Application Note

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 multiple Flash banks, such as the TMS320F28003x.

Table of Contents

List of Figures

Trademarks

C2000™ is a trademark of Texas Instruments.
All trademarks are the property of their respective owners.

1 Introduction

In applications like server power supplies, 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. 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 Key Innovations

There are two key innovations in the proposed solution:

- A Compiler LFU initialization routine that initializes variables in the new firmware, *executing in tandem* with Control ISRs of the old firmware. Preceding this, the download and installation (programming) of the new firmware occurs, also in tandem with the Control ISRs of the old firmware. These steps can be time-consuming, but the tandem execution means that application functionality is not compromised while the necessary steps for LFU execute.
- Switchover to new firmware at the best time (beginning of the idle time) with interrupts disabled for a very short period of time (< 100 CPU clock cycles), made possible through *hardware LFU support* in the MCU (swapping of interrupt vectors and function pointers). This step is very short, and decoupling this from the previous step allows activation of new firmware to be very fast.

3 Building Blocks for LFU

The LFU design consists of several building blocks:

- A desktop Host application that issues an LFU command
- An LFU bootloader on the target device's Flash to communicate with the Host and implement LFU
- A communication peripheral connecting the host to the target (for example, Serial Communication Interface (SCI))
- An LFU-ready application to be downloaded and activated
- Compiler with LFU support
- The target MCU with LFU related hardware support, such as Flash memory with multiple physically separate Flash banks, and so forth. Dual or more Flash banks allow application firmware resident on one Flash bank to execute, while the other Flash bank is being updated.

4 Details of Proposed Solution

4.1 Flash Bank Organization

The dual-bank Flash is partitioned as shown in Figure 4-1. Two sectors in each bank are allocated to the LFU 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 with application firmware, and which bank contains the most recent application firmware version. Thus, bank selection logic is the entry point of the software system. The SCI kernel implements the transfer of the image from the host, and programming of Flash through Flash programming APIs (resident 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/verification is about to begin
- KEY– The firmware in a bank is considered valid if this location contains a specific pattern
- Firmware Revision number (REV) – used by the bank selection logic to determine the newer firmware version between banks 0 and 1

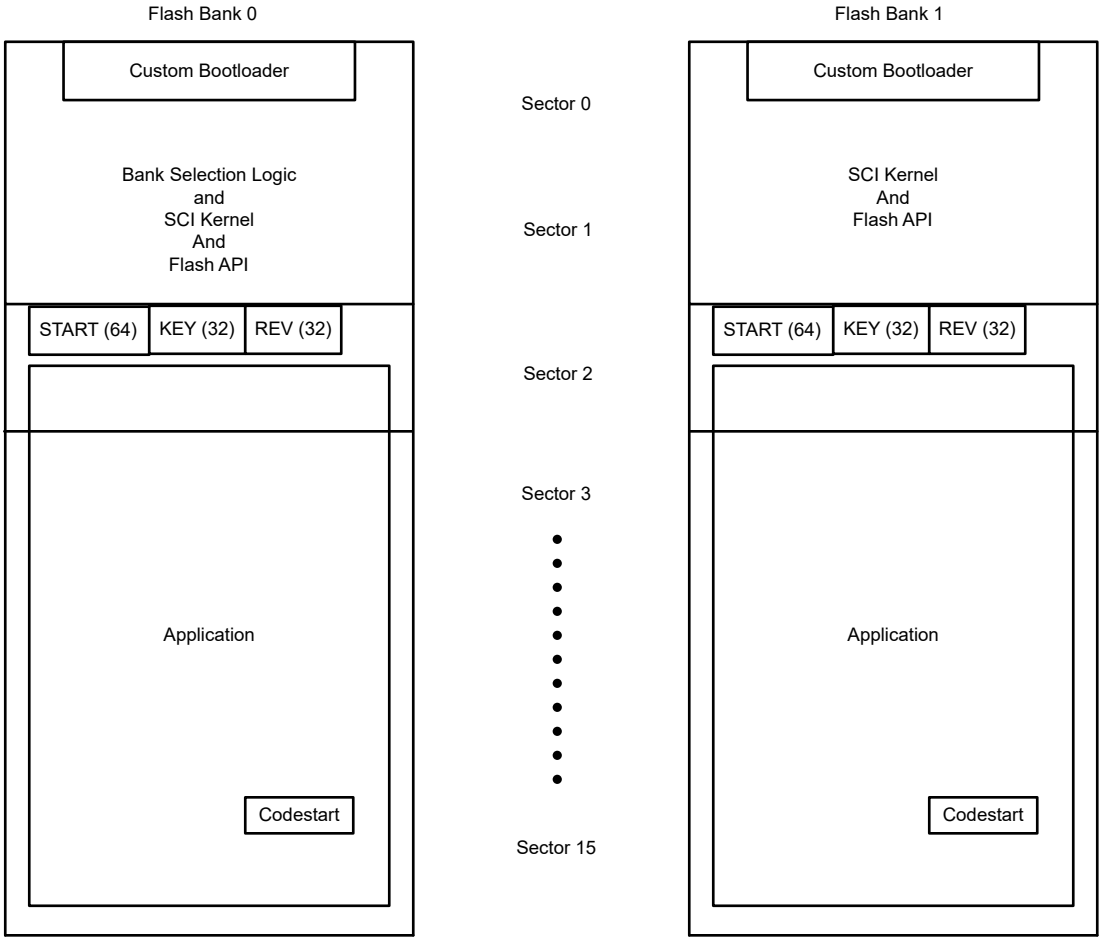

Figure 4-1. Dual-Bank Flash Partitioning

4.2 LFU Concepts and Factors Impacting Performance

The key considerations when creating LFU-ready firmware are operational continuity during LFU, and LFU switchover time. These two are closely related. Operational continuity is achieved through the persistence of state, keeping existing static and global variables in RAM at the same addresses between firmware versions, and avoiding re-initialization of those variables when the new firmware is activated. LFU Compiler support enables this.

Activating the new firmware involves branching to the LFU entry point of the new firmware, executing the compiler's LFU initialization routine, arriving inside the main() of the new image, and performing additional initialization. This is where interrupts are briefly disabled, and initialization that needs interrupts to be disabled is performed (for example, 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, 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 active and inactive banks need to be tracked, and you need to use the linker command file to create application firmware targeted to specific banks.

Function pointers and Interrupt vectors need to be re-initialized inside main(), since their locations are 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. LFU-specific hardware features (Interrupt vector swapping, RAM block swapping) enable a *significant* reduction in the LFU switchover time.

4.3 Hardware Support for LFU

4.3.1 Multiple Flash Banks

To enable seamless transfer of control from old firmware to new firmware multi-bank Flash support is a critical feature. Flash technology used on the devices does not permit simultaneous Reads and Writes to a Flash bank, so this model allows one bank to execute firmware, and other bank(s) to be programmed.

4.3.2 Interrupt Vector Table Swap

Multiple components of switchover time were analyzed to assess optimization opportunities and the update of the interrupt vector map entries was found to be one of the prime factors impacting switchover time. The number of vectors to be updated varies from a few to the entire table (192 vectors). The update for a single entry can take about 5 cycles and hence the update of the vector table itself can take up to 960 cycles (4.8 us at 200MHz).

In order to reduce the switchover time, a shadow vector memory and capability to swap it with active vector memory is implemented. The switchover code can update the shadow vector memory when the application execution is in progress. Once the vector memory is updated, the swap be completed in one clock cycle. Both memories can be used in ping-pong fashion for successive software upgrades.

A representative implementation of Interrupt vector swap is shown in Figure 4-2. Figure 4-2(a) is the configuration before swap and Figure 4-2(b) after the swap. The overall vector memory is divided into two blocks – Block A that spans from address 0x0000_0D00 to 0x0000_0EFF and Block B that spans from address 0x0100_0900 to 0x0100_0AFF. Block A holds the active vector table and Block B holds the shadow vector table. During LFU, shadow memory entries get updated before switchover, and the swap is executed during switchover. This reduces the switchover time from a maximum of 960 cycles to a single cycle.

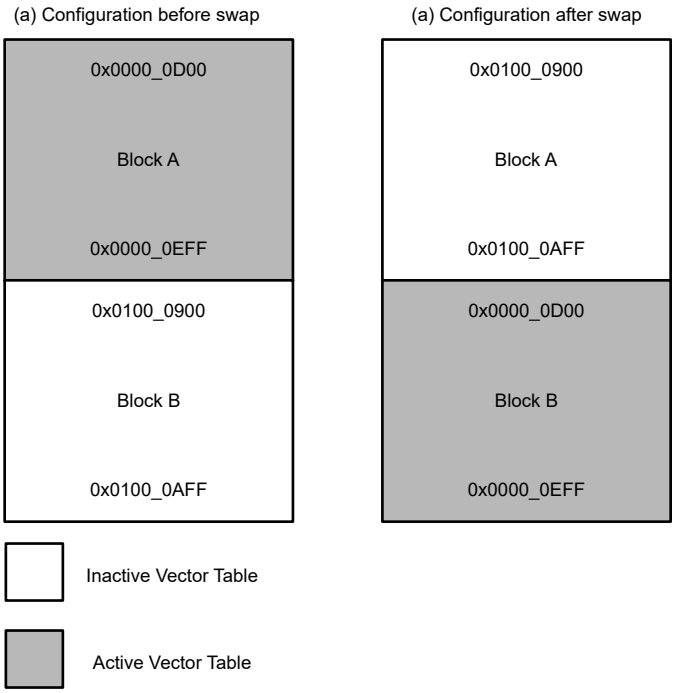

Figure 4-2. Interrupt Vector Swap

4.3.3 RAM Block Swap

Similar to Vector Table Swap physical RAM memory blocks can be swapped as shown in Figure 4-3.

If physical memory Block 1 contains function pointers for the current firmware, then the same relative locations in physical memory of Block 2 can be populated with function pointers of the new firmware before LFU switchover. During LFU switchover, a simple swap operation is initiated by the user application code that takes just 1 CPU clock cycle. This allows user-application code to maintain function pointers in LS0, yet have two different physical blocks that map to the LS0 address range. After the swap, the physical RAM block previously mapped to the Block1 address space would now be mapped to the Block0 address space, and vice versa, enabling seamless function pointer access in the new firmware.

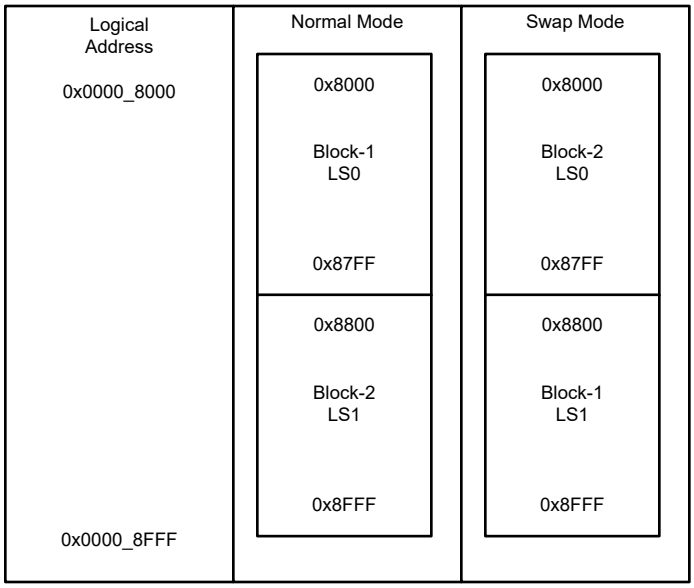

Figure 4-3. RAM Block Swap

4.3.4 Hardware Register Flags

Execution can reach main() either after a device reset and through a C initialization routine call to main(), or during LFU, from the LFU entry point calling the LFU Compiler initialization routine and then main(). In the former case, device specific initializations in main() need to occur, but not in the LFU case. To distinguish the two, a hardware flag in a register is supported, which indicates whether LFU is currently active or not.

4.4 LFU Compiler Support

The Compiler provides the following support for LFU:

1. LFU Initialization routine, named __TI_auto_init_warm
2. LFU attributes for variables
3. LFU modes

The Compiler requires the old firmware executable to be provided as a reference image while building the new firmware executable. This allows the compiler to identify common variables and their locations, and also identify new variables and deleted variables.

The compiler defines two new LFU 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 Compiler’s LFU initialization routine, named __TI_auto_init_warm(). Examples of 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;

With the above assignments, the generated memory map file contains ".TI.bound" sections corresponding to the preserve variables and a single ".TI.update" section where all the update variables are collected into.

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 has the following properties:

- With a reference (*old*) image provided, common variables do not even need to be specified as *preserve*. This will be the default attribute for common variables, and they are not initialized in the Compiler's LFU initialization routine. Hence, it maintains the state.
- Any new variables that do not have any attributes specified are assigned addresses, but these variables are not initialized in the Compiler's LFU initialization routine. The Compiler's LFU initialization routine initializes variables only when declared with the *update* attribute.

4.5 Application LFU Flow

The detailed steps associated with LFU are outlined below:

1. *Boot to Bank Selection Logic and Execute Application*: 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 for valid applications in the Flash banks, picks the recent most version, and branches to the corresponding firmware version's entry point (codestart). The entry point is the gateway to the C runtime initialization routine and main() of the application.

2. *Initiate LFU*: The user invokes LFU on the target MCU through a host initiated LFU command from the host side.

3. *Receive LFU command in Application*: (Step 1 in Figure 4-4) The application receives the LFU command in its SCI Receive Interrupt ISR (CommandLogISR).

4. *Parse LFU command and Branch to LFU Bootloader*: (Step 2 in Figure 4-4) A specific background task function (Run LFU) parses the LFU command, disables the SCI interrupt, and branches to a LFU function in the LFU bootloader in the same Flash bank, located at a fixed address.

5. *Download new Firmware and Program to Flash*: (Step 3 in Figure 4-4) The LFU function in the LFU bootloader receives an application image from the host and programs it into the Inactive Flash bank. At this point, background task functions in the old firmware have stopped executing, however Control ISRs from the old firmware continue executing to keep the application functionality unaffected.

6. *Branch to LFU entry point of new Firmware*: (Step 4 in Figure 4-4) When the new Firmware is successfully downloaded and programmed, the custom bootloader branches to the LFU entry point (C_int_LFU) of the new application image, located at a fixed address for each Flash bank, and is different from the regular Flash boot entry point (codestart).

6. *Execute Compiler LFU Initialization routine and branch to main() of new Firmware*: The function at the LFU entry point does the following:

a. The compiler's LFU initialization routine (__TI_auto_init_warm) is invoked. This initializes any variables that have been indicated as needing initialization. (Step 5 in Figure 4-4).

b. A flag is set in a hardware LFU register to indicate LFU is in progress.

c. main() is called. (Step 6 in Figure 4-4).

7. *Perform LFU specific initializations in main() prior to switchover*: (Step 7 in Figure 4-4) In main(), initialization progresses depending on whether or not LFU is in progress, by checking the above-mentioned flag.

If the flag is set, an LFU initialization function (Init_LFU) copies over any user indicated code from Flash to RAM memory. Next, it updates the inactive interrupt vector table with the interrupt vector locations corresponding to the new firmware. Similarly, a set of inactive function pointers is also updated corresponding to function pointer locations in the new firmware.

8. *Wait for optimal LFU switchover point*: (Step 8 in Figure 4-4) A simple state machine with software flags is used to determine the end of a Control ISR and the beginning of idle time. This is the optimal time to switchover (IdentifyIdleTime), since it allows maximizing the utilization of idle time between Control loop interrupts.

9. *Execute the LFU switchover*: (Step 9 in Figure 4-4) It is important to note that even though execution is already inside main() of the new firmware, old Control loop ISRs are still executing to keep the application functionality unaffected. Once the optimal LFU switchover point is identified, the LFU switchover steps (ActivateApp) occur. First, global interrupts are disabled. Hardware Interrupt vector table swapping and RAM block swapping are executed. Then the stack pointer is re-initialized, and global interrupts are re-enabled. Now, ISRs and background task functions of the new firmware begin executing, representing completion of the LFU switchover. Since global interrupts are disabled for a short duration, an interrupt occurring during this time would continue to stay latched and interrupt the CPU when global interrupts are re-enabled.

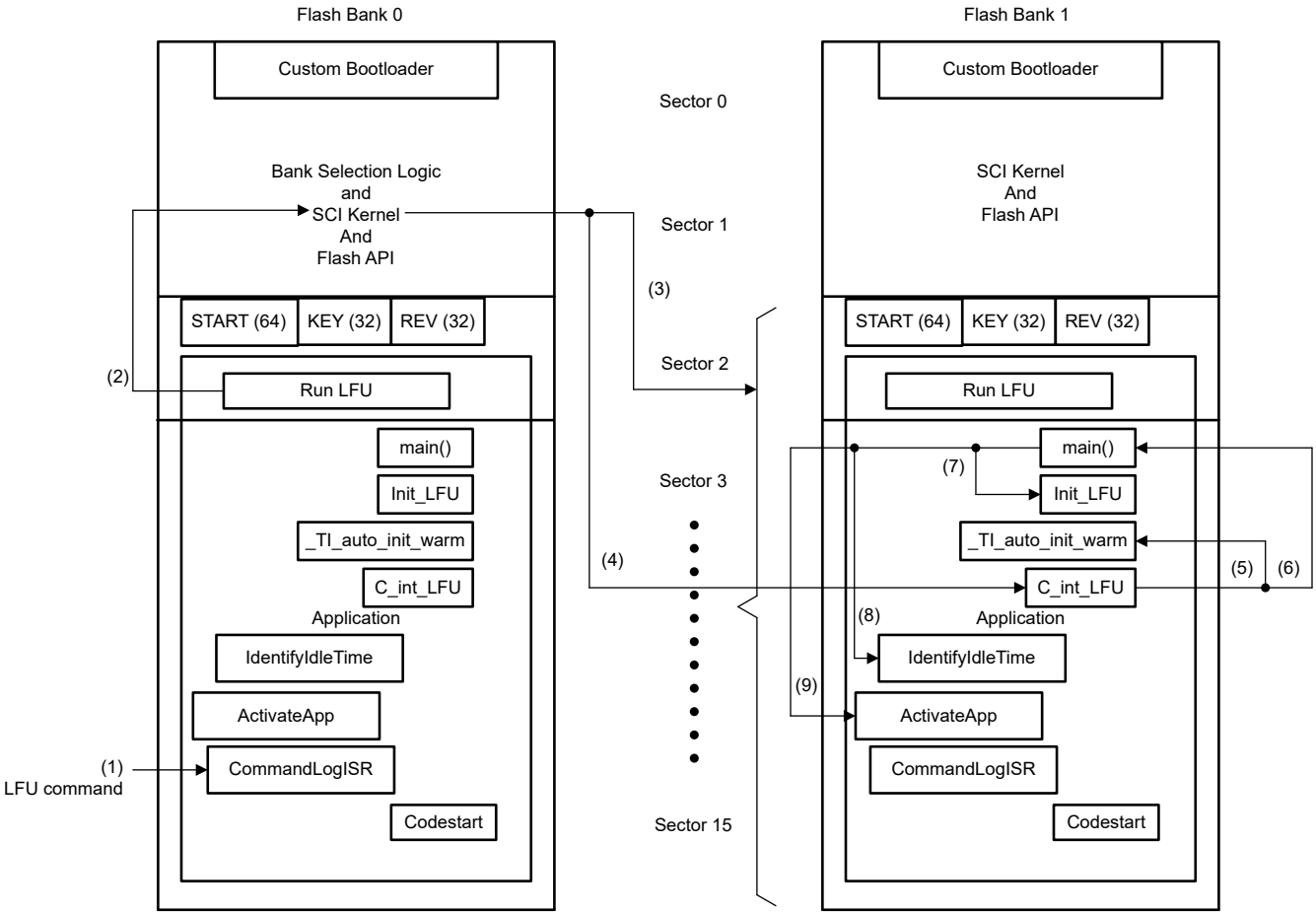

Figure 4-4. Application LFU Flow

5 Results and Conclusion

In Figure 5-1, the steps that occur prior to the LFU switchover are depicted: download firmware and program Flash bank, LFU Compiler Initialization routine (the function __TI_auto_init_warm()), then LFU specific initializations in main() (the function init_lfu()).

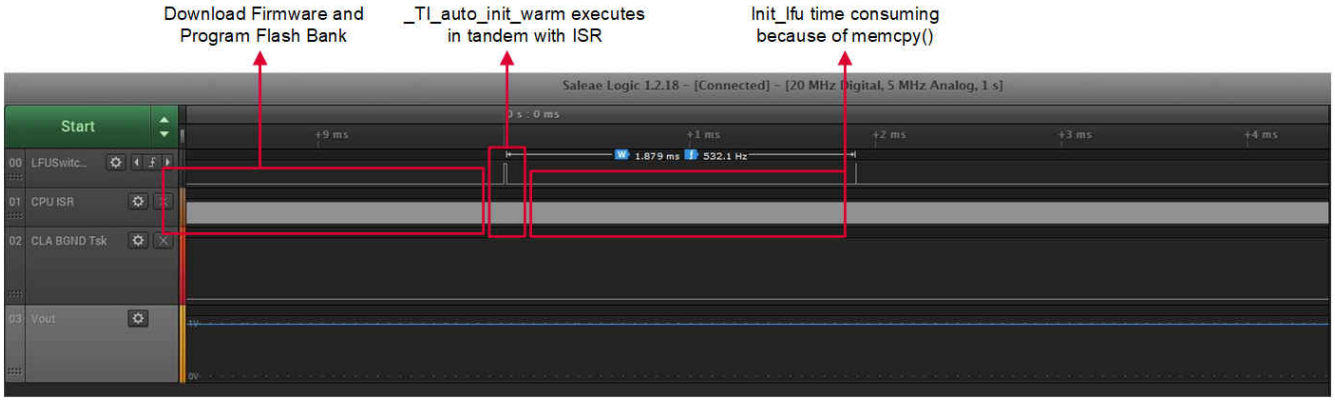

Figure 5-1. Steps Before LFU Switchover

Figure 5-2 illustrates the actual LFU switchover. The topmost waveform (00) represents the LFU switchover, the second waveform (01) represents the ISR CPU load (80% for old firmware and 40% for new firmware), and the bottom waveform (03) represents the regulated output voltage. The switchover occurs in 0.6 µs (or 72 CPU clock cycles), which includes function calls and GPIO set/reset times. Switchover occurs about 40 cycles after the ISR ends. It takes about 20 cycles to exit the ISR, and then about 15 cycles to exit the loop that is waiting for the optimal time to switchover.

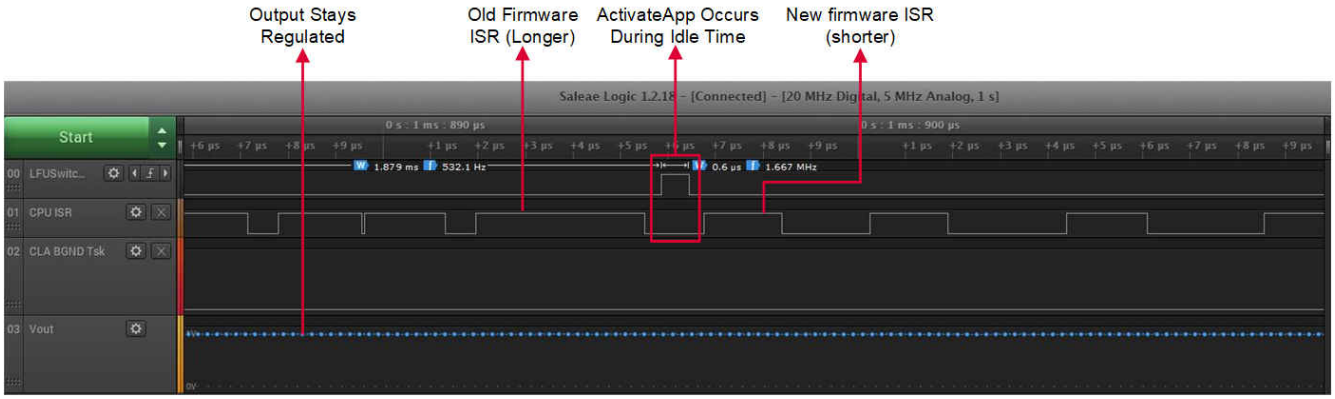

Figure 5-2. LFU Switchover Steps

This application note demonstrates the systematic implementation of LFU for real-time control applications and specifically high availability systems needing operation without downtime. Switchover to new firmware is able to be completed within 10s of CPU clock cycles with the available LFU building blocks, including a novel application LFU software flow, hardware LFU support, and Compiler LFU support.

6 Revision History

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

Changes from Revision A (August 2021) to Revision B (September 2022)	Page
• Updated the numbering format for tables, figures and cross-references throughout the document	2
• Added Section 2	2

IMPORTANT NOTICE AND DISCLAIMER

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

These resources are intended for skilled developers designing with TI products. You are solely responsible for (1) selecting the appropriate TI products for your application, (2) designing, validating and testing your application, and (3) ensuring your application meets applicable standards, and any other safety, security, regulatory or other requirements.

These resources are subject to change without notice. TI grants you permission to use these resources only for development of an application that uses the TI products described in the resource. Other reproduction and display of these resources is prohibited. No license is granted to any other TI intellectual property right or to any third party intellectual property right. TI disclaims responsibility for, and you will fully indemnify TI and its representatives against, any claims, damages, costs, losses, and liabilities arising out of your use of these resources.

TI’s products are provided subject to TI’s Terms of Sale or other applicable terms available either on ti.com or provided in conjunction with such TI products. TI’s provision of these resources does not expand or otherwise alter TI’s applicable warranties or warranty disclaimers for TI products.

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

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