ARL-TN-1272 ● AUG 2025

Serial Reprogramming without Boot-Mode-Select Hardware on Texas Instruments TMS320F283xD Digital Signal Processor

by Franklin Shedleski

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TN-1272 ● AUG 2025

Serial Reprogramming without Boot-Mode-Select Hardware on Texas Instruments TMS320F283xD Digital Signal Processor

Franklin Shedleski
DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
August 2025	Technical Note	9 February 2024	23 February 2024

4. TITLE AND SUBTITLE
Serial Reprogramming without Boot-Mode-Select Hardware on Texas Instruments TMS320F283xD Digital Signal Processor

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Franklin Shedleski

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-WE Aberdeen Proving Ground, MD 21005	ARL-TN-1272

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCID: Franklin Shedleski, 0009-0000-3939-5810

14. ABSTRACT
Modern devices employ microcontrollers to perform many computational tasks. Due to the complexity of these tasks, it is necessary to enable in-place software upgrades. Many form-factor designs do not allow for the standard JTAG programming interface; thus, the implementation of an alternative firmware upgrade path is required. Out of the box, the TMS320F2837xD supports reprogramming over serial by requiring an end user to flip dip switches on the development card. Many applications cannot expose these dip switches to the end user; therefore, an alternative method of putting the device in serial programming mode is required. In this technical note, I present a full software solution leveraging the device's boot rom so that an untrained end user can upgrade device firmware with a simple button click in a user interface.

15. SUBJECT TERMS
serial reprogramming, firmware update, TMS320F2837xD, Weapons Sciences, embedded development, embedded software

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	16

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Franklin Shedleski	(520) 691-5195

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

1. Introduction

In many microcontroller applications, the microcontroller is required to boot from flash on power cycle. This allows the device to always run an application from nonvolatile memory, preventing the need for reprogramming after power loss. The TMS320F2837xD is a dual-core digital signal processor (DSP) produced by Texas Instruments (TI) widely used in controls applications. On the TMS320F2837xD, flash boot is mutually exclusive with all other boot modes. The intended path for a firmware upgrade is to have the end user manipulate a dip switch while the device is unpowered to cause the microcontroller to boot into SCI (serial communications interface) boot mode or a similar boot mode, whereby a new program can be loaded to the flash (Roberson et al. 2023). This is not possible in our design as these dip switches are not accessible to an end user. To avoid the long lead time and cost associated with redesigning the electrical and mechanical implementations, I developed a software solution by hijacking the boot rom of the device.

2. Implementation

Our current application makes use of both cores on the TMS320F2837xD. CPU1 runs an application logic loop while CPU2 marshals logging and control data streams to a connected host device via universal asynchronous receiver/transmitter (UART) serial.

The serial reprogramming software solution is implemented in three parts. A graphical user interface (GUI) is used to connect to the device from a host PC over USB and command CPU2 to initiate the start of a firmware upgrade. Next, the microcontroller stops our application code and branches to SCI_Boot in the boot rom section. Finally, the GUI disconnects and starts TI's out-of-the-box serial reprogramming tool as though the device had just hard booted into SCI boot mode.

The TMS320F2837xD utilizes CPU1 as the main processing core. Thus, it must perform all peripheral multiplexing and has control over resetting and booting CPU2. This complicates the serial reprogramming process as the serial hardware used for reprogramming is currently in use by the second core for application-specific purposes.

Once CPU2 has received a host command to enter the reprogramming state, it stops its application code, releases the flash pump and UART peripherals, and signals CPU1 using an inter-processor communication (IPC) interrupt. CPU1 must shut down all currently running application code and multiplex the UART hardware, general-purpose input/output (GPIO) pins, and flash hardware back to itself from

CPU2. CPU1 then resets the UART peripheral configuration to prepare for it to be used by the SCI_Boot function. Finally, CPU1 multiplexes the necessary GPIO pins from CPU2 to itself.

Now, CPU1 must branch into the SCI_Boot function in the boot rom section of the microcontroller one-time programmable zone. Each CPU on the TMS320F2837xD has a read-only boot rom section, which is run on start-up to configure clocks and select the correct boot mode. Typically, this boot rom will only run the boot function associated with the current boot pin configuration. The application must boot from FLASH on power cycle, so the boot rom always selects and runs FLASH_Boot(). It can, however, still branch directly into the SCI_Boot() function to highjack its functionality. This is done by calling the function by address using raw C pointers. The address to branch to was found by loading the debug symbols for our specific chip, TMS320F2379D, from TI's C2000ware software development kit, then searching for the function definition in the expression window and disassembly. Calling SCI_Boot from our application code allows the out-of-the-box TI Serial Programming Tool to behave as though the device had properly hard-booted in SCI_Boot mode. After the TI Serial Programming Tool has finished loading a boot kernel, CPU1 resets CPU2, then branches to the kernel.

CPU2 also must be put into a state where the CPU1 can boot it. While this can be done several ways, the easiest is to hold CPU2 in reset and then allow the CPU1 boot kernel to force CPU2 to branch to SCI_Boot. However, if CPU2 is reset while CPU1 is using the UART device, it will cause a desync, which breaks the transaction. Thus, CPU2 sits in a spin–wait loop of no-op instructions while CPU1 loads its kernel into RAM. CPU1 then manually resets CPU2 and waits 100 cycles before launching the CPU1 boot kernel from RAM. CPU1 completes its firmware upgrade to flash then boots CPU2 allowing it to download a boot kernel. The CPU2 boot kernel is immediately run allowing the host to load the firmware upgrade into flash. After both cores have been upgraded, the device is power cycled by the end-user to clear the kernels from RAM and run the new firmware from flash. For a full graphical synopsis of this control flow, see Figure 1. For a minimal implementation example, see pseudo code in the Appendix.

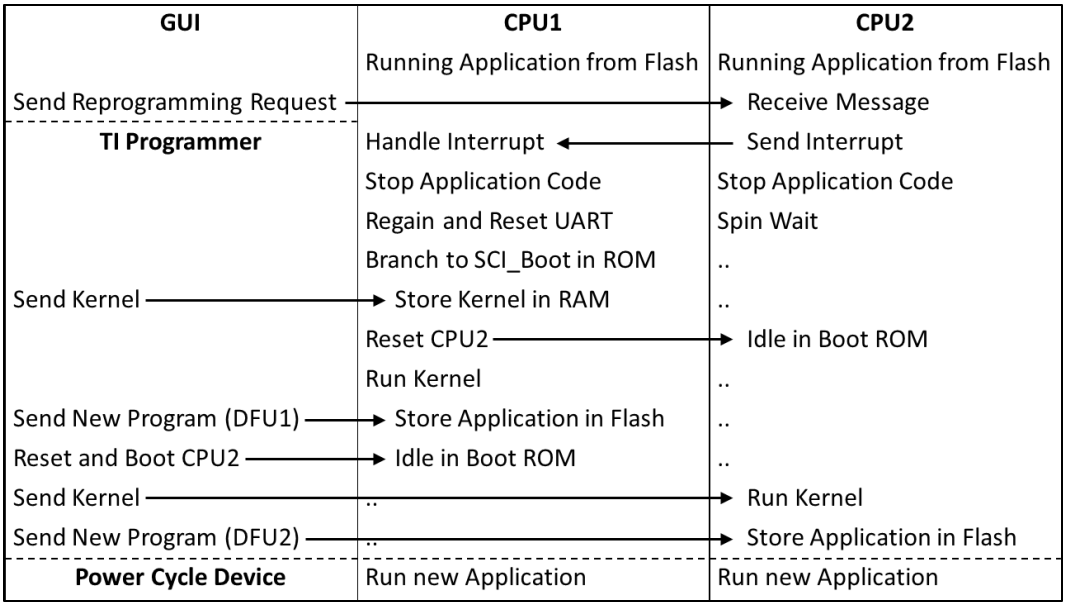

Figure 1. Control flow diagram for reprogramming process.

3. References

Roberson C, Rao S, Gudivada V. Serial flash programming of C2000 microcontrollers [application note]. Texas Instruments; 2023. p. 26.

Appendix. Minimal Pseudo Code Example

CPU2.cpp:

```
/* CPU2 message handler */
__attribute__((ramfunc)) static inline void OnMessage(Message
*msg)
{
    // stop main loop
    C2toC1_ipc.state = C2_SERIALPROGRAMMING;

    // stop running interrupts
    DINT;

    // release flash peripheral
    Flash_releasePumpSemaphore(FLASHPUMPSEMAPHORE_BASE);

    // pop ipc interrupt on cpu1
    IPC_setFlagLtoR(IPC_TYPE, IPC_FLAG_SERIAL_REPROGRAMMING);
}

/* CPU2 Example superloop */
for(;;)
{
    if(C2toC1_ipc.state == C2_RUNNING)
    {
        // application code
        ...
    }
    else if(C2toC1_ipc.state == C2_SERIALPROGRAMMING)
    {
        // state machine in serial programming. Do nothing
        asm("   NOP ");
    }
}
```

CPU1.cpp:

```
/* CPU1 interrupt for IPC Flag 0 from CPU2 */
__interrupt void IPC_Serial_Reprogramming_isr(void) {
    if(C2toC1_ipc.state == C2_SERIALPROGRAMMING) {
        // stop main loop
        C1toC2_ipc.state = C1_SERIALPROGRAMMING;

        // stop application code
        ...

        // stop interrupts
        DINT;
        Interrupt_disableGlobal();
        IER = 0x0000U;
        IFR = 0x0000U;

        // clear ipc interrupt
        IPC_unregisterInterrupt(IPC_CPU1_L_CPU2_R,
                IPC_INT_SERIAL_REPROGRAMMING);
        IPC_ackFlagRtoL(IPC_CPU1_L_CPU2_R,
                IPC_FLAG_SERIAL_REPROGRAMMING);
```

```
        Interrupt_clearACKGroup(INTERRUPT_ACK_GROUP1);
    }
}

/* CPU1 Example Superloop */
for(;;)
{
    switch(C1toC2_ipc.state)
    {
        case C1_RUNNING:
        {
            // Application Code
            ...
            break;
        }
        case C1_SERIALPROGRAMMING:
        {
            // reset registers and regain control of scia

            // partition sciA and gpios to CPU1
            SysCtl_selectCPUForPeripheral(SYSCTL_CPUSEL5_SCI,  1,
                    SYSCTL_CPUSEL_CPU1);
            BSP_GPIO_setup(UARTA_RX_PIN, GPIO_PIN_TYPE_STD,
                    UARTA_RX_PIN_CFG, GPIO_DIR_MODE_IN,
                    GPIO_CORE_CPU1);
            BSP_GPIO_setup(UARTA_TX_PIN, GPIO_PIN_TYPE_STD,
                    UARTA_TX_PIN_CFG, GPIO_DIR_MODE_OUT,
                    GPIO_CORE_CPU1);

            // reset sciA to default register values
            SCI_disableModule(SCIA_BASE);
            SCI_disableFIFO(SCIA_BASE);
            SCI_resetRxFIFO(SCIA_BASE);
            SCI_resetTxFIFO(SCIA_BASE);
            SCI_resetChannels(SCIA_BASE);
            SCI_disableInterrupt(SCIA_BASE, 0xFF);
            SCI_clearInterruptStatus(SCIA_BASE, 0xFF);
            SCI_performSoftwareReset(SCIA_BASE);
            HWREGH(SCIA_BASE + SCI_O_CCR)   = 0;
            HWREGH(SCIA_BASE + SCI_O_CTL1)  = 0;
            HWREGH(SCIA_BASE + SCI_O_HBAUD) = 0;
            HWREGH(SCIA_BASE + SCI_O_LBAUD) = 0;
            HWREGH(SCIA_BASE + SCI_O_CTL2)  = 0x00C0;
            HWREGH(SCIA_BASE + SCI_O_RXST)  = 0;
            HWREGH(SCIA_BASE + SCI_O_RXEMU) = 0;
            HWREGH(SCIA_BASE + SCI_O_RXBUF) = 0;
            HWREGH(SCIA_BASE + SCI_O_TXBUF) = 0;
            HWREGH(SCIA_BASE + SCI_O_FFTX)  = 0xA000;
            HWREGH(SCIA_BASE + SCI_O_FFRX)  = 0x201F;
            HWREGH(SCIA_BASE + SCI_O_FFCT)  = 0;
            HWREGH(SCIA_BASE + SCI_O_PRI)   = 0;
            __asm("   RPT #100 || NOP"); // delay

            // branch to boot rom to start serial programming
            // SCI_Boot function Address 0x003fedbc.
            // 0x81 param is serial alternative boot
            uint32_t runAddr = 0;
```

```
            runAddr = ((uint32_t (*)(uint32_t
mode))0x003fedbc)(0x81);
            __asm("   RPT #100 || NOP"); // delay

            // reset CPU2
            EALLOW;
            HWREG(DEVCFG_BASE + SYSCTL_O_CPU2RESCTL)= 0xA5A50001;
            __asm("   RPT #100 || NOP"); // wait for cpu2 reset
            HWREG(DEVCFG_BASE + SYSCTL_O_CPU2RESCTL)= 0xA5A50000;
            __asm("   RPT #100 || NOP"); // wait for cpu2 reset
            EDIS;

            // run the new program (the boot kernel)
            ((void (*)())runAddr)();

            // infinite loop in case something goes wrong
            // once programmer finishes it will rest core
            while(1)
            {
                __asm("   NOP ");
            }
            return;
        }
    }
}
```

List of Symbols, Abbreviations, and Acronyms

CPU	central processing unit
DSP	digital signal processor
GPIO	general-purpose input/output
GUI	graphical user interface
IPC	inter-processor communication
JTAG	Joint Test Action Group
OTP	one-time programming
RAM	random-access memory
ROM	read-only memory
SCI	serial communications interface
TI	Texas Instruments
UART	universal asynchronous receiver/transmitter

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CI TECH LIB