM and Boot to M1RAM" boot mode. This is the number of words to be copied from CPU1TOCPU2MSGRAM1 to CPU2 M1RAM. If not using this boot mode, set value to 0x0.
15:8	Reserved	-	Reserved
7:0	CPU2 Boot Mode	0x00 = None/Wait Boot 0x01 = IPC Message RAM copy and boot to M1RAM 0x03 = Flash Boot Option 0 (Sector 0) 0x05 = Boot to M0RAM 0x0A = Secure Flash Boot Option 0 (Sector 0) 0x0B = Boot to User OTP 0x23 = Flash Boot Option 1 (Sector 4) 0x2A = Secure Flash Boot Option 1 (Sector 4) 0x43 = Flash Boot Option 2 (Sector 8) 0x4A = Secure Flash Boot Option 2 (Sector 8) 0x63 = Flash Boot Option 3 (Sector 13) 0x6A = Secure Flash Boot Option 3 (Sector 13)	Sets the boot mode for CPU2

Table 2-8. CPU1TOCPU2IPCBOOTMODE Register Details



2.2 Emulation Boot

If a JTAG debugger is connected to the device, then the device enters emulation boot mode. Like standalone boot mode, boot options other than those in default boot table can be accessed by programming the emulation boot registers located in RAM, lending users the capability to program boot configurations repeatedly. Thus, emulation boot allows users to test boot configurations and view the state of the boot ROM with a debugger (see Section 5.3).

Note

TI highly recommends using emulation mode to debug and verify correctness of the device boot configurations before attempting to program the OTP, as you can only program the OTP once.

- EMU-BOOTPIN-CONFIG is the emulation equivalent of Z1-OTP-BOOTPIN-CONFIG/Z2-OTP-BOOTPIN-CONFIG, and can be programmed to experiment with different boot modes without writing to OTP.
- EMU-BOOTDEF-LOW/EMU-BOOTDEF-HIGH are the emulation equivalents of Z1-OTP-BOOTDEF-LOW/Z1-OTP-BOOTDEF-HIGH.

The emulation locations located in RAM can be written to as many times as needed at the locations defined in Table 2-9 using the memory browser in CCS:

	U U
Boot ROM Name	Register Address
EMU-BOOTPIN-CONFIG	0x0000 0D00
EMU-GPREG2	0x0000 0D02
EMU-BOOTDEF-LOW	0x0000 0D04
EMU-BOOTDEF-HIGH	0x0000 0D06

Table 2-9. Emulation Boot Register Locations

At the start of the emulation boot mode, the EMU-BOOTPIN-CONFIG and EMU-BOOTDEF locations are checked, and the EMU-BOOTPIN-CONFIG-KEY is verified.

- If EMU-BOOTPIN-CONFIG-KEY is equal to 0xA5, then the CPU emulates standalone boot, essentially using the OTP definitions if already programmed.
- If EMU-BOOTPIN-CONFIG-KEY is equal to 0x5A, then the emulation BOOTDEF options are decoded using the specified BMSPs and the selected boot mode is executed.



3 Programming the Flash

Before attempting to program the device, understand how the non-volatile memory of C2000 devices works. The Flash memory on C2000 devices allow users to easily erase and re-program the device without losing data after loss of power. Erase operation set all bits in a given sector to 1, while programming operations selectively clear bits to 0.

During development, an application executable can be programmed into the Flash memory using Code Composer Studio[™] (CCS) [1]. When CCS identifies that the application code is mapped into the Flash memory, the On-Chip Flash Plugin is automatically invoked to load the executable to the Flash. By default, the plugin erases the Flash before programming, generates the ECC for the executable, and then programs and verifies the application into the Flash.

The Flash Plugin GUI can be used when connected to the target CPU core and found in CCS at:

• For CCS v12, Tools > On-Chip Flash



Figure 3-1. On-Chip Flash Tool Location in CCS v12

 For CCS v20, right-click the intended CPU in the *Debug* view and navigate to *Properties* > *Flash Settings* (*Flash Settings* is found under the *Categories* drop-down menu)

erties for: TMS320F28	28P550SJ9_LaunchPad.ccxmi	×
_		
re Texas Instru	ruments XDS110 USB Debug Probe_0/C28xx_CPU1	~
tegory Flash Settin	ings	~
Program/M	femory Load Options	
C28xx Disa	assembly Style Options	
Flash Settin	ngs	
CPUCLK (MHz)	150	43
Jownload Settings		
Beert terret befer		
Reset target before	e hash programming/operations	
Erase and Progra	am	
O Program Only		
O Verify Only		
J Perform Blank Che	eck before loading data to Flash memory	
Verify Flash after P	Program	
🛾 Auto ECC Generati	tion	
ank0 Program CMD	DWEPROTA 0x 0	
		Cancel Save and Close

Figure 3-2. On-Chip Flash Tool Location in CCS v20



TI also offers application flashing with UniFlash [2], a standalone JTAG based Flash programming tool with a smaller footprint than that of CCS due to less debug support. Nonetheless, UniFlash provides all the GUI operations that the CCS On-Chip Flash plugin does.

3.1 Flash API

Flash operations on all C2000 operations are performed by the CPU. Algorithms are loaded into RAM and executed by the CPU to perform any Flash operations. For instance, erasing or programming the Flash of a C2000 device with CCS entails loading Flash algorithms into the RAM over JTAG and having the CPU execute them.

All Flash operations are performed using the Flash Application Programming Interface (API); the device-specific library and reference guide is available in C2000Ware [3] at "libraries/flash_api". Applications that require erase or program Flash at runtime can link the Flash API library to perform Flash programming.

However, applications that call the Flash API are not advised to execute from the same Flash bank, since erasing or programming the Flash while also executing code introduces race conditions and undefined behavior. Hence, the Flash API needs to be executed from the RAM or another Flash bank (if additional banks exists for the same core).

This can be achieved by allocating the Flash API to the ".ti.ramfunc" section in the linker command file, designating a Flash load address and RAM run address, and then copying the functions to RAM in the main function before executing. Comprehensive details on Flash programming on C2000 devices can be referenced in [4] and [5].

Note

On multicore devices, one CPU cannot access another CPU's allocated Flash bank. For instance, the Flash bank of the CPU2can only be programmed by executing the Flash API from CPU2 RAM.

3.2 Flash Kernels

Given that the bootloaders in the boot ROM can only load code into RAM, flash kernels serve as a link between the ROM and flash by providing a mechanism for flash-based firmware upgrades in the field over various communication protocols (SCI, CAN, I2C, and so forth.). Programming the flash can be achieved by using the ROM bootloader to download the flash kernel to RAM, and then running the flash kernel in the RAM to download the application to the flash using the Flash API.

Before any application data is received, the flash kernel erases the flash of the device to prepare for programming. Once the host starts sending the application code, a buffer is used to hold the received contiguous blocks 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. After the entire application is received and programmed to flash, the flash kernel branches to the entry point of the application.

TI has developed flash kernels to load code from the RAM to the flash based on the boot ROM source code, and can be found in the *examples* folder of C2000Ware. A similar development flow can be applied by the user to implement a custom bootloader for a specific application. The boot ROM source code used by device boot flow, including the peripheral bootloaders, can be viewed in C2000Ware at:

 C2000Ware_x_xx_xx > libraries > boot_rom > DEVICE_FAMILY > REV# > rom_sources > DEVICE_FAMILY_ROM > bootROM > source

On devices with multiple flash banks, the flash kernel projects can be adapted to execute from the flash instead of the RAM to program another flash bank. This allows users to simply jump to the flash kernel located in the flash and avoid using the bootloaders in the boot ROM. For more details on how to implement this, refer to Section 5.2.



Note

On multicore devices (that is, F2838x, F2837xD, or F28P65x), CPU1 and CPU2 kernel projects can utilize the bootloader in CPU1 to download a modified bootloader (that is, flash kernel) for CPU2 to download the CPU2 application image.

Once CPU1 operations are completed, CPU1 can write the CPU2 kernel into shared message RAM (CPU1TOCPU2MSGRAM) with an instruction for CPU2 to branch to the CPU2 entry point in the CPU2 IPC message copy destination RAM (M1RAM) during the CPU2 boot sequence. After the branch instruction is written to M1RAM and the CPU2 boot sequence is complete, CPU2 starts execution from M1RAM and branches to CPU2 kernel entry point. The CPU1 kernel is waiting for the CPU2 kernel commands to finish before proceeding.

Please refer to the device-specific technical reference manual for more details on IPC protocol.

All in all, the general flow for using a flash kernel is follows:

- 1. Reset the device and use the intended boot mode
- 2. Transfer the flash kernel from the host to the device via the bootloader in the boot ROM
- 3. The flash kernel takes control after the ROM bootloader is complete
- 4. The kernel erases the old application code from flash memory on the controller
- 5. Kernel configures a connection with host and receive the new application code using the intended peripheral communications protocol
- 6. Kernel writes the newly received application code to flash memory and transfers control to the application
- 7. Newly received code from host executes



4 Bootloading Code to Flash

At the beginning of device boot up, the device decides if firmware is programmed into the Flash that needs to be executed or if code needs to be loaded in using a ROM loader. This is determined by examining the BMSPs, either defined by the user in the OTP, the emulation registers, or following the factory default boot configuration.

However, if a bootloader is selected that loads code from an external host, the application image needs to formatted and delivered in a specific procedure to be successfully programed into the RAM. Flash kernels can then be applied to bridge the gap between the Flash and RAM by linking the flash APIs to enable the kernels to erase and program flash.

This section demonstrates how to program and/or execute an application onto Flash for the most common boot modes (Flash, SCI, CAN, CAN-FD, and USB) by walking through the entire flow, from device boot configuration to flash kernel execution.

4.1 C2000 Hex Utility

The ROM loader requires data to be presented as a data stream and boot table. The structure is common to all ROM loaders and is described in detail in the *Bootloader Data Stream Structure* section of the device-specific TRM. Users can easily generate applications in this format 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 line in the project properties, as seen in Figure 4-1. The hex2000 utility can also be configured in a GUI by enabling the *C2000 Hex Utility* in the project properties and selecting the necessary conversion options.

In case of C2000 ROM bootloaders, the following line needs to be added to the post-build steps in the CCS Build (Project Properties > CCS Build > Steps > Post-build steps) for the firmware project loaded into the on-chip flash and the flash kernel project loaded into RAM:

SCI, CAN, CAN-FD Bootloading:

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

USB Bootloading:

"\${CG_TOOL_HEX}" "\${BuildArtifactFileName}" -boot -b -o "\${BuildArtifactFileBaseName}.dat"

Properties for led_ex1_blinky		- 🗆 X
type filter text	Build	⇔ ◄ ⇔ ▼ ∦
> Resource		
General	Configuration: CPU1_LAUNCHXL_FLASH [Active]	Manage Configurations
> SysConfig		
 C2000 Compiler Processor Options 	🗟 Builder 🗟 Validator 🗟 Variables 👼 Environment 🌫 Steps 🖉 Link Order 😁 Dependencies	
Optimization	Pre-build steps	
Include Options		A
Performance Advisor Predefined Symbols		
> Advanced Options	4	T I
> C2000 Linker	Description	
C2000 Hex Utility [Disable		
> Debug		
	Post-build steps	
	if \$[GENEKALE_DJAGHAM] == 1 [NubE_TOL] if \$[GENEKATE_DJAGHAM] == 1 \$[NubE_TOL] \$[COUWARE_ROOT]/driverlib/.meta/generate_dia "\$[CG_TOOL_HEX]" \$[BuildArtifactFileName]" -boot -sci8 -a -o "\$[BuildArtifactFileBaseName].txt"	agrams.js" "\$(C2000WARE_F
	Description:	
	Alternatively, create and define a new custom post-build step tool with the name:	
Show advanced settings	App	ply and Close Cancel

Figure 4-1. Adding Post-Build Steps to Invoke the Hex Utility



The hex utility supports creation of the boot table required for the SCI, SPI, I2C, CAN, and Parallel I/O loaders. The hex utility adds the required information to the file such as the key value, reserved bits, entry point, address, block start address, block length and terminating value. The contents of the boot table vary slightly depending on the boot mode and the options selected when running the hex conversion utility. The actual file format required by the host (ASCII, binary, hex, and so on) differs from one specific application to another and some additional conversion can be required.

See the TMS320C28x Assembly Language Tools User's Guide [11] for detailed description of the hex2000 options used to generate a boot table.

4.2 Common Boot Modes

This section gives a complete description of the entire flow of programming and executing an application onto Flash for the most common boot modes (Flash, SCI, CAN, CAN-FD, and USB).

Note

This section predominantly focuses on the boot execution paths when the emulator (JTAG) is connected and no BMSPs are used (zero-pin boot). The emulation procedure can easily be translated to standalone boot with different BMSP configurations by referring to Section 5.4 or Section 5.5, as the emulation boot configuration registers in RAM are formatted identically to the OTP registers.

4.2.1 Boot to Flash

Note

Although these steps were conducted on an F2800157 LaunchPad, the general flow can be easily applied to any C2000 devices that support custom BMSPs and boot definition tables (all devices provided in Table 2-3). Refer to the device-specific TRM for details for the device that is intended to boot load on.

If the user needs to boot to code already programmed in the on-chip flash, then users can either use the default BMSPs to boot to flash entry point address 0x0008 0000, or configure the BOOTPIN-CONFIG and BOOTDEF registers to boot to a specific flash address.

Note

Certain devices feature a secure flash boot option present on devices that can be used to perform an application boot from flash with an additional security layer of boot code authentication before the actual code execution. Please refer to *Secure BOOT on C2000 Device* [13] for the general procedure of enabling a secure boot. Users can verify the availability of the secure flash boot feature in the device-specific TRM.

The default BMSPs to configure boot to flash can be found in the TMS320F280015x Real-Time Microcontrollers data sheet. If the user sets both GPIO24 and GPIO32 to 1, then the boot ROM branches to flash entry address 0x0008 0000 without needing to configure the device registers.

However, if users need to boot to a different flash sector, then the BOOTCONFIG and BOOTDEF registers need to be configured for the specific boot option. Refer to the *GPIO Assignments* section in the data sheet to find which boot option to configure to reach the intended flash entry point. These steps describe how to emulate boot to flash entry point 0x0009 0000. For example, boot option 0x63.

Note

An application can be loaded into different flash locations by using linker command files. Refer to the *Compiler Tools User Manual* description on Linker Command Files [12] for details on how to load code into a particular flash address.

- 1. Open CCS to a workspace.
- 2. Select View > Target Configurations.



le Edit	Viev	Project Run Tool	s Scripts	Window	Help
) – 🛛 🖞	0	Resource Explorer			-
Project E:	ø	Resource Explorer Offli	ine		3
Boot_	٨	Getting Started			
🕮 boot_	V	CCS App Center			
🥌 Examı 📛 f2800	Θ	GUI Composer™			>
🕮 f2800	6	Project Explorer			
👺 flash_l	2	Problems	Alt	+Shift+Q, X	
G flash_		Console	Alt	+Shift+Q, C	
led_e>	0	Advice			
📛 secon	*	Debug			
	0	Memory Browser			
	1111	Registers			
	eg:	Expressions			
	(x)=	Variables	Alt	+Shift+Q, V	
		Disassembly			
	θ ₀	Breakpoints	Alt	+Shift+Q, B	
	Ξλ	Modules			
	<i>§</i>	Terminal			
	9 2 ,	Scripting Console			
	2	Target Configurations			
	85	Outline	Alt	+Shift+Q, O	
	5	Stack Usage			
	5	Memory Allocation			
	۲	Optimizer Assistant			
		Other	Alt	+Shift+Q, Q	

Figure 4-2. Opening the Target Configuration Menu in CCS

- Users can import a project for this device to CCS and use that to connect to the device, or copy the target configuration file (.ccxml) from C2000Ware (C2000Ware x_xx_xx > device_support > DEVICE_FAMILY > common > targetConfigs) to the User Defined target configurations.
 - a. Find the device target config and then manually launch by right clicking.

Target Configurations	×		🕱 🗯 🧇 🖃 🗖
type filter text			
 Projects User Defined TMS320F2800 TMS320F28P6 f28002x corm 	New Target Configuration Import Target Configuration		
f2807x.ccxml	X Delete Rename & Refresh	Delete F2 F5	
	Launch Selected Configuration		
	Set as Default Link File To Project	>	
	Properties	Alt+Enter	

Figure 4-3. Launching a Target Configuration in CCS

4. When CCS brings up the debug window, select the intended CPU and connect to the target.



Figure 4-4. Connecting to the Target Core in CCS

- 5. If a window pops up stating there is a break in the boot ROM with no debug information available, or outside of program code, then follow Section 5.3 to debug the boot ROM.
- 6. Once the symbols are loaded, open the memory browser by going to View > Memory Browser.





Figure 4-5. Navigating to the Memory Browser in CCS

- In the memory browser tab, navigate to address 0xD00. Recall that the 0xD00 location specifies the BMSPs with the validity key (EMU-BOOTPIN-CONFIG) and 0xD04-0xD05 specifies the boot definitions (EMU-BOOTDEF-LOW).
- The objective is to configure a zero-pin boot to flash address 0x0009 0000, so all BMSPs need to disabled and the EMU-BOOTDEF-LOW needs to be set to 0x63 in the lowest index. If the boot option is programmed to any other entry in EMU-BOOTDEF-LOW, then the intended boot mode is not selected.
 - a. Set 0xD00-0xD01 (EMU-BOOTPIN-CONFIG) to 0x5AFF FFFF.
 - b. Set 0xD04 (EMU-BOOTDEF-LOW) to 0x0063.

Note

Zero-pin boot means that the device automatically boots to the first entry defined the BOOTDEF table. This is achieved by disabling all BMSPs, thus allowing the device to only consider one boot option.

Memory Br	owser × 🖉 ▼ 🖉 ▼ 🛷 🕸 😫 🕄	- 0
Data ~	0xD00	S
Data:0xd00 -	0xD00 <memory 1="" rendering=""> ×</memory>	
16-Bit Hex -	ГI Style	
0×00000D00	FFFF 5AFF 0000 0000 0063 0000 0000 0000 0000 00	
0×00000D0C	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0×00000D18	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x00000D24	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x00000D30	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x00000D3C	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x00000D48	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x00000D54	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0×00000D60	0000 0000 0000 0000 0000 0000 0000 0000 0000	- 11
0×00000D6C	0000 0000 0000 0000 0000 0000 0000 0000 0000	- 18
0x00000D78	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x00000084	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0×00000D90	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x00000D9C	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0×00000DA8	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0×00000DB4	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x00000DC0	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x00000DCC	0000 0000 0000 0000 0000 0000 0000 0000 0000	
0x00000DD8	0000 0000 0000 0000 0000 0000 0000 0000 0000	

Figure 4-6. Emulating a Zero-pin Boot to Flash (0x0009 0000)

9. Reset the CPU and perform an external reset (XRSn). Then, click on Resume to begin the boot sequence.

10. Now, the device boots to flash address 0x0009 0000 on reset, as specified by boot option 0x63.



4.2.2 SCI Boot

Note

Although these steps were conducted on an F2800157 LaunchPad, the general flow can be easily applied to any C2000 devices that support custom BMSPs and boot definition tables (all devices provided in Table 2-3). Refer to the device-specific TRM for the device that is intended to boot load on.

The SCI flash kernel is based on the ROM bootloader, communicating with the host PC application provided in C2000Ware (C2000Ware_x_xx_xx_x > utilities > flash_programmers > serial_flash_programmer) and providing feedback to the host on the receiving of packets and completion of commands given.

Note

This section details CPU1 SCI boot loading. For a more detailed explanation on the SCI kernel commands and functionality, or steps on how to use CPU2 or Connectivity Manager (CM) SCI bootloader, refer to the *Serial Flash Programming of C2000 Microcontrollers application note* [12].

Flash kernel source and project files for CCS are provided in C2000Ware, in the examples directory of the corresponding device. 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 project name with a .txt extension.

1. Find the SCI flash kernel project for the intended device in C2000Ware and import into CCS.

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
F2838x	RAM	CPU1-CPU2 C2000Ware_x_x_xx > driverlib > f2838x>examples>c28x_dual>flash_kernel CPU1-CM C2000Ware_x_x_xx > driverlib > f2838x>examples>c28x_cm>flash_kernel
F28002x	RAM, Flash with LDFU	C2000Ware_x_xx_xx_xx > driverlib > f28002x > examples > flash, select flash_kernel_ex3_sci_flash_kernel
F28003x	RAM, Flash with LDFU	C2000Ware_x_xx_xx_xx > driverlib > f28003x > examples > flash, select flash_kernel_ex3_sci_flash_kernel
F280013x	RAM	C2000Ware_x_xx_xx_xx > driverlib > f280013x > examples > flash, select flash_kernel_ex3_sci_flash_kernel

Device	Build Configurations	Location
F280015x	RAM	C2000Ware_x_xx_xx_xx > driverlib > f280015x > examples > flash, select flash_kernel_ex3_sci_flash_kernel
F28P65x	RAM	C2000Ware_x_xx_xx_xx > driverlib > f28p65x > examples > c28x_dual > flash_kernel
F28P55x	RAM	C2000Ware_x_xx_xx_xx > driverlib > f28p55x > examples > flash, select f28p55x_flash_ex3_sci_flash_kernel

- 2. Make sure that the Active Build Target Configuration of the SCI flash kernel project is set to *RAM* because the kernel needs to be linked for execution in the RAM.
- 3. Build the kernel 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.
 - a. If txt output is not generated, then follow Section 4.1 to make sure that the correct post-build step to generate the hex file is defined.



Figure 4-7. Finding the Converted SCI Kernel Output File

- 4. Repeat the build process for the firmware code that is loaded into the flash by the kernel.
 - a. Confirm that the Active Build Target Configuration is set for the *Flash*.
 - b. Confirm that the correct post-build step to generate the txt file is defined.
 - c. Build the firmware project.

After building the kernel and firmware projects in CCS, set up the device hardware correctly to communicate with the host PC running the serial_flash_programmer executable provided in C2000Ware. The first task is to make sure the boot mode select pins are configured properly to boot the device to SCI boot mode. If the user needs to load code from an external host in the on-chip flash, then the default BMSPs or BOOTPIN-CONFIG and BOOTDEF registers can configure SCI boot.

The default BMSPs to enable SCI boot can be found in the TMS320F280015x Real-Time Microcontrollers data sheet. If the user sets GPIO24 to 0 and GPIO32 to 1, then the boot ROM jumps to the SCI bootloader with SCIRXDA to GPIO28 and SCITXDA to GPIO29 without needing to program the device registers.

However, if the user wants the flexibility of using SCI boot with different GPIO assignments, then the OTP or emulation BOOTCONFIG and BOOTDEF registers need to be configured for the specific boot option. Refer to the *GPIO Assignments* section in the data sheet to find which SCI boot option fits the GPIO requirements.

When using the serial_flash_programmer executable, the appropriate SCI bootloader GPIO pins to the Rx and Tx pins need to be connected to the host PC COM port. A transceiver is often needed to convert a Virtual



COM port from the PC to GPIO pins that can connect to the device. On some systems, like the controlCARD or LaunchPad, an FTDI chip is used to interface the GPIO pins used for SCI communication to a USB Virtual COM port.

For the F2800157 LaunchPad, the PC must connect to the USB on the LaunchPad and use the UART routing on the device to connect to the GPIO pins to the XDS110 COM Port. The schematics for the UART routing can be found in C2000Ware_x_xx_xx > boards > (LaunchPads or controlCARDs) > DEVICE_NAME > Rev# > documentation. For F2800157, SCIRXDA is internally routed to GPIO28 and SCITXDA to GPIO29 using the default SCI_SEL settings, so boot option 0x01 needs to be configured.



Figure 4-8. F2800157 LaunchPad UART Routing Schematic

However, users can also elect to use jumpers to externally route the XDS110 COM port to the GPIO BoosterPack[™] (BP) header. This is helpful if alternative SCI GPIO assignments are selected and a connection to the XDS COM port is required. The steps below demonstrate how to externally route the XDS110 COM port to GPIO28 or GPIO29 on the F2800157 LaunchPad.

1. Make sure the UART routing is set to connect to BoosterPack[™] (BP) for GPIO28 and GPIO29 (SCI_SeI1 = 1); not internally to the XDS110 COM port. This allows the SCI-A signals to output in the BP header pins.



Figure 4-9. Routing the SCI TX or RX (GPIO28 or GPIO29) Signals to the BoosterPack (BP) Pins



- Remove the jumpers for the TXD and RXD pins on the J101 header. Note that the XDS RXD pin (closer to the XDS circuit) is connected to the MCU TXD, and is the pin closer to the XDS circuit. Similarly, the XDS TXD is connected to the MCU RXD.
- 3. Attach a jumper wire from XDS TXD to GPIO28 (SCI-A RX) and a jumper wire from the XDS RXD to GPIO 29 (SCI-A TXD) according to the *XDS110 Target Interface* section in the board schematic.



Figure 4-10. Jumping the XDS TX, RX to the SCI TX, RX GPIOs

Now, the device needs to be set up to emulate a zero-pin SCI boot with boot option 0x01, SCIRXA = GPIO28 and SCITXA = GPIO29.

- 1. Open CCS to a workspace.
- 2. Select View > Target Configurations.



Figure 4-11. Opening the Target Configuration Menu in CCS

- Users can import a project for this device to CCS and use that to connect to the device, or copy the target configuration file (.ccxml) from C2000Ware (C2000Ware_x_xx_xx_x > device_support > DEVICE_FAMILY > common > targetConfigs) to the User Defined target configurations.
 - a. Find the device target config and then manually launch by right-clicking.





Figure 4-12. Launching a Target Configuration in CCS

4. When CCS brings up the debug window, select the intended CPU and connect to the target.



Figure 4-13. Connecting to the Target Core in CCS

- 5. If a window pops up stating there is a break in the boot ROM with no debug information available, or outside of program code, then follow Section 5.3 to debug the boot ROM.
- 6. Once the symbols are loaded, open the memory browser by going to View > Memory Browser.



Figure 4-14. Navigating to the Memory Browser in CCS

- 7. In the memory browser tab, navigate to address 0xD00. Recall that the 0xD00 location specifies the BMSPs with the validity key (EMU-BOOTPIN-CONFIG) and 0xD04-0xD05 specifies the boot definitions (EMU-BOOTDEF-LOW).
- The objective is to configure a zero-pin SCI boot with SCIRXDA to GPIO28 and SCITXDA to GPIO29, so all BMSPs need to disabled and the EMU-BOOTDEF-LOW needs to be set to 0x01 in the lowest index. If the boot option is programmed to any other entry in EMU-BOOTDEF-LOW, then the intended boot mode is not selected.
 - a. Set 0xD00-0xD01 (EMU-BOOTPIN-CONFIG) to 0x5AFF FFFF.
 - b. Set 0xD04 (EMU-BOOTDEF-LOW) to 0x0001.

Note

Zero-pin boot means that the device automatically boots to the first entry defined the BOOTDEF table. This is achieved by disabling all BMSPs, thus allowing the device to only consider one boot option.

Memory Browser ×	🤹 🕶 🧑 💌 🍪 💆 📑 🖻 🕴	- 0
Data v 0xD00		1
Data:0xd00 - 0xD00 <n< td=""><td>Nemory Rendering 1> ×</td><td></td></n<>	Nemory Rendering 1> ×	
16-Bit Hex - TI Style	\checkmark	
0×00000000 FFFF 5AFF	0000 0000 0001 0000 0ACC 0008 0ACC 0008 0ACC 0008	
0×00000D0C 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000018 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000D24 0C6C 0008	3 0C62 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000D30 0ACC 0008	3 ØACC 0008 ØACC 0008 ØACC 0008 ØACC 0008 ØACC 0008	
0x0000D3C 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000D48 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000D54 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000D60 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000D6C 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	- 8
0x00000D78 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	- 8
0x00000084 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000090 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x0000009C 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000DA8 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000DB4 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000000 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	
0x00000DCC 0ACC 0008	3 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008 0ACC 0008	

Figure 4-15. Emulating a Zero-pin SCI Boot with SCIRXDA to GPIO28 and SCITXDA to GPIO29

- 9. Reset the CPU and perform an external reset (XRSn). Then, click on Resume to begin the boot sequence.
- 10. If there is a break in the boot ROM with no debug information available, or outside of program code, then follow the Section 5.3 to load the boot ROM symbols. Afterwards, confirm that the device is in SCI autobaud lock.

Now, the SCI bootloader (with GPIO assignments as specified by the boot option 0x01) in the ROM begins executing and waits to autobaud lock with the host (for the *A* character to be received to determine the baud rate at which the communications occurs). At this point, the device is ready to receive code from the host.

The command line PC utility is a programming design that can easily be incorporated into scripting environments for applications like production line programming. This was written using Microsoft Visual Studio[®] in C++. The project and 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, make sure that the target board has been reset and is currently in the SCI boot mode as configured above, and connected to the PC COM port. The command line usage of the tool for a single core SCI boot is described below, where -d, -k, -a, -p are mandatory parameters. If the baud rate is omitted, then the baud rate is set to 9600 by default. More details on the parameters of the utility is detailed in [12].

serial_flash_programmer.exe -d DEVICE -k KERNEL_FILE -a APPLICATION_FILE -p COM# -b BAUDRATE -v

- 1. Find the XDS110 UART COM port by navigating to the Device Manager > Ports (COM & LP).
 - Ports (COM & LPT)
 Intel(R) Active Management Technology SOL (COM3)
 XDS110 Class Application/User UART (COM6)
 XDS110 Class Auxiliary Data Port (COM8)

Figure 4-16. Finding the XDS COM Port in Device Manager

 Navigate to the folder containing the compiled serial_flash_programmer executable (C2000Ware_x_xx_xx_xx > utilities > flash_programmers > serial_flash_programmer). Run the executable serial_flash_programmer.exe with the following command.

serial_flash_programmer.exe -d DEVICE_NAME -k <path_to_kernel_hex> -a <path_to_application_hex>
-p COM# -v



Note

Both the flash kernels and flash application must be in the SCI8 boot format as discussed in Section 4.1.

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

1. The serial_flash_programmer prints the options to the screen to choose from that is sent to the device kernel. Select *1-DFU CPU1* to flash the application to CPU1. In this case, the original command already specified the application file, so no additional information is required at this point.

Z Windows PowerShell X + v		×
5==5		
8f==8f		
d0==d0		
23==23		
c4==c4		
6==6		
6==6		
0==0		
1==1		
9a==9a		
6==6		
0==0		
6==6		
0==0		
0==0		
0==0		
Bit rate /s of transfer was: 1240.378418		
Kernel Loaded! Booting Kernel		
Done waiting for kernel boot		
Attempting autobaud to send function message		
What operation do you want to perform?		
3-Verity CPUI Zene 1		
0 bonc		

Figure 4-17. Commanding the Serial Flash Programmer to Download the Application

 After the execution of the command, the application needs to be executed. To run the application, select 6-Run CPU1 and specify branch address 0x0008 0000 (flash entry point of application). The application that was successfully SCI boot loaded onto the device now executes the application loaded in flash.



Figure 4-18. Running the Application After Loading the Application into Flash

4.2.3 CAN Boot

Note

Although these steps were conducted on an F280039C LaunchPad, the general flow can be easily applied to any C2000 devices that support custom BMSPs and boot definition tables (all devices provided in Table 2-3). Refer to the device-specific TRM for the device details that is intended to boot load on.

The DCAN flash kernel is based on the ROM bootloader, communicating with the host PC application provided in C2000Ware (C2000Ware_x_xx_xx_ > utilities > flash_programmers > dcan_flash_programmer) and providing feedback to the host on the receiving of packets and completion of commands given. The source and executable for the host application are found in the dcan_flash_programmer folder. For a more detailed explanation on kernel functionality, refer to the *CAN Flash Programming of C2000 Microcontrollers application note* [16].

The following devices are supported by the dcan_flash_programmer in C2000Ware:

- 1. F28003x
- 2. F280015x
- 3. F28P65x

Note

In the F280015x LaunchPad, the built-in CAN transceiver and connecter does not map to any possible CAN GPIO assignments and requires soldering to short the correct GPIOs to the transceiver [32].

Note

The term DCAN used in this report is defined as DCAN flash kernels, DCAN flash programmer, and refer to the Controller Area Network communication interface (CAN) [13] Version D designed by Bosch. The DCAN flash programmer described in this document refers to the CAN module.

DCAN Flash Kernel is based off DCAN 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 F28P65x and F280015x DCAN flash kernels erase the flash of the device readying for programming. The F28P65x and F280015x DCAN flash kernel projects allow the user to specify which flash banks and flash sectors to erase before the application is programmed. This is discussed in more detail in the CAN Flash Programming of C2000 Microcontrollers application note [16].

After the appropriate locations in flash memory are erased, the application load begins. A buffer is used to hold the received contiguous blocks 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.

Before writing to a sector for the first time, the F28003x DCAN flash kernel checks to see if the sector has been erased. If the sector has not been erased, then the F28003x Flash kernel has the Flash API execute an erase operation. After this, a buffer is filled up with content to be written into Flash, and a program command is sent from the Flash API. Once the write has occurred, the Flash kernel has the Flash API verify that the segment was written into Flash at the correct address. Once the kernel has copied everything to Flash, the project jumps to the entry address of the image.

After the DCAN module is initialized in the flash kernel, this module waits for the host to send in the firmware image. The flash kernel receives 8 bytes at a time from the host and places the contents into an intermediate RAM buffer. This buffer is then written into Flash in 128-bit or 512-bit increments.

- F28P65x and F280015x DCAN flash kernel projects support 512-bit programming
 - If desired, there is also a 128-bit programming project available for the F28P65x devices
 - The F280015x flash API supports 128-bit programming, but the flash kernel was implemented using 512-bit programming
- F28003x DCAN flash kernel project supports 128-bit programming



All of the sections of the firmware image stored in flash are aligned according to the number of bits being programmed at once.

- If programming 128-bits at once (F28003x), then the flash sections of the application are aligned to a 128-bit boundary. In the linker command file for the firmware image, all initialized sections need to be mapped to Flash sectors, and after each mapping, an ALIGN(8) directive needs to be added to verify the 128-bit alignment.
- If programming 512-bits at once (F280015x and F28P65x), then the flash sections of the application are aligned to a 512-bit boundary. In the linker command file for the firmware image, all initialized sections need to be mapped to Flash sectors, and after each mapping, an ALIGN(32) directive needs to be added to verify the 512-bit alignment.

Note

The ALIGN(x) directive inserts padding bytes until the programmed location becomes aligned on a x word boundary. For C2000, the word size is 16-bits, so 16-bit programming requires ALIGN(1), 32-bit programming requires ALIGN(2), and so on.

Flash kernel source and project files for CCS are provided in C2000Ware, in the examples directory of the corresponding device. 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 DCAN ROM bootloader and is saved as the project name with a .txt extension.

The correct GPIO assignments needed for CAN routing in the LaunchPad can be confirmed by inspecting the schematics for the *CAN Transceiver and Connector* in C2000Ware_x_xx_xx > boards > (LaunchPads or controlCARDs) > DEVICE_NAME > Rev# > documentation. On LAUNCHXL-F280039C, the transceiver RXD is internally routed to GPIO5 and TXD to GPIO4, hence boot option 0x02 needs to be configured.



Figure 4-19. F280039C LaunchPad CAN Transceiver and Connector Schematic

- 1. Find the DCAN flash kernel project for the intended device in C2000Ware and import into CCS. For example, the DCAN flash kernel for F28003x is found at C2000Ware_x_x_xx > driverlib > f28003x > examples > flash.
- 2. Make sure that the Active Build Target Configuration of the DCAN flash kernel project is set to *RAM*, since the kernel needs to be linked for execution in the RAM.
- If using a LaunchPad, then apply the predefined symbol _LAUNCHXL_F280039C in Project Properties > CCS Build > C2000 Compiler > Predefined Symbols to use correct GPIO assignments for the LaunchPad in device.h.

Properties for flash_kernel_ex4	t_can_flash_kernel	– 🗆 X
type filter text	Predefined Symbols	↓ ↓ ↓ ₹ 8
> Resource		
CCS General	Configuration: CPU1 RAM [Active]	Manage Configurations
✓ CCS Build		
C2000 Complier		
Ontimization	Pre-define NAME (define, -D)	🗟 🛍 🗟 🖗 😓
Include Options	\$(COM TI C2000WARE SYMBOLS) =	
Performance Advisor		
Predefined Symbols		
> Advanced Options		
> C2000 Linker		
> C2000 Hex Utility		
> C/C++ Build		
> C/C++ General	Undefine NAME (undefine, -U)	월 최 월 취 분]
> Debug		
Project Natures		
Project References		
Run/Debug Settings		
Itide advanced settings	A	pply and Close Cancel

Figure 4-20. Adding a Predefined Symbol for the Correct LaunchPad CAN GPIO Assignments

- 4. At power-up, the device boot ROM is clocked from an on-chip 10MHz oscillator (INTOSC2). This value needs to be set as the primary internal clock source in the flash kernel and is the default clock at reset. In *device.h*, uncomment and use *#define USE_PLL_SRC_INTOSC* on line 295. Comment out *#define USE_PLL_SRC_XTAL*.
- 5. In *DCAN_boot.c*, confirm that the local copy of *bootloader_can_timing.h* is included on line 64 instead of the auto-generated header file found in the *include* folder.



Figure 4-21. Including the Local CAN Timing Header File

- 6. In *bootloader_can_timing.h*, confirm lines 51-53 defines the following CAN timing settings:
 - a. 1Mbps bitrate
 - b. 20Mhz CAN clock
 - c. 20ms bit time

```
48 //
49 // CAN bit timing settings for running CAN at 100kbps with a 20MHz crystal
50 //
51 #define CAN_CALC_BITRATE 1000000U
52 #define CAN_CALC_CANCLK 2000000U
53 #define CAN_CALC_BITTIME 20U
```

Figure 4-22. Confirming the CAN Timing Settings

7. Build the kernel 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 DCAN ROM bootloader and is saved as the example name with a txt extension.



a. If txt output is not generated, then follow the steps in Section 4.1 to make sure that the correct post-build step to generate the hex file is defined.



Figure 4-23. Generating the DCAN kernel txt output

- 8. Open the flash kernel txt file in a text editor and change bytes 3-4 to be *C0* 7*A*. These bytes contain metadata to configure the bitrate of the CAN bus.
- 9. Repeat the build process for the firmware code that is loaded into the flash by the kernel.
 - a. Confirm that the Active Build Target Configuration is set for the Flash.
 - b. Confirm that the correct post-build step to generate the hex file is defined.
 - c. Build the firmware project.

After building the kernel and firmware projects in CCS, set up the device hardware correctly to be able to communicate with the host PC running the dcan_flash_programmer provided in C2000Ware. The first task to do is make sure the boot mode select pins are configured properly to boot the device to CAN boot mode. If the user needs to load code from an external host in the on-chip flash, then users can either use the default BMSPs, if supported, to configure CAN boot or configure the BOOTPIN-CONFIG and BOOTDEF registers.

The default BMSPs to enable CAN boot can be found in the TMS320F28003x Real-Time Microcontrollers data sheet. If the user sets GPIO24 to 1 and GPIO32 to 0, then the boot ROM jumps to the CAN bootloader with CANRXA to GPIO5 and CANTXA to GPIO4 without needing to program the device registers.

However, if the user wants the flexibility of using CAN boot with different GPIO assignments, then the OTP or emulation BOOTCONFIG and BOOTDEF registers need to be configured for the specific boot option. Refer to the *GPIO Assignments* in the TMS320F28003x Real-Time Microcontrollers data sheet to find which CAN boot option fits the GPIO requirements.

The hardware components needed to run the examples are a C2000 device connected to a CAN transceiver and a PEAK PCAN-USB Pro FD analyzer.

- 1. The LaunchPad devices contain an onboard CAN transceiver. Confirm that the PEAK PCAN-USB Pro FD Analyzer is connected to the LaunchPad through the ground, CAN-Lo and CAN-Hi connections.
- 2. Confirm that the onboard CAN Routing switch needs to be set low (to XCVR) for the transceiver to communicate using the GPIOs.





Figure 4-24. Setting the CAN Routing Switch

Note

For controlCards, a custom designed CAN transceiver board and the HSEC-180-pin controlCard Docking Station needs to be used. The custom-designed transceiver board is connected to the controlCard using four connections: ground, 3.3V, CANTX and CANRX.

Now, the device needs to be set up to emulate a CAN boot with boot option 0x02, CANRXA = GPIO5 and CANTXA = GPIO4. There is only ≈ 10 second timeout window to send the first CAN frame to device.

- 1. Open CCS to a workspace.
- 2. Select View > Target Configurations.



Figure 4-25. Opening the Target Configuration Menu in CCS

- Users can import a project for this device to CCS and use that to connect to the device, or copy the target configuration file (.ccxml) from C2000Ware (C2000Ware_x_xx_xx_x > device_support > DEVICE_FAMILY > common > targetConfigs) to the User Defined target configurations.
 - a. Find the device target config and manually launch by right clicking.





Figure 4-26. Launching a Target Configuration in CCS

4. When CCS brings up the debug window, select the intended CPU and connect to the target.



Figure 4-27. Connecting to the Target Core in CCS

- 5. If a window pops up stating there is a break in the boot ROM with no debug information available, or outside of program code, then follow the steps in Section 5.3 to debug the boot ROM.
- 6. Once the symbols are loaded, open the memory browser by going to View > Memory Browser.



Figure 4-28. Navigating to the Memory Browser in CCS

- 7. In the memory browser tab, navigate to address 0xD00. Recall that the 0xD00 location specifies the BMSPs with the validity key (EMU-BOOTPIN-CONFIG) and 0xD04-0xD05 specifies the boot definitions (EMU-BOOTDEF-LOW).
- 8. The objective is to configure a zero-pin CAN boot with CANRXA = GPIO5 and CANTXA = GPIO4, so all BMSPs need to disabled and the EMU-BOOTDEF-LOW needs to be set to 0x02 in the lowest index. If the boot option is programmed in any other entry in EMU-BOOTDEF-LOW, then the intended boot mode is not selected.
 - a. Set 0xD00-0xD01 (EMU-BOOTPIN-CONFIG) to 0x5AFF FFFF.
 - b. Set 0xD04 (EMU-BOOTDEF-LOW) to 0x0002.

Note

Zero-pin boot means that the device automatically boots to the first entry defined the BOOTDEF **table**. This is achieved by disabling all BMSPs, thus allowing the device to only consider one boot option.



Bremory Br	owser × 🔹 🔹 🖛 🖛 🗢 😵 🏟 📑 🖆 🕴 🗖 🗖
Data ~	0xD00
Data:0xd00 -	0xD00 <memory 1="" rendering=""> ×</memory>
16-Bit Hex -	TI Style 🗸
0×00000D00	FFFF 5AFF C238 DB5A 0002 9B31 2D41 0008 2D41 0008
0x00000D0A	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008
0x00000D14	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008
0x0000001E	2D41 0008 2D41 0008 2D41 0008 2EE4 0008 2EDA 0008
0x00000D28	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008
0x00000D32	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008
0x00000D3C	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008
0x00000D46	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008
0x00000D50	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008
0x00000D5A	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008
0x00000D64	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008
0x0000006E	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008
0x00000D78	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008

Figure 4-29. Emulating a Zero-pin CAN Boot with CANRXA=GPIO5 and CANTXA=GPIO4

- 9. Reset the CPU and perform an external reset (XRSn). Then, click on *Resume* to begin the boot sequence.
- 10. If there is a break in the boot ROM with no debug information available, or outside of program code, then follow the Section 5.3 to load the boot ROM symbols. Afterwards, confirm that the device is in CAN boot.

Now, the CAN bootloader (with GPIO assignments as specified by the boot option 0x02) in the ROM begins executing and waits for a CAN frame to be transmitted from the host. At this point, the device is ready to receive code from the host.

Note

The CAN bootloader has a 10 second window for accepting CAN frames. If offered in the device boot options, then the SEND_TEST modes remove the timeout. These options use the same GPIO pins for communication as equivalent boot options. However, the GPIO pins transmit a frame before beginning the boot-loading process to help ascertain proper functionality of the module. In the case of F280039x devices, boot options 0x82 can be used to avoid the timeout while using the same GPIO assignments as boot option 0x02.

The command line PC utility is a programming solution that can easily be incorporated into scripting environments for applications like production line programming. This was written using Microsoft Visual Studio in C++. The project and source can be found in C2000Ware (C2000Ware_x_xx_xx > utilities > flash_programmers > dcan_flash_programmer).

The host is responsible for sending the DCAN kernel image and flash (firmware) image to the MCU. The PEAK PCAN-USB Pro FD CAN bus analyzer is used as the host. The flash programmer project is built and run on Visual Studio 2019. The host programmer uses the PCAN_Basic API from PEAK [17]. The PCAN_Basic API can be used to send and receive CAN frames on the CAN analyzer.

Note

The PEAK PCAN-USB Pro FD CAN bus analyzer is backwards compatible to receive both CAN frames as well as CAN-FD frames.

On the F28003x device, the clock to the CAN module is switched to the external clock source by the Boot ROM. The external clock is 20MHz in the LaunchPad and the controlCard. The Boot ROM configures the nominal bit rate to be 100Kbps. The host CAN programmer configures the PEAK CAN analyzer to have the same clock and nominal bit rate value.

The host initializes the analyzer for CAN usage, sends the kernel over in 2-byte increments, and then sends over the image in 8-byte increments with a delay of 10ms between each frame to give the Flash API time to program the data received into Flash. Once the firmware image has been written, the host CAN programmer exits.

To use this tool to program the C2000 device, make sure that the target board has been reset and is currently in the CAN boot mode as configured above, and connected to the PEAK PCAN-USB Pro FD CAN bus analyzer. The command line usage of the tool for a single core CAN boot is described below, where -d, -k, and -a are



mandatory parameters. Verbose output can be enabled with -v. More details on the parameters of the utility are detailed in [4].

dcan_flash_programmer.exe -d DEVICE -k KERNEL_FILE -a APPLICATION_FILE -v

- Navigate to the folder containing the compiled dcan_flash_programmer executable (C2000Ware_x_xx_xx_xx > utilities > flash_programmers > dcan_flash_programmer).
- 2. Open a terminal and run the executable dcan_flash_programmer with the following command:

dcan_flash_programmer.exe -d DEVICE_NAME -k <path_to_kernel_hex> -a <path_to_application_hex> -v

This first loads the DCAN flash kernel into RAM of the device using the bootloader. The bytes transferred over the CAN bus can be seen in the terminal. Then, the kernel executes and loads and programs flash with the file specified by the '-a' command line argument as seen in Figure 4-30 and Figure 4-31. The kernel branches to the application and begins executing if successfully loaded into the flash.



Figure 4-30. DCAN Flash Programmer Kernel Loaded







4.2.4 CAN-FD Boot

Note

Although these steps were conducted on an F280039C LaunchPad, the general flow can be easily applied to any C2000 devices that supports custom BMSPs and boot definition tables (all devices are provided in Table 2-3). Refer to the device-specifc TRM for the device that is intended to boot load on.

ROM bootloaders can only load code into RAM, which is why ROM bootloaders are used to load in flash kernels to allow code to be stored in the flash, as described in Section 3.2. The CAN flash kernel is based on the ROM bootloader, communicating with the host PC application provided in C2000Ware (C2000Ware_x_xx_xx_xx > utilities > flash_programmers > can_flash_programmer) and providing feedback to the host on the receiving of packets and completion of commands given. The source and executable for the host application are found in the can_flash_programmer folder. For more information on kernel functionality, refer to the CAN Flash Programming of C2000 Microcontrollers application note [16].

The following devices are supported by the can_flash_programmer in C2000Ware:

- 1. F28003x
- 2. F28P55x
- 3. F28P65x

Note

The term MCAN, MCAN flash kernels, CAN flash programmer, and so forth refer to the Modular Controller Area Network (MCAN) in this document. MCAN is an interchangeable term with Controller Area Network Flexible Data-Rate (CAN-FD) [15]. The CAN flash programmer described in this document refers to the MCAN module.

The flash kernel project is modeled after the MCAN ROM bootloader. This goes straight into the MCAN_Boot function which has been modified to write to Flash. The MCAN module initialization for the flash kernel is the same as the bootloader. The clock source for the MCAN module, the nominal and data bit rates, GPIO pins, and so forth, are set by the kernel on initialization according to the boot mode.

Before any application data is received, the F28P55x kernel erases the flash of the device, readying for programming. Additionally, the F28P55x MCAN flash kernel project allows the user to specify which flash banks and flash sectors to erase before the application is programmed. This is described in more detail in the *Custom Flash Bank and Sector Erase* section of the CAN Flash Programming of C2000 Microcontrollers application note [16].

The F28003x kernel checks each flash sector if the flash sector been erased before programming the application. If the flash sector is has not been previously erased, then the sector is erased and the application data is written.

The F28P65x kernel erases the flash at the beginning of the application download process. Erasing flash can take a few seconds. Note, that while the application load presents as failed, the flash is being erased.

After the appropriate locations in flash memory are erased, the application load begins. The flash kernel receives 64 bytes at a time from the host and places the contents into an intermediate RAM buffer. This buffer is then written into Flash in 128-bit or 512-bit increments.

- F28003x and F28P65x MCAN flash kernels write 128-bits at a time
- F28P55x MCAN flash kernel writes 512-bits at a time

After the RAM buffer is filled up with content to be written into Flash, a program command is sent from the Flash API. Once the write has occurred, the Flash kernel has the Flash API verify that the segment was written into Flash at the correct address. Once the kernel has copied everything to Flash, the project jumps to the entry address of the image.

All of the sections of the firmware image stored in flash must be aligned according to the number of bits being programmed at once.



- If programming 128-bits at once (F28003x and F28P65x), then the flash sections of the application are aligned to a 128-bit boundary. In the linker command file for the firmware image, all initialized sections need to be mapped to Flash sectors. After each mapping, an ALIGN(8) directive needs to be added to verify the 128-bit alignment.
- If programming 512-bits at once (F28P55x), then the flash sections of the application are aligned to a 512-bit boundary. In the linker command file for the firmware image, all initialized sections need to be mapped to Flash sectors. After each mapping, an ALIGN(32) directive needs to be added to verify the 512-bit alignment.

Note

The ALIGN(x) directive inserts padding bytes until the programmed location becomes aligned on a x word boundary. For C2000, the word size is 16-bits, so 16-bit programming requires ALIGN(1), 32-bit programming requires ALIGN(2), and so on.

The protocol used to transfer the application data follows the MCAN ROM loader protocol. With the original MCAN ROM loader protocol, nominal bitrate used is 1Mbps and transmits 64 bytes per frame from the host to the target device for nominal bit timing. The data bitrate used by the protocol is 2Mbps for data bit timing.

Flash kernel source and project files for CCS are provided in C2000Ware in the corresponding examples directory of the device. 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 MCAN ROM bootloader and is saved as the project name with a txt extension.

The correct GPIO assignments needed for CAN routing in the LaunchPad can be confirmed by inspecting the schematics for the *CAN Transceiver and Connector* in C2000Ware_x_xx_xx > boards > (LaunchPads or controlCARDs) > DEVICE_NAME > Rev# > documentation. On LAUNCHXL-F280039C, the transceiver RXD is internally routed to GPIO5 and TXD to GPIO4, hence BOOTDEF 0x02 needs to be configured.



Figure 4-32. F280039C LaunchPad CAN Transceiver and Connector Schematic

- Find the MCAN flash kernel project for the intended device in C2000Ware and import into CCS. For example, the MCAN flash kernel for F28003x is found at C2000Ware_x_x_xx > driverlib > f28003x > examples > flash
- 2. Make sure that the Active Build Target Configuration of the MCAN flash kernel project is set to *RAM* because the kernel needs to be linked for execution in the RAM.
- 3. Add the predefined symbol *_LAUNCHXL_F280039C* in Project Properties > CCS Build > C2000 Compiler > Predefined Symbols to use correct GPIO assignments for the LaunchPad in device.h.

Properties for flash_kernel_ex4	I_can_flash_kernel	– 🗆 X
type filter text	Predefined Symbols	⇔ ▼ ⇔ ≋
Resource CCS General CCS Build C2000 Compiler Processor Options Optimization Include Options Performance Advisor Predefined Symbols Advanced Options C2000 Linker C2000 Linker C2000 Hex Utility Builders	Configuration: CPU1_RAM [Active] Pre-define NAME (define, -D) \$(COM TI C2000WARE SYMBOLS) LAUNCHXL F280039C CPU1	Manage Configurations মি জ ভা থা থা
 C/C++ Build C/C++ General Debug Project Natures Project References Run/Debug Settings 	Undefine NAME (undefine, -U)	ର ଛି ହି। ହି।
⑦ Hide advanced settings		Apply and Close Cancel

Figure 4-33. Adding a Predefined Symbol for the Correct LaunchPad CAN GPIO Assignments

- 4. At power-up, the device boot ROM is clocked from an on-chip 10MHz oscillator (INTOSC2). This value needs to be set as the primary internal clock source and is the default clock at reset. In device.h, uncomment and use #define USE_PLL_SRC_INTOSC on line 295. Comment out #define USE_PLL_SRC_XTAL.
- 5. Build the kernel 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 MCAN ROM bootloader and is saved as the example name with a txt extension.
 - a. If txt output is not generated, then follow the steps in Section 4.1 to make sure that the correct post-build step to generate the hex file is defined.



Figure 4-34. Generating the CAN Kernel txt Output

- 6. Repeat the build process for the application code that is loaded into the flash by the kernel.
 - a. Confirm that the Active Build Target Configuration is set for *Flash*.
 - b. Confirm that the correct post-build step to generate the hex file is defined.
 - c. Build the firmware project.

After building the kernel and firmware projects in CCS, set up the device hardware correctly to be able to communicate with the host PC running the can_flash_programmer provided in C2000Ware. The first task to do is make sure the boot mode select pins are configured properly to boot the device to CAN-FD boot mode.



If the user needs to load code from an external host in the on-chip flash, then users need to configure the BOOTPIN-CONFIG and BOOTDEF registers since this is not available as a default boot option. Refer to the GPIO Assignments in the TMS320F28003x Real-Time Microcontrollers data sheet to find which CAN-FD boot option fits the GPIO requirements.

The hardware components needed to run the examples are a C2000 device connected to a CAN transceiver and a PEAK PCAN-USB Pro FD analyzer.

- 1. The LaunchPad devices contain an onboard CAN transceiver. Confirm that the PEAK PCAN-USB Pro FD Analyzer is connected to the LaunchPad through the ground, CAN-Lo and CAN-Hi connections.
- 2. Confirm that the onboard CAN Routing switch needs to be set low (to XCVR) for the transceiver to communicate using the GPIOs.



Figure 4-35. Setting the CAN Routing Switch

Note

For controlCards, a custom-designed CAN transceiver board needs to be used, and the HSEC-180-pin ControlCard Docking Station. The custom-designed transceiver board is connected to the ControlCard using four connections: ground, 3.3V, CANTX and CANRX.

The device needs to be set up to emulate a zero-pin CAN boot with boot option 0x02, CANRXA = GPIO5 and CANTXA = GPIO4. Now, the device needs to be set up to emulate a zero-pin CAN-FD boot with boot option 0x08, CANRXA = GPIO5 and CANTXA = GPIO4. Note that there is only approximately 10 seconds to send the first CAN frame to device.

- 1. Open CCS to a workspace.
- 2. Select View > Target Configurations.



w	orkspac	ce_v1a	- seconda	ry_boot	tloader/	CPU1_FL/	ASH/primar	y_appli
ile	Edit	View	Project	Run	Tools	Scripts	Window	Help
3 -		Ø	Resource	xplore	r			
Pro	oiect E:	0	Resource I	Explore	r Offline			3
19	boot_	3	Getting St	arted				
6	boot_	v	CCS App (Center				
10 11	Examı f2800	Q	GUI Comp	oser™				>
6	f2800	6	Project Ex	olorer				
1	flash_l	8	Problems			Alt	+Shift+Q, >	c l
6	flash_l	۵	Console			Alt	+Shift+Q, C	2
1	led_e>	8	Advice					
	secon	*	Debug					
b an an		â	Memory B	rowser				
		1111	Registers					
		65	Expression	IS				
		(*)=	Variables			Alt	+Shift+Q, \	/
			Disassemb	bly				
		•	Breakpoin	ts		Alt	+Shift+Q, E	3
		عا	Modules					
		<i>.</i> ,	Terminal					
		ĝa,	Scripting (Console	•			
		2	Target Cor	nfigurat	tions			
		85	Outline			Alt	+Shift+Q, C)
		5	Stack Usag	je				
		=	Memory A	llocatio	on			
		۲	Optimizer	Assista	int			
			Other			Alt	+Shift+Q, C	2

Figure 4-36. Opening the Target Configuration Menu in CCS

- Users can import a project for this device to CCS and use that to connect to the device, or copy the target configuration file (.ccxml) from C2000Ware (C2000Ware x_xx_xx > device_support > DEVICE_FAMILY > common > targetConfigs) to the User Defined target configurations.
 - a. Find the device target config and then manually launch by right-clicking:

Target Configurations	×		🕱 🗯 🔶 🖃 🗖
type filter text			
> >	Wew Target Configuration Import Target Configuration Mexer Target Configuration Delete Rename Delete F2		
ſ	Launch Selected Configuration		
	Set as Default Link File To Project	>	
	Properties	Alt+Enter	

Figure 4-37. Launching a Target Configuration in CCS

4. When CCS brings up the debug window, select the intended CPU and connect to the target.



Figure 4-38. Connecting to the Target Core in CCS

- 5. If a window pops up stating there is a break in the boot ROM with no debug information available, or outside of program code, then follow the steps in Section 5.3 to debug the boot ROM.
- 6. Once the symbols are loaded, open the memory browser by going to View > Memory Browser.





Figure 4-39. Navigating to the Memory Browser in CCS

- In the memory browser tab, navigate to address 0xD00. Recall that the 0xD00 location specifies the BMSPs with the validity key (EMU-BOOTPIN-CONFIG) and 0xD04-x0D05 specifies the boot definitions (EMU-BOOTDEF-LOW).
- The objective is to configure a zero-pin CAN boot with CANRXA = GPIO5 and CANTXA = GPIO4, so all BMSPs need to disabled and the EMU-BOOTDEF-LOW needs to be set to 0x08 in the lowest index. If the boot option is programmed to any other entry in EMU-BOOTDEF-LOW, then the intended boot mode is not selected.
 - a. Set 0xD00-0xD01 (EMU-BOOTPIN-CONFIG) to 0x5AFF FFFF.
 - b. Set 0xD04 (EMU-BOOTDEF-LOW) to 0x0008.

Note

Besides the lower eight bytes in EMU-BOOTDEF-LOW, EMU-BOOTDEF-LOW and EMU-BOOTDEF-HIGH can be any set to any value as the values are ignored in zero-pin boot.

I Memory Br	rowser × 🦉 🕶 🐼 💌 🛷 😻 🍪 🗂 😁 🖇	- 0
Data ~	0xD00	P
Data:0xd00 -	0xD00 <memory 1="" rendering=""> ×</memory>	
16-Bit Hex - 1	TI Style \vee	
0×00000D00	FFFF 5AFF C628 DF58 0008 9B30 2D41 0008 2D41 0008	
0×00000D0A	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	
0×00000D14	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	
0×00000D1E	2D41 0008 2D41 0008 2D41 0008 2EE4 0008 2EDA 0008	
0x00000D28	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	
0x00000D32	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	
0x00000D3C	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	- 8
0×00000D46	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	- 8
0x00000D50	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	
0x00000D5A	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	
0x00000D64	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	
0x0000006E	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	
0×00000D78	2D41 0008 2D41 0008 2D41 0008 2D41 0008 2D41 0008	

Figure 4-40. Emulating a Zero-pin MCAN Boot with CANRXA=GPIO5 and CANTXA=GPIO4

- 9. Reset the CPU and perform an external reset (XRSn). Then, click on Resume to begin the boot sequence.
- 10. If there is a break in the boot ROM with no debug information available, or outside of program code, then follow the steps in Section 5.3 to load the boot ROM symbols. Afterwards, confirm that the device is in CAN boot.

Now, the MCAN bootloader (with GPIO assignments as specified by the boot option 0x08) in the ROM begins executing and waits to for a CAN frame to be transmitted from the host. At this point, the device is ready to receive code from the host.

Note

The MCAN bootloader has a 10 second window for accepting CAN-FD frames. If offered in the device boot options, then the SEND_TEST modes remove the timeout. These options use the same GPIO pins for communication as equivalent boot options. However, the GPIO pins transmit a frame before beginning the boot-loading process to help ascertain proper functionality of the module. In F280039x devices, boot options 0x68 can be used to avoid the timeout while using the same GPIO assignments as boot option 0x08.

The command line PC utility is a programming solution that can easily be incorporated into scripting environments for applications like production line programming. This was written using Microsoft Visual Studio in C++. The project and source can be found in C2000Ware (C2000Ware_x_xx_xx > utilities > flash_programmers > can_flash_programmer).

The host is responsible for sending the MCAN kernel image and flash (firmware) image to the MCU. The PEAK PCAN-USB Pro FD CAN bus analyzer is used as the host. The flash programmer project is built and run on Visual Studio 2019. The host programmer uses the PCAN_Basic API from PEAK [17]. The PCAN_Basic API can be used to send and receive CAN-FD frames on the CAN analyzer.

Note

The PEAK PCAN-USB Pro FD CAN bus analyzer is backwards compatible to receive both CAN frames as well as CAN-FD frames.

On the F28003x device, the clock to the MCAN module is switched to the external clock source by the Boot ROM. The external clock is 20MHz in the LaunchPad and the ControlCard. The Boot ROM configures the nominal bit rate to be 1Mbps, and the data bit rate to be 2Mbps. The host CAN programmer configures the PEAK CAN analyzer to have the same clock, nominal and data bit rate values.

The host initializes the analyzer for CAN-FD usage, sends the kernel over in 64-byte increments, and sends over the image in 64-byte increments with a delay of 100ms between each frame to give the Flash API time to program the data received into Flash. Once the firmware image has been written, the host CAN programmer exits.

To use this tool to program the C2000 device, make sure that the target board has been reset and is currently in the CAN boot mode as configured above, and connected to the PEAK PCAN-USB Pro FD CAN bus analyzer. The command line usage of the tool for a single core MCAN boot is described below, where -d, -k, and -a are mandatory parameters. Verbose output can be enabled with -v. More details on the parameters of the utility is detailed in [14].

can_flash_programmer.exe -d DEVICE -k KERNEL_FILE -a APPLICATION_FILE -v

- Navigate to the folder containing the compiled dcan_flash_programmer executable (C2000Ware x xx xx xx > utilities > flash programmers > can flash programmer).
- 2. Open a terminal and run the executable can flash programmer.exe with the following command.

can_flash_programmer.exe -d DEVICE_NAME -k <path_to_kernel_hex> -a <path_to_application_hex> -v

This loads the MCAN flash kernel into RAM of the device using the bootloader. The bytes transferred over the MCAN bus can be seen in the terminal. Then, the kernel executes and loads and programs flash with the file specified by the '-a' command line argument, as seen in Figure 4-41 and Figure 4-42. The kernel branches to the application and begins executing if successfully loaded into the flash.



Figure 4-41. MCAN Flash Programmer Kernel Loaded



Figure 4-42. MCAN Flash Programmer Application Loaded

4.2.5 USB Boot

Note

Since C2000 LaunchPads do not have a USB peripheral allowing for USB data transfer, unlike controlCards, these steps are based on a F28379D controlCARD Rev 1.3. More importantly, early silicon versions have a known bug in the USB bootloader preventing the device to communicate with the host PC, therefore, at least Rev C silicon for F28379D must be used.

The USB flash kernel is based on the ROM bootloader, communicating with the host PC application provided in C2000Ware (C2000Ware $x_x x_x x_x x$ utilities > flash_programmers > usb_flash_programmer) and essentially performing the same operations as the ROM bootloader. Because the ROM bootloader is equipped with the flash API, the bootloader is able to erase and program flash to perform the firmware update.



Note

This section demonstrates CPU1 USB boot loading. For a more detailed explanation on the USB kernel functionality and CPU2 usage, refer to the USB Flash Programming of C2000 Microcontrollers application report [20].

Flash kernel source and project files for CCS are provided in C2000Ware, in the examples directory of the corresponding device. 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 project name with a .dat extension.

- 1. Find the USB flash kernel project for the intended device in C2000Ware and import into CCS.
 - a. For F2837xD devices, the kernels can be found in C2000Ware_x_xx_xz_ > device_support > f2837xD > dual > F2837xD_usb_flash_kernels.
- 2. Make sure that the USB flash kernel project Active Build Target Configuration is set to *RAM*, since the kernel needs to be linked for execution in the RAM.



Figure 4-43. Generating the Correct USB Kernel Bin Output

3. Build the kernel project. These projects have a post-build step in which the compiled and linked .out file is converted into the correct binary format needed by the USB ROM bootloader and is saved as the example name with a .dat extension.

Note

If .dat output is not generated, then follow the steps in Section 4.1 to verify that the correct post-build step to generate the binary file is defined.

- 4. Repeat the build process for the firmware code that is loaded into the flash by the kernel.
 - a. Confirm that the Active Build Target Configuration is set for Flash.
 - b. Follow the steps in Section 4.1 to verify that the correct post-build step to generate the binary file is defined.
 - c. Build the firmware project.

After building the kernels in CCS, set up the device correctly to communicate with the host PC running the usb_flash_programmer.

1. Connect the mini-USB and micro-USB from the controlCARD to the host PC.

- 2. Set controlCARD GPIOs to debug using CCS and the On-Card XDS100v2 Emulator [20]. Refer to the *ROM Bootloader* section in the USB Flash Programming of C2000[™] Microcontrollers application report for the connectivity options.
 - a. Set A:SW1 Position 1 to On (up) and Position 2 to Off (down).



Figure 4-44. Configure the F2837xD controlCard to Debug Using CCS and the Onboard XDS Emulator

- 3. Open CCS to a workspace.
- 4. Select View > Target Configurations.



Figure 4-45. Opening the Target Configuration Menu in CCS

- Users can import a project for this device to CCS and use that to connect to the device, or copy the target configuration file (.ccxml) from C2000Ware (C2000Ware_x_xx_xx_x > device_support > DEVICE_FAMILY > common > targetConfigs) to the User Defined target configurations.
 - a. Find the device target config and then manually launch by right-clicking.



Figure 4-46. Launching a Target Configuration in CCS

6. When CCS brings up the debug window, select the intended CPU and connect to the target.



Figure 4-47. Connecting to the Target Core in CCS

- 7. If a window pops up stating there is a break in the boot ROM with no debug information available, or outside of program code, then follow the steps in Section 5.3 to debug the boot ROM.
- 8. Once the symbols are loaded, open the memory browser by going to View > Memory Browser.



Figure 4-48. Navigating to the Memory Browser in CCS

- 9. In the memory browser tab, navigate to address 0xD00.
- 10. Set device up for emulation *USB Boot*. The BMODE value is defined as 0x0C in the *Configuring Get Boot Options* section of the TMS320F28003xD Real-Time Microcontrollers technical reference manual [23]. Program 0xD00 to 0x0C5A.



- memory	0.0005	- · · ·																
Data	~ 0)xd00)													6	51	
Data:0xd00	<men< td=""><td>nory F</td><td>Rend</td><td>ering</td><td>1></td><td>x /</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></men<>	nory F	Rend	ering	1>	x /												
16-Bit Hex	- TI St	yle		\sim														
0x00000D00	0C5A	20C3	836C	3A7B	D6D5	9FEB	2ACC	0008										
0x00000D10	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0x00000D20	2ACC	0008	2ACC	0008	2896	0008	2B8C	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0×00000D30	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0×00000D40	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0x00000050	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
000000D60	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
3×00000D70	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0800000x6	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
3×00000D90	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		i.
0x00000DA0	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0×00000DB0	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0x00000DC0	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		1
0x00000DD0	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		2
0×00000DE0	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0×00000DF0	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0×00000E00	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0x00000E10	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0x00000E20	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0x00000E30	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0x00000E40	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		
0x00000E50	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008	2ACC	0008		

Figure 4-49. Configuring the F2837xD controlCard to Emulate USB Boot

- 11. Reset the CPU and then click on *Resume* to enable the USB boot sequence.
- 12. If there is a break in the boot ROM with no debug information available, or outside of program code, then follow the steps in Section 5.3 to load the boot ROM symbols. Afterward, confirm that the device is in USB boot in the boot ROM.

Command line options and file IO can be done through the C standard library, but USB operations can only be done through the device driver framework of the operating system. 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 C2000Ware uses WinUSB, but source code is provided for both libraries. The WinUSB drivers need to be installed on the device for the USB bootloader to run.

- 1. Go to the Device Manager and right-click on the unrecognized device under *Universal Serial Bus controllers*. The F28379D displays as *Stellaris Device Firmware Upgrade*.
- Select Update Drivers > Browse my computer for drivers > Let me pick from a list of available drivers on my computer > Have Disk.
 - a. Input this into the Copy from manufacturer's files from: C2000Ware_x_xx_xx\utilities\flash_programmers\usb_flash_programmer\windows_driver\x86

earch for drivers in this I	ocation:	
C:\ti\C2000Ware_5_04_00	0_00\utilities\flash_programmers\usb_flash	F ~ Browse
→ Let me pick from This list will show av	m a list of available drivers on m ailable drivers compatible with the device,	y computer and all drivers in the sam

Figure 4-50. Browsing C2000Ware for the USB Windows Drivers



3. Now, the WinUSB drivers are installed onto the device and appear as *F28x7x USB Boot Loader* under the *Texas Instruments Microcontrollers* category if done successfully.



Figure 4-51. The F28x7x USB Boot Loader device appearing in the Device Manager

4. Reset the CPU and perform an external reset (XRSn). Then, click on *Resume* to begin the boot sequence.

The command line PC utility is a lightweight (approximately 64KB executable) programming solution that can easily be incorporated into scripting environments for applications like production line programming. This was written using Microsoft Visual Studio in C++. The project and source can be found in C2000Ware (C2000Ware_x_xx_xx_x > utilities > flash_programmers > usb_flash_programmer > src > VS2010_USBLoader2000).

Input filepaths are loaded in ascending order on the command line, so the flash kernel filepath must be provided listed first to be loaded into the RAM. After the flash kernel is loaded, the ROM transfers control and the kernel begins to execute to program the application to the flash. The flash kernel must prepare the device for flash programming before the device is ready to begin communications, so a small delay is needed. During this time, the flash kernel configures the PLL and flash wait states.

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. This is after the entry point is received that the kernel begins to erase the flash. Erasing the flash can take a few seconds. Note, that while the application load looks as if failed, the flash is being erased. Once the flash is erased, the application load continues by transferring each block of application code and programming to flash.

- 2. Open a terminal and run the executable usb_flash_programmer.exe with the following command:

usb_flash_programmer.exe <path_to_kernel_dat> <path_to_application_dat>

Note

Both the flash kernels and flash application MUST be in the binary boot format as discussed earlier in Section 4.1.

Once the application is programmed into flash, the flash kernel attempts to run the application by branching to the entry point that was transferred at the start of the application load process.



5 FAQ

This section details recommendations to common roadblocks encountered by users when attempting to boot load across the various peripheral boot modes.

5.1 Selecting the BMSP GPIOs with a Software-based Implementation

Question: Is there a way to implement a software-based boot loading process without manual intervention?

Answer: C2000[™] Software Controlled Firmware Update Process application report [7] describes a softwarecontrolled firmware update process on C2000 devices using existing boot modes without the need to manually select boot mode. The method described in this document directly applies to F28004x device and can be applied to legacy devices with necessary modifications.

5.2 Running a Flash Kernel from the Flash Instead of the RAM

Question: How to modify the flash kernels to run from the flash instead of the RAM?

Answer: On multibank devices, users can implement a custom bootloader in the flash to directly load application code onto the other flash banks. On single-bank devices, a custom bootloader in the flash can perform firmware upgrades on individual sectors within the same flash bank, but the Flash API must execute from RAM.

Note Flash API functions and the application functions that call Flash API must not be executed from the same bank. Executing the Flash API on the same bank can introduce race conditions when programing and erasing data. The Flash API needs to be executed from the RAM or another flash bank (if another bank exists for the same core). [6]

Using a custom bootloader provides more flexibility to the user in the boot flow, allowing for different peripheral GPIOs to be used as opposed to only having access to a limited, predefined set of GPIO assignments in the boot ROM. Having a custom bootloader also effectively reduces the number of steps in the bootloading process as there's no need to load an intermediate flash kernel into the RAM to program the flash.

Moreover, having a flash-based bootloader can also act as a fail-safe if the firmware upgrade process fails. Booting to code that does not get upgraded (that is, the flash-based bootloader) minimizes the chance that the device tries to boot to corrupted code, since the bootloader can use multiple methods to verify the existing firmware (key value, checksum, and so forth.). The same can be achieved in a RAM-based flash kernel too, but it's not practical to load this in every time the device boots.

One way to implement a custom bootloader is to modify the existing flash kernel projects in C2000Ware to execute from the flash instead of the RAM. This requires modifications to the kernel's linker command file, which specifies where the application is loaded in memory, and adding a pre-defined symbol to be used by the compiler.

Note

The following steps modify the F28P55x SCI flash kernel to execute from the flash, but the same flow can be applied to any C2000 device with an existing flash kernel project in C2000Ware.

1. Right-click the project and select Add Files/Folders.

Ch	EXPLORER		8003x_flash_bank0_LDFU_lnk.cmd	Deleted)	C flash_ker
5	> OPEN EDITORS		flash_kernel_ex3_sci_flash_kern	el 👌 🖷	28p55x_flash_ker
Q	WORKSPACE_CCSTHEIA PT_flash_kernel_ex3_sci_flash_kernel_(CPU1_RAM)	1	2 {		
je	> Ct_ CPU1_RAM > Ct_ device > Ct_ targetConfigs		New File New Folder Add Files/Folders		
₽ B	 28p55x_flash_kernel_ex3.cmd C cpu1bootrom.h C cpu1boot_boot_modes.h C dcsm.c 		Reveal in File Explorer Open in Integrated Terminal Select for Compare		Ctrl+Alt+P 💟
~	D _☉ driverlib.lib C f28p55x_flash_kernel_ex3_boot.c		Find in Folder		Alt+Shift+F
	 C f28p55x_flash_kernel_ex3_erase.c C f28p55x_flash_kernel_ex3_erase.h C f28p55x_flash_kernel_ex3_sci_boot.c C f28p55x_flash_kernel_ex3_sci_flash_kernel.c 		Copy Paste Copy Path Copy Relative Path	Ctrl+K,	Ctrl+C Ctrl+V Alt+Shift+C Ctrl+Shift+C
	 C f28p55x_flash_kernel_ex3_sci_get_function.c C f28p55x_flash_kernel_ex3_verify.c C f28p55x_flash_kernel_ex3_verify.h C FAPI_F28P55x_EABI_v4.00.00.lib 	×	Build Project(s) Clean Project(s) Rebuild Project(s)		Ctrl+B
	 flash_kernel_ex3_codestartbranch.asm flash_kernel_ex3_commands.h flash_programming_128p55x.h flashani 512hit programming [CD11 514511 		Build Configurations System Configurations Executable Actions		> > >
	 C_flashapi_ex1_programming [CPU1_FLASH] C_led_ex1_blinky [CPU1_RAM] 	12 6) Elash Project		F5 Ctrl+F5
	> 🗠 sysctl_ex2_xclkout_config [CPU1_LAUNCHXL_RAN	4]	Delete Duplicate		Delete
			Properties		Alt+Enter

Figure 5-1. Adding a File to a Project into CCS

- 2. Navigate to the following path in C2000Ware and select the generic flash linker command file for the intended device.
 - a. C2000Ware_X_XX_XX\device_support\DEVICE_NAME\common\cmd
- 3. Right-click the RAM linker command file and select *Exclude from Build*, so the flash linker command file is used in the compiler build.



Figure 5-2. Excluding the RAM Linker Command File from the Compiler Build

Note

With the generic flash linker command file, the kernel is loaded into flash bank 0 (entry point at 0x80000). If users want to place the kernel in a different flash bank, then modify the SECTIONS allocated to flash bank 0 in the linker command file to the intended flash bank.

Furthermore, in MEMORY portion of the linker command file, BEGIN must also be updated to the intended flash entry point location (codestart) as needed, including the flash bank origin and length specifications to account for the codestart allocation at the entry point.

Refer to [33] for more details on linker command file directives.

4. Create a new build configuration by right-clicking the project and navigating to *Build Configurations* > *Manage...*



Figure 5-3. Creating a New Build Configuration for the Project

a. Name the build configuration CPU1_FLASH and opt to copy settings from the CPU1_RAM configuration.

		_	 _		
Build-confi	igurations				+ 🖻 🖉 🗸
錄 CPU1_I	RAM active			/	
Create	new build-cor	nfiguration	/		×
Name		CPU1_FLASH			
Descri	ption				
Copy s	settings from	CPU1_RAM			~
				Cancel	ОК
					Close

Figure 5-4. Creating the CPU1_FLASH Build Configuration

- 5. Set the active build configuration to CPU1_FLASH.
- 6. In device.c, the time critical code and flash setup code (called *Ramfuncs*) is copied to the RAM. Moreover, the flash initialization function to setup flash waitstates must reside in RAM. Since the kernel is executing in the flash now, the symbol *_FLASH* must be defined for these conditions to occur.
 - a. Right-click and navigate to the *Properties > Tools > C2000 Compiler > Predefined Symbols*. Define the symbol _*FLASH*.



Properties for: flash_kernel_ex3_sci_flas	h_kernel [CPU1_FLASH]	×
 ✓ General IN Dependencies () Variables ✓ Build i Esteps ⊙ Link Order 	Pre-define NAME (define, -D) \$(COM_TI_C2000WARE_SYMBOLS) CPU1 _FLASH	$+ - arrho \wedge \vee$
 ✓ 2000 Compiler ✓ 2000 Compiler Processor Options Optimization Include Options Performance Advisor 	Undefine NAME (undefine, -U)	
Predefined Symbols > Advanced Options > ∅ C2000 Linker > ∅ C2000 Hex Utility Executable Actions Clang-Tidy		ø
Debug		Cancel Save and Close

Figure 5-5. Add the Predefined Symbol _FLASH to Initialize Flash Functions

7. Re-build the flash kernel project with the *CPU1_FLASH* build configuration. Now, the flash kernel is built to execute from the flash and can be programmed onto the device flash.

Note

Since the kernel is already loaded onto the device, then the serial_flash_programmer_appln.exe in C2000Ware must be used. This executable can be called with the same command line arguments as the normal serial flash programmer, but the kernel argument (-k) is not required.

5.3 No Symbols Defined When Debugging Boot ROM

Question: When I debug the boot ROM, the ROM states there is a break in the boot ROM with no debug information available, or outside of program code. What am I missing?

Answer: Users can step through the device boot ROM by loading the boot ROM symbols (.out file) to the device. Loading symbols can be a valuable debug method. This option adds the symbols available in the generated project .*out* file for debugging purposes instead of loading the actual .*out* program onto the core by CCS. This is also why users can use this method with the boot ROM or built-in bootloaders to debug and get enhanced visibility.

1. Open CCS to a workspace.

2. Select View > Target Configurations.



Figure 5-6. Opening the Target Configuration Menu in CCS

Users can import a project for this device to CCS and use that to connect to the device, or copy the target configuration file (.ccxml) from C2000Ware (C2000Ware_x_xx_xx_x > device_support > DEVICE_FAMILY > common > targetConfigs) to the User Defined target configurations in this window (View > Target Configurations). Either way, find the device target config and manually launch by right-clicking:

R Target Configurations	×		🗑 🗶 🔗 🖃 🗖
type filter text			
 > i Projects > User Defined I TMS320F2800 I TMS320F28P6 I TMS320F28P6 	New Target Configuration Import Target Configuration		
i 128002x.ccxmi i f28004x.ccxmi i f2807x.ccxmi	 Delete Rename Refresh 	Delete F2 F5	
	Launch Selected Configuration		
	Set as Default Link File To Project	>	
	Properties	Alt+Enter	

Figure 5-7. Launching a Target Configuration in CCS

4. When CCS brings up the debug window, select the intended CPU and connect to the target.



Figure 5-8. Connecting to the Target Core in CCS

5. At this point, a window pops up stating there is a break in the boot ROM with no debug information available, or outside of program code.

Break at address "0x3fdf2a" with no o	debug information available, or outside of program code
View Disassembly	
Configure when this editor is shown	Preferences

Figure 5-9. CCS View When No Boot ROM Symbols Are Loaded



6. Navigate to the toolbar and click the button to Load Symbols.

	<u>a</u> -	🔊 🔅 🔹 🕹 💣 🔹 🎋 👻 🔍 💿 🔦 💌 🖾 🖉 💉 🖅 📑	
Project Explorer ×	0	Load Program	Ctrl+Alt+L
> 👺 boot_ex1_cpu1_secure_flash	2	Reload Program	Ctrl+Alt+R
> Boot_ex1_cpu1_secure_flash_cpu1	٨	Load Symbols	
> 😂 Example_28335_Flash	\$\$	Add Symbols	
	ø	Verify Program	
f280015x_flash_kernel_ex5_dcan_flash_kernel i i	*	Remove All Symbols	
## flash_kernel_ex4_can_flash_kernel	-	· · · · · · · · · · · · · · · · · · ·	
Flash_kernel_ex5_dcan_flash_kernel	è,	C:\ti\libraries\\Release\F280015x_ROM.out	
Bed_ex1_blinky	Ð	C:\Users\workspace_v12\\CPU1_FLASH\secondary_bootloader.out	
> 🐸 led_f28003x	3	C:\Users\workspace_v12\\CPU1_FLASH\primary_application.out	
Secondary_bootloader [Active - CPU1_FLA]	ڪ	C:\ti\libraries\\Release\F280015x_CPU1_Full_ROM.out	
	₫	C:\Users\workspace_v12\\CPU1_LAUNCHXL_FLASH\led_f28003x.out	
	ڪ	C:\ti\C2000Ware_5_01_00_00\\Release\F280013x_ROM.out	
	ڪ	C:\Users\workspace_v12\\CPU1_FLASH\boot_ex1_cpu1_secure_flash.out	
		C:\Users\workspace_v12\\CPU1_FLASH\boot_ex1_cpu1_secure_flash.hex	
		C:\Users\workspace_v12\\CPU1_FLASH\boot_ex1_cpu1_secure_flash.out	
	2	$\label{eq:cluster} C:\label{eq:cluster} C:$	

Figure 5-10. Navigating to Load Symbols in CCS

- 7. Load the boot ROM .out file. This can be found in C200Ware at:
 - a. C2000Ware_x_xx_xx > libraries > libraries > boot_rom > DEVICE_FAMILY > REV# > rom_sources > CPU# > ccs_files > Release
- 8. If a window pops up stating the source file cannot be found, then users can select *Locate File* and find in C2000Ware at:
 - a. C2000Ware_x_xx_xx > libraries > boot_rom > DEVICE_FAMILY > REV# > rom_sources > CPU# > DEVICE_FAMILY_ROM > bootROM > source

Can't find a source file at	
"C:/Projects/Boot_ROM/bootrom_f28006x/F28006x_ROM_	dev_PG1.0/cpu1/F28006x_ROM/bootROM/source/cpubrom_init_boot.as
m"	
Locate the file or edit the source lookup path to include its	location.
View Disassembly	
Locate File	

Configure when this editor is shown Preferences...

Figure 5-11. Locating the Boot ROM Source Files

The file opens to show the current instruction in the boot ROM and users can go through how to debug, reset and restart the device.

5.4 Writing Values in the OTP Using the On-Chip Flash Tool

This section demonstrates how to program the OTP using the On-Chip Flash Tool with two example use cases.

Note

Although this section is based on F280015x devices, the same flow can be applied to any C2000 device that supports custom BMSPs and boot mode tables.

Device specific information can be found in the *Boot ROM* chapter of the device's Technical Reference Manual (TRM).

- 1. After launching a debug session with the intended CPU core, open the On-Chip Flash Tool (see Section 3 for how to find in CCS).
- 2. Find GPREG(BOOTCTRL) under Zone 1 or 2 (recall Zone 2 takes precedence over Zone 1). Users can write to the OTP-BOOTPIN-CONFIG and OTP-BOOTDEF-LOW/OTP-BOOTDEF-HIGH registers.



a. Table 2-7 shows the locations for the boot configuration registers on F280015x devices, and can be found in the *Boot ROM Configuration Registers* section in the device-specific TRM. The *Register Name* column is how the boot ROM registers are referenced in the On-Chip Flash tool.

Example 1: Zero Boot Modes Select Pins

This use case exhibits a scenario for an application that does not wish to use any BMSPs and always have the device boot to Flash entry point 0x88000.

Refer to *GPIO Assignments* section in the device TRM for values to set in the table. For Flash entry points, see the *Entry Points* section in the TRM details about the entry point addresses for various boot modes. These entry points direct the boot ROM what address to branch to at the end of booting as per the selected boot mode.

- 1. Program the BOOTPIN_CONFIG location in OTP as follows:
 - a. Set BOOTPIN_CONFIG.BMSP0 to 0xFF (disabled)
 - b. Set BOOTPIN_CONFIG.BMSP1 to 0xFF (disabled)
 - c. Set BOOTPIN_CONFIG.BMSP2 to 0xFF (disabled)
 - d. Set BOOTPIN_CONFIG.KEY to 0x5A for the boot ROM to treat these register bits as valid and use the custom boot table
- 2. Program the BOOTDEF location options for the device. This essentially sets up a device-specific boot mode table.
 - a. Set BOOTDEF.BOOTDEF0 to 0x23 for booting to Flash (entry address option 1). This sets Flash boot to boot table index 0.

Figure 5-12 shows the completed input fields in the On-Chip Flash tool to program this example.

GPREG(BOOTCTRL)	
Z1-GPREG1 (0x78008)(32 bits) 0x	SAFFFFF
Z1-GPREG2 (0x7800A)(32 bits) 0x	FFFFFF
Z1-GPREG3 (0x7800C)(32 bits) 0x	FFFFF23
Z1-GPREG4 (0x7800E)(32 bits) 0x	FFFFFF
Program	

Figure 5-12. Example 1: Flash Plugin Boot Configuration Programmed

BMSP Index	BOOTDEF
0	0x23 (Flash Boot to address 0x88000)

Example 2: Two Boot Modes Select Pins

This use case demonstrates a more common scenario for an application using two boot mode select pins to select between CAN, Secure Flash, and SCI boot in the custom boot table.

Refer to *GPIO Assignments* section in the device TRM for values to set in the table. For Flash entry points, see the *Entry Points* section in the TRM details about the entry point addresses for various boot modes. These entry points direct the boot ROM what address to branch to at the end of booting as per the selected boot mode.

- 1. Program the BOOTPIN_CONFIG location in OTP as follows:
 - a. Set BOOTPIN_CONFIG.BMSP0 to a user specified GPIO, such as 0x2A for GPIO42
 - b. Set BOOTPIN_CONFIG.BMSP1 to a user specified GPIO, such as 0x58 for GPIO88
 - c. Set BOOTPIN_CONFIG.BMSP2 to 0xFF (disabled)
 - d. Set BOOTPIN_CONFIG.KEY to 0x5A for the boot ROM to treat these register bits as valid and use the custom boot table.



- Program the BOOTDEF location options for the device. This essentially sets up a device-specific boot mode table.
 - a. Set BOOTDEF.BOOTDEF0 to 0x6A for Secure Flash boot (entry address option 3). This sets Secure Flash boot to boot table index 0.
 - b. Set BOOTDEF.BOOTDEF1 to 0x22 for CAN boot option 1. This sets CAN boot to boot table index 1.
 - c. Set BOOTDEF.BOOTDEF2 to 0x41 for SCI boot option 2. This sets CAN boot to boot table index 2.

Figure 5-13 shows the completed input fields in the On-Chip Flash tool to program this example.

GPREG(BOOTCTRL)	
Z1-GPREG1 (0x78008)(32 bits) 0x	5AFF582A
Z1-GPREG2 (0x7800A)(32 bits) 0x	FFFFFF
Z1-GPREG3 (0x7800C)(32 bits) 0x	FF41226A
Z1-GPREG4 (0x7800E)(32 bits) 0x	FFFFFF
Program	

Figure 5-13. Example 2: Flash Plugin Boot Configuration Programmed

Table 5-2. Two boot mode beleet I in bonngaration		
BMSP Index	BOOTDEF	
0	0x6A (Secure Flash Boot to address 0x90000)	
1	0x22 (CAN boot 1 with alt. GPIOs)	
2	0x41 (SCI boot 2 with alt. GPIOs)	
3	Don't care (unused)	

Table 5-2. Two Boot Mode Select Pin Configuration

5.5 Writing Values in the OTP Using the Flash API Plugin

Note

Although this section is based on F280015x devices, the same flow can be applied to any C2000 device that supports custom BMSPs and boot mode tables.

Device specific information can be found in the *Boot ROM* chapter of the device's Technical Reference Manual (TRM).

The custom boot configurations in Section 5.5 can also be written into DCSM OTP using the compiler's RETAIN and DATA_SECTION pragmas outlined in [31]. The steps below follow the example in Example 2: Two Boot Modes Select Pins.

1. Open any of the C2000Ware examples in CCS and add the following code snippet above the main function (it can be anywhere in the file just outside the functions). These lines program the BOOTPIN-CONFIG and BOOTDEF-LOW registers, device respectively.

```
#pragma RETAIN(otp_z1_data_1)
#pragma DATA_SECTION(otp_z1_data_1,"dcsm_zsel_z1");
const long otp_z1_data_1 = 0x5AFF582A;
#pragma RETAIN(otp_z1_data_2)
#pragma DATA_SECTION(otp_z1_data_2,"dcsm_zsel_z1_2");
const long otp_z1_data_2 = 0xFF41226A;
```



 In the linker command file for the project, add the following lines defining the DCSM OTP BOOTPIN-CONFIG and BOOTDEF-LOW memory map locations as defined in the *Boot ROM Configuration Registers* section of the device-specific TRM. The addresses below are for the DCSM user OTP boot configuration registers on F280015x devices.

```
MEMORY {
    PAGE 0:
        DCSM_ZSEL_Z1_P0: origin = 0x078008, length = 0x000002 // Z1-OTP-BOOTPIN-CONFIG
        DCSM_ZSEL_Z1_P1: origin = 0x07800C, length = 0x000002 // Z1-OTP-BOOTDEF-LOW
}
SECTIONS {
    dcsm_zsel_z1_1 :> DCSM_ZSEL_Z1_P0, PAGE = 0
    dcsm_zsel_z1_2 :> DCSM_ZSEL_Z1_P1, PAGE = 0
}
```

3. Rebuild the example and load to the target by JTAG in CCS. The program loader and Flash API plugin in CCS handles the writing of these values into OTP locations. These values have to be selected and written carefully as the OTP locations cannot be reprogrammed.

6 Summary

Embedded processors often require flexible programming methods when JTAG debug probes cannot be reliably used, especially in field-deployable systems. C2000 microcontrollers address this need by providing bootloading utilities in the Boot ROM to enable device firmware upgrades. These bootloaders allow users to load application code from an external host into RAM using various communication interfaces, with the ability to use flash kernels to subsequently program the on-chip flash memory. This document describes the fundamental bootloading configurations and details how to leverage common boot modes to load application code into C2000 devices.



7 References

- 1. Texas Instruments, Code Composer Studio[™] integrated development environment (IDE), webpage
- 2. Texas Instruments, UniFlash flash programming tool, webpage
- 3. Texas Instruments, C2000Ware for C2000 MCUs, webpage
- 4. Texas Instruments, [E2E] FAQ on Flash software for C2000 MCUs, webpage
- 5. Texas Instruments, [E2E] FAQ on C2000 CCS Flash plugin and UniFlash, webpage
- 6. Texas Instruments, [E2E] FAQ on Flash API usage for C2000 devices, webpage
- 7. Texas Instruments, C2000[™] Software Controlled Firmware Update Process , application report
- 8. Texas Instruments, C28x Academy, Dual Code Security Module (DCSM), webpage
- 9. Texas Instruments, C2000[™] DCSM Security Tool , application report
- 10. Texas Instruments, C2000 Multicore Development User Guide
- 11. Texas Instruments, TMS320C28x Assembly Language Tools, user's guide
- 12. Texas Instruments, TI Arm Clang Compiler Tools User's Guide Linker Description
- 13. Texas Instruments, Secure BOOT on C2000 Device, application report
- 14. Texas Instruments, Serial Flash Programming of C2000™ Microcontrollers, application note
- 15. Texas Instruments, Introduction to the Controller Area Network (CAN), application report
- 16. Texas Instruments, CAN Flash Programming of C2000 Microcontrollers, application note
- 17. PEAK System, PCAN-Basic API
- 18. Texas Instruments, Getting Started with the MCAN (CAN FD) Module, application note
- 19. Texas Instruments, [E2E] TMS320F28377D: 28377D USB Bootloader, webpage
- 20. Texas Instruments, USB Flash Programming of C2000™ Microcontrollers, application report
- 21. Texas Instruments, XDS100 Debug Probe, product page
- 22. Texas Instruments, Delfino[™] TMS320F28379D controlCARD R1.3, user's guide
- 23. Texas Instruments, TMS320F28003xD Real-Time Microcontrollers, technical reference manual
- 24. Texas Instruments, TMS320F280015x Real-Time Microcontrollers, technical reference manual
- 25. Texas Instruments, TMS320F28P65x Real-Time Microcontrollers, technical reference manual
- 26. Texas Instruments, TMS320F2837xD Real-Time Microcontrollers, technical reference manual
- 27. Texas Instruments, TMS320F28P55x Real-Time Microcontrollers, technical reference manual
- Texas Instruments, TMS320F2838xD Real-Time Microcontrollers With Connectivity Manager Technical Reference Manual (SPRUII0)
- 29. Texas Instruments, TMS320F280015x Real-Time Microcontrollers, data sheet
- 30. Texas Instruments, TMS320F280039x Real-Time Microcontrollers, data sheet
- 31. Texas Instruments, F021 Flash API Version 2.01.01, reference guide
- 32. Texas Instruments, TMS320F280013x/15x Flash API Version 2.00.10.00, reference guide
- 33. Texas Instruments, TI Linker Command File Primer, webpage
- 34. Texas Instruments, TMS320C28x Optimizing C/C++ Compiler, user's guide
- 35. Texas Instruments, [E2E] LAUNCHXL-F2800157 CAN Flash Programming, webpage

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 Copyright © 2025, Texas Instruments Incorporated