Design Guide: TIDM-02011

Live Firmware Update Reference Design with C2000™ Real-Time MCUs

Sira Rao

Description

This reference design illustrates Live Firmware Update (LFU) without device reset on two C2000™ real-time MCUs, one with hardware features supporting LFU. LFU is illustrated on both the C28x CPU and the Control Law Accelerator (CLA). The software available with this design helps accelerate your time to market. LFU without device reset is an important consideration for high availability systems similar to Server power supply units (PSU), where downtime needs to be minimized. This reference design also has an example that illustrates Firmware Over-the-Air (FOTA) functionality.

Resources

TIDM-02011	Design Folder
TIDM-DC-DC-BUCK	Product Folder
TMS320F28003x,TMS320F28004x	Product Folder
BOOSTXL-BUCKCONV	Product Folder
LAUNCHXL-F280039C, LAUNCHXL-F280049C	Product Folder
C2000WARE-DIGITALPOWER-SDK	Software Folder

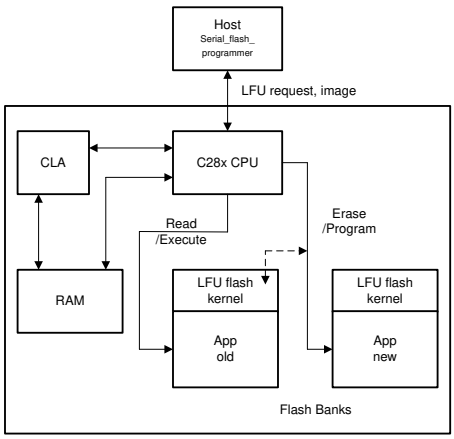
Ask our TI E2E™ support experts

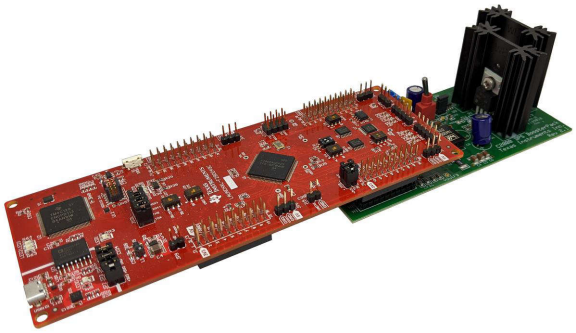
Features

- LFU Reference Design based on the C2000™ Digital Power BoosterPack™ plug-in module
- Reference software demonstrating LFU switchover without device reset
- LFU examples illustrating use of LFU hardware features on MCU
- LFU examples illustrating seamless switchover without loss of real-time interrupts
- LFU examples for C28x CPU as well as CLA
- Compiler LFU support integrated, Embedded Application Binary Interface (EABI) output format
- Example for FOTA

Applications

- Merchant Network and Server PSU





1 System Description

In applications similar to server power supply, metering, and so on, the system is desired to be run continuously to reduce downtime. But typically, during firmware upgrades, due to bug fixes, new features, and or performance improvements, the system is removed from service causing downtime for associated entities as well. This can be handled with redundant modules but with an increase in total system cost. An alternate approach, called Live Firmware Update (LFU), allows updating the firmware while the system is still operating. Switching to new firmware can be done either with or without resetting the device, with the latter being more complex.

This reference guide presents details on LFU without Device Reset using two Flash banks on a TMS320F28003x or TMS320F28004x device, detailing the specific challenges involved and suggestions on how to address them. LFU is implemented on the *C2000™ Digital Power Buck Converter BoosterPack* reference design. The document illustrates LFU capabilities with the main control loop running on either the C28x CPU or the CLA.

1.1 Key System Specifications

Table 1-1. Key System Specifications

PARAMETER	SPECIFICATIONS
LFU Switchover Time	LFU switchover must complete within the idle time available. LFU switchover time represents the time for which interrupts are disabled. Idle time represents the longest interval between Interrupt Service Routines (ISRs). It will depend on system parameters such as interrupt rate and ISR CPU load

2 System Overview

2.1 Block Diagram

The block diagram of an LFU based system is shown in Figure 2-1.

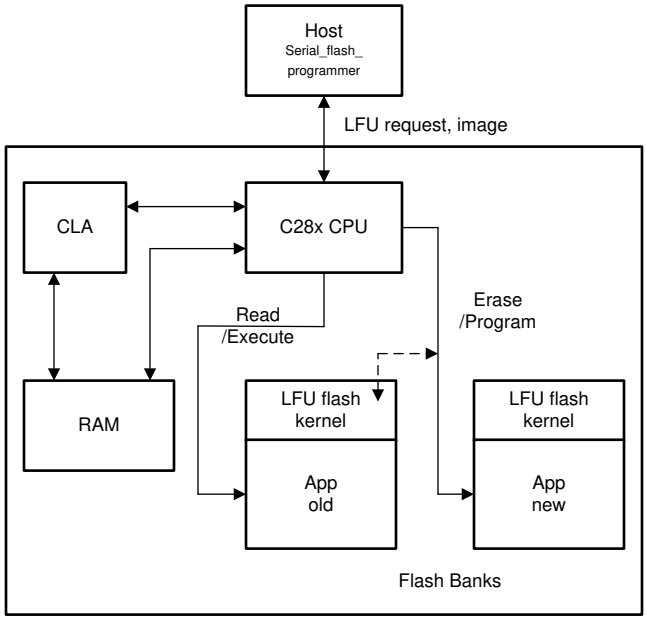


Figure 2-1. TIDM-02011 Block Diagram

2.2 Design Considerations

2.2.1 Building Blocks

The LFU design consists of a number of building blocks – a desktop Host application from where the LFU command is issued, a Custom Bootloader on the target device's Flash which communicates with the Host and enables LFU, a communication peripheral connecting the host to the target (for example, SCI/UART, CAN, I2C, and so on), the application to be downloaded and executed which is LFU compatible, Compiler with LFU support, the MCU with LFU related hardware support, and Flash memory with multiple physically separate Flash banks. Dual or more Flash banks allows application firmware resident on one Flash bank to execute, while the other Flash bank is updated.

2.2.2 Flash Partitioning

Figure 2-2 shows how the dual-bank Flash is partitioned. Two sectors in each bank are allocated to the custom bootloader, which comprises of Flash bank selection logic, the SCI kernel, and Flash APIs. These do not change during firmware upgrades. Bank 1 does not contain bank selection logic. The rest of the Flash sectors in the bank are allocated to the application. Bank selection logic allows the bootloader to determine which, if any, of the Flash banks are programmed, and which bank contains the more recent application firmware version. By implication, this function is the entry point of the software system. The SCI kernel is a function that implements the transfer of the image from the host, and programming of Flash through Flash programming APIs (either in Flash or in ROM). A few locations in Sector 2 are reserved to store the below information:

- START – Indicates that Flash erase is complete and program/verficication is about to begin.
- Firmware Revision number (REV) – Used by the bank selection logic to determine the newer firmware version between banks 0 and 1.
- KEY – The firmware in a bank is considered valid if this location contains a specific pattern.

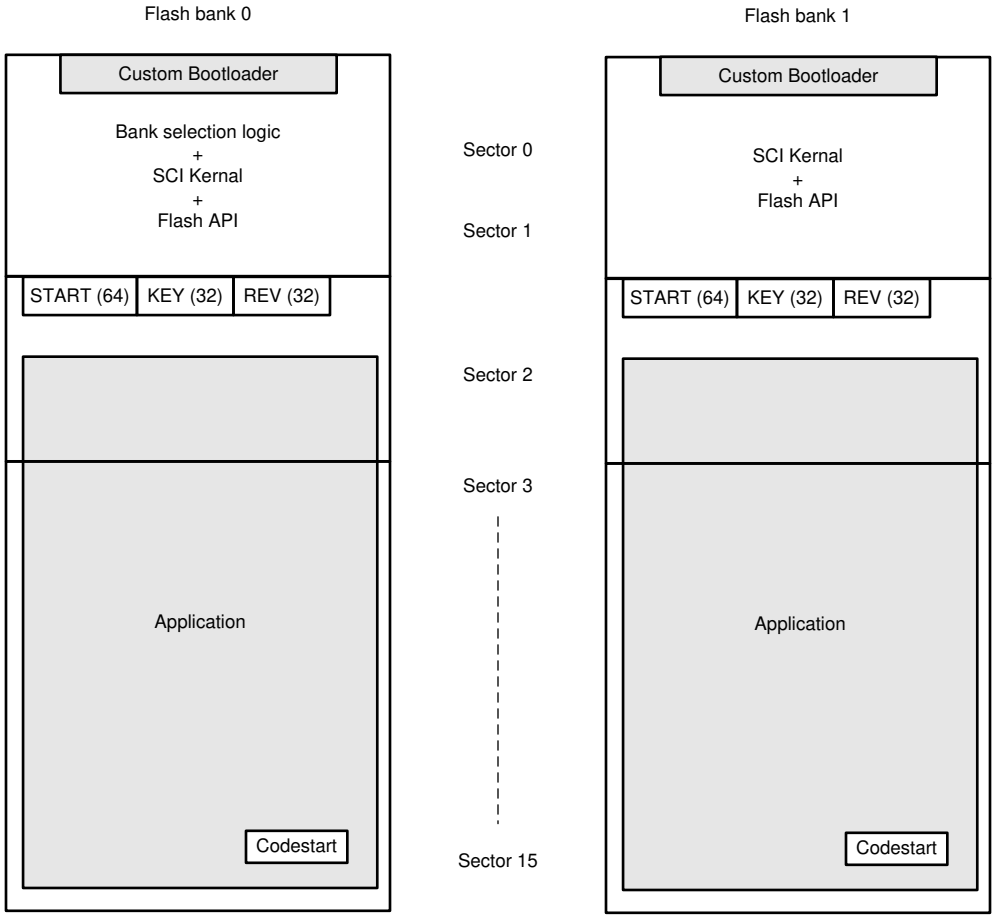


Figure 2-2. Dual Flash Bank Partitioning

2.2.3 LFU Switchover Concepts

The key considerations when preparing firmware for LFU are operational continuity and LFU switchover time. Operational continuity is achieved through persistence of state, which means keeping common static and global variables in RAM at the same addresses between firmware versions, and avoiding re-initialization of those variables when the new firmware takes effect. Compiler support for LFU is used to enable persistence of state.

Activating the new firmware involves branching from old firmware to the LFU entry point of the new firmware, execute the compiler's LFU initialization routine, arrive inside main() of the new image, and perform any additional initialization. This is where interrupts are briefly disabled, initialization that needs interrupts to be disabled is performed (e.g. Interrupt vector updates, function pointer updates), before interrupts are re-enabled. This last time interval is defined as the LFU switchover time.

LFU is simplified when there is hardware support to swap **Flash banks [2]**, where either Flash bank can be mapped to a fixed address space, considered the *Active* bank. The *Inactive* bank is mapped to a different address space, and is the bank that is updated. C2000™ MCUs do not currently support Flash bank swap, so the user will need to keep track of the Flash bank where application firmware will be resident, and make the necessary assignments and adjustments in a linker command file.

Function pointers and Interrupt vectors need to be re-initialized inside main(), since their locations will be different between Flash banks. C2000™ MCUs support a large number of interrupt vectors (typically 192), so it is not practical to re-initialize all of them. Usually, only a few are used, and the rest are assigned to a default vector. The F28003x device contains LFU specific hardware features (Interrupt vector swapping, RAM block swapping) that enable reduction in the LFU switchover time.

If there are changes to array sizes or addition of variables to a structure, the user needs to manage these appropriately, by using pragmas early in the development cycle to place arrays and structures at fixed locations, but with sufficient headroom to account for their potential growth in future firmware. With this approach, only newly added fields need to be initialized.

2.2.4 Application LFU Flow

Figure 2-3 shows the LFU software flow diagram. After a device reset, execution always begins in the bank selection logic, which determines which Flash bank to execute from based on the firmware revision field, and pass control to the corresponding application. After necessary system initialization and enabling of interrupts, a real-time control loop executes within an ISR corresponding to a specific interrupt vector. During the idle time between ISRs, a background loop consisting of lower priority functions executes. If the host issues an LFU command, it triggers an SCI Receive interrupt in the MCU, and a corresponding ISR (lower in priority than the control-loop ISR) executes and identifies a host command request.

In the background loop, the command is parsed, the LFU request is identified, control passes to the custom bootloader i.e. SCI Flash kernel, which downloads the new application image from the host and programs the appropriate Flash bank. If the application on Flash bank 1 is running, then control passes to the SCI Flash Kernel on bank 1, so as to program bank 0. Once the new application image is in Flash, the process of switching over to new firmware can begin.

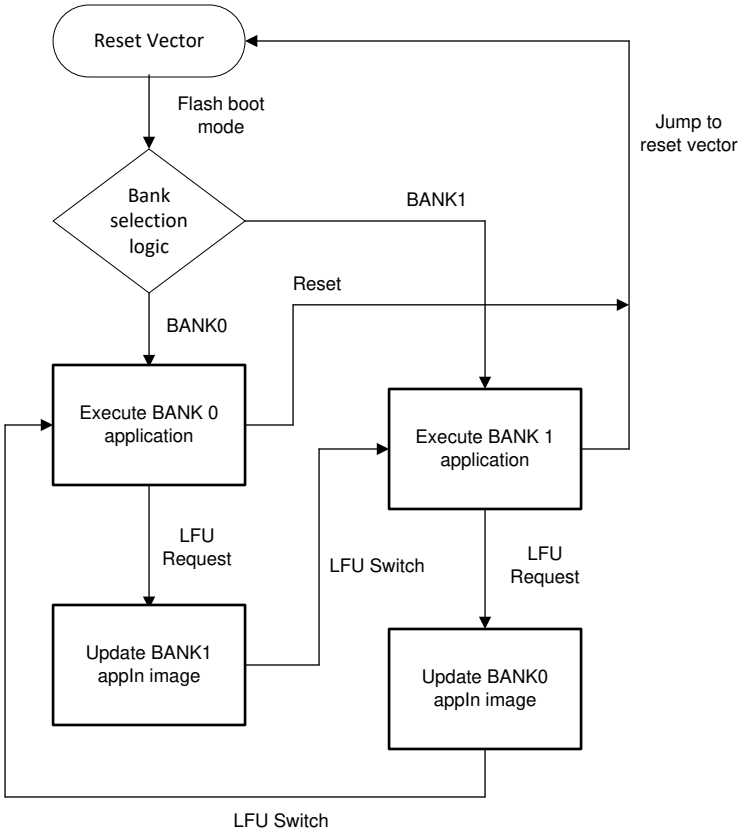


Figure 2-3. LFU Software Flowchart

3 Hardware, Software, Testing Requirements, and Test Results

3.1 Hardware Requirements

The user needs the following components:

1. F28003x Launchpad (LAUNCHXL-F280039C)

   Or F28004x Launchpad (LAUNCHXL-F280049C)
2. Booster pack (BOOSTXL-BUCKCONV)
3. A micro-USB to USB cable to connect the Launchpad to a Computer
4. A 9 V, 2 A DC bench supply
5. Two Banana to bare-wire cables
6. An Oscilloscope or similar (for example, a Saleae Logic Analyzer)
7. A multi-meter

3.2 Software Requirements

The user needs the following software:

1. Code Composer Studio™ software v10.1.0 (CCS) or later, running TI Compiler C2000 v21.6.0.LTS or later.

   The recommended compiler version is v22.6.3.LTS or later, which contains many LFU compiler bug fixes. A new LFU feature that simplifies LFU ease of use, along with the aforementioned LFU compiler bug fixes, is available in v25.11.0.LTS or later.
2. C2000WARE-DIGITALPOWER-SDK v4.01.00.00 or later, which includes software for this design. The software is available at DigitalPower Software Development Kit (SDK) for C2000 MCUs.

3.2.1 Software Package Contents

Table 3-1 lists the key target executable files that will be needed to run the LFU examples. The locations of the projects, the **specific project build configurations** that need to be used to build the output executables, and the locations where the output executables need to be placed are also mentioned.

Table 3-1. Software Package Contents of F28004x Example

FILE/FOLDER NAME	CONTROL LOOP RUNS ON?	PROJECT BUILD CONFIGURATION	DESCRIPTION
flashapi_ex2_sci_kernel.out	NA	BANK0_LDFU_ROM	Output file after building the **flashapi_ex2_sci_kernel** project, which is the Custom Bootloader (the project will be located at <C2000Ware_DigitalPower_SDK_path>\solutions\tidm_02011\f28004x\examples\flash\CCS)
flashapi_ex2_sci_kernel.out	.NA	BANK1_LDFU_ROM	Same as above
buck_F28004x_lfuBANK0FLASH.txt	CPU	BANK0_FLASH	Output file post conversion to .txt of the **buck_F28004x_lfu** project, which is the Application (the project will be located at <C2000Ware_DigitalPower_SDK_path>\solutions\tidm_02011\f28004x\ccs) Project is built with compiler pre-defined symbol BUCK_CONTROL_RUNNING_ON_CPU Copy the generated .txt file to <C2000Ware_DigitalPower_SDK_path>\c2000ware\utilities\flash_programmers\serial_flash_programmer
buck_F28004x_lfuBANK1FLASH.txt	CPU	BANK1_FLASH	Same as above

Table 3-1. Software Package Contents of F28004x Example (continued)

FILE/FOLDER NAME	CONTROL LOOP RUNS ON?	PROJECT BUILD CONFIGURATION	DESCRIPTION
buck_F28004x_lfu_controlloopBANK0FLASH.txt	CPU	BANK0_FLASH	Output file post conversion to .txt of the **buck_F28004x_lfu_controlloop** project, which is the Application (the project will be located at <C2000Ware_DigitalPower_SDK_path>\solutions\tidm_02011\f28004x\ccs) Project is built with compiler pre-defined symbol BUCK_CONTROL_RUNNING_ON_CPU Copy the generated .txt file to <C2000Ware_DigitalPower_SDK_path>\c2000ware\utilities\flash_programmers\serial_flash_programmer
buck_F28004x_lfu_controlloopBANK1FLASH.txt	CPU	BANK1_FLASH	Same as above
buck_F28004x_lfuBANK0FLASH_cla.txt	CLA	BANK0_FLASH	Output file post conversion to .txt of the **buck_F28004x_lfu** project, which is the Application (the project will be located at <C2000Ware_DigitalPower_SDK_path>\solutions\tidm_02011\f28004x\ccs) Project is built with compiler pre-defined symbol BUCK_CONTROL_RUNNING_ON_CLA. Resulting .txt is renamed to include an "_cla" Copy the generated .txt file to <C2000Ware_DigitalPower_SDK_path>\c2000ware\utilities\flash_programmers\serial_flash_programmer
buck_F28004x_lfuBANK1FLASH_cla.txt	CLA	BANK1_FLASH	Same as above
buck_F28004x_lfu_controlloopBANK0FLASH_cla.txt	CLA	BANK0_FLASH	Output file post conversion to .txt of the **buck_F28004x_lfu_controlloop** project, which is the Application (the project will be located at <C2000Ware_DigitalPower_SDK_path>\solutions\tidm_02011\f28004x\ccs) Project is built with compiler pre-defined symbol BUCK_CONTROL_RUNNING_ON_CLA. Resulting .txt is renamed to include an "_cla" Copy the generated .txt file to <C2000Ware_DigitalPower_SDK_path>\c2000ware\utilities\flash_programmers\serial_flash_programmer
buck_F28004x_lfu_controlloopBANK1FLASH_cla.txt	CLA	BANK1_FLASH	Same as above
serial_flash_programmer_appln.exe	.exe	-	This is the host side serial flash programmer executable for loading the Application to Flash on the target device It is present at <C2000Ware_DigitalPower_SDK_path>\c2000ware\utilities\flash_programmers\serial_flash_programmer

Table 3-2. Software Package Contents of F28003x Example

FILE/FOLDER NAME	CONTROL LOOP RUNS ON?	PROJECT BUILD CONFIGURATION	DESCRIPTION
flash_kernel_ex3_sci_flash_kernel.out	NA	BANK0_LDFU	Output file after building the **flash_kernel_ex3_sci_flash_kernel** project, which is the Custom Bootloader (the project will be located at <C2000Ware_DigitalPower_SDK_path>\solutions\tidm_02011\f28003x\examples\flash\CCS)
flash_kernel_ex3_sci_flash_kernel.out	.NA	BANK1_LDFU	Same as above
buck_F28003x_lfuBANK0FLASH.txt	CPU	BANK0_FLASH	Output file post conversion to .txt of the **buck_F28003x_lfu** project, which is the Application (the project will be located at <C2000Ware_DigitalPower_SDK_path>\solutions\tidm_02011\f28003x\ccs) Project is built with compiler pre-defined symbol BUCK_CONTROL_RUNNING_ON_CPU Copy the generated .txt file to <C2000Ware_DigitalPower_SDK_path>\c2000ware\utilities\flash_programmers\serial_flash_programmer
buck_F28003x_lfuBANK1FLASH.txt	CPU	BANK1_FLASH	Same as above
buck_F28003x_lfuBANK0FLASH_cla.txt	CLA	BANK0_FLASH	Output file post conversion to .txt of the **buck_F28003x_lfu** project, which is the Application (the project will be located at <C2000Ware_DigitalPower_SDK_path>\solutions\tidm_02011\f28003x\ccs) Project is built with compiler pre-defined symbol BUCK_CONTROL_RUNNING_ON_CLA. Resulting .txt is renamed to include an “_cla” Copy the generated .txt file to <C2000Ware_DigitalPower_SDK_path>\c2000ware\utilities\flash_programmers\serial_flash_programmer
buck_F28003x_lfuBANK1FLASH_cla.txt	CLA	BANK1_FLASH	Same as above
serial_flash_programmer_appln.exe	.exe	-	This is the host side serial flash programmer executable for loading the Application to Flash on the target device It is present at<C2000Ware_DigitalPower_SDK_path>\c2000ware\utilities\flash_programmers\serial_flash_programmer

Note

The buck_F28003x_lfu project also contains a BANK0_FLASH_BANK10COPY build configuration, which is an alternate configuration where the application is always built to be Loaded to Bank1 and Run from Bank0. In this configuration, BANK1_TO_0COPY is a pre-defined symbol which allows the modified functionality to be implemented.

Similarly, the flash_kernel_ex3_sci_flash_kernel project also contains a BANK0_LDFU_BANK1TO0COPY build configuration, which is an alternate configuration that supports the use-case above i.e. where the application is always built to be Loaded to Bank1 and Run from Bank0. In this configuration, BANK1_TO_0COPY is a pre-defined symbol which allows the modified functionality to be implemented.

This allows the developer to build an image for LFU without having to know which Bank the image is going to reside in. The downside to this is the Bank1 to Bank0 copy, which needs to occur before activating the new image. This copy is done by the Flash kernel. This takes up time, during which the application cannot be running.

The build configurations have been updated in later releases of TIDM-02011 (C2000WARE-DIGITAL-POWER-SDK v5.06.00 onwards). The new configurations for F28003x are VER0_BANK0_FLASH, VER1_BANK1_FLASH, VER2_BANK0_FLASH, and VER3_BANK1_FLASH. Each subsequent version is built with the previous version as the LFU reference image. This allows a user to emulate LFU in the field by switching from VER0 to VER1 to VER2 to VER3.

Additionally, later releases of TIDM-02011 also contain a F28003x projectspec (buck_lfu_kernel.projectspec) that combines the application and the custom bootloader into a single project. With LFU in this project, the entire Flash bank (bootloader + application) is updated during LFU, not just the application.

3.2.2 Software Structure

Figure 3-1 shows the software directory structure for the LFU solution for F28003x. The same structure exists for F28004x. The solutions folder contains a tidm_02011 folder, inside which is the LFU implementation for the TIDM-DC-DC-BUCK solution. The following folders are present:

- /lfu contains source and header files specific to LFU
- /drivers contains HAL (hardware abstraction layer) source and header files
- /ccs contains CCS projectspecs
- /cmd contains linker command files
- /buck contains buck_main.c, buck_clatasks.cla, main.syscfg, and other header files
- /examples contains the custom bootloader (SCI Flash kernel)

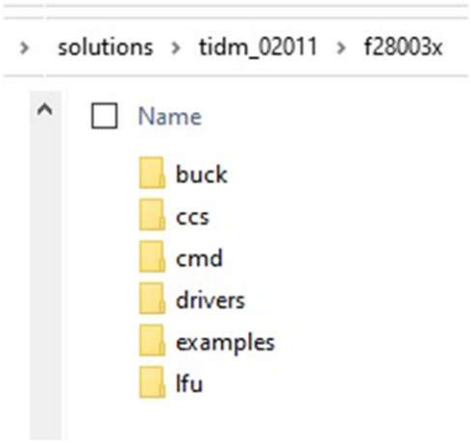


Figure 3-1. LFU Solution Software Directory Structure for F28003x

3.3 Introduction to the TIDM-DC-DC-BUCK

The TIDM-DC-DC-BUCK solution illustrates how to implement Digital Power control on C2000 MCUs. It is used in TIDM-02011 to illustrate LFU. Notable features of this solution are mentioned below:

- Runs on a TMS320F28004x dual-bank Flash MCU clocked at 100MHz

  This reference design TIDM-02011 also adds support for the above TIDM-DC-DC-BUCK example to run on a TMS320F28003x MCU (containing up to 3 Flash banks) clocked at 120MHz.
- The control loop ISR runs at 200kHz. Refer to BUCK_DRV_EPWM_SWITCHING_FREQUENCY in buck_settings.h
- A series of background tasks run when the Control loop ISR is not executing:
  - It rotates between A, B, and C type tasks
  - A type tasks run at 1kHz rate (rotates between A1, A2, A3 functions)
  - B type tasks run at 100Hz rate(rotates between B1, B2, B3 functions)
  - C type tasks run at 10Hz rate (rotates between C2, C2, C3 functions)
- The ISR as well as select Background task functions run from RAM.

3.4 Test Setup

The remainder of the document demonstrates test results assuming the Launchpad is used. If the user wants to use the ControlCard instead, CONTROLCARD needs to be a pre-defined symbol in the Application projects before building them.

3.4.1 Loading the Custom Bootloader and Application to Flash using CCS

1. Set the Launchpad in Flash boot mode by moving GPIO24 to OFF (1) and GPIO32 to OFF (1) on boot select switch. Refer to the C2000™Piccolo™F28004x Series LaunchPad™ Development Kit user's guide or *C2000™Piccolo™ F28003x Series LaunchPad™ Development Kit* for details.
2. Apply power to the board by connecting the micro USB cable to the computer and the Launchpad.
3. Use CCS to download the custom bootloader to Flash bank 0.
   a. With F28004x, program the Bank0_LDFU_ROM build configuration .out of the custom bootloader (located at <C2000Ware_DigitalPower_SDK_path>\c2000ware\driverlib\f28004x\examples\flash\CCS) using CCS.
   b. With F28003x, program the Bank0_LDFU build configuration .out of the custom bootloader (located at <C2000Ware_DigitalPower_SDK_path>\solutions\tidm_02011\f28003x\examples\flash\CCS) using CCS.

   For this step, use a target configuration file that erases the entire Flash. Refer to Figure 3-2 for details.

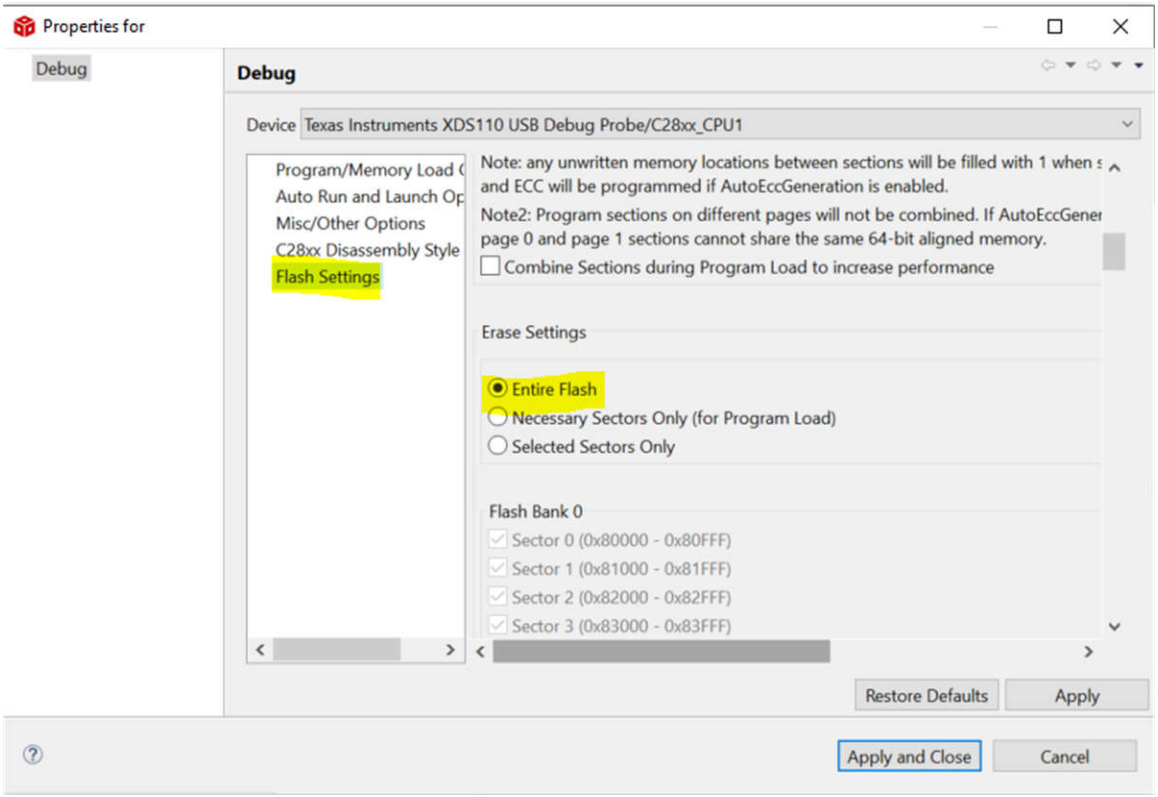


Figure 3-2. Target Configuration File with Erase Settings to Erase Entire Flash

4. Once the custom bootloader is programmed in Flash, click on Run in CCS, and then execute the following commands from a windows command prompt:
   - cd <C2000Ware_DigitalPower_SDK_path>\c2000ware\utilities\flash_programmers\serial_flash_programmer
   - On F28004x: serial_flash_programmer_**appln**.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28004x_lfuBANK1FLASH.txt -b 9600 -p COM11
   - On F28003x: serial_flash_programmer_**appln**.exe -d f28003x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28003x_lfuBANK1FLASH.txt -b 9600 -p COM11
   - In the above command, f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt is needed but not used. This is because the serial_flash_programmer_appln.exe is used, which programs only the application firmware, not the kernel.
   - Also note that in the above command, “COM11” will have to be replaced with the specific COM port associated with your connection. This can be identified through Device Manager – Ports – XDS110 Class Application/User UART
   - Enter “8 – Live DFU” – this will program the Bank1_Flash build configuration of the application firmware to Flash Bank1
   - Enter “0 – Done” when complete
5. At this point, the custom bootloader is programmed on Flash bank 0 and the application is programmed on Flash bank 1.
6. Next the custom bootloader is programmed on Flash bank 1 and the application on Flash bank 0. Use CCS to download the custom bootloader to Flash bank 1. With F28004x, program the Bank1_LDFU_ROM build configuration .out of the custom bootloader (located at <C2000Ware_DigitalPower_SDK_path>\c2000ware\driverlib\f28004x\examples\flash\CCS) using CCS. With F28003x, program the Bank1_LDFU build configuration .out of the custom bootloader (located at <C2000Ware_DigitalPower_SDK_path>\solutions\tidm_02011\f28003x\examples\flash\CCS) using CCS.

For this step, use a target configuration file that erases only the necessary sectors, not the entire Flash. Refer to Figure 3-3 for details.

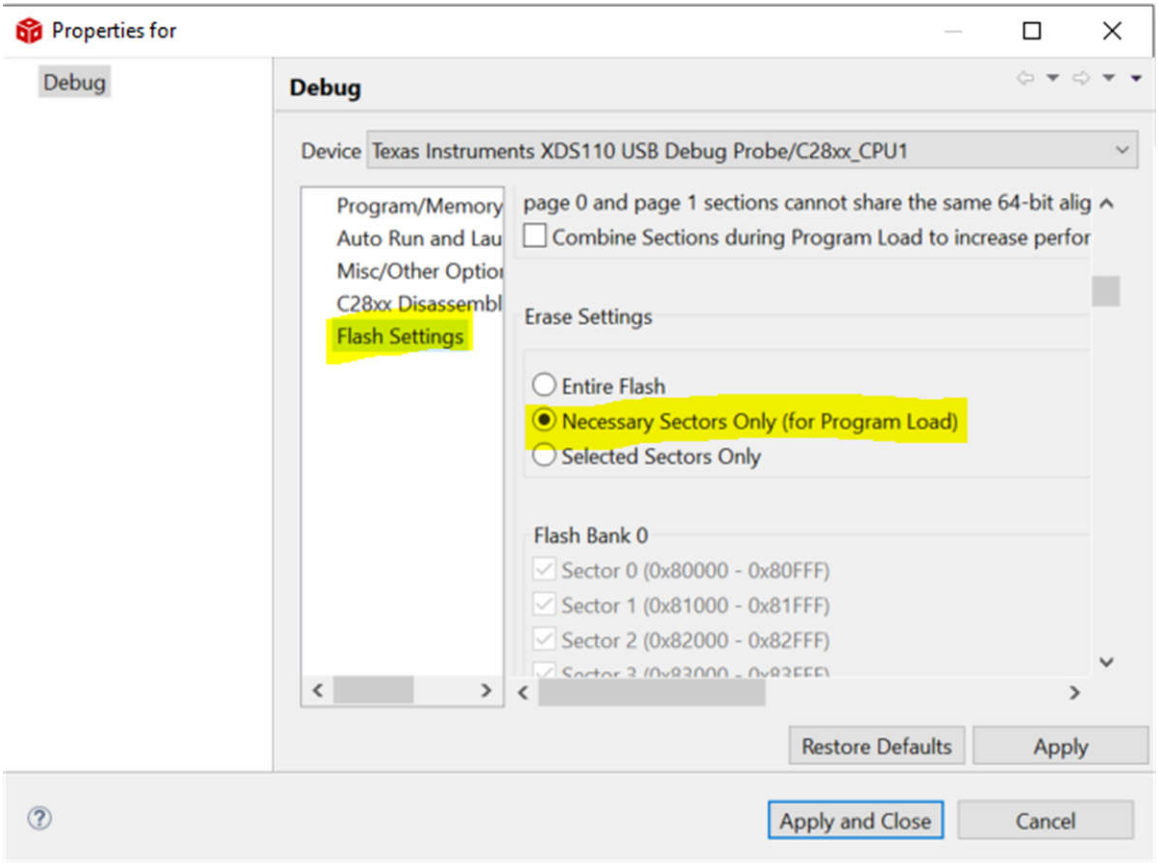


Figure 3-3. Target Configuration File with Erase Settings to Erase Necessary Sectors Only

7. Once the custom bootloader is programmed in Flash, click on Run in CCS, and then execute the following commands from a windows command prompt:
   - cd <C2000Ware_DigitalPower_SDK_path>\c2000ware\utilities\flash_programmers\serial_flash_programmer
   - On F28004x: serial_flash_programmer_**appln**.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28004x_IfuBANK0FLASH.txt -b 9600 -p COM11
   - On F28003x: serial_flash_programmer_**appln**.exe -d f28003x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28003x_IfuBANK0FLASH.txt -b 9600 -p COM11
   - In the above command, f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt is needed but not used. This is because the serial_flash_programmer_appln.exe is used, which programs only the application firmware, not the kernel.
   - Also note that in the above command, “COM11” will have to be replaced with the specific COM port associated with your connection. This can be identified through Device Manager – Ports – XDS110 Class Application/User UART
   - Enter “8 – Live DFU” – this will program the Bank0_Flash build configuration of the application firmware to Flash Bank0
   - Enter “0 – Done” when complete
8. Reset the board. Now both Flash banks have custom bootloaders and Application Images.

3.5 Test Results

3.5.1 Running the LFU Demo with Control Loop Running on the CPU

With both flash banks of the device programmed with the custom bootloader and Application images, the LFU demo is now ready to run in Standalone mode.

1. Switch to Flash boot mode (it should already be in this mode at this point).
2. Connect the Booster pack to the Launchpad as shown in Figure 3-4. The Launchpad is above the Boosterpack. Launchpad headers J5-J7 connect to Boosterpack headers H1-H2. Launchpad headers J6-J8 connect to Boosterpack headers H3-H4. This represents Launchpad Site2 in the project, in main.syscfg (Powerstage Parameters – Hardware – Launchpad Site).

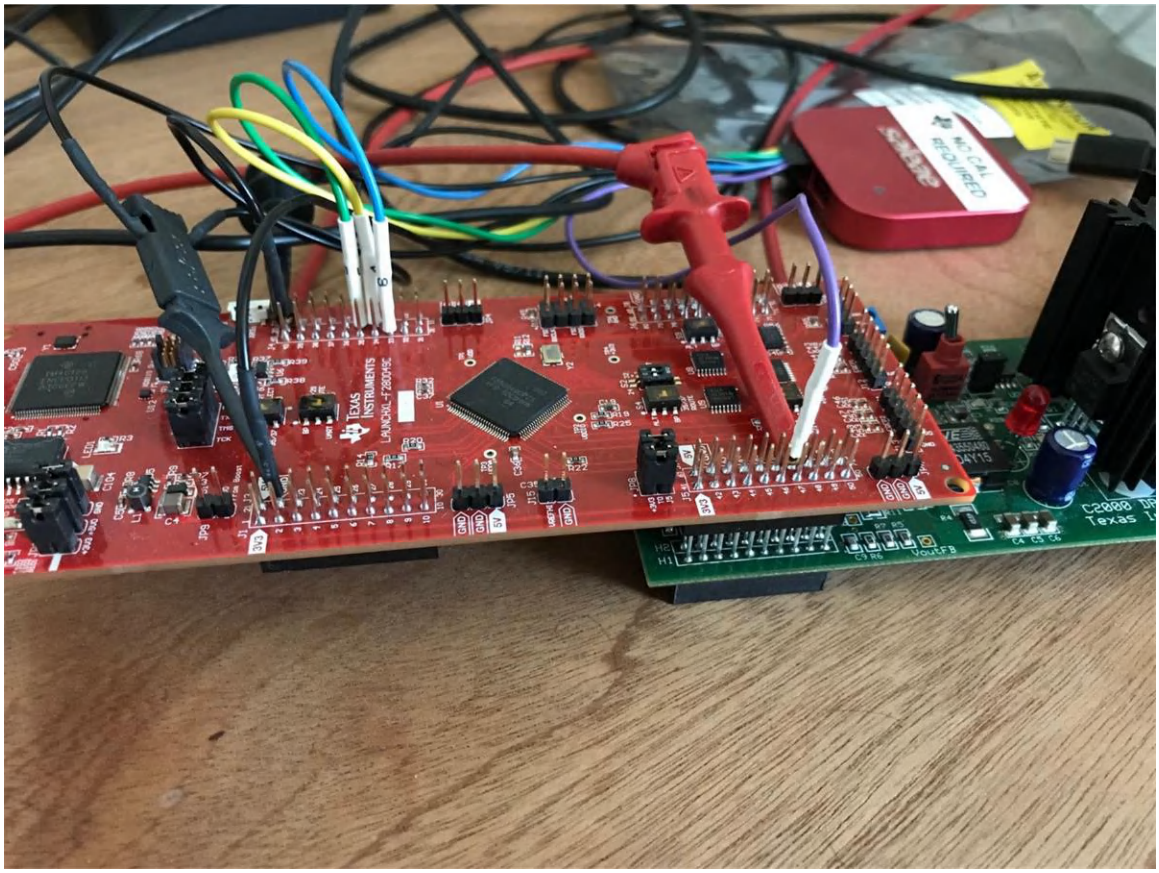


Figure 3-4. Connecting the Booster Pack to the F28004x Launchpad

3. Connect the banana to bare-wire cables from the DC-bench supply to the Booster pack at JP1 with the correct polarity ([JP1 +]Vin and [JP1 GND]GND).
4. Set the DC-bench supply to output 9V. Enable Power.
5. Turn SW1 to the ON position.
6. Connect an oscilloscope (or similar) to sense the output voltage, as well as 2 additional signals – the ISR CPU load, as well as LFU switchover time. Connections can be made according to the description below. Also use a multi-meter to monitor the regulated output voltage.
   - Output voltage – on header J7 of the Launchpad, signal 67. This represents the regulated output voltage.
   - ISR CPU load - on header J2 of the Launchpad, signal 15. This represents the CPU load of the Control loop ISR.
   - LFU Switchover time - on header J2 of the Launchpad, signal 14. This represents the time taken to perform LFU from the old to the new application image.
7. Apply power to the board by connecting the micro USB cable to the computer and the Launchpad. Note that it is important for this step to occur **after** the DC-bench supply is already powering the Booster Pack and SW1 is ON.
8. This should cause the Control loop ISR to start executing. By default, this will run Build2 of TIDM-DC-DC-BUCK. This is closed-loop voltage regulation using VMC (voltage mode control). However, the background tasks will not yet be executing as the software initialization step is still incomplete. This is the SCI Autobaud

lock step. To enable SCI Autobaud lock, execute a command from the Windows command prompt as follows:

- On F28004x: serial_flash_programmer_**appIn**.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28004x_lfuBANK1FLASH.txt -b 9600 -p COM11
- On F28003x: serial_flash_programmer_**appIn**.exe -d f28003x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28003x_lfuBANK1FLASH.txt -b 9600 -p COM11
- after issuing the command, do not select any option just yet
- just this much should suffice for SCI autobaud lock, and the Background tasks will start executing. When this happens, sensing operations will commence and the regulated output voltage will show as 1V on the multi-meter (reflective of the 2V regulated output voltage).
- If there is a Red LED populated on the Booster pack, it will light up.
- Red LED4 (GPIO23) on the Launchpad is controlled by BUCK_HAL_toggleRunLed() in B1() in buck_main.c. Since this is a background task function, it will start toggling. The toggling frequency is set to a smaller value when the application is running from Bank0 than Bank1.

9. The programming steps above programmed the TIDM-DC-DC-BUCK application on Flash Bank1 first, then on Flash Bank0. But the firmware versions on both would be 0xFFFE. When they are equal, the bank selection logic in the custom bootloader will deem the lower number bank i.e. Flash Bank0 as the most recent application version, and will execute this.
   - Green LED5 (GPIO34) on the Launchpad is ON when code is running from Bank1, and is OFF when code is running from Bank0. Since code is now running from Bank0, this LED will be OFF.
10. In Step 8, a command was issued to enable SCI Autobaud lock, and the command prompt was waiting for a user input
    - Enter “8 – Live DFU” – this will program the Bank1_Flash build configuration of the TIDM-DC-DC-BUCK Application to Flash Bank1
    - Enter “0 – Done” when complete
    - While the new image is downloading to Flash, LED4 on the Launchpad will not toggle, because Background tasks are stopped during the image download.
    - After programming the Bank1_Flash image to Flash Bank1, this image will automatically start executing. The user will now notice the following:
      – Green LED5 on the Launchpad is ON . This is because code is now running from Bank1.
      – The Red LED on the Booster pack is still ON. But Red LED4 on the Launchpad is not toggling. This is because the Background tasks are not yet enabled. Similar to above, a command can be issued to enable SCI Autobaud lock. The following command may be used
        - On F28004x: serial_flash_programmer_**appIn**.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28004x_lfuBANK0FLASH.txt -b 9600 -p COM11
        - On F28003x: serial_flash_programmer_**appIn**.exe -d f28003x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28003x_lfuBANK0FLASH.txt -b 9600 -p COM11
        - after issuing the command, do not select any option just yet
      – Output voltage continues to stay at 1V throughout this process. **This is because no device reset was issued after LFU, and the switchover from the old to the new application firmware occurred during the idle time between interrupts.**
11. For the next LFU switchover, the user can set the oscilloscope to trigger on the “LFU switchover time” signal. This will allow the user to visually inspect when the switchover occurs, how long it takes, etc.
    - In step 8, a command was issued to enable SCI Autobaud lock, and the command prompt was waiting for a user input
    - Enter “8 – Live DFU” – this will program the Bank0_Flash build configuration of the TIDM-DC-DC-BUCK Application to Flash Bank0
    - Enter “0 – Done” when complete
    - While the new image is downloading to Flash, LED4 on the Launchpad will not toggle, because Background tasks are stopped during the image download.
    - After programming the Bank0_Flash image to Flash Bank0, this image will automatically start executing. The user will now notice the following:

- Green LED5 on the Launchpad is OFF. This is because code is now running from Bank0
- The Red LED on the Booster pack is still ON. But Red LED4 on the Launchpad is not toggling. This is because the Background tasks are not yet enabled. Similar to above, a command can be issued to enable SCI Autobaud lock. The following command may be used
  - On F28004x: serial_flash_programmer_**appIn**.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28004x_lfubank1FLASH.txt -b 9600 -p COM11
  - On F28003x: serial_flash_programmer_**appIn**.exe -d f28003x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28003x_lfubank1FLASH.txt -b 9600 -p COM11
  - after issuing the command, do not select any option just yet
- Output voltage continues to stay at 1V throughout this process. **This is because no device reset was issued after LFU, and the switchover from the old to the new application firmware occurred during the idle time between interrupts**.
- Refer to Figure 3-5 for details and to visually confirm the above statements. The signals shown are LFU switchover, CPU ISR load, and the regulated output voltage.

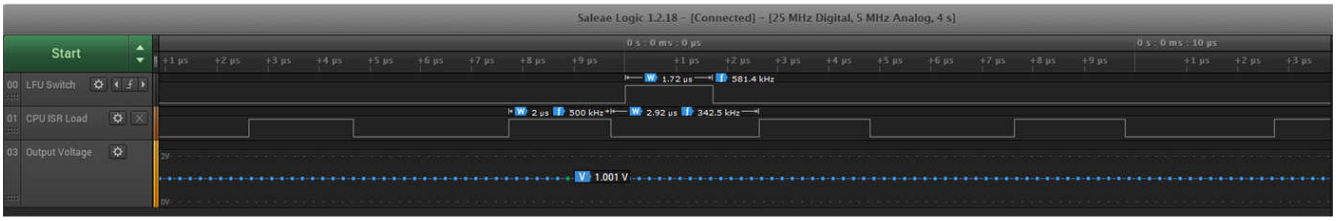


Figure 3-5. LFU Switchover Timing – Control Loop on CPU

12. Repeat the above steps as needed. When Bank0 is active, issue an LFU command to program and switchover to Bank1. When Bank1 is active, issue an LFU command to program and switchover to Bank0.

3.5.2 Running the LFU Demo with Control Loop Running on the CLA

If the user wants to generate the .txt files for the projects built with the control loop running on the CLA, the only change necessary is to change the pre-defined compiler symbol BUCK_CONTROL_RUNNING_ON_CPU to BUCK_CONTROL_RUNNING_ON_CLA. Refer to Figure 3-6 for details. The generated .txt file is renamed with a _cla to distinguish it from the corresponding .txt file running on the CPU.

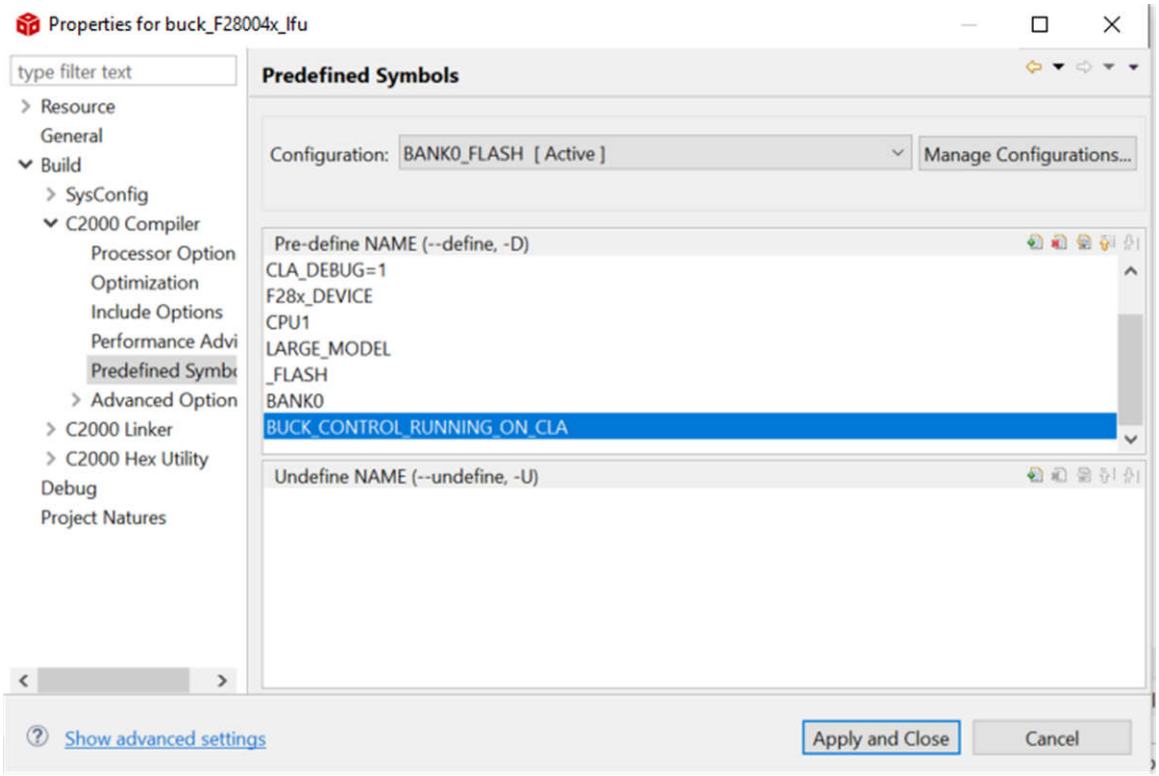


Figure 3-6. Pre-defined Symbol for CLA Build of Project

Once the user has run the LFU demo with the Control loop running on the CPU, running the LFU demo with the Control loop running on the CLA is straightforward, and attention needs to be paid to only a few points:

1. If the device already contains the Application files corresponding to the CPU side control loop, this update can be made using the same LFU commands as in the previous section, except with the updated .txt names corresponding to the CLA build (mentioned in Table 3-1). For example, when updating from BANK0_FLASH to BANK1_FLASH, execute the command:
   - On F28004x: serial_flash_programmer_**appln**.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28004x_lfuBANK1FLASH**_cla**.txt -b 9600 -p COM11
   - On F28003x: serial_flash_programmer_**appln**.exe -d f28003x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel-CPU1-RAM.txt -a buck_F28003x_lfuBANK1FLASH**_cla**.txt -b 9600 -p COM11
2. The CLA setup function occurs in main() on a device reset (not after an LFU switch), so it is important to reset the device after running the LFU, so that this initialization is performed. For example, with control loop running on the CPU, assume that BANK0_FLASH was updated last. This means the firmware on BANK0 is executing. So the user will need to execute the LFU command to update BANK1_FLASH (with the CLA side executable). After the LFU update is complete, a device reset is required. ***The device reset needs to be done only once*** .

   Then the user can perform additional LFU updates with the CLA side firmware executables, without device reset.
3. Figure 3-7 and Figure 3-8 demonstrate LFU switchover with the Control loop running on the CLA. There is a background task that toggles a GPIO every 1ms, and this GPIO is available on header J4 of the Launchpad, signal 33. Note how the LFU switchover in this case can, in general, overlap with ISR execution, because the ISR executes on the CLA whereas LFU occurs on the CPU. This is not an issue in general, but there can be scenarios where this is not acceptable. The signals shown in Figure 3-7 and Figure 3-8 are LFU switchover, CLA ISR load, CLA background task execution, and the regulated output voltage.

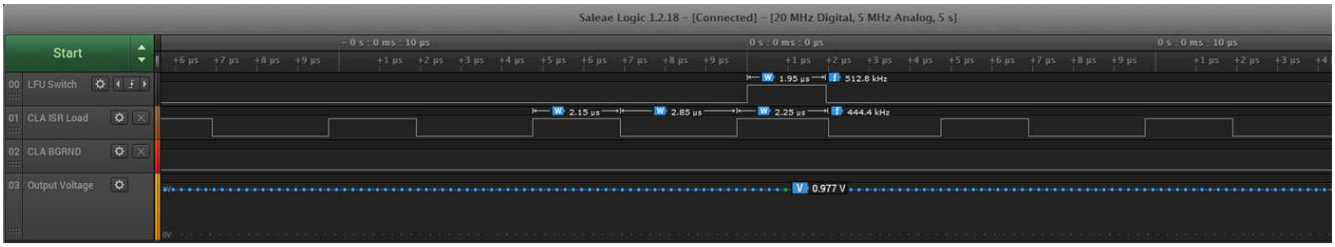


Figure 3-7. LFU Switchover Timing – Control loop on CLA

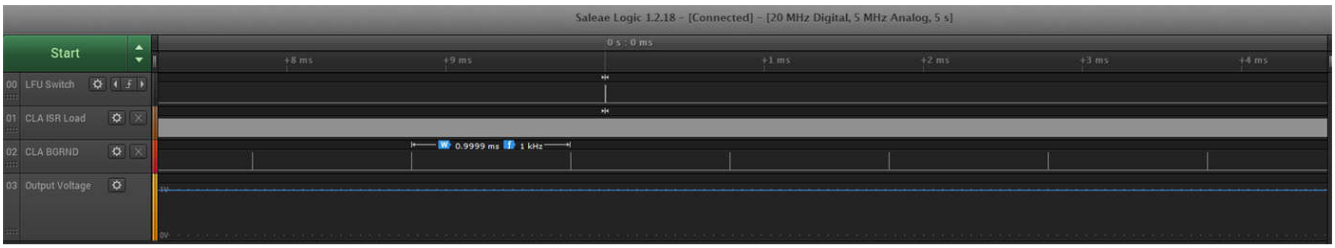


Figure 3-8. CLA Background Task

3.5.3 LFU Flow on the CPU

In the above LFU demo, two application images were used. One that runs on Flash Bank0, and another that runs on Flash Bank1. The BANK0_FLASH build configuration is considered the “old” or “reference” firmware, and the BANK1_FLASH build configuration is considered the “new” firmware. These two applications are otherwise identical. There are no source code differences between them; however the new firmware has 25 new floating point variables that are defined and initialized. These two applications are implemented through two build configurations of the TIDM-DC-DC-BUCK solution project – BANK0_FLASH, and BANK1_FLASH. As the name suggests, BANK0_FLASH executes from Flash Bank0, and BANK1_FLASH from Flash Bank1. These two build configurations share the same source files, but contain different linker command files. Also, at various places in the code, Macros “#ifdef BANK0” and “ifdef BANK1” control execution. They run the same Control loop ISR and Background tasks.

The following represents the high-level LFU flow when the Control loop runs on the CPU.

1. On device reset, execution starts at the default boot to flash entry point, 0x80000, which is where the bank selection logic function is located. This function checks if there is a valid application present in either or both Flash banks, and if one is present, it picks the more recent version and branches to it (either 0x8EFF0 or 0x9EFF0). These are the code_start locations for the respective applications, from where execution enters the C runtime initialization routine (_c_int00), and into main() of the corresponding application. If neither Flash bank contains a valid application, execution waits to auto-baud lock with the host, and for the host to send an image over SCI.
2. User invokes LFU command “8 Live DFU” through Windows command prompt.
3. The target device receives a command ID “0x700” in the SCI Receive Interrupt ISR.
4. In main(), in the background loop, a function BUCK_LFU_runLFU() is called. When the command ID matches “0x700”, the SCI interrupt is disabled and execution branches to the address of the Live DFU (liveDFU()) function in the custom bootloader. If the Application in Bank0 is executing, then the branch is made to the custom bootloader in Bank0 (address 0x81000). If the Application in Bank1 is executing, then the branch is made to the custom bootloader in Bank1 (address 0x91000).
5. liveDFU() in the custom bootloader receives an application image from the host and programs it into Flash. After completing, execution depends on whether the macro LFU_WITH_RESET is defined. If it is, the Watchdog is configured to generate a Reset signal, and then enabled, so a device reset occurs. If the macro is not defined, execution branches to the **LFU entry point of the new application image**. This is 0x8EFF8 for Bank 0 and 0x9EFF8 for Bank 1. This is different from the regular Flash boot entry point.
6. The function c_int_lfu() is located at 0x8eff8 on Bank 0 and 0x9eff8 on Bank 1. This function enables LFU switchover without a device reset. In this function:

a. The compiler's LFU initialization routine (__TI_auto_init_warm()) is invoked. This initializes any variables that have been indicated as needing initialization. So it initializes the 25 new floating point variables defined in the BANK1_Flash build configuration.
b. A flag is set to indicate LFU is in progress. On F28004x, this is done using a software variable lfuSwitch. On F28003x, this is done by setting the LFU.CPU bit of the LFUConfig SysCtl register using LFU_setLFUCPU().
c. main() is called

7. In main(), initialization progresses depending on whether or not LFU is in progress. This is done by accessing the LFU.CPU bit of the LFUConfig SysCtl register on F28003x, or lfuSwitch on F28004x. If the value is 0, initialization progresses as if a device reset occurred.
8. If the value is not 0, then limited initialization is performed. First, init_lfu() is executed. This function copies over code from Flash to RAM, corresponding to Program code that the user has indicated needs to run from RAM. Next, on F28003x, it updates the inactive PIE interrupt vector table using Shadow_Interrupt_Register(). On F28004x, the interrupt vector table swapping hardware feature is not available, so this is not performed. On F28003x, a set of inactive function pointers is also updated. On F28004x, the RAM block swapping hardware feature is not available, so this is not performed.

   init_lfu() should also be used for any new peripheral configuration (that do not impact operation of the old firmware).
9. Next, a variable lfuSwitch_start is set to Lfu_switch_wait_for_isr. Execution waits here until the next Control loop ISR executes, where lfuSwitch_start moves from Lfu_switch_wait_for_isr to Lfu_switch_ready_to_switch. **This helps synchronize the LFU switchover to the end of the Control loop ISR, which allows maximizing the utilization of the idle time between Control loop interrupts**.
10. When execution proceeds, the LFU switchover steps occur. First, global interrupts are disabled. On F28003x, PIE interrupt vector table swapping and RAM block swapping are executed. On F28004x, every used PIE interrupt vector needs to be individually updated here. Likewise, every function pointer needs to be individually updated here. Any existing peripheral re-configuration needs to be done here. Then C28x CPU side stack pointer initialization is performed, and global interrupts are re-enabled. **This represents the end of the LFU switchover**.
11. Global interrupts are disabled for a short duration. If a peripheral interrupt occurs during this time, it would continue to stay latched and interrupt the CPU when global interrupts are re-enabled. This is done to avoid unpredictable behavior in the unlikely event an interrupt occurs and this vector table is accessed while it is being updated.
12. Figure 3-9 shows another LFU use-case where control loop parameters are updated between firmware versions. In practice, this can be done in real-time using the Compensation Designer, but this use-case is included for illustrative purposes. This corresponds to the **buck_F28004x_lfu_controlloop** project. In this project, the BANK0_FLASH build configuration contains coefficients that correspond to a smaller gain, Kdc = 4000. The BANK1_FLASH build configuration contains coefficients that correspond to a larger gain Kdc = 38904 (refer to the function BUCK_initControlLoopGlobals() in buck.h). This leads to poorer transient performance with the BANK0_FLASH build configuration and more optimal transient performance with the BANK1_FLASH build configuration, when Active Load is enabled. If the device already contains the Application files corresponding to the **buck_f28004x_lfu** project (or even the CLA side), this update can be run using the same LFU commands as in the previous section, except with the updated .txt names corresponding to this project as shown in Table 3-1. In the **buck_f28004x_lfu_controlloop** project, Active load is enabled in main() on a device reset (not after an LFU switch), so it is important to reset the device after running the LFU, so that this initialization is performed. ***The device reset needs to be done only once*** .

    Then the user can perform additional LFU updates with the controlloop project executables, without device reset.

Note

At present, the Controlloop example with Active load enabled is created only on F28004x.

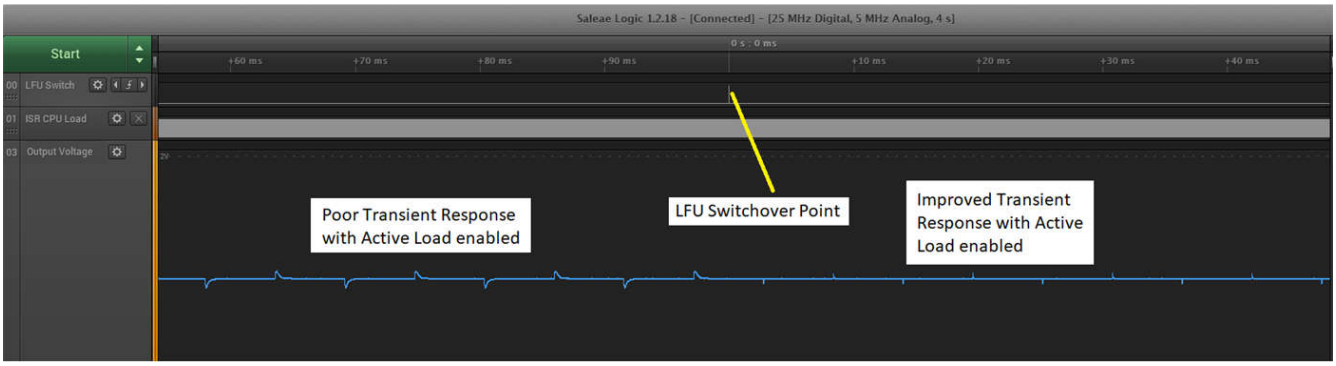


Figure 3-9. LFU Switchover with Transient Performance Improvement – Control Loop on CPU

3.5.4 LFU Flow on the CLA

The following represents the high-level LFU flow when the Control loop runs on the CLA.

1. Flash partitioning and high level LFU software flow both remain the same.
2. User invokes LFU command *8 Live DFU* through Windows command prompt..
3. The target device receives a command ID *0x700* in the SCI Receive Interrupt ISR.
4. In main(), in the background loop, a function BUCK_LFU_runLFU() is called. When the command ID matches *0x700*, the SCI interrupt is disabled and execution branches to the address of the Live DFU function in the custom bootloader. If the Application in Bank0 is executing, then the branch is made to the custom bootloader in Bank0 (address 0x81000). If the Application in Bank1 is executing, then the branch is made to the custom bootloader in Bank1 (address 0x91000).
5. liveDFU() in the custom bootloader receives an application image from the host and programs it into Flash. After completing, execution depends on whether the macro LFU_WITH_RESET is defined. If it is, the Watchdog is configured to generate a Reset signal, and then enabled, so a device reset occurs. If the macro is not defined, execution branches to the **LFU entry point of the new application image**. This is 0x8EFF8 for Bank 0 and 0x9EFF8 for Bank 1. This is different from the regular Flash boot entry point.
6. The function c_int_lfu() is located at 0x8eff8 on Bank 0 and 0x9eff8 on Bank 1. This function enables LFU switchover without a device reset. In this function:
   a. The compiler's LFU initialization routine (__TI_auto_init_warm()) is invoked. This initializes any CPU variables that have been indicated as needing initialization. **CLA variable initialization on LFU is not supported by __TI_auto_init_warm() and needs to be managed in user code.**
   b. A flag is set to indicate LFU is in progress. On F28004x, this is done using a software variable lfuSwitch. On F28003x, this is done by setting the LFU.CPU bit of the LFUConfig SysCtl register using LFU_setLFUCPU().
   c. main() is called
7. In main(), initialization progresses depending on whether or not LFU is in progress. This is done by accessing the LFU.CPU bit of the LFUConfig SysCtl register on F28003x, or lfuSwitch on F28004x. If the value is 0, initialization progresses as if a device reset occurred.
8. If the value is not 0, then limited initialization is performed. First, init_lfu() is executed and performs operations described in the previous section.
   a. In this case, with control running on the CLA, a memcpy() is needed to copy code from Flash to RAM.This corresponds to linker command file sections Cla1Prog and.const_cla, and the control loop ISR. In preparation for this memcpy, the corresponding LSRAM sections are reconfigured to where the CPU is the Master of these sections. After the memcpy, these LSRAM sections are once again configured to be shared between the CPU and the CLA.
   b. The CLA background task is disabled.
9. To determine the correct time to execute LFU switchover when the control loop runs on the CLA and a CLA background task is present, the logic is slightly different. First, BUCK_LFU_getBackgroundTaskControlRegister() is used to read the BGSTART bit of the MCTLBGRND register. If the read back value is 0, it means the CLA BGRND task is neither running nor pending. The application is deemed ready for LFU switchover. If the read back value is 1, then the CPU sets the variable

lfuSwitch_start to Lfu_switch_waiting_to_switch_cla. An end-of-task interrupt from the CLA BGRND task to the CPU causes the execution of an ISR BUCK_LFU_CLA_BGRND_ISR where lfuSwitch_start changes from Lfu_switch_waiting_to_switch_cla to Lfu_switch_ready_to_switch_cla.

Note

LFU switchover waits until the CLA background task has stopped because, unlike the other CLA tasks, the CLA background task can be preempted and does not have to run to completion. If a switchover stops the background task while it is executing, it can leave the task is a "non-clean" state. The goal is for the switchover to occur only after all the tasks in the old firmware have completed execution.

10. Further, inside the BUCK_LFU_CLA_BGRND_ISR, after the variable change above, BUCK_LFU_setupCLALFU() is called within which the following steps occur:
    a. CLA task vectors and Background task vector are mapped to the appropriate tasks.
    b. The ISR corresponding to the end-of-task interrupt from the CLA BGRND task to the CPU is registered.
    c. The CLA background task is enabled.
11. When the CLA task vectors are updated, peripheral interrupts would continue to occur. However, CLA tasks always run to completion. Due to this property, no context violation occurs.
12. Another important point to note here is that the .scratchpad section (corresponding to CLA tasks and functions) needs to be assigned to a separate memory section, different from .bss and .bss_cla sections. During LFU, the CLA ISR could be running while the C28x CPU is initializing the new variables. The ISR could be accessing variables located in .bss and .bss_cla sections, and using the .scratchpad as well. In parallel, the variable initialization would be updating .bss_cla sections. To avoid any .scratchpad corruption, their separation is important.
13. Also note that in this case, it is possible for the CLA ISR time during LFU switchover to be increased slightly. This is due to LSRAM memory access conflicts between the CLA and C28x CPU. The CLA ISR runs using .scratchpad and .bss (both located within the RAMLS7 block) while the C28x CPU initializes new variables of the CLA in .bss_cla (also located within the RAMLS7 block).
14. Figure 3-10 shows another LFU use-case where control loop parameters are updated between firmware versions. In practice, this can be done in real-time using the Compensation Designer, but this use-case is included for illustrative purposes. This corresponds to the **buck_F28004x_lfu_controlloop** project, built with the compiler pre-defined symbol BUCK_CONTROL_RUNNING_ON_CLA. In this project, the BANK0_FLASH build configuration contains coefficients that correspond to a smaller gain, Kdc = 4000. The BANK1_FLASH build configuration contains coefficients that correspond to a larger gain Kdc = 38904 (refer to the function BUCK_initControlLoopGlobals() in buck.h). This leads to poorer transient performance with the BANK0_FLASH build configuration and more optimal transient performance with the BANK1_FLASH build configuration, when Active Load is enabled. If the device already contains the Application files corresponding to the **buck_f28004x_lfu** project (or even the CLA side), this update can be run using the same LFU commands as in the previous section, except with the updated .txt names corresponding to this project as shown in Table 3-1. In the **buck_f28004x_lfu_controlloop** project, Active load is enabled in main() on a device reset (not after an LFU switch), so it is important to reset the device after running the LFU, so that this initialization is performed. For example, with control loop running on the CPU, assume that BANK1_FLASH was updated last. This means the firmware on BANK1 is executing. So the user will need to execute the LFU command to update BANK0_FLASH (with the CLA side controlloop project executable), **not BANK1_FLASH**. After the LFU update is complete, a device reset is required. ***The device reset needs to be done only once*** .

    Then the user can perform additional LFU updates with the controlloop project executables, without device reset.

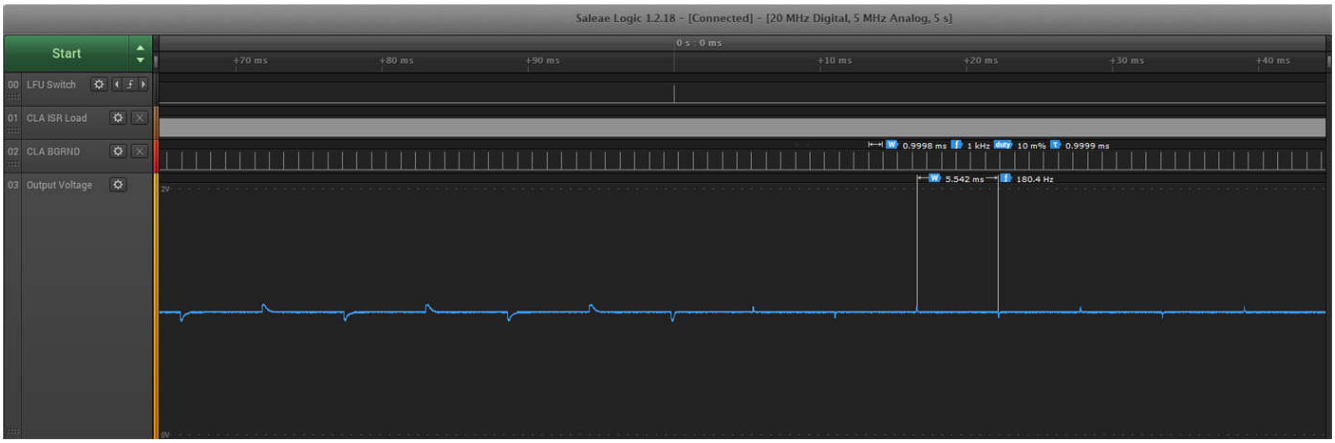


Figure 3-10. LFU Switchover with Transient Performance Improvement – Control Loop on CLA

3.5.5 Limitations and Key Concerns

1. **Absence of A/B swapping:** Since A/B swapping is not supported, the user needs to build firmware for a specific Flash bank. For example, FW_version0 would be targeted to Bank0, FW_version1 to Bank1, FW_version2 to Bank0, FW_version3 to Bank1, and so on.
2. **Use of 3rd Flash bank:** LFU implementation on F28003x is limited to 2 Flash banks in the design. However, the device can contain up to 3 Flash banks, and the LFU implementation can be extended to cover 3 banks. One potential use of a third Flash bank is to store the custom bootloader, with the application firmware residing in the first and second Flash banks. This removes the need for the custom bootloader to be placed in the first and second Flash banks.
3. **Ping-pong approach to code in RAM:** Any old firmware running from RAM could be running during the LFU process and therefore has to be maintained and untouched by the new firmware. The init_lfu() function which copies code from Flash to RAM of code in the new firmware that needs to run from RAM, should not corrupt the RAM space the old firmware was using. In this sense, a ping-pong like scheme would need to be designed by the user, with a portion of the RAM memory allocated for the LFU image.
4. **Interupt configuration:** Interrupt mapping of existing interrupts to the shadow PIE vector table is done in init_lfu() using Shadow_Interrupt_register(). The same process can be followed for any new interrupts that need to be set up. However, they should be enabled (using Interrupt_enable()) only during the LFU switchover. Similarly, any interrupts in the old firmware that are removed will need to be disabled (using Interrupt_disable()) only during the LFU switchover.
5. **Stack initialization:** this is done in the LFU switchover in this example so that the SP is reset just prior to executing ISRs of the new firmware. However, main() of the new firmware has already begun executing. If there are local variables in main(), then resetting SP during the LFU switchover can cause a problem. Those local variable values can potentially be lost. An alternate option is to reset SP in the LFU entry point function c_int_lfu(). This entry point is branched to directly, and while inside this function, no ISRs are running, therefore it is safest to reset SP here.
6. **Variables (symbol) management:** Update variables are placed in the .TI.update section, whereas preserved variables are placed in the .TI.bound section. But an update variable in one firmware version would become a preserved variable in a later firmware version. It would therefore move from the .TI.update section to the .TI.bound section, while remaining at the same address. This is possible since .TI.bound sections are not required to be contiguous.
7. **Management of static libraries:** this example illustrates LFU management of global or static variables in the user's application. It is important to note that if the application links any static libraries which contain global or static variables, LFU management needs to consider those static libraries as well. Library source files need to specify attributes on global or static variables, and the library needs to be compiled by specifying a reference firmware image (just like the firmware it is linked in does). If this is not done, globals within the linked libraries may move location between firmware versions, where the user may incorrectly assume they are preserved.
8. **#pragma DATA_SECTION:** this pragma is used to put data in specific sections other than .bss. Another common use case is for peripheral memory mapped registers that have specific locations, and are marked

NOINIT so as to avoid erroneous zero initialization on startup. The compiler does not perform LFU initialization of sections defined as DATA_SECTIONS.

9. **Fapi_setActiveFlashBank():** application code does not need to execute Fapi_setActiveFlashBank(). This is managed in the Flash bootloader. Additionally, it is important to disable Flash prefetch before executing this function. Flash prefetch can be enabled after. Otherwise, it can cause an ITRAP. Details are present in the F28003x device errata.
10. **Robustness:** in the current example, Flash programming failures or communication issues are not handled as they should be. The user's LFU code should handle these. If a Flash Program/Verify occur occurs, no specific action is taken. In ldfuCopyData(), the section where the START field is programmed checks for a Flash FSM error and calls fmstatfail(). However, in the loop that programs the entire image, this check is not present. Instead a variable 'fail' is incremented.
11. **Bank selection logic:** this is present only on Flash Bank0.
12. **Background tasks:** Once LFU command processing begins, the background tasks of TIDM-DC-DC-BUCK stop running. If users want to implement LFU and want the background loop or portions of it to continue running during LFU command processing, they may want to consider moving those portions into an ISR (for example, a CPUTimerISR).
13. **DCSM Security:** On F28003x, after LFU, reading DCSM space locks the device for that zone (no need of reset after DCSM configuration to lock the device).
14. **Object output type:** the compiler supports LFU only when the object output type for the application firmware is EABI.
15. The Control loop ISR is specified with a “#pragma INTERRUPT(ISR_name, HPI).” The HPI refers to High Priority Interrupt, which uses a fast context save and cannot be nested. The SCI Receive interrupt ISR is not specified as HPI. So it defaults to LPI or Low Priority Interrupt, which can be nested. Furthermore, the Control loop ISR is triggered by an ADCB1 interrupt, which belongs to Interrupt group 1 on F28004x/ F28003x, higher in priority than the SCIA_RX interrupt, which belongs to Interrupt group 9.
16. The default ISR assigned to PIE vectors that are not explicitly assigned remains unchanged.
17. SFRA is disabled during LFU – this is because SFRA and the LFU Host share the same SCI peripheral. With the current Hardware configuration, it is not possible to support both, since the Launchpad only supports one SCI channel from the host to the device.
18. Since Flash bank swapping is not supported, a specific Firmware version has to be mapped to a specific Flash Bank.
19. This comment is specific to F28004x - C2000Ware contains Flash API libraries built for COFF and for EABI. The EABI-based library is a Flash API library that runs from ROM. This is included in the application project. For consistency, ROM build configurations are used with the custom bootloader (flashapi_ex2_sci_kernel) project as well.
20. **LFU switchover timing:** If the user's application contains multiple ISRs, including Nested ISRs, LFU switchover will need to occur in idle time when none of the ISRs are running. The user may choose, based on their specific application use-case, the optimal idle time period if there are idle-time periods of varying durations. The entire LFU process, including the final LFU switchover, occurs in the background, for example, during idle-time.
21. **C28x and CLA LFU switchover:** they occur asynchronously, based on the assumption that code running on the C28x and code running on the CLA are independent enough to allow this asynchronous switchover. If they are dependent, and a synchronous switchover is required, then the process of LFU switchover needs to be modified. First, the CLA LFU switchover time needs to be identified. This corresponds to the "CLA background task stopped" interrupt from the CLA to the C28x CPU. Now the CLA is ready for switchover, but should not switchover. The C28x LFU switchover time now needs to be identified. Once this is identified, both the C28x and CLA are ready for switchover, and can switchover simultaneously.

3.5.6 Preparing Firmware for LFU

To perform LFU while making substantial changes between the old and new application images, the user needs to be aware of the following:

1. LFU Compiler support helps maintain state of common global variables (preserving their address in RAM, and avoiding their initialization during LFU switchover).

But the source or header files containing their definition/declaration should have the same name between the new firmware and the old (reference) firmware.

2. __TI_auto_init_warm() executes in tandem with the old application's ISRs, therefore it does not matter how long __TI_auto_init_warm() takes. **This means there is no limitation on the number of variables that need to be initialized**.
3. LFU Switchover timing – this is important when the control loop ISRs run on the C28x CPU. On the F28003x, LFU hardware features on the device like PIE vector swapping and RAM block swapping allow significant flexibility in the number of interrupt vectors and function pointers that need to be updated on LFU. Irrespective of the number of vectors or function pointers, a single cycle swap implements is all that is needed. However, on the F28004x, these hardware features are not present, so each PIE vector and function pointer needs to be individually updated, which proportionately increases LFU switchover time. If this exceeds idle time, then interrupt execution is affected, which is not acceptable.
4. LFU Switchover timing - in general, this is not an issue when the control loop ISRs run on the CLA. Disabling global interrupts affects only the C28x CPU, not the CLA. CLA tasks (other than the background task) are not disabled and re-enabled during LFU.
5. Another important aspect to consider is RAM memory overlaps between the custom bootloader (SCI Flash Kernel) and the Application:
   a. In general, avoid RAM section overlaps between SCI Flash Kernel and the Application. If this is not possible, verify using the generated .map files for the SCI Flash Kernel and the Application that there are no RAM memory overlaps, as this will break functionality.
   b. In the case of LFU with the CLA, some LSRAM sections are designated in the Application as Program and some as Data. Ensure that this does not conflict with the SCI Flash Kernel. In other words, do not place Application Program in sections that the SCI Flash kernel is using for data, and vice versa.

3.5.7 LFU Compiler Support

This section describes the steps involved in utilizing compiler support for LFU.

1. The Compiler version required for LFU support is 21.6.0.LTS or later.
2. **Assuming the BANK0_FLASH build configuration is the old firmware, the path to its output executable needs to be provided as a reference image to the BANK1_FLASH build configuration**. This will allow the compiler to identify common variables and their locations, and also identify new variables. This is done as follows (for F28004x) in the BANK1_FLASH build configuration projectspec:
   --lfu_reference_elf=${CWD}\..\BANK0_FLASH\buck_F28004x_lfu.out

   Likewise, for F28003x, --lfu_reference_elf=${CWD}\..\BANK0_FLASH\buck_F28003x_lfu.out
3. The compiler defines 2 new attributes for variables, called *preserve* and *update*. Preserve is used to maintain the addresses of common variables between firmware versions. Update is used to indicate new variables that the compiler can assign addresses without constraints and also initialize during the LFU initialization routine __TI_auto_init_warm(). Examples for how these attributes can be used are listed below:
   float32_t __attribute__((preserve)) BUCK_update_test_variable1_cpu;
   float32_t __attribute__((update)) BUCK_update_test_variable2_cpu;
4. If the user builds the BANK1_FLASH configuration as above, with the BANK0_FLASH image as the reference image, then the generated .map file will contain .TI.bound sections corresponding to the preserve variables. Additionally, if the user specifies variables with the update attribute, the .map file will contain a single .TI.update section where all the update variables are collected into. **They will not be placed in the .bss or .data sections. The user needs to define and allocate a .TI.update section in the linker command file. For the CLA LFU use-case, the recommendation is that the user handle it manually in their LFU code. CLA variable initialization on LFU is not supported by __TI_auto_init_warm() and needs to be managed in user code. The compiler will not handle warm initialization of CLA update variables.**
5. To make things easier for the application developer, different LFU modes are available. The default mode is called *preserve* (**not to be confused with the corresponding variable attribute described above**), which can be explicitly specified as follows in the BANK1_FLASH build configuration projectspec:
   --lfu_default=preserve

   This mode has the following properties:

   a. If a reference (*old*) image is provided, then common variables don't need to be specified as *preserve*. This will be the default attribute for common variables, and the RTS library will **not** initialize them in the LFU initialization routine. This helps maintain state.
   b. Any new variables that do not have any attributes specified will be assigned addresses, but they also will **not** be initialized in the warm-start LFU routine. **If the user wants the LFU initialization routine to initialize the new variables, they need to be declared with the *update* attribute.**

6. The complete list of LFU modes supported by the compiler in this release are called "none" and "preserve". They have the following properties:
   a. none: Do not preserve any global and static variable addresses by default or initialize any variables during warm start by default.
      i. If preserve attribute is explicitly specified, then preserve the address of the variable.
      ii. If update attribute is explicitly specified, then initialize the value of the variable during warm start. Address can move in memory.
   b. preserve: Preserve all global and static variables addresses found in the reference ELF unless the update attribute is specified for a variable.
      i. No need to specify preserve attribute for common variables. If preserve attribute is explicitly specified for a variable in reference ELF, it has the same behavior as not having been specified.
      ii. If update attribute is explicitly specified, then initialize the value of the variable during warm start. Otherwise, do not initialize during warm start. In both cases, address can move in memory.

Compiler version 25.11.0.LTS and later versions support a new mode called "all". This will be the default mode, and will always preserve (i.e. Same address and no LFU initialization) global/static variables that are common between the old and new firmware, and will always update (i.e. Goes into .TI.update and LFU initialization) global/static variables that are present only in the new firmware. This means it is not necessary for the user to explicitly add an attribute to global/static variables unless they want a different behavior e.g. They want a variable that's common between old and new firmware to get LFU initialization.

---

7. The RTS library provides an LFU initialization routine (__TI_auto_init_warm()). It initializes any new variables per the rules described above.
   a. The routine performs initialization of C28x CPU side global and static variables. This includes zero initialization (default) and non-zero initialization (if a non-zero value is specified).
   b. The routine must not be used for CLA side global and static variables.

For additional information, refer to the LFU section of the TMS320C28x Optimizing C/C++ Compiler User's Guide.

3.5.8 Robustness

The sequence of Flash programming events related to LFU is:

- In liveDFU(), the custom bootloader writes a START field to a specific Flash location in the Flash bank being programmed
- Then the host transfers data to the device block by block (multiple bytes), which is stored in a buffer, checksum is returned to the host
- Then the custom bootloader erases the corresponding Flash sector (if not already erased)
- After the entire application image has been programmed, the custom bootloader writes a KEY field and updates the VERSION field in the Flash bank programmed. This indicates the presence of a valid application image in that Flash bank.

If the LFU process is interrupted due to a Power loss or Communication issue:

- If the interruption did not result in a device reset (e.g. failure due to communication issue), then the custom bootloader will not be able to complete downloading the application image. However, the old ISRs from the application will continue to execute, but not the background tasks. Another LFU command cannot be initiated until a device reset is performed.

- The Flash bank that was being written to would be partially programmed.
- However, since the VERSION is not updated, on the next device reset, the custom bootloader will branch to the old application in Flash and full services are resumed, with the ability to again perform LFU.

3.5.9 LFU Use-Cases

A number of use cases can be envisioned with LFU. They are listed below. A, B, C, D, and so on refer to firmware versions.

1. A→B →C →D →E
   - This is the typical use-case we prepare for
   - A serves as a reference image to build B, B as a reference image for C, and so on
2. A→B →A →B →A
   - Another use-case we may encounter, where for testing purposes, or in the field if you want to revert to the original image if a problem is found with the new image
   - A will serve as a reference image to build B, so when you switch from A to B you can use the Compiler's warm initialization routine. Because A was provided as reference, the compiler knows the variable differences between A and B, and it will place the variables unique to B in a "TI.update" section. This is the only section that will be initialized by the compiler in its __TI_auto_init_warm() routine when LFU switching from A to B
   - When switching from B to A, the situation is different. A was built standalone, so it does not have a ".TI.update" section, the compiler does not know which variables are unique to A (relative to B), so __TI_auto_init_warm() will not do anything
   - **Is this use-case feasible? Yes**, the user can still switch from B back to A. Just that the user cannot leverage the Compiler's __TI_auto_init_warm() to initialize any variables unique to A. The user will need to use Macros to manually initialize these unique variables in main() of A. For example, if A is in Flash Bank0, and B in Flash Bank1, A can have initialization code in main() like this:
     #ifdef BANK0
     [initialize variables unique to A]
     #endif
   - **In fact, the current LFU example is illustrates A-B-A-B switching**. The difference is that in the current example, B has new variables compared to A, but A does not have any unique variables compared to B.
3. Skipped updates – assume field locations don't update firmware versions as they become available, but skip updates. For example:

   Field_location_1: A→B →C →D →E

   Field_location_2: A →C →D →E

   Field_location_3: A→B → D →E

   Is this use-case feasible? No. There are two issues.

   - One, LFU by nature is incremental. So, each image builds on the other. The .TI.update section generated by the compiler is specific to the reference elf used in generating that image. If the user wants to update relative to an older image, then manual effort would be involved in understanding the variable differences between those 2 images, and manual initialization of the unique variables would need to be performed. A bigger issue is state. Suppose B introduced a new variable "var_x" that then became a common variable across all future images. And the user is updating from A to D. Now, D assumes var_x is a common variable because it is present in C, and therefore doesn't initialize it. However, relative to A, var_x is new. So not initializing it can cause problems.
   - Two, with our implementation, the user also needs to be aware of the specific Bank the image is targeted to. So, in this example, A, C, E would be on Bank0, and B, D would be on Bank1. So, it would not be possible to update from A to C as indicated in Field_location_2.
   - If the user builds the new firmware without using the old firmware as reference, there is no guarantee that common global variables will remain at the same addresses, so state cannot be preserved. So the application behavior may be unexpected after the LFU switchover., unless the entire C initialization routine is executed during the switchover. This may be too time consuming and exceed available LFU switchover time.

4 FOTA Example

4.1 Abstract

This section illustrates an LFU example on the F28003x, where the firmware executable is always Loaded to Flash Bank1 and Runs from Flash Bank0. A copy from Flash Bank1 to Flash Bank0 occurs after the firmware executable is programmed to Flash Bank1. The advantage of this approach is that users need to maintain only one linker command file for their project, and do not need to keep track of the Flash Bank on which the firmware will run after the LFU operation.

This example can also be used as a reference for FOTA.

4.2 Introduction

Where remapping of Flash banks is not available, each Flash bank is mapped to a fixed memory address. During LFU, the firmware executable is programmed to the Flash bank that is currently inactive, while the application continues to run from the Flash bank that is currently active. With this approach, the user needs to be aware of which Flash bank their firmware executable is targeted for. Thus, they need to maintain 2 linker command files for their project (if they are using 2 Flash banks). This can be cumbersome, so an alternate solution is proposed and implemented here.

In this approach, the firmware executable is always built to be Loaded to Flash Bank1 and Run from Flash Bank0. This can be done with just one linker command file. Similar to how functions in an application are Loaded to Flash and Run from RAM for performance improvement, a memory copy is needed here as well. This is implemented in the LFU bootloader i.e. Flash kernel. The Flash Bank1 to Bank0 memory copy takes about 1 second to complete.

Flash APIs and the functions that call it need to run from RAM to allow the Bank1 to Bank0 copy to occur, since the Flash bootloader is located in Flash. This RAM can be freed up for use by the application during its C initialization routine. The application's RAM can overlap with the bootloader's RAM in this case.

As mentioned before, this example can be used as **a reference for FOTA**, with a few functional features not implemented:

- Rollback – to support a rollback, a copy needs to be done from Flash Bank0 to Bank2, prior to the Bank1 to Bank0 copy. This can done exactly along the lines of the Bank1 to Bank0 copy illustrated in the Flash kernel.
- Reset – this example does not implement a full device reset after the Bank1 to Bank0 copy. That is how FOTA would work. In this example, once the memory copy is complete, the Flash kernel directly branches to the entry point of the application, where c_int00 is called and then main().

4.3 Hardware Requirements

The following are required to run the example:

1. F28003x ControlCARD and USB-C cable
2. ControlCARD Docking station [R4.1]
3. Logic Analyzer like Saleae Logic 8

4.4 Software Requirements

The software required to run this example are:

1. flash_kernel_ex3_sci_flash_kernel project
2. buck_F28003x_lfu project

4.5 Running the example

The steps to run the example are:

1. Launch CCS, import the following projects available within the tidm_02011 directory - buck_f28003x_lfu, and flash_kernel_ex3_sci_flash_kernel. Build the BANK0_LDFU_BANK1TO0COPY build configuration of the flash_kernel_ex3_sci_flash_kernel project.
2. Build the BANK0_FLASH_BANK10COPY build configuration of the buck_F28003x_lfu project, with **BANK0_V1** declared as a predefined symbol. Rename the built .txt file from buck_F28003x_lfuBANK0FLASH.txt to buck_F28003x_lfuBANK0FLASH_v1.txt and copy it to C2000Ware_DigitalPower_SDK_xx_xx_xx\c2000ware\utilities\flash_programmers\serial_flash_programmer
3. Build the BANK0_FLASH_BANK10COPY build configuration of the buck_F28003x_lfu project, with **BANK0_V2** declared as a predefined symbol. Rename the built .txt file from buck_F28003x_lfuBANK0FLASH.txt to buck_F28003x_lfuBANK0FLASH_v2.txt and copy it to C2000Ware_DigitalPower_SDK_xx_xx_xx\c2000ware\utilities\flash_programmers\serial_flash_programmer
4. Launch a target configuration file (which erases all of Flash) for F28003x, connect to the F28003x target on the ControlCARD, and program flash_kernel_ex3_sci_flash_kernel.out to the device. This places the SCI Flash kernel i.e. LFU bootloader in Sectors 0 and 1 of Flash Bank0.
5. Once programming is complete, execution stops in the bankSelect() function. Click on Run.
6. Open a Windows command prompt, and change directory to the serial_flash_programmer directory within the DigitalPower SDK. Then issue the usual LFU command to program buck_F28003x_lfuBANK0FLASH_v1.txt to the target, which will program the application firmware executable to Bank1, then copy it from Bank1 to Bank0, then branch to the application entry point in Bank0 and begin execution. LED D2 on the top right corner of the ControlCARD will begin blinking. Disconnect CCS. This completes the “Production programming” steps.
7. To test LFU/FOTA updates in the field, repeat above step with the usual LFU command to program buck_F28003x_lfuBANK0FLASH_v2.txt to the target, which will program the application firmware executable to Bank1, then copy it from Bank1 to Bank0, then branch to the application entry point in Bank0 and begin execution.
8. With _v1, LED D2 blinks at a lower frequency than with _v2. Since the LED blinking occurs within an ISR, it continues even during the LFU process. LED blinking stops briefly when the Flash Bank1 to Bank0 copy and initialization of new code occurs, which lasts about 1 second.

5 Design and Documentation Support

5.1 Software Files

To download the software for this reference design, go to DigitalPower Software Development Kit (SDK) for C2000 MCUs.

5.2 Documentation Support

1. Texas Instruments, LFU e2e FAQ
2. Texas Instruments, Live Firmware Update (Without Reset) of C2000 F29x MCUs
3. Texas Instruments, TMS320C28x Optimizing C/C++ Compiler User's Guide
4. Texas Instruments, *TMS320F28003x Real-Time Microcontrollers Technical Reference Manual*
5. Texas Instruments, *TMS320F28004x Real-Time Microcontrollers Technical Reference Manual*
6. Texas Instruments, *Live Firmware Update without Device Reset on C2000™ MCUs user guide*
7. Texas Instruments, *Live Firmware Update with Device Reset on C2000™ MCUs User's Guide*
8. Texas Instruments, *TIDM-DC-DC-BUCK C2000™ Digital Power BoosterPack™*
9. Texas Instruments, *C2000™Piccolo™ F28004x Series LaunchPad Development Kit*
10. Texas Instruments, *C2000™Piccolo™ F28003x Series LaunchPad Development Kit*
11. Texas Instruments, *SCI Flash Kernel F28004x project in C2000Ware*
12. Texas Instruments, *LFU LED Blinky F28004x example in C2000Ware*
13. Texas Instruments, TMS320C28x Optimizing C/C++ Compiler User's Guide

5.3 Support Resources

TI E2E™ support forums are an engineer's go-to source for fast, verified answers and design help — straight from the experts. Search existing answers or ask your own question to get the quick design help you need.

Linked content is provided "AS IS" by the respective contributors. They do not constitute TI specifications and do not necessarily reflect TI's views; see TI's Terms of Use.

5.4 Trademarks

C2000™, TI E2E™, BoosterPack™, Code Composer Studio™, Piccolo™, F28004x Series LaunchPad™, LaunchPad™, are trademarks of Texas Instruments.
All trademarks are the property of their respective owners.

6 Terminology

CCS	Code Composer Studio
CLA	Control Law Accelerator
ISR	Interrupt Service Routine
LFU	Live Firmware Update
MCU	Microcontroller Unit
PSU	Power Supply Unit
SCI	Serial Communication Interface
UART	Universal Asynchronous Receiver-Transmitter

7 About the Author

Sira Rao is in the Application Specific MCU (ASM) business unit at Texas Instruments. He graduated from the Georgia Institute of Technology with a PhD in Electrical Engineering in 2007. His interests include embedded systems, computer architecture, and signal processing.

8 Revision History

Changes from Revision D (December 2022) to Revision E (August 2025) **Page**

- Updated Software Requirements 8
- Updated LFU Flow on the CPU 19
- Updated LFU Flow on the CLA 21
- Updated Limitations and Key Concerns 23
- Updated Preparing Firmware for LFU 24
- Updated LFU Compiler Support 25
- Updated Introduction 28
- Updated Documentation Support 30
- Updated About the Author 32

Changes from Revision C (August 2022) to Revision D (December 2022) **Page**

- Updated *Software Package Contents of F28003x Example* table..... 8
- Updated *Test Setup* topic..... 12
- Updated *LFU Flow on the CLA* topic..... 21
- Added additional steps to the *Assumptions* topic..... 23
- Updated *LFU Use-Cases* topic..... 27
- Added *FOTA Example* topics..... 28

Changes from Revision B (July 2022) to Revision C (August 2022) **Page**

- Updated *LFU Flow on the CPU* topic..... 19
- Updated *LFU Flow on the CLA* topic 21

Changes from Revision A (April 2021) to Revision B (July 2022) **Page**

- Updated description..... 1
- Added support for LFU on TMS320F28003x MCU..... 1
- Added F28003x and LAUNCHXL-F280039C 1
- Updated features..... 1
- Added F28003x..... 3
- Added definition of LFU Switchover time..... 3
- Added Compiler, MCU LFU hardware support..... 5
- Updated topic title to LFU Switchover Concepts..... 6
- Updated to note that Indicated Compiler support for LFU is available..... 6
- Updated definition of LFU switchover time..... 6
- Added note on F28003x LFU hardware features..... 6
- Updated descriptions, removed comment about ISRs in RAM..... 6
- Added LAUNCHXL-F280039C..... 8
- Updated Compiler and DigitalPower SDK version requirements..... 8
- Updated title of Table 3-1 to *Software Package Contents of F28004x example* 8
- Updated path to custom bootloader of F28004x from within C2000Ware to DPSDK..... 8
- Deleted flashapi_ex2_sci_kernel-CPU1-RAM.txt, and serial_flash_programmer.exe rows from the table to simplify demo..... 8
- Added a new Table for Software Package Contents of F28003x example..... 8
- Updated contents for F28003x..... 11
- Updated LFU Solution Software Directory Structure image..... 11

- Added note indicating TIDM-02011 runs TIDM-DC-DC-BUCK example on F28003x. Original TIDM-DC-DC-BUCK design still runs only on F28004x.....12
- Deleted *Loading the Custom Bootloader and Application to Flash without using CCS* topic.....12
- Updated Loading the Custom Bootloader and Application to Flash without using CCS is deleted.....12
- Added details for F28003x execution.....15
- Added details for F28003x execution.....17
- Updated F28003x implementation.....19
- Added note about F28003x LFU implementation limited to two banks. Can be extended to three.....23
- Deleted note on memcpy().....23
- Added note on NOINIT.....23
- Updated title to *Preparing Firmware for LFU* 24
- Updated not on Compiler support and initializing variables.....24
- Updated not on LFU switchover timing.....24
- Deleted note erlated to compiler initialization bug.....24
- Updated compiler version for LFU support.....25
- Added F28003x updates 25
- Deleted notes related to compiler initialization time and number of variables 25
- Deleted note related to compiler initialization bug 25
- Updated details on what happens during specific scenarios.....26
- Added F28003x TRM and LAUNCHXL-F280039C 30

Changes from Revision * (December 2020) to Revision A (April 2021) **Page**

- Added C2000WARE-DIGITALPOWER-SDK software hyperlink.....1
- Updated *C:\ti\c2000\C2000Ware_DigitalPower_SDK_version_num* with *<C2000Ware_DigitalPower_SDK_path>* throughout the document.....1
- Deleted *Before reading this section, it is important for users to read and follow the initial steps outlined in the readme.txt document present in the root directory of the software package (tidm_dc_dc_buck_lfu.zip).* 8
- Deleted reference to .zip 8
- Updated *20.8.0.STS* to *20.12.0.STS* 8
- Added *...these files are no longer present in package, need to be built.* 8
- Updated *build configuration* to *project build configuration* 8
- Added *...software structure is different.* 11
- Updated LFU Solution Software Directory Structure image.....11
- Added *CPU1_RAM build configuration is not used in this step* 12
- Updated step 2.....17
- Updated step 12.....19
- Updated steps 10, 11 and 15 21
- Updated step 9.....23
- Updated step 1 24
- Updated steps 3, 5.b, 7, and 9.c.....25
- Updated 8.0.STS to 20.12.0.STS.....25
- Deleted MSS.....27
- Updated *To download the software for this reference design, go to the DigitalPower Software Development Kit (SDK) for C2000 MCUs site.* 30

IMPORTANT NOTICE AND DISCLAIMER

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

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, 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 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, TI’s General Quality Guidelines, 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. Unless TI explicitly designates a product as custom or customer-specified, TI products are standard, catalog, general purpose devices.

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

Last updated 10/2025