Live Firmware Update Without Device Reset on C2000™ MCUs

Baskaran Chidambaram and Sira Rao

ABSTRACT

This document presents details on live firmware update (LFU) without device reset on devices with two Flash banks, detailing the challenges involved and suggestions on how to address them. For simplicity of illustration, an LED based example is used (included as part of C2000ware).

Contents

List of Figures

Trademarks

C2000, Code Composer Studio are trademarks of Texas Instruments.
All other trademarks are the property of their respective owners.

1 Introduction

In applications like server power supply, metering, and so forth 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 increase in total system cost. An alternate approach, called Live Firmware Update (LFU), allows firmware to be updated 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.

2 Resources Required for LFU

LFU is feasible when the device has enough resources of various kinds – CPU bandwidth, Memory, and Peripheral availability:

- CPU Bandwidth – The new firmware has to be transferred using a communication peripheral and written to Flash memory while the application is still operating. This means CPU need to have enough available bandwidth to support LFU.
- Memory – The non-volatile memory that is used here is Flash memory. Flash read and write operations cannot be simultaneously performed on the same Flash bank. However read and write operations can be simultaneously performed on different Flash banks. Hence, the ideal scenario is for the device to contain dual Flash banks. In devices with single Flash banks, LFU is particularly challenging, but still feasible provided:
  - The complete application code (or a portion of it that controls the output(s) the user is interested in) and Flash APIs need to run from RAM memory while the new firmware is updated to Flash. This means there should be enough RAM memory that can be utilized.
  - Some devices support Flash APIs in ROM. In those devices, the application code can run from RAM and Flash APIs can run from ROM memory, thus reducing the RAM requirement.
- Peripheral Availability – A spare communication peripheral using the new image can be transferred from the host to the device.

LFU is easier to implement in devices with multiple Flash banks. In this document and the reference example, it is assumed that the device has two Flash banks. The device considered is TMS320F28004x, which has dual flash banks. Their address space is 64K x 16 each, with addresses ranging from 0x80000-0x8FFFF, and 0x90000-0x9FFFF.

3 Memory Layout

The Flash memory has been partitioned as shown in Figure 1.

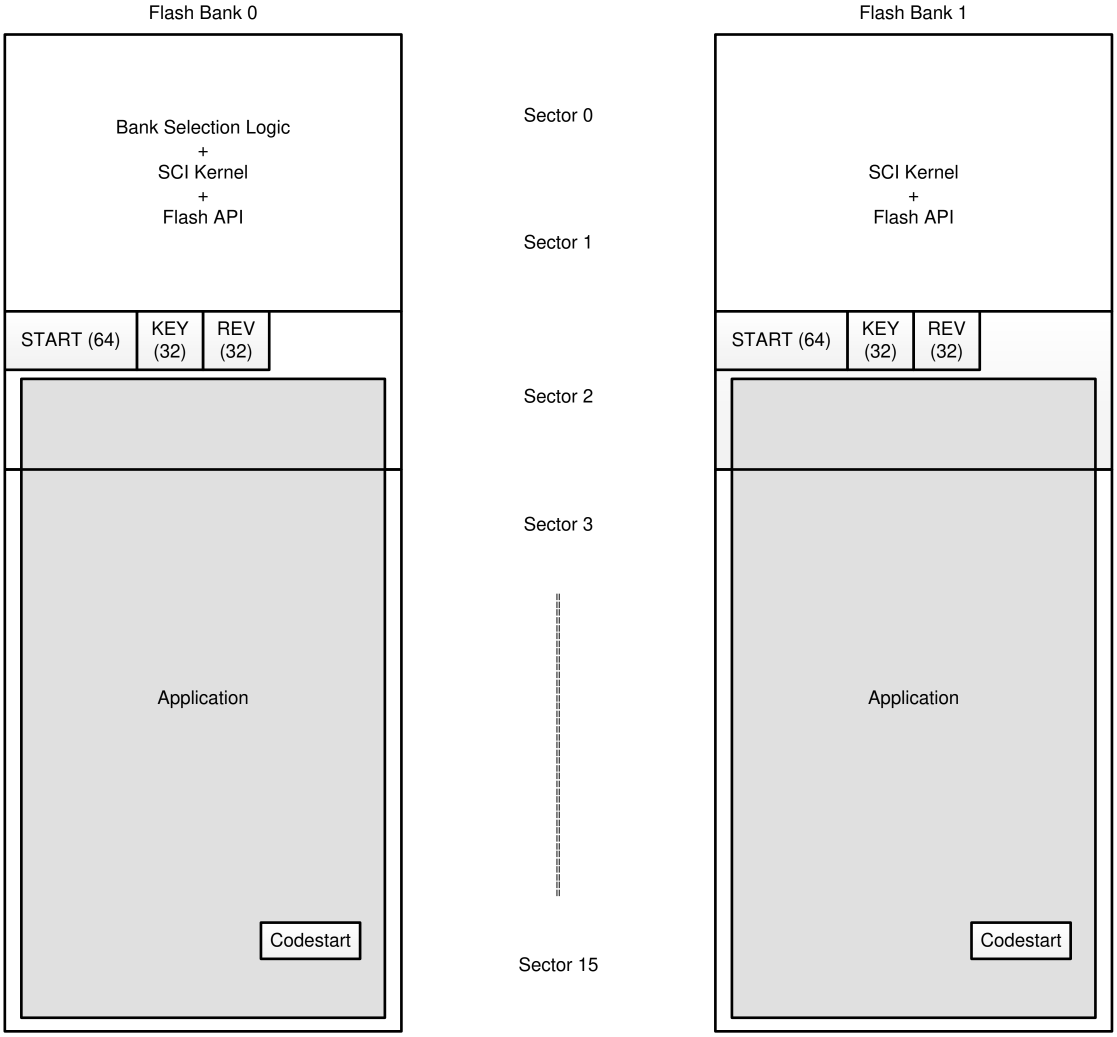


Figure 1. Flash Memory Contents for Bank 0 and Bank 1

Assuming each Flash bank has 16 sectors, two sectors have been allocated to Static code (code that will not change between applications). This is described in Section 4.

A few locations in sector 2 have been reserved to store the below entries:

- START – when this 64-bit field is set in a specific Flash bank (to 0x5A5A5A5A5A5A5A5A) by the Serial Communications Interface (SCI) Flash Kernel, it indicates that the corresponding Flash bank has been erased (Application sectors) and programming/verification is about to begin. In this example, the START field is located at addresses 0x82000 in BANK0 and 0x92000 in BANK1.
- REV – this represents a 32-bit Firmware revision number that is set by the SCI Flash Kernel that is used by the Bank Selection Logic to determine which of the latest images is between Flash bank 0 and 1. REV starts at 0xFFFFFFFF and is decremented on each subsequent Flash programming cycle. Thus, the Bank with the lower revision number is considered the latest one. Firmware revision field is handled by the flash kernel for simplicity. In practice, the last downloaded image may not be the latest firmware version and the application image would contain the firmware version, but that assumption is made with this model where the kernel updates the REV field. In this example, the REV field is located at address 0x82006 in BANK0 and 0x92006 in BANK1. As an example, if the REV in BANK0 is FFFF FFFA, and the REV in BANK1 is FFFF FFF9, BANK1 will be considered the latest firmware and will execute.

- KEY– The image in a bank is considered as valid if this location contains a valid KEY (0x5B5B5B5B). This Key is written to by the SCI Flash Kernel, and is read and tested by the Bank Selection Logic. In this example, the KEY field is located at addresses 0x82004 in BANK0 and 0x92004 in BANK1.

The rest of sector 2 and sectors 3-15 can be used to store the application image. This allows the static code in sectors 0 and 1 to be programmed once, and remain unchanged during LFU.

4 Static Code in LFU

Static code that is used to support LFU consists of:

- Bank Selection Logic – when there are two (or more) Flash banks containing application firmware, logic that determines which Flash bank to boot is necessary. A common implementation of this logic centers on a firmware revision number. As described, the lower revision number is the latest image in this Example. Bank selection logic is placed at the default flash boot address (0x80000 in F28004x), so that once Boot ROM code completes execution, execution will transfer to bank selection logic. Bank selection logic is only included in Bank0, not in Bank1.
- Flash Kernel – it is the job of the Flash kernel to receive the Firmware image from the host using a peripheral, and call the Flash programming APIs to write it to flash memory. In this document, the SCI flash Kernel is used since the SCI peripheral is used to transfer the new firmware image. The detailed step by step flow is documented in the file header of flashapi_ex2_ldfu.c.
  - In a blank device (where application firmware is not present in either flash banks), the bank selection logic will identify there is no valid KEY in either Flash bank, and will wait for an LFU command over SCI. This will use the Flash Kernel to update the firmware (on to bank1 as the kernel code is first programmed to bank0, and therefore executes from bank0).
  - If one or both banks contains a valid application, bank selection logic will transfer control to the code entry point (codestart) of the corresponding bank. In this example, the codestart address is 0x8EFF0 for Bank0 and 0x9EFF0 for Bank1.
  - During LFU, the application will make a call to the Flash Kernel to receive and update the firmware.
- Flash API – Flash APIs provide interfaces to erase and program Flash memory. These APIs need to run from the bank which is NOT being updated.

The static code is configured as a single example - flashapi_ex2_sci_kernel project (included in C2000Ware at <C2000Ware>\driverlib\f28004x\examples\flash). This example supports multiple build configurations, of which those relevant to LFU are listed below:

- BANK0_LDFU - Links the bank selection logic and flash kernel to Bank 0 (addresses 0x80000 - 0x81FFF). Uses Flash API symbols in flash.
- BANK0_LDFU_ROM - Links the bank selection logic and flash kernel to Bank 0 (addresses 0x80000 - 0x81FFF). Uses Flash API symbols in ROM; Rev A of F28004x cannot be used with this build configuration, since it does not support Flash APIs in ROM.

  The two configurations above support programming application image onto Bank1.
- BANK1_LDFU - Links the flash kernel to Bank 1 (0x90000 - 0x91FFF). Uses Flash API symbols in flash.
- BANK1_LDFU_ROM - Links the flash kernel to Bank 1 (0x90000 - 0x91FFF). Uses Flash API symbols in ROM; Rev A of F28004x cannot be used with this build configuration, since it does not support Flash APIs in ROM.

The two configurations above support programming application image onto Bank0.

For more details, see the flashapi_ex2_sciKernel.c in the flashapi_ex2_sci_kernel project (included in C2000Ware at <C2000Ware>\driverlib\f28004x\examples\flash).

5 LED Example Application and LFU Flow

This example (flashapi_ex5_lfu_no_reset project) is designed to demonstrate LFU with the LED blinking periodically. This is achieved using the Live Device Firmware Update (Live DFU or LDFU) command, which is part of SCI kernel. This is to be used with the Serial Flash Programmer (PC tool).

In this example, an SCI auto baud lock is performed and the byte used for auto baud lock is echoed back. Two interrupts are initialized and enabled: SCI Rx FIFO interrupt and CPU Timer 0 interrupt. The CPU Timer 0 interrupt occurs every 1 second; the interrupt service routine (ISR) for CPU Timer 0 toggles an LED based on the build configuration that is running.

- LED1 is toggled with the BANK0_FLASH build configuration. “BANK0” is a pre-defined symbol in this build configuration, and when this symbol is defined, the project sets up GPIOs associated with LED1.
- LED2 is toggled with the BANK1_FLASH build configuration. “BANK1” is a pre-defined symbol in this build configuration, and when this symbol is defined, the project sets up GPIOs associated with LED2.

The application images generated by building the above build configurations are the ones that will be used to illustrate LFU in this document. Note that other than the changes described above between the two build configurations, there are no other differences between the two application images. Hence, this is a relatively simple example for LFU illustration.

The SCI Rx FIFO interrupt is set for a FIFO interrupt level of 10 bytes. The number of bytes in a packet from the Serial Flash Programmer (when using the LDFU command) is 10. When a command is sent to the device from the Serial Flash Programmer, the SCI Rx FIFO ISR receives a command from the 10 byte packet in the FIFO. If the command matches the Live Device Firmware Update (Live DFU) command, then the code branches to the Live DFU function (liveDFU()) located inside of the SCI Flash Kernel (flashapi_ex2_ldfu.c) for the corresponding bank. So if the application on Bank0 is executing, control will pass to liveDFU() on Bank0, located at 0x81000. If the application on Bank1 is executing, control will pass to liveDFU() on Bank1, located at 0x91000. Within this function, execution passes to the ldfuLoad() function in order to erase the appropriate bank, load a hex formatted program (in the appropriate SCI boot format) into flash, and verify the program.

Once the firmware is updated on the alternate flash bank, the PC is made to jump to the entry point (codestart) of the new image, which branches to the C runtime initialization routine and then calls main() of the application. In the new image, during LFU flow (refer to the lfuSwitch variable in flashapi_ex5_lfu_no_reset.c) most of the variable and peripheral initializations can be skipped to quickly run the application and service critical interrupts. If the application runs as a result of LFU, then the LED is made to blink at a faster rate (400 ms – ON for 400 ms, OFF for 400 ms) to indicate successful LFU switchover to the new application.

Figure 2 depicts the flow of the code at a high level after the code enters main() of the application. For more details, see the flashapi_ex5_lfu_no_reset.c located in the flashapi_ex5_lfu_no_reset project (included in C2000Ware at <C2000Ware>\driverlib\f28004x\examples\flash).

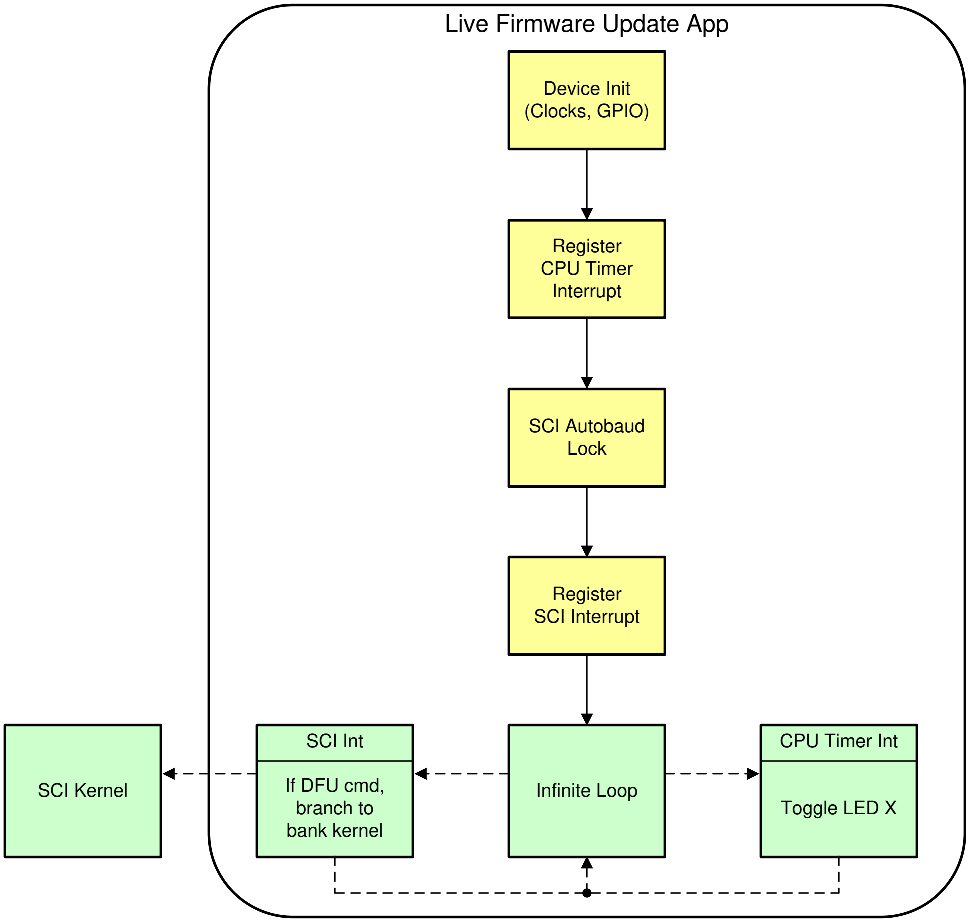


Figure 2. Code Flow After Entering main() of Application

5.1 Managing Variables in LFU

The key factor in LFU is to switch to new firmware as soon as possible and also retain context from execution of old firmware. This can be achieved by ensuring that variables are retained at the same address in RAM in both firmware images. This also helps in skipping initialization of these variables which reduces initialization time. If the Compiler does not provide explicit support to maintain variables at the same addresses between firmware images, it will be the user’s responsibility to do so. Users may use compiler directives to aid with this (#pragma DATA_SECTION (myVar,"MYSECTION").

It is also important to note that variable initialization, which occurs during C runtime initialization prior to main(), is handled differently depending on whether the executable output format is COFF or EABI. With COFF, uninitialized global variables are not initialized during C runtime initialization. Initialized variables are assigned to their default values. So if context needs to be retained from an old to a new image, nothing extra needs to be done for uninitialized variables.

With EABI, however, uninitialized global variables are initialized to ‘0’ during C runtime initialization. Initialized variables are assigned to their default values. So if context needs to be retained from old to new image, uninitialized global variables need to be explicitly defined with the NOINIT qualifier - #pragma NOINIT (myVar).

To address initialized variables, with both COFF and EABI, the user will need to ensure that address in RAM is maintained, and initialization is avoided to reduce switchover time during LFU. As mentioned earlier, this example does that and illustrates how the user may go about doing that.

If the new firmware contains new variables, there are no constraints on their address in RAM. If they are initialized variables, they will be initialized during C runtime initialization. If they are uninitialized variables, and the output format is EABI, they will need to be specified with the NOINIT qualifier so that time is not spent initializing them to 0.

If there are changes to array sizes or addition of variables to a structure, the user needs to manage these appropriately. One option is to copy data from the old array/struct to new array/struct, but this is inefficient and leads to increased initialization time. Another option is to use the DATA_SECTION pragma to place arrays and structures at fixed locations, but with sufficient headroom for potential growth of these arrays and structures in future firmware. With this approach, only newly added fields need to be initialized.

NOTE: The LED example project's output format is COFF.

6 Running the LED Example

6.1 Serial Flash Programmer Update

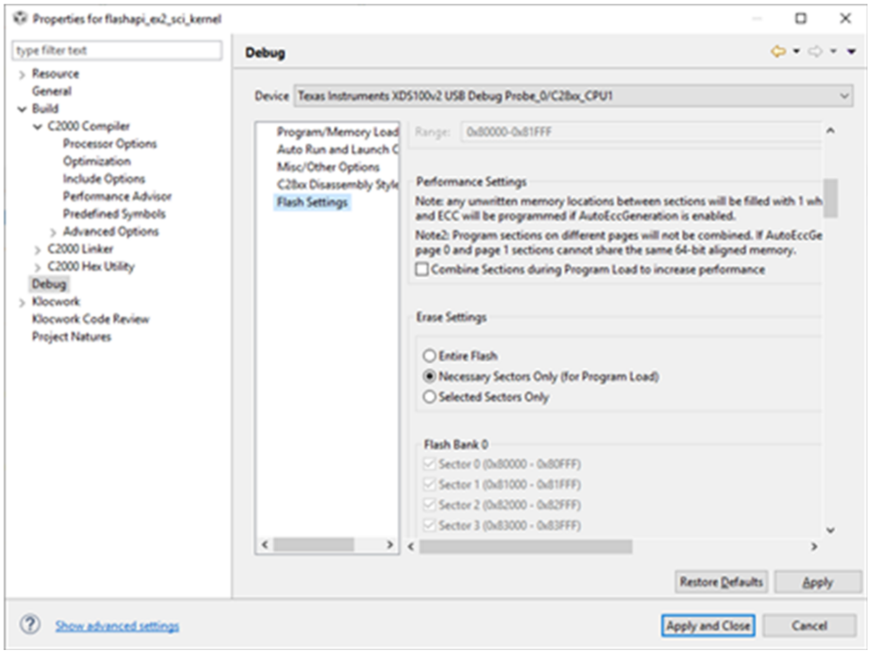
The serial flash programmer (serial_flash_programmer.exe) supplied with C2000ware takes both the kernel and application image as parameters. Typically, the kernel is transferred first over to the SCI bootloader and executed from RAM or Flash on the device. The kernel program then takes the application image over SCI (from serial programmer running on PC) and programs the application image in flash memory.

In the case of LFU, the static content including flash kernel is first programmed to flash sectors 0 and 1 of Flash banks 0 and 1. This is described in Section 6.2. After this, the serial flash programmer needs to be modified to transfer only the application image. This can be done by commenting the line “#define kernel” in serial_flash_programmer.cpp. The serial flash programmer can be regenerated by compiling the project in Visual C (called serial_flash_programmer_appln.exe). The pre-built executable is placed at <C2000Ware>\utilities\flash_programmers\serial_flash_programmer\). Thus, the user needs to take no action here.

6.2 Programming Static Code – Loading via Code Composer Studio™ (CCS)

The hardware used in the illustrated steps is an F28004x ControlCARD on a ControlCARD docking station Rev4.1. If a JTAG connection is available, then CCS can be used to load Flash banks 0 and 1 with static code.

NOTE: Make sure you have the settings in CCS (or your target configuration file – by right clicking and selecting properties) as shown in Figure 3 before loading the images. Build the flashapi_ex2_sci_kernel project in both BANK0_LDFU and BANK1_LDFU configurations, and load each to target. The CCS flash plugin will load the contents on to flash.



Device Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1

Range: 0x80000-0x81FFF

Performance Settings

Note: any unwritten memory locations between sections will be filled with 1 wh
and ECC will be programmed if AutoEccGeneration is enabled.

Note2: Program sections on different pages will not be combined. If AutoEccGe
page 0 and page 1 sections cannot share the same 64-bit aligned memory.

□ Combine Sections during Program Load to increase performance

Erase Settings

○ Entire Flash
● Necessary Sectors Only (for Program Load)
○ Selected Sectors Only

Flash Bank 0

☑ Sector 0 (0x80000 - 0x80FFF)
☑ Sector 1 (0x81000 - 0x81FFF)
☑ Sector 2 (0x82000 - 0x82FFF)
☑ Sector 3 (0x83000 - 0x83FFF)

Figure 3. Flash Settings to Only Erase Necessary Sectors

1. Load BANK0 static image (BANK0_LDFU of flashapi_ex2_sci_kernel project).

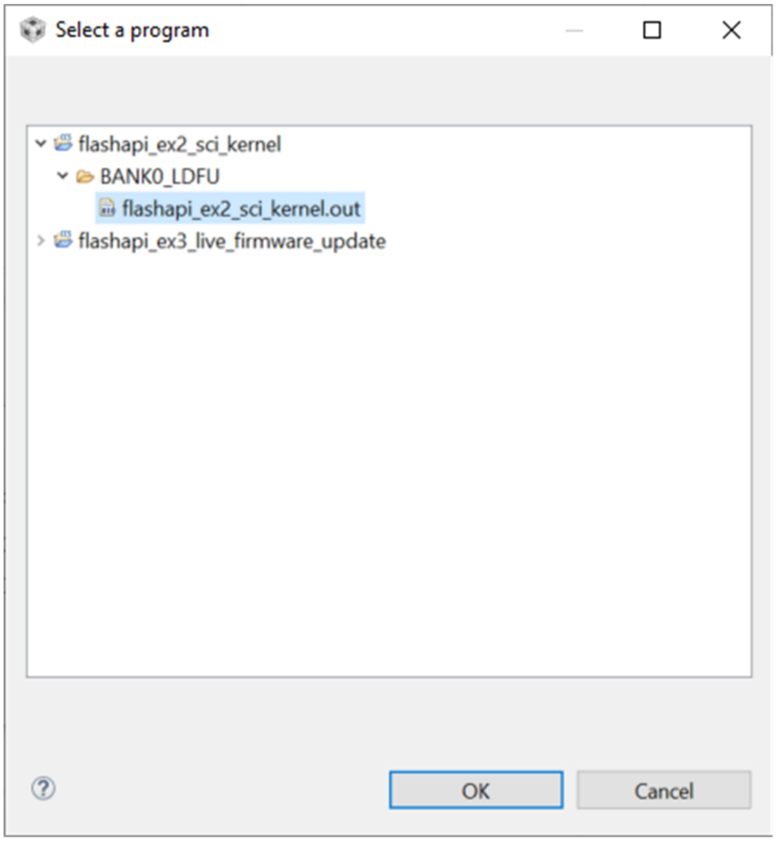
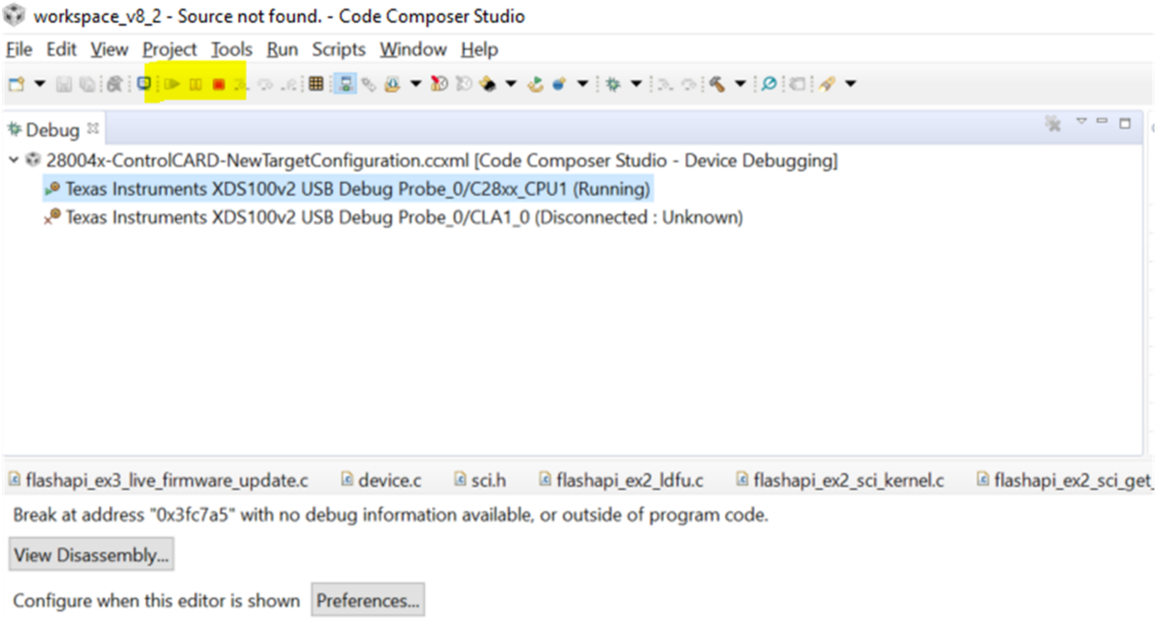

Select a program
flashapi_ex2_sci_kernel
BANK0_LDFU
flashapi_ex2_sci_kernel.out
flashapi_ex3_live_firmware_update
OK
Cancel

Figure 4. Selecting Kernel to Load to Flash Bank 0

After the static contents are loaded on BANK0, the first thing that happens is the execution of the bank selection logic that will determine that application firmware is not programmed in either bank.

Control will pass to the flash kernel, which will be ready to program the application in BANK1. The CCS view will look Figure 5 (the program will not stop at main(), but will be running, awaiting a SCI command).


workspace_v8_2 - Source not found. - Code Composer Studio

File Edit View Project Tools Run Scripts Window Help
Debug

28004x-ControlCARD-NewTargetConfiguration.ccxml [Code Composer Studio - Device Debugging]
Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1 (Running)
Texas Instruments XDS100v2 USB Debug Probe_0/CLA1_0 (Disconnected : Unknown)

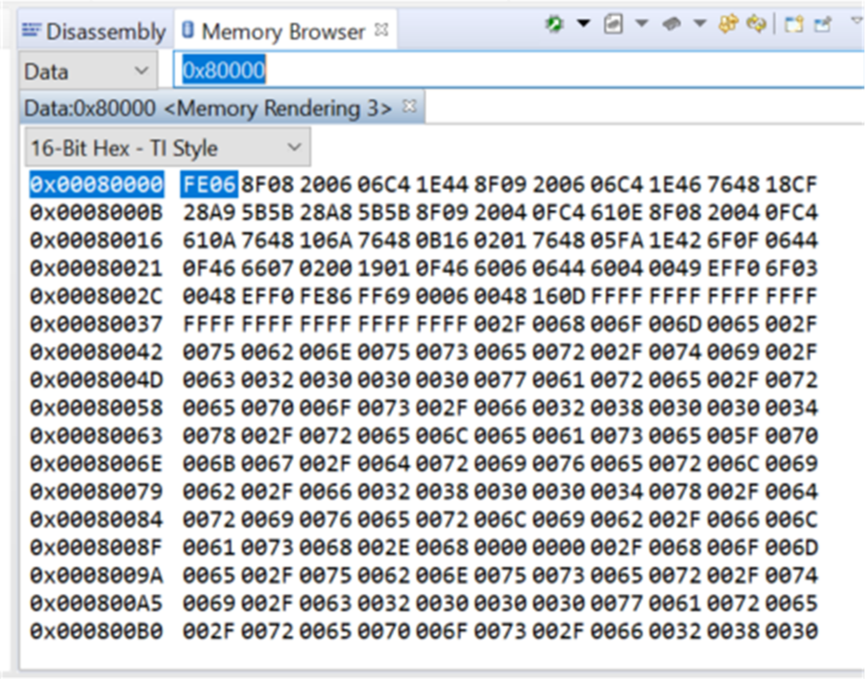
flashapi_ex3_live_firmware_update.c
device.c
sci.h
flashapi_ex2_ldfu.c
flashapi_ex2_sci_kernel.c
flashapi_ex2_sci_get_

Break at address "0x3fc7a5" with no debug information available, or outside of program code.

Configure when this editor is shown

Figure 5. CCS Window view After Programming Bank 0 Flash Kernel

2. Verify the Kernel content of BANK0 by opening the Memory Browser window in CCS, and entering address 0x80000.



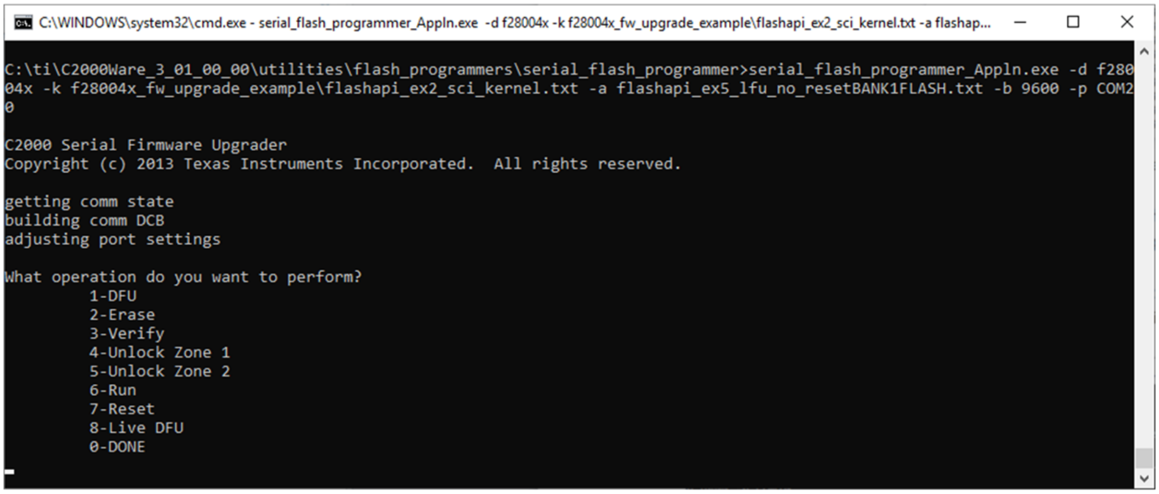
```
0x00080000  FE06 8F08 2006 06C4 1E44 8F09 2006 06C4 1E46 7648 18CF
0x0008000B  28A9 5B5B 28A8 5B5B 8F09 2004 0FC4 610E 8F08 2004 0FC4
0x00080016  610A 7648 106A 7648 0B16 0201 7648 05FA 1E42 6F0F 0644
0x00080021  0F46 6607 0200 1901 0F46 6006 0644 6004 0049 EFF0 6F03
0x0008002C  0048 EFF0 FE86 FF69 0006 0048 160D FFFF FFFF FFFF FFFF
0x00080037  FFFF FFFF FFFF FFFF FFFF 002F 0068 006F 006D 0065 002F
0x00080042  0075 0062 006E 0075 0073 0065 0072 002F 0074 0069 002F
0x0008004D  0063 0032 0030 0030 0030 0077 0061 0072 0065 002F 0072
0x00080058  0065 0070 006F 0073 002F 0066 0032 0038 0030 0030 0034
0x00080063  0078 002F 0072 0065 006C 0065 0061 0073 0065 005F 0070
0x0008006E  006B 0067 002F 0064 0072 0069 0076 0065 0072 006C 0069
0x00080079  0062 002F 0066 0032 0038 0030 0030 0034 0078 002F 0064
0x00080084  0072 0069 0076 0065 0072 006C 0069 0062 002F 0066 006C
0x0008008F  0061 0073 0068 002E 0068 0000 0000 002F 0068 006F 006D
0x0008009A  0065 002F 0075 0062 006E 0075 0073 0065 0072 002F 0074
0x000800A5  0069 002F 0063 0032 0030 0030 0030 0077 0061 0072 0065
0x000800B0  002F 0072 0065 0070 006F 0073 002F 0066 0032 0038 0030
```

Figure 6. CCS Memory Browser View to Verify Successful Kernel Programming of Bank 0

3. Now switch to the Windows command prompt, and execute the command below:
   serial_flash_programmer_appln.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel.txt -a flashapi_ex5_lfu_no_resetBANK1FLASH.txt -b 9600 -p COMx

   where, x = COM port corresponding to the JTAG connection between the PC and the target board.

   flashapi_ex5_lfu_no_resetBANK1FLASH.txt is generated by building the CCS project flashapi_ex5_lfu_no_reset in build configuration BANK1_FLASH. The COM port number can be found by looking up Ports in Device Manager. This will be populated by virtue of the fact that there is a USB cable connected between the target and the computer, which provides both JTAG and SCI functionality. The example uses a baud rate of 9600.

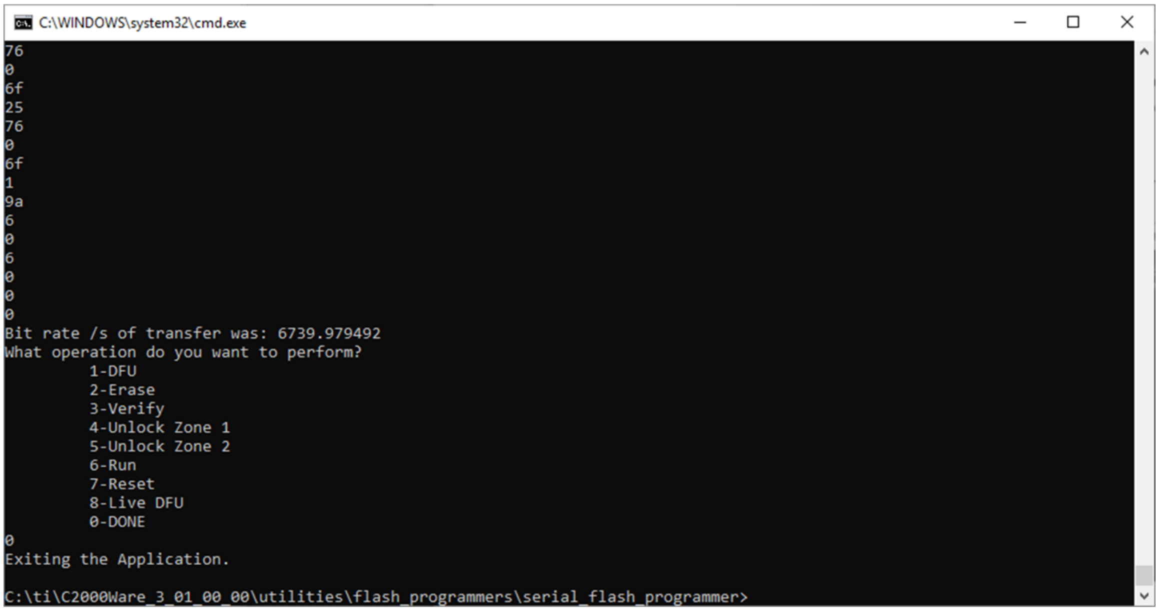


C:\WINDOWS\system32\cmd.exe - serial_flash_programmer_Appln.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel.txt -a flashap...

```
C:\ti\C2000Ware_3_01_00_00\utilities\flash_programmers\serial_flash_programmer>serial_flash_programmer_Appln.exe -d f280
04x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel.txt -a flashapi_ex5_lfu_no_resetBANK1FLASH.txt -b 9600 -p COM2
0

C2000 Serial Firmware Upgrader
Copyright (c) 2013 Texas Instruments Incorporated.  All rights reserved.

getting comm state
building comm DCB
adjusting port settings

What operation do you want to perform?
         1-DFU
         2-Erase
         3-Verify
         4-Unlock Zone 1
         5-Unlock Zone 2
         6-Run
         7-Reset
         8-Live DFU
         0-DONE
```

Figure 7. LFU Serial Command Invoked From Windows Command Prompt

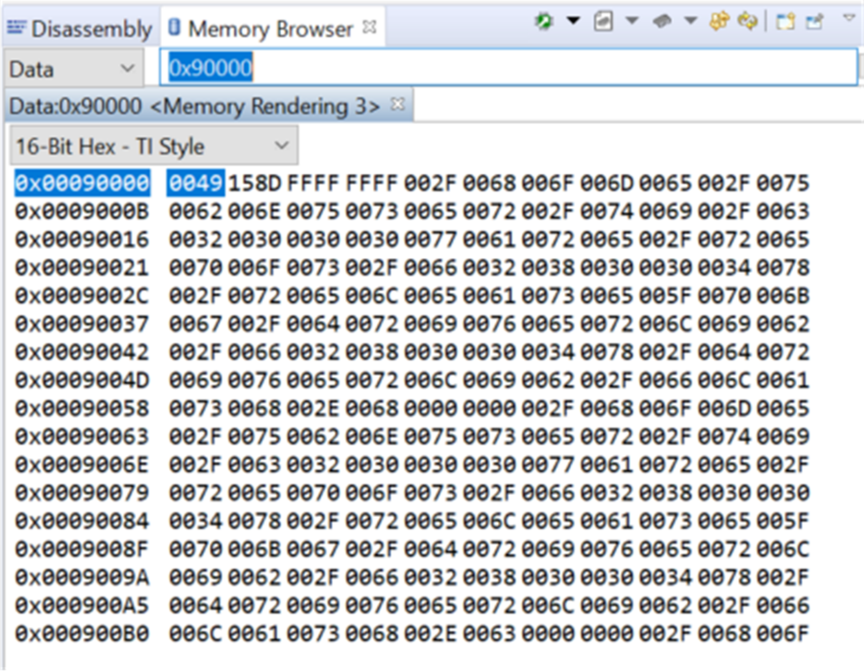
4. Once LDFU (8) command is selected, the kernel will receive and program the application in BANK1. The application size is about 35KB. Download time will be about 30 seconds.



```
76
0
6f
25
76
0
6f
1
9a
6
0
6
0
0
0
Bit rate /s of transfer was: 6739.979492
What operation do you want to perform?
         1-DFU
         2-Erase
         3-Verify
         4-Unlock Zone 1
         5-Unlock Zone 2
         6-Run
         7-Reset
         8-Live DFU
         0-DONE
0
Exiting the Application.

C:\ti\C2000Ware_3_01_00_00\utilities\flash_programmers\serial_flash_programmer>
```

Figure 8. Successful Completion of LFU Command to Program Flash Bank 1

5. After transfer is complete, enter 0 to indicate end of command operations. Verify the Application content of BANK1 by opening the Memory Browser window in CCS, and entering address 0x92000.



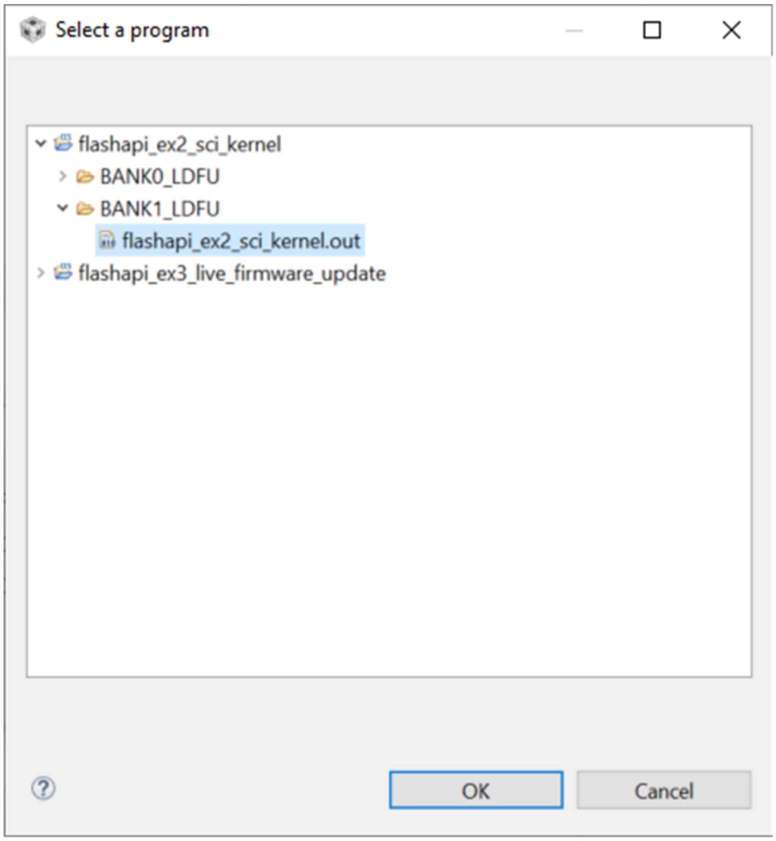
```
0x00090000  0049 158D FFFF FFFF 002F 0068 006F 006D 0065 002F 0075
0x0009000B  0062 006E 0075 0073 0065 0072 002F 0074 0069 002F 0063
0x00090016  0032 0030 0030 0030 0077 0061 0072 0065 002F 0072 0065
0x00090021  0070 006F 0073 002F 0066 0032 0038 0030 0030 0034 0078
0x0009002C  002F 0072 0065 006C 0065 0061 0073 0065 005F 0070 006B
0x00090037  0067 002F 0064 0072 0069 0076 0065 0072 006C 0069 0062
0x00090042  002F 0066 0032 0038 0030 0030 0034 0078 002F 0064 0072
0x0009004D  0069 0076 0065 0072 006C 0069 0062 002F 0066 006C 0061
0x00090058  0073 0068 002E 0068 0000 0000 002F 0068 006F 006D 0065
0x00090063  002F 0075 0062 006E 0075 0073 0065 0072 002F 0074 0069
0x0009006E  002F 0063 0032 0030 0030 0030 0077 0061 0072 0065 002F
0x00090079  0072 0065 0070 006F 0073 002F 0066 0032 0038 0030 0030
0x00090084  0034 0078 002F 0072 0065 006C 0065 0061 0073 0065 005F
0x0009008F  0070 006B 0067 002F 0064 0072 0069 0076 0065 0072 006C
0x0009009A  0069 0062 002F 0066 0032 0038 0030 0030 0034 0078 002F
0x000900A5  0064 0072 0069 0076 0065 0072 006C 0069 0062 002F 0066
0x000900B0  006C 0061 0073 0068 002E 0063 0000 0000 002F 0068 006F
```

Figure 9. CCS Memory Browser View to Verify Successful Programming of Application on Bank 1

The kernel will also update the KEY and revision number in BANK1 sector 2. Now the static image, in programmed BANK0 and application image, is programmed in BANK1.

6. At this point, the user can reset the board to see LED2 start blinking.

7. Load BANK1 static image (BANK1_LDFU of flashapi_ex2_sci_kernel project).

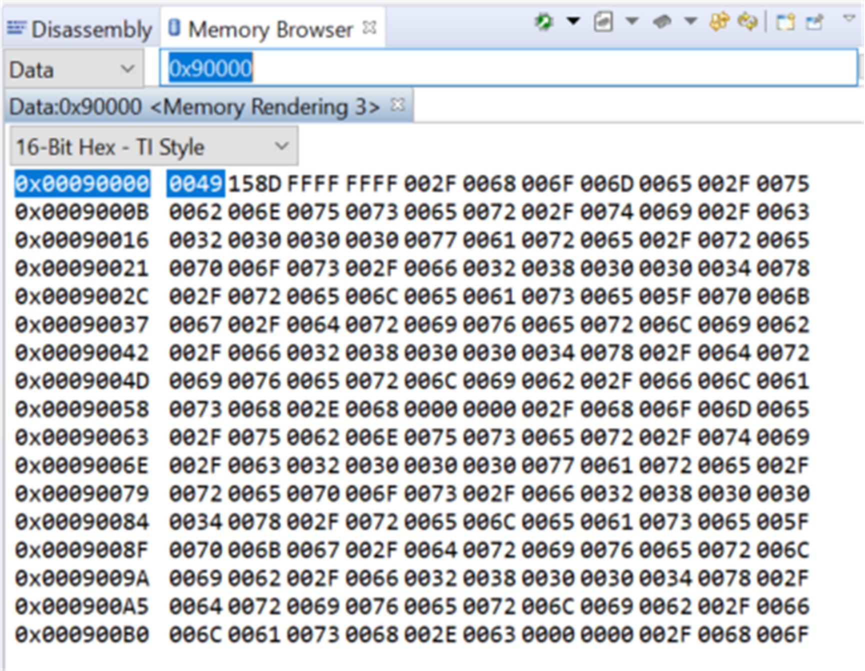


Select a program

- flashapi_ex2_sci_kernel
  - BANK0_LDFU
  - BANK1_LDFU
    - flashapi_ex2_sci_kernel.out
- flashapi_ex3_live_firmware_update

Figure 10. Selecting Kernel to Load to Flash Bank 1

8. Verify the Kernel content of BANK1 by opening the Memory Browser window in CCS, and entering address 0x90000.



Data 0x90000

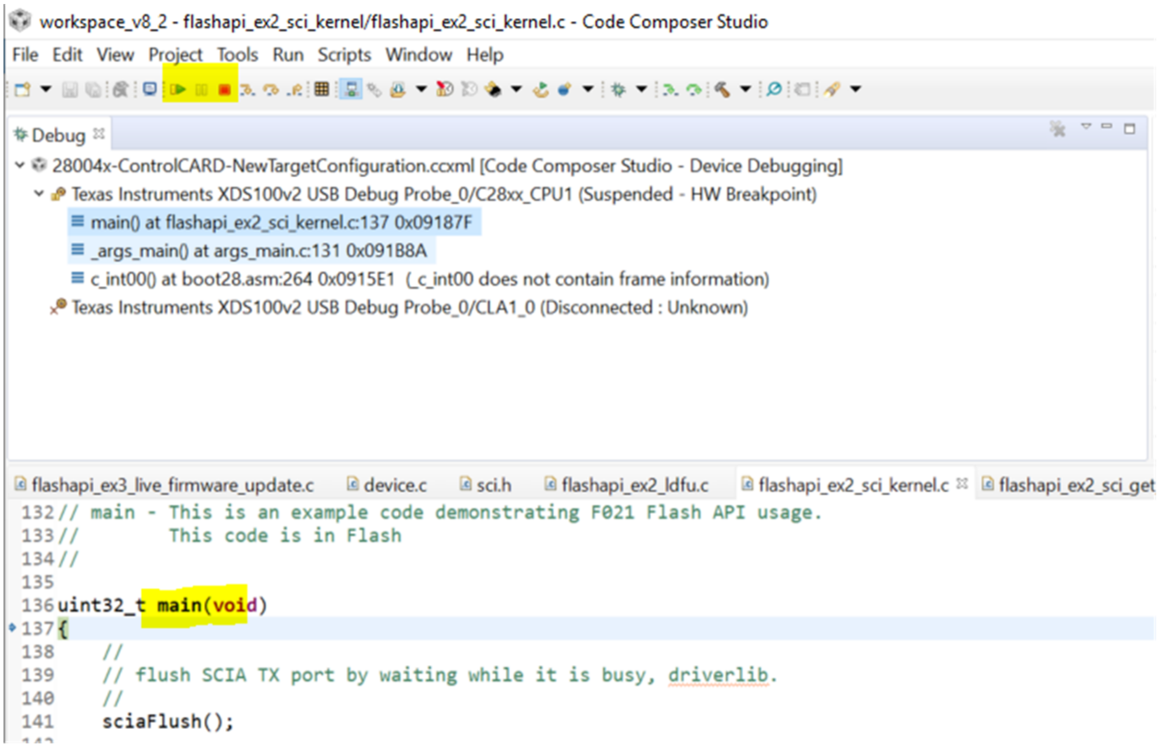
Data:0x90000 <Memory Rendering 3>

16-Bit Hex - TI Style

```
0x00090000  0049 158D FFFF FFFF 002F 0068 006F 006D 0065 002F 0075
0x0009000B  0062 006E 0075 0073 0065 0072 002F 0074 0069 002F 0063
0x00090016  0032 0030 0030 0030 0077 0061 0072 0065 002F 0072 0065
0x00090021  0070 006F 0073 002F 0066 0032 0038 0030 0030 0034 0078
0x0009002C  002F 0072 0065 006C 0065 0061 0073 0065 005F 0070 006B
0x00090037  0067 002F 0064 0072 0069 0076 0065 0072 006C 0069 0062
0x00090042  002F 0066 0032 0038 0030 0030 0034 0078 002F 0064 0072
0x0009004D  0069 0076 0065 0072 006C 0069 0062 002F 0066 006C 0061
0x00090058  0073 0068 002E 0068 0000 0000 002F 0068 006F 006D 0065
0x00090063  002F 0075 0062 006E 0075 0073 0065 0072 002F 0074 0069
0x0009006E  002F 0063 0032 0030 0030 0030 0077 0061 0072 0065 002F
0x00090079  0072 0065 0070 006F 0073 002F 0066 0032 0038 0030 0030
0x00090084  0034 0078 002F 0072 0065 006C 0065 0061 0073 0065 005F
0x0009008F  0070 006B 0067 002F 0064 0072 0069 0076 0065 0072 006C
0x0009009A  0069 0062 002F 0066 0032 0038 0030 0030 0034 0078 002F
0x000900A5  0064 0072 0069 0076 0065 0072 006C 0069 0062 002F 0066
0x000900B0  006C 0061 0073 0068 002E 0063 0000 0000 002F 0068 006F
```

Figure 11. CCS Memory Browser View to Verify Successful Kernel Programming of Bank 1

After the static contents are loaded on to BANK1, execution stops at main(). This occurs for the kernel on Bank1 because bank selection logic resides only on Bank0, not on Bank1. Thus, for Bank1, execution flow follows the conventional flow of codestart leading up to main().


workspace_v8_2 - flashapi_ex2_sci_kernel/flashapi_ex2_sci_kernel.c - Code Composer Studio
File Edit View Project Tools Run Scripts Window Help
Debug
flashapi_ex3_live_firmware_update.c
device.c
sci.h
flashapi_ex2_ldfu.c
flashapi_ex2_sci_kernel.c
flashapi_ex2_sci_get

28004x-ControlCARD-NewTargetConfiguration.ccxml [Code Composer Studio - Device Debugging]
Texas Instruments XDS100v2 USB Debug Probe_0/C28xx_CPU1 (Suspended - HW Breakpoint)
main() at flashapi_ex2_sci_kernel.c:137 0x09187F
_args_main() at args_main.c:131 0x091B8A
c_int00() at boot28.asm:264 0x0915E1 (_c_int00 does not contain frame information)
Texas Instruments XDS100v2 USB Debug Probe_0/CLA1_0 (Disconnected : Unknown)

```
// main - This is an example code demonstrating F021 Flash API usage.
//        This code is in Flash
//

uint32_t main(void)
{
    //
    // flush SCIA TX port by waiting while it is busy, driverlib.
    //
    sciaFlush();
```

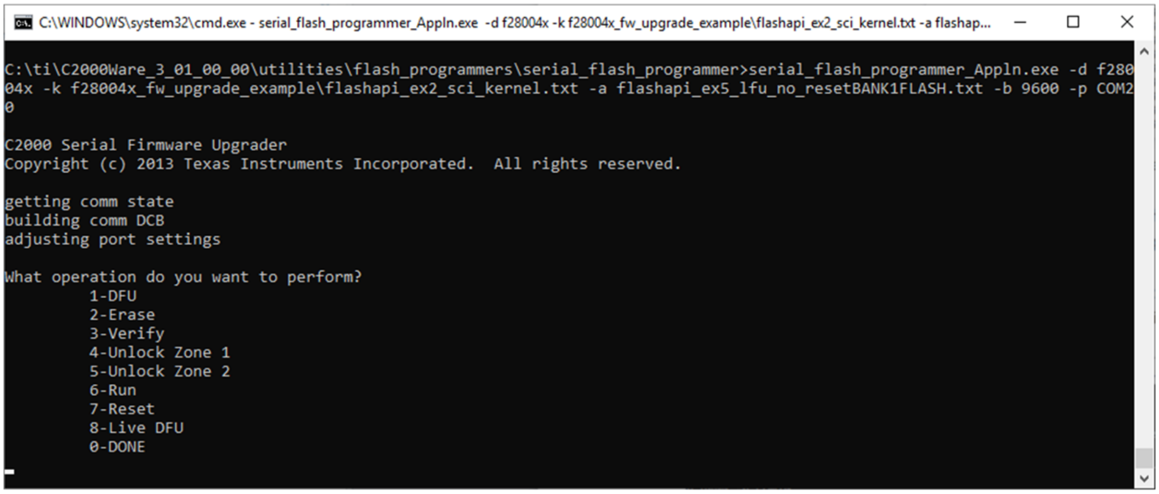
Figure 12. CCS Window View After Programming Bank 1 Flash Kernel

9. Press Run in CCS, the bank selection logic will run and execute application from BANK1. Then, execute the below command from command line in PC.

   serial_flash_programmer_appln.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel.txt -a flashapi_ex5_lfu_no_resetBANK0FLASH.txt -b 9600 -p COMx

   where, x = COM port corresponding to the JTAG connection between the PC and the target board.

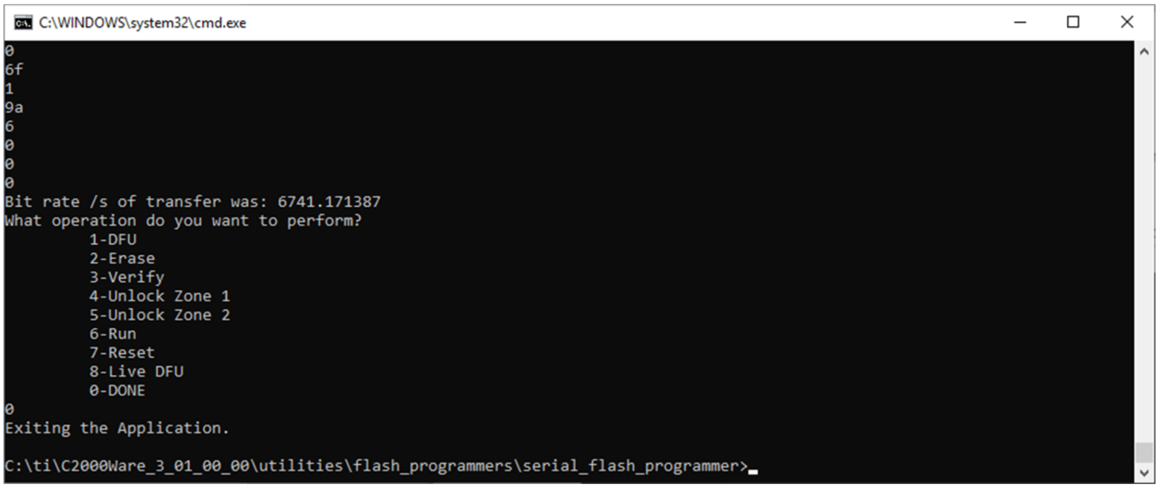
   flashapi_ex5_lfu_no_resetBANK0FLASH.txt is generated by building the CCS project flashapi_ex5_lfu_no_reset in build configuration BANK0_FLASH.


C:\WINDOWS\system32\cmd.exe - serial_flash_programmer_Appln.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel.txt -a flashap...

```
C:\ti\C2000Ware_3_01_00_00\utilities\flash_programmers\serial_flash_programmer>serial_flash_programmer_Appln.exe -d f280
04x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel.txt -a flashapi_ex5_lfu_no_resetBANK1FLASH.txt -b 9600 -p COM2
0

C2000 Serial Firmware Upgrader
Copyright (c) 2013 Texas Instruments Incorporated.  All rights reserved.

getting comm state
building comm DCB
adjusting port settings

What operation do you want to perform?
         1-DFU
         2-Erase
         3-Verify
         4-Unlock Zone 1
         5-Unlock Zone 2
         6-Run
         7-Reset
         8-Live DFU
         0-DONE
```

Figure 13. LFU Serial Command Invoked From Windows Command Prompt

10. Control will pass to flash kernel from the application and, once LDFU (8) command is selected, the kernel will receive and program the application in BANK0.
11. After transfer is complete, enter 0 to indicate end of command operations.



```
0
6f
1
9a
6
0
0
0
Bit rate /s of transfer was: 6741.171387
What operation do you want to perform?
         1-DFU
         2-Erase
         3-Verify
         4-Unlock Zone 1
         5-Unlock Zone 2
         6-Run
         7-Reset
         8-Live DFU
         0-DONE
0
Exiting the Application.

C:\ti\C2000Ware_3_01_00_00\utilities\flash_programmers\serial_flash_programmer>
```

Figure 14. Successful Completion of LFU Command to Program Flash Bank 0

   The kernel will also update the KEY and revision number in BANK0 sector 2. Now the static image in programmed BANK1 and application image is programmed in BANK0.

12. At this point, as before, the user can reset the board, to see LED1 start blinking.

6.3 Live Firmware Update of Application

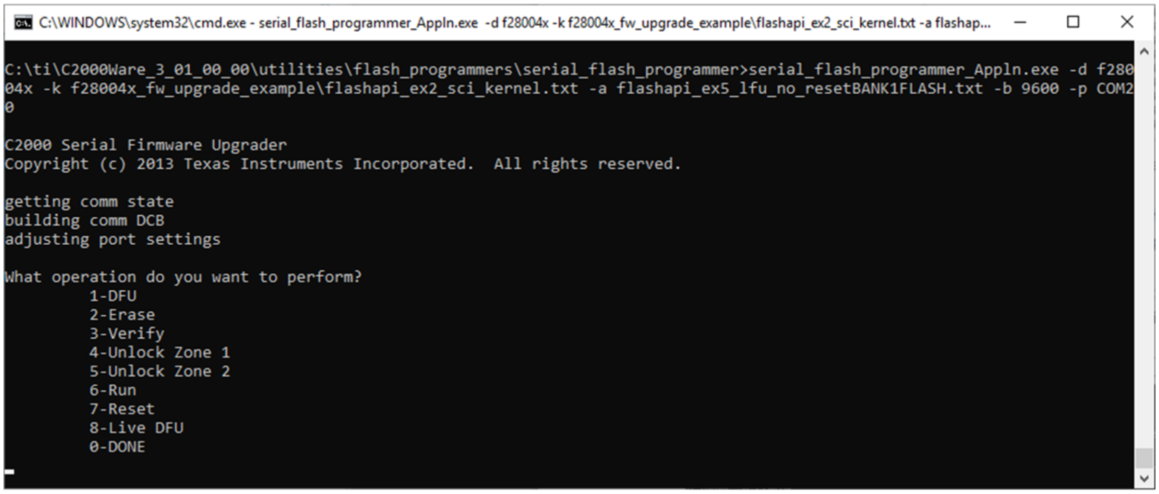
After programming the static contents, disconnect the debugger and set the boot mode switches to flash boot mode. When the device boots up, it will jump to Flash. The default flash entry point is 0x80000, which is where the static code (Bank selection logic + SCI Flash Kernel) has been programmed in Bank0. The bank selection logic will execute and determine that valid images exist in both Banks 0 and 1 (based on KEY and revision number). Bank0 will be selected to run because, in Section 6.1, it was programmed later so it will be deemed the newest version based on the REV field. So the application firmware in Bank0 will be executed.

The application blinks LED1 every 1 second. At the same time the application also monitors the serial port to check if it is getting any image for live firmware update.

To perform live firmware updates, build the updated application for BANK1 configuration and execute the command shown in Figure 15 from the host PC in the command line. Enter ‘8’ for LDFU when the menu is listed. serial_flash_programmer_appln.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel.txt -a flashapi_ex5_lfu_no_resetBANK1FLASH.txt -b 9600 -p COMx

where, x = COM port corresponding to the JTAG connection between the PC and the target board.

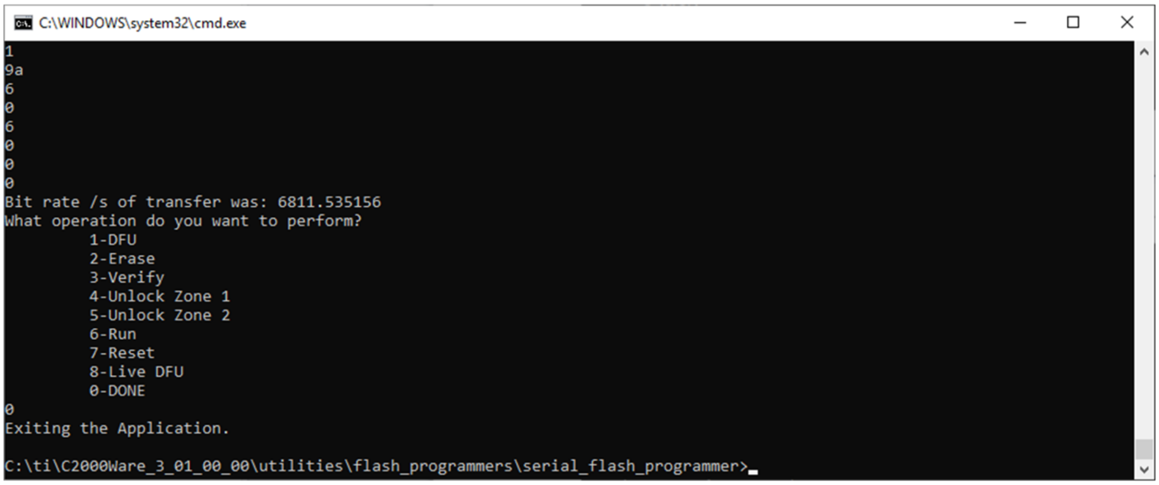
flashapi_ex5_lfu_no_resetBANK1FLASH.txt is generated by building the CCS project flashapi_ex5_lfu_no_reset in build configuration BANK1_FLASH.


C:\WINDOWS\system32\cmd.exe - serial_flash_programmer_Appln.exe -d f28004x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel.txt -a flashap...

```
C:\ti\C2000Ware_3_01_00_00\utilities\flash_programmers\serial_flash_programmer>serial_flash_programmer_Appln.exe -d f280
04x -k f28004x_fw_upgrade_example\flashapi_ex2_sci_kernel.txt -a flashapi_ex5_lfu_no_resetBANK1FLASH.txt -b 9600 -p COM2
0

C2000 Serial Firmware Upgrader
Copyright (c) 2013 Texas Instruments Incorporated.  All rights reserved.

getting comm state
building comm DCB
adjusting port settings

What operation do you want to perform?
         1-DFU
         2-Erase
         3-Verify
         4-Unlock Zone 1
         5-Unlock Zone 2
         6-Run
         7-Reset
         8-Live DFU
         0-DONE
```

Figure 15. LFU Serial Command Invoked From Windows Command Prompt

If it receives an image, control will jump to flash kernel in Bank0 and update the firmware image on Bank1. After transfer is complete, enter 0 to indicate end of command operations.



```
1
9a
6
0
6
0
0
0
Bit rate /s of transfer was: 6811.535156
What operation do you want to perform?
         1-DFU
         2-Erase
         3-Verify
         4-Unlock Zone 1
         5-Unlock Zone 2
         6-Run
         7-Reset
         8-Live DFU
         0-DONE
0
Exiting the Application.

C:\ti\C2000Ware_3_01_00_00\utilities\flash_programmers\serial_flash_programmer>
```

Figure 16. Successful completion of LFU Command to Program Flash Bank

The user may note a couple of key observations here:

1. When the new image is being downloaded to Bank1, the application continues to run on Bank0 (LED1 continues to blink at the usual 1s rate). This is because LFU processing occurs in a background loop, not the SCI Receive ISR. This allows other interrupts, such as the CPU Timer ISR that toggles the LEDs, to be serviced.
2. Once the image download to Bank1 is complete, execution jumps to the code entry point of Bank 1’s application image (codestart), which branches to the C runtime initialization routine and then calls main(). Variable initialization in the C runtime initialization routine, and peripheral initialization in main() is excluded to speed up switchover time. Application execution continues, with the execution of the image on Bank1. Note now that LED2 is blinking, and at a faster rate (400ms), indicating that device reset did not occur during switchover.
3. Perform a device reset, and LED2 will blink, but at the usual 1s rate.
4. This process can be repeated multiple times to update the image in alternate banks. Note that the LFU command can be sent multiple times before a device reset is applied. For example, if after an LFU, LED2 is blinking at 400ms rate, LFU can be performed to update Bank0, after which LED1 will blink at 400ms rate, and so on.

Figure 17 illustrates the flow described above.

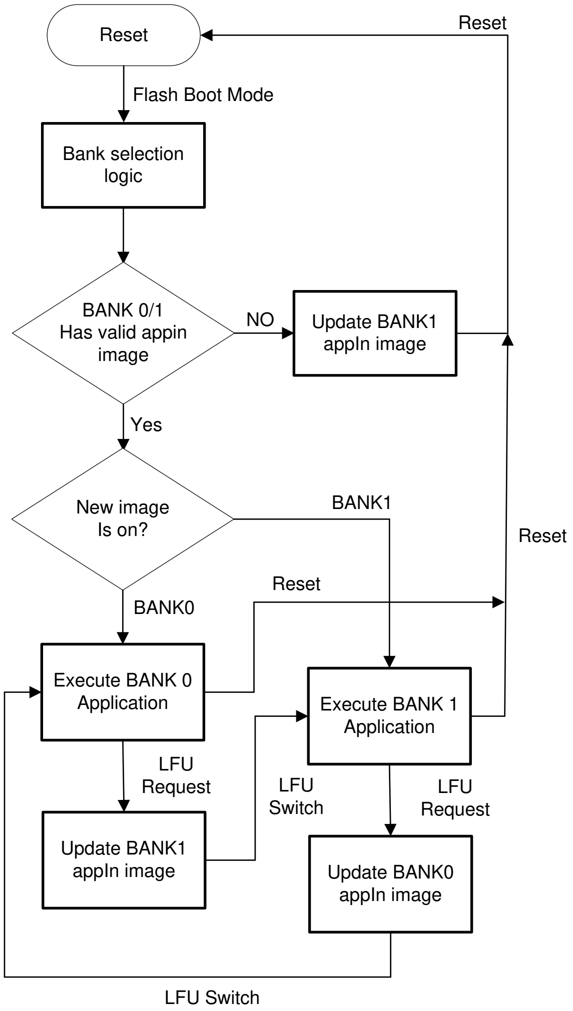


Figure 17. LFU Code Flow Diagram

6.4 Limitations and Troubleshooting

- One point for the user to note is that since the Bank Selection Logic resides in Bank0, if Flash corruption occurs when the Application is being updated on Bank0, it is possible that the Static contents of Bank0, although located at different sectors, end up corrupted as well. This would include the Bank selection logic + SCI Flash Kernel + Flash APIs (if running from Flash). The user would then have to repeat the steps involved in programming static code to get the system operational again.
- While programming the Flash kernel in Section 6.2, the erase settings for Flash should be set to “Necessary Sectors only”, otherwise, while Programming Kernel on BANK1, the Application on BANK1 will be erased.
- The Kernel “maintains” the firmware revisions, considering the most recently downloaded image as the latest version. In practice, firmware revisions may be maintained on Applications themselves.
- In this example, the Application file names mention the Flash Bank they are intended for. This is because each Application is built for a specific bank, with the dependencies in the linker command file for codestart, .text, .cinit, and so forth. Thus, the user needs to keep track of the Bank a specific Application image is intended for.

IMPORTANT NOTICE AND DISCLAIMER

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

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

TI’s products are provided subject to TI’s Terms of Sale (www.ti.com/legal/termsofsale.html) 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.

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