TMS320F281x Boot ROM Serial Flash Programming

Jeff Stafford

ABSTRACT

This application report describes the implementation of TI's Flash application program interface (API), the software interface to TI's Flash algorithms. Understanding the fundamentals of the Flash API documentation prior to using this application report is important. This document does not replace the Flash API documentation; instead, it guides you among several sets of TI documentation with the Flash API documentation being the most critical. For a complete list of related documents, see the References section in this application report.

The option available for serial-based Flash programming on the TMS320F281x devices in-circuit at this time is Spectrum Digital's SDFlash utility. This is an excellent off-the-shelf Windows® based tool for programming the TMS320F281x devices in-circuit using IEEE Standard 1149.1-1990, IEEE Standard Test Access Port and Boundary-Scan Architecture or system communications interface (SCI). It is the recommended path when a GUI-based solution will work. If your programming options do not include a Windows-based PC, this document will help you configure a custom test fixture.

Example software is provided in applicable sections of the document. The hardware used in this document includes:

- Spectrum Digital's F2812 eZdsp™ and IEEE Std. 1149.1-1990 (JTAG) emulator (XDS510™ universal serial bus (USB))
- Link-research RS-232 interface board

Project collateral and source code discussed in this application report can be downloaded from the following URL: http://www-s.ti.com/sc/techlit/spraaq2.zip.

Contents

List of Figures

List of Tables

1 Introduction

Serial-based (RS-232) Flash programming for the TMS320F2812, TMS320F2811, or TMS320F2810 (F281x) is a popular method used to program devices in-circuit. One of its main advantages is reducing the handling of the parts which reduces the risks of damage that would come from off-board Flash programming options. The SCI communication could be from a PC, production line in-circuit tester, or another processor.

Each of the steps required for Flash programming through the F281x Boot read-only memory (ROM) SCI-A option are discussed in this document:

- Interfacing with the Boot ROM to transfer kernel software and Flash algorithms into target random access memory (RAM)
- Transferring and programming application code into target Flash
- Minimizing Flash programming time

Any communication option supported by the board design can be used with the custom Flash programming API provided by TI: *Download TMS320F2810, TMS320F2811 and TMS320F2812 Flash API* (http://www-s.ti.com/sc/techlit/sprc125.zip). The F281x Boot ROM provides options to transfer the Flash API to RAM using the SCI, serial peripheral interface (SPI), or parallel general-purpose input/output (GPIO). For other communication interfaces, you can provide these in a one-time programmable (OTP) memory or in a protected sector of Flash. See Figure 1 for an illustration of these bootloading options.
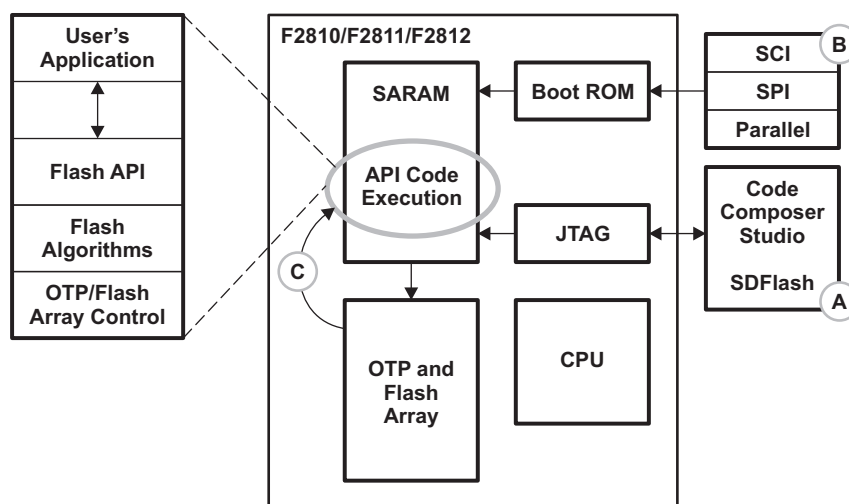
Figure 1. F281x Flash Boot-Loading Options

This application report uses the SCI-A Flash boot-loading option of the Boot ROM.

The option available for SCI-based Flash programming on the TMS320F281x in-circuit at this time is Spectrum Digital's SDFlash utility. Spectrum Digital describes the software tool as a Windows GUI based utility that allows you to flash program a DSP Target using a Spectrum Digital JTAG Emulator. Each target requires Flash programming algorithms specific to that specific DSP and the Flash memory on that board.

For a complete list of C2000™ Flash options, visit www.ti.com.

2 Methodology

The key components to the procedure described here are:

- The F281x Boot ROM's SCI-A
- Communication Kernel and Flash API (CKFA)
- Your application code (AppCode)

The basic procedure is:

- The CKFA is transferred to the F281x internal RAM through the SCI under Boot ROM control
- The AppCode is transferred to the F281x RAM and programmed into Flash under CKFA control

> **Note:** The CKFA code used in this application report is based on the Flash API example code that is part of *Download TMS320F2810, TMS320F2811 and TMS320F2812 Flash API* (http://www-s.ti.com/sc/techlit/sprc125.zip); therefore, the Flash API documentation is an excellent reference for the API, the example code provided with the API, and this document. Closely consider the Flash API documentation prior to any modifications to the CKFA source code.

2.1 Transferring CKFA to F281x RAM

First, transfer the CKFA binary file to its LOAD addresses in unsecured RAM using the Boot ROM SCI-A option (Figure 2). Since the F281x could be in a locked state, the CKFA code must first run from unsecured RAM. For more information, see the Code Security Module (CSM) section in the *TMS320x281x DSP System Control and Interrupts Reference Guide* (SPRU078).
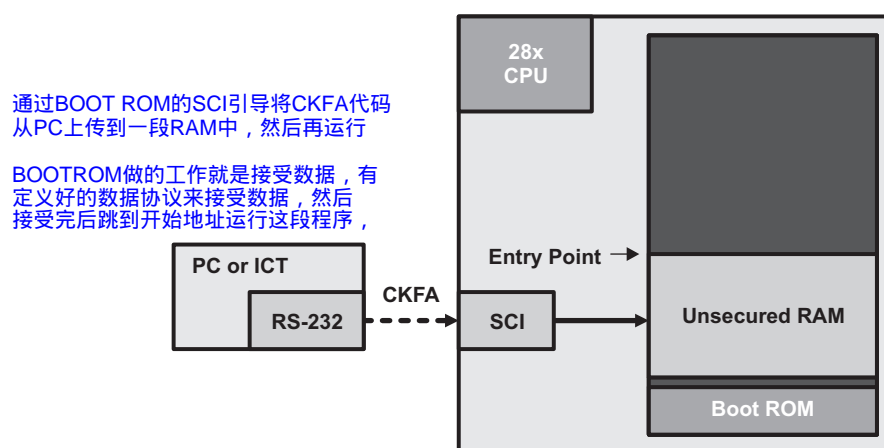
Figure 2. Transfer CKFA to RAM LOAD Addresses

Once the Boot ROM code completes the transfer of the CKFA, it transfers control of the F281x CPU to CKFA (Figure 3). After the CSM is unlocked, the CKFA code has access to all internal RAM and Flash.
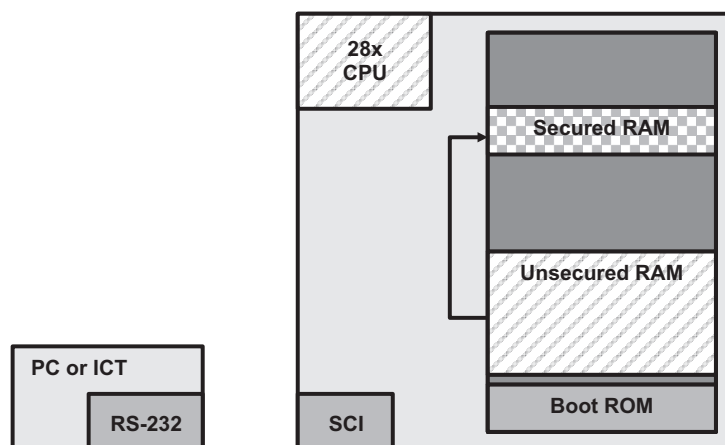
Figure 3. CKFA Transfer to RAM RUN Addresses

The LOAD addresses are defined by the linker command file that the CKFA code uses. If the unlocking process is successful, then the CKFA code copies a main portion of itself from its load addresses in unsecured RAM to its RUN addresses in secured RAM. If the unlock attempt is unsuccessful, the CKFA code transmits this status using the SCI.

Using secured RAM is not the goal of the CKFA. Instead, this procedure makes the large RAM of H0 available for the upcoming AppCode transfer. If the CKFA software was small enough, it would be transferred to the unsecured RAM blocks of MO/M1 and executed from there.

RAM

2.2   *Transferring and Programming Application Code*

With the CKFA code executing and controlling the target's SCI-A peripheral, the application code's binary file is transferred to the 4 KW RAM buffer 1 (Figure 4). Once this buffer is full, the Flash programming is started and the CKFA code transfers the next 4 KW AppCode block to RAM buffer 2 (Figure 5).

CKFA   SCI   BOOTROM
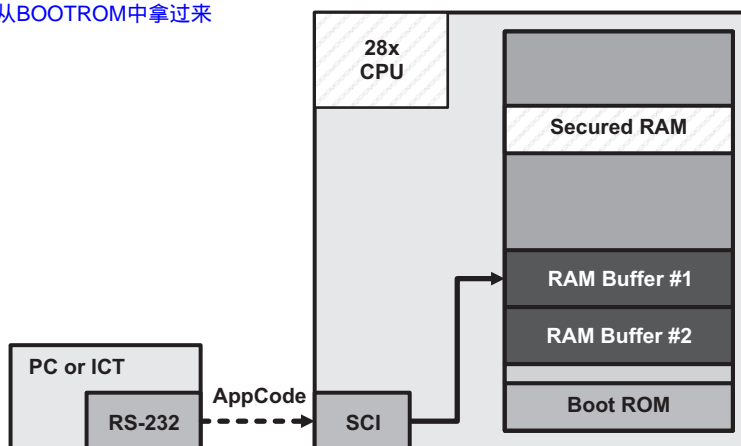
Figure 4. CKFA Transfers AppCode to RAM Buffer 1

The Flash API functions provide a call-back function that allows the SCI to continue to transfer data to RAM while Flash is being programmed. The CKFA code uses the call-back function capability to program Flash with RAM:

- Buffer 1 data while filling buffer 2 with the next AppCode block
- Buffer 2 data while filling buffer 1 with the next AppCode block

Flash programming begins when RAM buffer 1 is filled. Programming time is reduced because the SCI characters are continuously received into the SCI 16-level FIFO while the Flash is being programmed.

The approach used in this application report requires complete allocation of the entire 64 KW of the F2810's Flash. This also applies to the 128 KW of Flash for the F2811 and F2812. All Flash addresses must be allocated. Therefore, the code with this document has a fixed Flash programming start address of 0×3E8000 for the F2810, and 0×3D8000 for the F2811. See Appendix A and Appendix B for the device memory map and Flash sector addresses.
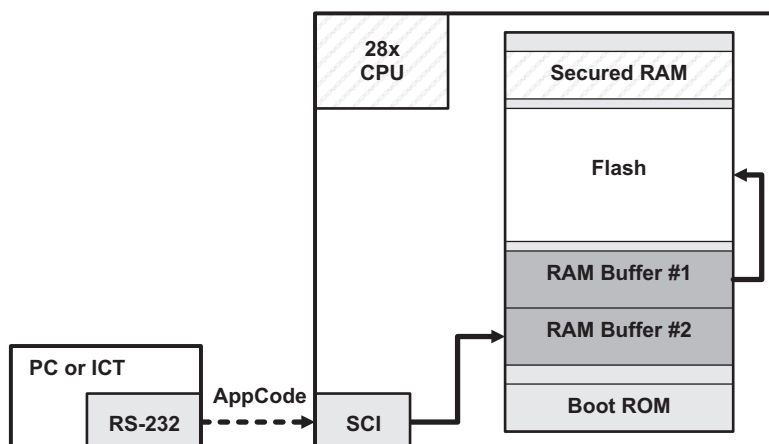
Figure 5. CKFA Starts Programming Flash

PC   FLASH

2.3   *Programming Completed and Entry Point to Application Code Defined*

When programming is complete, the F281x program counter is set to the entry point of Flash (0×3F7FF6) (Figure 5 and Appendix B). The digital signal processor (DSP) is now ready to run the application code programmed into Flash. This code executes at reset if the GPIO pins related to the boot modes are configured for Flash execution (Figure 6).
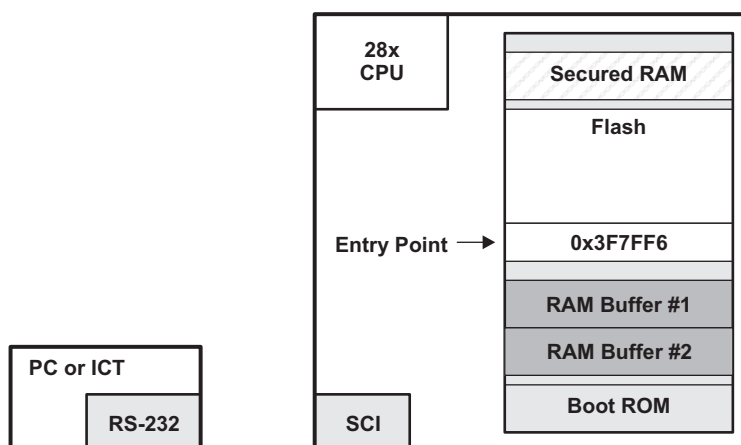
Figure 6. Flash Programming Completed

3 Procedure

Programming your application code into Flash using the approach presented in this application report consists of preparing the application and the CKFA software, establishing serial communications, and then doing the actual Flash programming. These steps are explained in detail in the next sections.

3.1 *Prepare Application Code*

To program the application code into Flash, the CKFA code receives the AppCode in binary form through the SCI and then programs the entire Flash range of addresses. The CKFA code requires the application code to do the following:

- Fill unused Flash addresses
- Create a single binary file for SCI
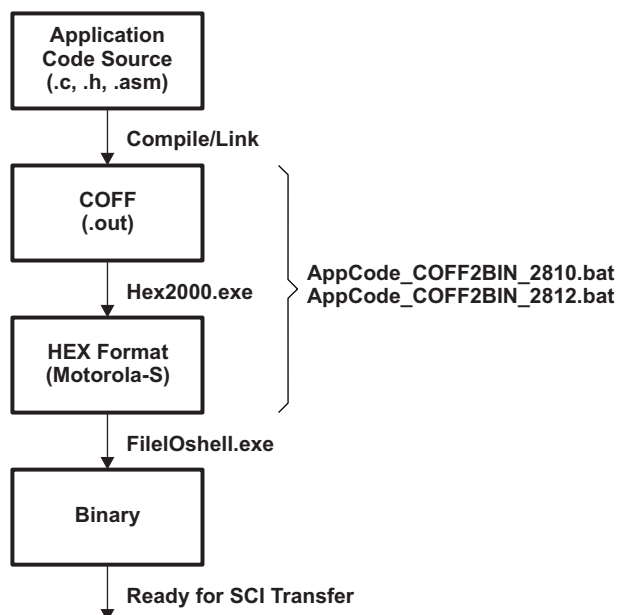
Figure 7. Overview of AppCode File Processing

3.1.1 Filling Unused Flash Addresses

The CKFA software controls the F281x DSP during the AppCode SCI transfer to RAM and subsequent Flash programming. The AppCode must be configured to fill all addresses of Flash, even unused addresses. This allows the CKFA software to process continuous blocks of data, which reduces Flash programming time.

In addition, filling unused Flash addresses with a specific constant value can add functional stability to your system. If the application software fetches an opcode from an address outside of the expected program range (software bug), you can force an illegal-instruction trap if you fill unused Flash addresses with a constant that is known to be an illegal op-code. If you fill Flash with the value of 0×FFFF, then you are guaranteed to generate an illegal opcode trap. For more information, see the *TMS320C28x DSP CPU and Instruction Set Reference Guide* (SPRU430).

Filling unused Flash with 0×FFFF also reduces Flash programming time. As the Flash programming step only writes 0s to the bits that require it, if all unused Flash bits are 1s then programming does not occur on these bits, reducing programming. TI ships F281x completely erased, which means every Flash location is 0×FFFF upon shipping. Therefore, the unused Flash addresses on a new part are already loaded with the recommended fill value of 0×FFFF.

3.1.1.1 *Select 64 KW or 128 KW Linker Command File*

Linker command files, for both the 64 KW F2810 and 128 KW F2811/F2812, are provided with this application report. Figure 8 shows that the F2810.cmd file is included in the AppCode project, but is excluded from the build process. The icon next to the F2810.cmd file is missing the down arrow, indicating that the F2810.cmd file is excluded from the build process.
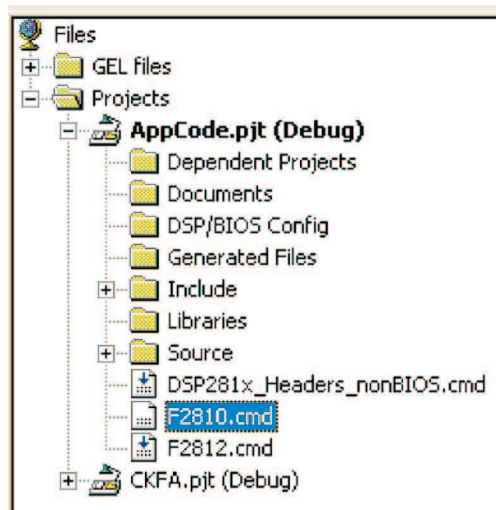
Figure 8. AppCode Project for 128 KW Flash

You can apply specific build options to each file in a project, such as excluding it from the project build. To set the individual file build options, right-click on a file, and select *File Specific Options* from the context menu. By unchecking the *Exclude file from build* option in the Build Option dialog, the F2812.cmd file is included in the next project build (Example 12). Reverse this process to exclude the F2810.cmd file from the next build. You only want the F2812.cmd or the F2810.cmd file, depending on your application software memory requirements.
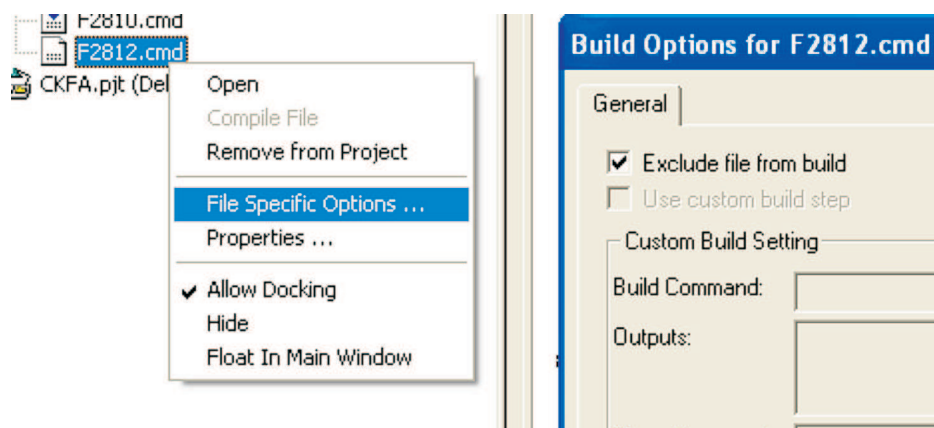
Figure 9. Excluding a Linker Command File From Build

The Code Composer Studio™ software project needs to be rebuilt, compiled, and linked after any source code or project build option modifications.

3.1.1.2 Using Linker to Fill Unused Flash Addresses

Figure 7 illustrates the build process used by the AppCode to go from source code to a single binary file. The linking stage can fill unused Flash addresses. The linker uses a command file to define the memory range to the target processor. It then directs the program and data sections from the AppCode into these defined memory ranges. If these ranges are not completely used, a fill value can be used to make sure that all addresses have been loaded. Example 1 shows how the fill value of 0×FFFF is being used to fill any unused addresses in the memory ranges named: FLASHE, FLASHD, and FLASHC. Note that FLASHB does not have a fill value. That is discussed in the next section when the hex converter fills this Flash section with the same 0×FFFF fill value.

Example 1. Linker Command File for Filling Unused Addresses With 0xFFFF

```
MEMORY
{
PAGE 0:    /* Program Memory */
   RAML0      : origin = 0x008000, length = 0x001000                      /* on-chip RAM block
L0 */
   FLASHE     : origin = 0x3E8000, length = 0x004000,  fill =0xFFFF    /* on-chip FLASH */
   FLASHD     : origin = 0x3EC000, length = 0x004000,  fill =0xFFFF    /* on-chip FLASH */
   FLASHC     : origin = 0x3F0000, length = 0x004000,  fill =0xFFFF    /* on-chip FLASH */
   FLASHB     : origin = 0x3F4000, length = 0x002000                      /* on-chip FLASH */
```

Example 2 shows the MAP file output from linking the AppCode. Note that the fill value of 0×FFFF is shown in the right-most column, and this fill value is associated with the memory ranges defined in the linker command file.

The FLASHB range does not have a fill value assigned to it, and this range is shown as having 0×2000 addresses that are unused. These are filled in the next section using the hex converter.

Example 2. Linker MAP File With 0xFFFF Fill Values

```
*******************************************************************************
        TMS320C2000 COFF Linker PC v4.1.0
*******************************************************************************
OUTPUT FILE NAME:   <./Debug/AppCode.out>
ENTRY POINT SYMBOL: "_c_int00"  address: 003ec000


MEMORY CONFIGURATION

              name            origin    length      used    attr    fill
         ---------------------  --------  ---------  --------  ----  --------
PAGE  0: RAML0                 00008000  00001000  00000086  RWIX
         FLASHE                003e8000  00004000  00004000  RWIX  ffff
         FLASHD                003ec000  00004000  00004000  RWIX  ffff
         FLASHC                003f0000  00004000  00004000  RWIX  ffff
         FLASHB                003f4000  00002000  00000000  RWIX
         FLASHA                003f6000  00001f80  00001f80  RWIX  ffff
         CSM_RSVD              003f7f80  00000076  00000076  RWIX
         BEGIN                 003f7ff6  00000002  00000002  RWIX
         CSM_PWL               003f7ff8  00000008  00000008  RWIX
         ROM                   003ff000  00000fc0  00000000  RWIX
         RESET                 003fffc0  00000002  00000000  RWIX
         VECTORS               003fffc2  0000003e  00000000  RWIX
```

3.1.1.3 Using the Hex Converter to Fill Unused Flash Addresses

The hex converter (HEX2000 utility) is used to convert the COFF formatted output from the linker to an ASCII hex file. The format of this ASCII hex file can be controlled from a command file, similar to the linker. Example 3 shows the F2810 HEX2000 command file used for the AppCode. HEX2000 is documented in the *TMS320C28x Assembly Language Tools User's Guide* (SPRU513).

Example 3. AppCode HEX2000 F2810 Command File

```
AppCode.out


-map AppCode_hex.map
-o AppCode.hex
-m
-memwidth 16
-image

ROMS
{
     FLASH2810:     origin = 0x3e8000, len = 0x10000, romwidth = 16, fill = 0xFFFF
}
```

In Example 4, FLASHB has the correct fill value of 0×FFFF, and the other Flash sections filled by the linker are shown with fill values represented by symbols starting with these characters: *$fillxxx*.

Example 4. Hex Converter MAP File With 0xFFFF Fill Values

```
*******************************************************************************
TMS320C2000 COFF/Hex Converter                                          v4.3.0
*******************************************************************************

INPUT FILE NAME: <AppCode.out>
OUTPUT FORMAT:   Motorola-S

PHYSICAL MEMORY PARAMETERS
   Default data width   :  16
   Default memory width :  16
   Default output width :   8


OUTPUT TRANSLATION MAP
-------------------------------------------------------------------------------
003e8000..003f7fff  Page=0  Memory Width=16  ROM Width=16  "FLASH2810"
-------------------------------------------------------------------------------
   OUTPUT FILES: AppCode.hex [b0..b15]

   CONTENTS: 003e8000..003e80ff   .econst Data Width=2
             003e8100..003ebfff   $fill000 Data Width=2
             003ec000..003ec3b9   .text Data Width=2
             003ec3ba..003ec43f   ramfuncs Data Width=2
             003ec440..003ec458   .cinit Data Width=2
             003ec459..003effff   $fill001 Data Width=2
             003f0000..003f3fff   $fill002 Data Width=2
             003f4000..003f5fff   FILL = 0000ffff
             003f6000..003f7f7f   $fill003 Data Width=2
             003f7f80..003f7ff5   csm_rsvd Data Width=2
             003f7ff6..003f7ff7   codestart Data Width=2
             003f7ff8..003f7fff   csmpasswds Data Width=2
```

3.1.2 Create AppCode Binary File

All unused Flash addresses have been filled. This section looks at how to create an AppCode's binary file suitable for CKFA's SCI transmission and Flash programming, see Figure 7.

HEX2000 produces an ASCII formatted file that could have been used in this application report, but there are two drawbacks:

- It takes ASCII 16-bits to represent 8-bits of binary data required for Flash programming
- Performing this conversion with CKFA adds to the overall Flash programming time

FileIOShell.exe is used to convert the ASCII output of HEX2000 to binary format. This utility is configured to convert from Motorola-S record (16-bit, big endian) to binary file formats. See Appendix G for Motorola-S record format.

Example 3 shows the HEX2000 F2810 command file instructing HEX2000 to generate a Motorola-S formatted ASCII file. The syntax of this file is described below:

- AppCode.out   = COFF executable input file
- -map AppCode_hex.map   = HEX2000 MAP output file (contents shown in Example 2)
- -o AppCode.hex   = hex file output
- -m   = Specify Motorola-S record format
- -memwidth 16   = 16-bit memory width, big endian
- -image   = this allows the fill parameter to be used

The output file, AppCode.hex, becomes an input to FileIOShell which converts it to binary form ready for CKFA controlled SCI transmission and Flash programming.

Example 5. AppCode.hex (ASCII Reader), Mot-S Input File to FileIOShell.exe

```
S00600004844521B
S2223E8000C1F5003EC1F5003EC1F5003EC1F5003EC1F5003EC1F5003EC1F5003EC1F5BD
S2223E800F003EC1F5003EC1F5003EC1F5003EC1F5003EC1F5003EC0B0003EC0B5003EAD
S2223E801EC0BA003EC0BF003EC0C4003EC0C9003EC0CE003EC0D3003EC0D8003EC0DDF3
```

Example 6. AppCode.bin (Binary Reader), Binary Output From FileIOShell.exe

```
c1 f5 00 3e c1 f5 00 3e c1 f5 00 3e c1 f5 00 3e c1 f5 00 3e c1 f5 00 3e c1 f5 00 3e c1 f5 00 3e
c1
f5 00 3e c1 f5 00 3e c1 f5 00 3e c1 f5 00 3e c1 f5 00 3e c0 b0 00 3e c0 b5
00 3e c0 ba 00 3e c0 bf 00 3e c0 c4 00 3e c0 c9 00 3e c0 ce 00 3e c0 d3 00 3e c0 d8 00
3e c0 dd
```

3.1.2.1   Generating AppCode.bin From CCS

In the Code Composer Studio project AppCode.pjt, build options are configured to call the AppCode_COFF2BIN_281x.bat on every build. There are two versions of this batch file, one for the 64 KW Flash range of the F2810 (AppCode_COFF2BIN_2810.bat), and one for the 128 KW range of the F2811/F2812 (AppCode_COFF2BIN_2812.bat). Essentially these files are identical, except that they use different command files for the HEX2000 utility to define the two different Flash ranges.

Example 7. AppCode_COFF2BIN_2810.bat

```
cd debug

C:\CCStudio_v3.1\C2000\cgtools\bin\hex2000.exe AppCode_hex_2810.cmd

FileIOShell.exe -I AppCode.hex -o AppCode.bin
```

Example 8. AppCode_COFF2BIN_2812.bat

```
cd debug

C:\CCStudio_v3.1\C2000\cgtools\bin\hex2000.exe AppCode_hex_2812.cmd

FileIOShell.exe -I AppCode.hex -o AppCode.bin
```

Configure the Code Composer Studio project build options to execute the batch file that corresponds to the memory range that your project is configured for.
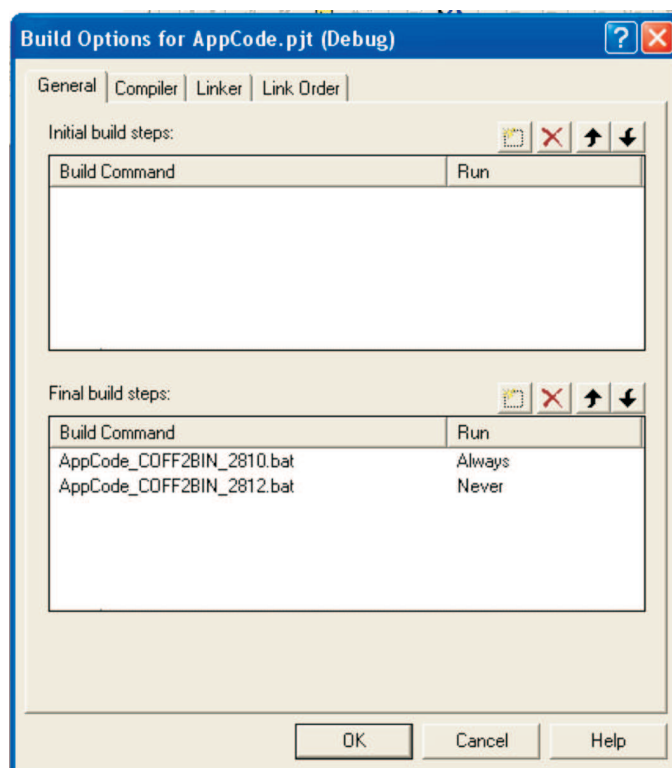
Figure 10. AppCode Code Composer Studio Project – Configuring COFF2BIN Batch File

3.1.2.2   Generating AppCode.bin Without Code Composer Studio

If Code Composer Studio is not being used, then convert either a COFF or a Motorola-S record file to the binary format suitable for CKFA SCI transmission and Flash programming. If the format of the input file is COFF, use either *AppCode_COFF2BIN_2810.bat* or *AppCode_COFF2BIN_2812.bat*. If the format of the input file is Motorola-S record, use the batch file *FileIOShell Only.bat*.

Example 9. FileIOShell Only.bat

```
FileIOShell.exe -I AppCode.hex -o AppCode.bin
```

3.1.3   Calculating Expected Checksum for Application Software

The application code's checksum can be calculated using Code Composer Studio's Flash programmer plug-in. The CKFA software uses the checksum to verify that the application code has been programmed into Flash correctly. CKFA   APP   FLASH

3.1.3.1 Calculating Checksum With Code Composer Studio

The checksum can be calculated using the Code Composer Studio on-chip Flash programmer after Flash has been programmed at least once. Flash is scanned through JTAG by the host PC, adding each 16-bit memory location to a total sum. In Figure 11, the on-chip Flash programmer calculated 0×6E13 for the application software's Flash CHECKSUM. CCS flash APP

Once the CHECKSUM has been calculated, it needs to be included into the CKFA software so that it can verify the Flash after the application code has been programmed. CKFA CKFA APP FLASH
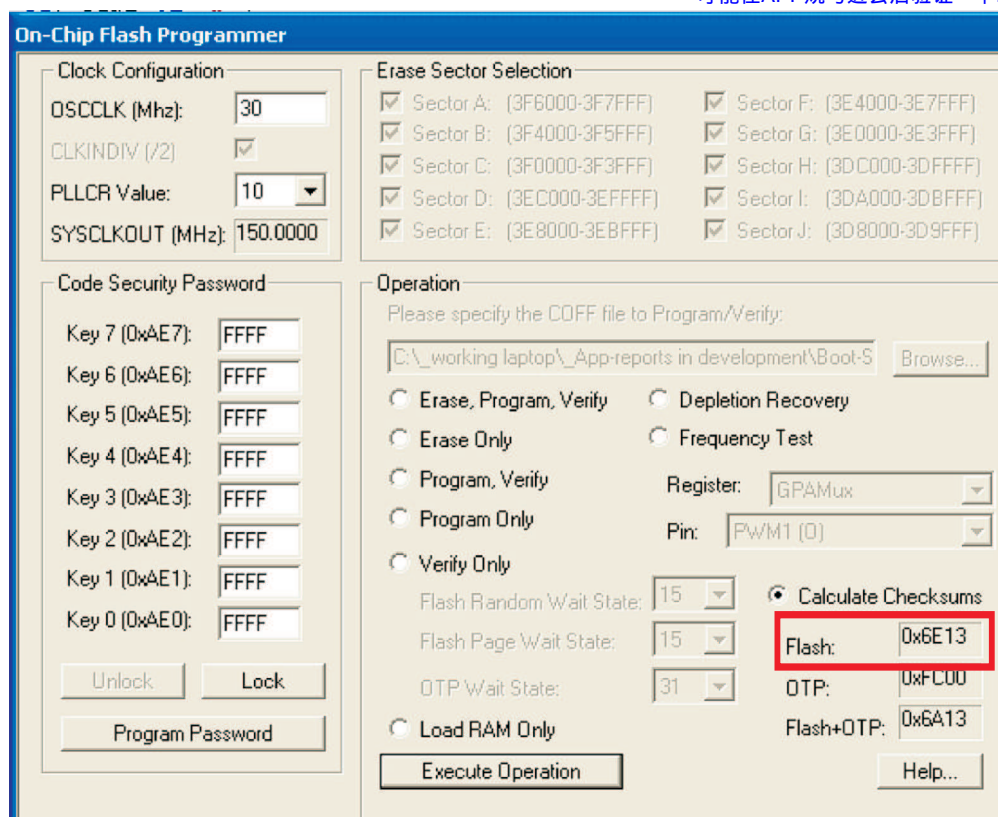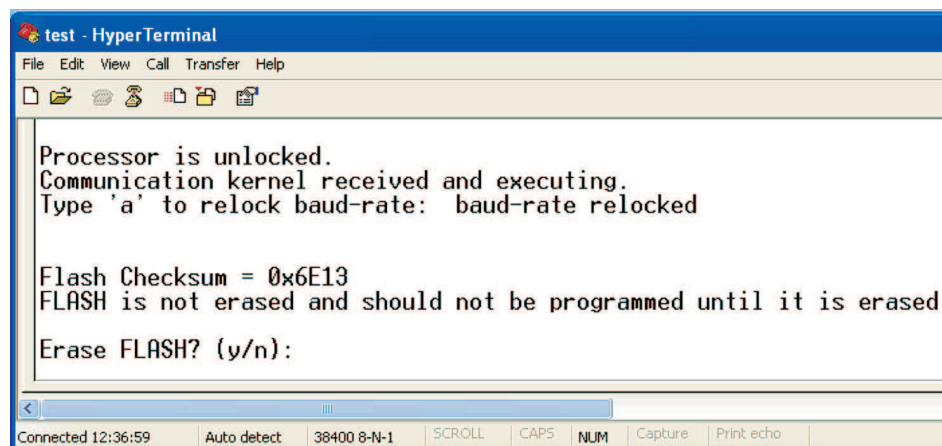
Figure 11. Code Composer Studio On-Chip Flash Programmer Calculating CHECKSUM

3.1.3.2 Calculate Checksum Without Code Composer Studio

The CKFA calculates the Flash checksum at the start of its execution to determine whether the Flash is in an erased condition. This calculation determines the AppCode checksum without using Code Composer Studio. The AppCode must be programmed into Flash using the CKFA. After Flash programming, the CKFA responds that there was a checksum error since the calculated value is different than the expected value, but the calculated value is transmitted by the CKFA. This value represents the AppCode checksum and can then be included in the CKFA software for verification of subsequent Flash programming.

CKFA   flash   APP   CKFA   flash   CKFA

Figure 12. CKFA Calculating AppCode Checksum at Start-Up

3.2   Prepare CKFA

The CKFA code consists of the communication kernel and the Flash API. As described earlier, the CKFA is transferred to the F281x RAM using the F281x SCI-A boot code. Once it is transferred to RAM, the SCI-A boot code transfers CPU control to the CKFA code so it can transfer the application code into RAM, and then program it into Flash. To do this, the following steps are required:

1. Configure the F281x phased-locked loop (PLL) for the desired CPU clock rate.

    The F2812 eZdsp uses a 30 MHz oscillator; therefore, the PLL was configured accordingly to output a 150 MHz CPU clock (maximum rate).

    Note: As stated in the *TMS320x281x Boot ROM Reference Guide* (SPRU095), the Boot ROM does not change the PLL state; therefore, the PLL setting may be different depending on whether the F281x is reset by the Code Composer Studio software or from a power cycle.

    ---

2. Update the CKFA with AppCode passwords to unlock the F281x's code security module (CSM).

    The CKFA software requires that the CSM is unlocked. TI ships devices with Flash completely erased and CSMs unlocked. The unlocking process is not required for first-time programming. For additional information on the CSM, see the *TMS320x281x DSP System Control and Interrupts Reference Guide* (SPRU078).

3. Configure the CKFA for 64 KW or 128 KW Flash size.

4. Create a single binary file for SCI transmission.
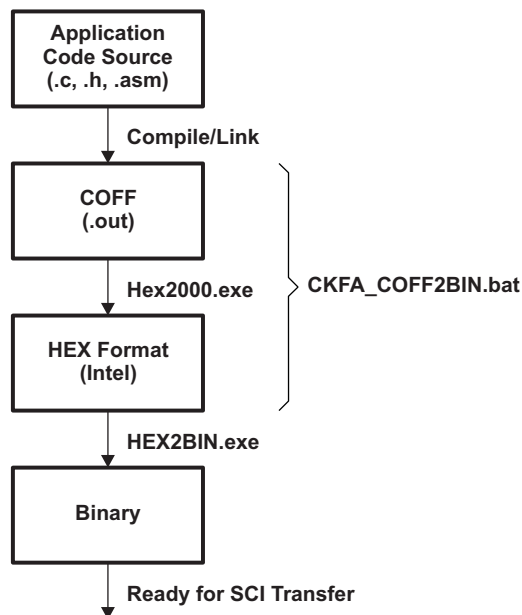
Figure 13. Overview of CKFA File Processing

3.2.1 Update Expected AppCode Checksum in CKFA Software

The expected AppCode checksum is stored in Example_Flash281x_API.c. After calculating a new checksum, update the 16-bit value for the constant CHECKSUM_EXPECTED. The CKFA software compares this value against its calculated checksum. Once the checksum is updated, the CKFA software needs to be compiled, linked, and a new CKFA binary file generated.

Example 10. Expected Checksum Used by CKFA – Example_Flash281x_API.c

```
#define    CHECKSUM_EXPECTED 0x6E13
```

3.2.2 Configure CKFA Software for Oscillator Frequency on F281x Target Board

Example_Flash281x_API.h contains the constant (PLLCR_VALUE) that defines the PLL multiplier setting. Flash281x_API_Config.h contains the constant (CPU_RATE) that defines the CPU_RATE for the system. The Flash timing parameters are based on this setting and if it is not set correctly, the Flash could be damaged.

Review the contents of these files to verify that PLLCR_VALUE and CPU_RATE are set to values that correspond to your F281x target board. If they are not, then make the required modifications and rebuild (compile, link, and convert to binary format) the CKFA software.

Example 11. PLL Setting Defined in Example281x_Flash281x_API.h

```
/*-----------------------------------------------------------------------------
      Specify the PLL Control Register (PLLCR) value.

       Uncomment the appropriate line by removing the leading double slash: //
       Only one statement should be uncommented.

       Your application must set the PLLCR Register before calling any
       of the Flash API functions.

       Example:  CLKIN is a 30 MHz crystal.
                 You need to have a 150 MHz CPU clock (SYSCLKOUT = 150 MHz).
                 In this case, PLLCR must be set to 10 (0x000A)
                 Uncomment the line: #define PLLCR_VALUE 10
                 Comment out the remaining lines with a double slash: //
-------------------------------------------------------------------------------*/

#define PLLCR_VALUE  0x000A      // SYSCLKOUT = (OSCLK*10)/2
// #define PLLCR_VALUE   0x0009     // SYSCLKOUT = (OSCLK*9)/2
// #define PLLCR_VALUE   0x0008     // SYSCLKOUT = (OSCLK*8)/2
```

Example 12. CPU_RATE Defined in Flash281x_API_Config.h

```
/*-----------------------------------------------------------------------------
    2. Specify the clock rate of the CPU (SYSCLKOUT) in ns.

       Take into account the input clock frequency and the PLL multiplier
       that your system will use.

       Use one of the values provided, or define your own.
       The trailing L is required tells the compiler to treat
       the number as a 64-bit value.

       Only one statement should be uncommented.

       Example:  CLKIN is a 30 MHz crystal. The PLL is enabled.

                 If your application will set PLLCR = 0xA then the CPU clock
                 will be 150 MHz CPU (SYSCLKOUT = 150 MHz).

                 In this case, the CPU_RATE will be 6.667L
                 Uncomment the line:  #define CPU_RATE  6.667L
-------------------------------------------------------------------------------*/

#define CPU_RATE       6.667L   // for a 150MHz CPU clock speed (SYSCLKOUT)
//#define CPU_RATE       7.143L   // for a 140MHz CPU clock speed (SYSCLKOUT)
//#define CPU_RATE       8.333L   // for a 120MHz CPU clock speed (SYSCLKOUT)

//-----------------------------------------------------------------------------
```

3.2.3   Configure CKFA for Correct Flash Range (64 KW or 128 KW)

Specify a device in file Flash281x_API_Config.h. Select either F2810 or F2811 by setting the desired processor #define to 1 in the file Flash281x_API_Config.h, used in the CKFA project.

Example 13. Specifying 64 KW or 128 KW Flash Range for CKFA

```
Setting for using the F2810 (Flash281x_API_Config.h):

#define FLASH_F2810   1
#define FLASH_F2811   0
#define FLASH_F2812   0


Setting for using the F2811 (Flash281x_API_Config.h):

#define FLASH_F2810   0
#define FLASH_F2811   1
#define FLASH_F2812   0
```

3.2.4   Create CKFA Binary File

Once the CKFA is configured for the correct rate, passwords, and Flash range, you must create the binary file suitable for Boot ROM SCI-A transmission.

3.2.4.1   Use Code Composer Studio to Rebuild CKFA Software

After modifications are made to the CKFA software, rebuild it using the Code Composer Studio projects that are included with this application report. The Code Composer Studio projects include additional build steps to convert the linked output into a binary format suitable for the SCI-A bootloader. It is important to use the included Code Composer Studio projects whenever rebuilding the CKFA or your application software.

From within the Code Composer Studio, open the CKFA project by selecting Open from the Project menu.

After verifying the settings of PLLCR_VALUE and CPU_RATE in Example_Flash281x_API.h and Flash281x_API_Config.h, respectively, select Rebuild All from the Project menu.

3.2.4.2 CKFA's Code Composer Studio Project Details

The Code Composer Studio project for the CKFA software consists of the following files (Figure 14).
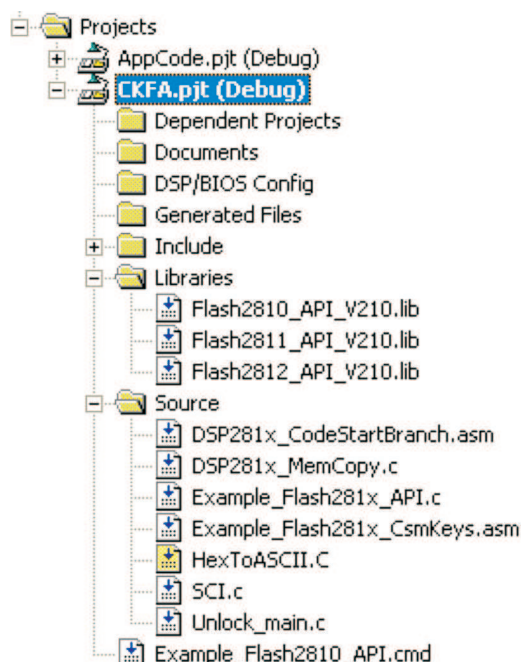
Figure 14. CKFA's Code Composer Studio Project

Note that the CKFA software is based on the Flash API example code. The difference between the Flash API example code and the CKFA code used in this application report is that the Flash API example does not transfer the code into RAM using the SCI-A boot option. Instead, it is designed so that the Flash API is already programmed into the F281x Flash and can be directly copied into RAM. This is a scenario typical for in-field programming. As demonstrated by this document, using the SCI-A boot option is typical for production programming of parts before shipping.

3.2.4.3 Generating CKFA.bin From Code Composer Studio

The CKFA project build options include a final build step that calls for a batch file that converts the COFF executable produced by building the project into a binary format suitable for SCI communication (Figure 15).

The F281x SCI-A boot option expects binary data transmission; ASCII-Hex format is not an option for the data stream. To convert a COFF executable to binary, the first step is to convert the COFF executable to ASCII-Hex format using the TI hex converter tool. This application report uses the Intel style ASCII-Hex format.
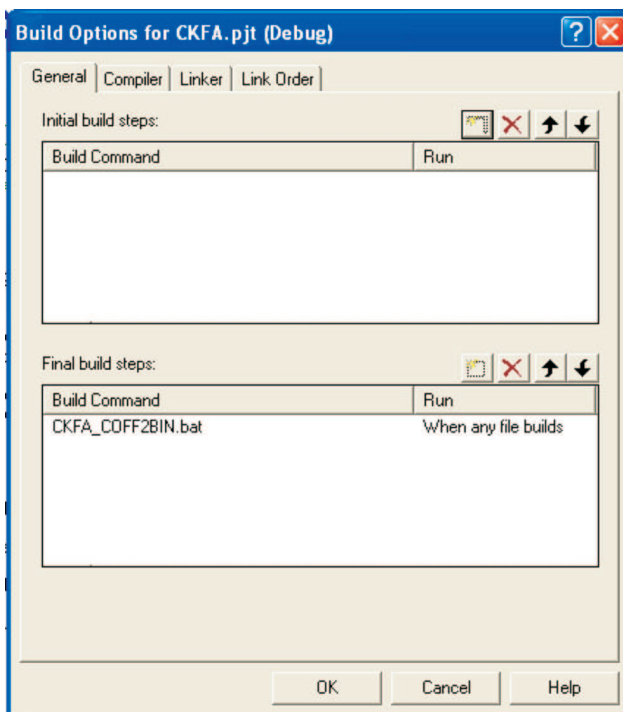
Figure 15. CKFA Project Build Options – Final Build Steps

The contents of the batch file, CKFA_COFF2BIN.bat, are listed below. It calls the C2000 Hex converter to convert the COFF executable, generated by the linker, to ASCII-Hex format (Intel). For more detailed information on the HEX-conversion utility, see the *TMS320C28x Assembly Language Tools User's Guide* (SPRU513).

Example 14. CKFA_COFF2BIN.bat

```
cd debug

C:\CCStudio_v3.1\C2000\cgtools\bin\hex2000.exe CKFA_hex.cmd

HEX2BIN CKFA.hex
```

The CKFA_hex.cmd contains the command-line instructions for the HEX2000 converter. The key output to this process is the CKFA.hex file that is used as the input to the HEX2BIN converter to generate the binary file.

- CKFA.out   = COFF executable input file
- -map CKFA_hex.map   = HEX2000 MAP output file
- -o CKFA.hex   = Hex file output
- -I   = Specify Intel record format

Example 15. CKFA_hex.cmd

```
CKFA.out

-boot
-sci8
-map CKFA_hex.map
-o CKFA.hex
-I
```

Once the file is in ASCII-Hex format, it can be converted to binary using any one of several widely available binary converters. For this application report, the Intel hex to binary converter used was HEX2BIN (http://gnuwin32.sourceforge.net/).

CKFA.bin follows the SCI-A bootloading option formatting expectations. See Section F.3 for the expected 8-bit SCI data stream format.

Example 16. CKFA.bin (Binary Reader), Binary Output From HEX2BIN.exe

```
aa 08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 f2 02 6b 01 00 00 00 01 04
fe 1f 76 c1 01 1a 1a 00 40 1a 18 ff df 1a 1a 00 20 00 8f 6c 02 40 76 37 01 00 8f 6c 02

KeyValue = 0x08AA
Reserved = 0x0000 ...
Entry Point = 0x000002F2
Block #1 Size = 0x016B
Block #1 Destination = 0x00000100
```

3.3 Establishing Communication With Boot ROM's SCI-A Code

The following steps must be verified to enable SCI communication with the F281x's Boot ROM code.

3.3.1 Configuring F281x Target Board for SCI-A Boot Option

At reset, the F281x scans four GPIO pins to determine the intended mode of operation. The *Jump to FLASH* option is the default mode, as it is implemented if there are no external connections to the four scanned GPIO pins at reset. See Table 1 for the GPIO pins that are scanned at reset. There is an internal pull-up resistor on GPIOF4 that makes the *Jump to FLASH* option default.

Table 1. Boot Mode GPIO Pins

Boot Mode Selected	GPIOF4 (SCITXDA)	GPIOF12 (MDXA)	GPIOF3 (SPISTEA)	GPIOF2 (SPICLK)
GPIO PU status[1]	PU	NoPU	NoPU	NoPU
Jump to Flash/ROM address 0×3F 7FF6 A branch instruction must be programmed here prior to reset to re-direct code execution as desired.	1	X	X	X
Call SPI_Boot to load from an external serial SPI EEPROM[2]	0	1	X	X
Call SCI_Boot to load SCI-A	0	0	1	1
Jump to H0 SARAM address 0×3F 8000[3]	0	0	1	0
Jump to OTP address 0×3D 7800	0	0	0	1
Call Parallel_Boot to load from GPIO Port B	0	0	0	0

[1]   PU - Pin has an internal pullup. NoPU = Pin doe not have an internal pullup.
[2]   Extra care must be taken on the external logic due to any toggling effect of the SPICLK to select a boot mode.
[3]   If the selected boot mode is Flash, H0, or OTP, then no external code is loaded by the bootloader.

Table 2 shows the jumper settings for the SCI_Boot option with the F2812 eZdsp used in this application report.

Table 2. F2812 eZdsp Jumper Settings

Boot ROM Options	JP7 (GPIOF4) (SCITXDA)	JP8 (GPIOF12) (MDXA)	JP11 (GPIOF3) (SPISTEA)	JP12 (GPIOF2) (SPICLK)
Jump to FLASH	1-2	X	X	X
Boot From SCI-A	2-3	2-3	1-2	1-2

3.3.2 Configuring Serial Communication Hardware

The F2812 eZdsp does not include an RS-232 transceiver. This application report uses the Link Research board LR-2812COM that connects directly to the F2812 eZdsp and provides an RS-232 link to the PC. Complete product information for the Link Research model LR-2812COM can be downloaded from the following URL: http://www.link-research.com/.

3.3.3 Configuring Serial Connection Software (PC)

HyperTerminal is the PC software program used in this application report for serial communications. First, you must configure HyperTerminal for the communications format used by the Boot ROM SCI-A boot option. Also, you should select a slower baud rate to start with, such as 9600 bps and then increasing it to 38400 or 57600 bps. This avoids having an immediate communications problem.

Figure 16 shows the configuration of serial port COM1 with the following settings:

- 115200 bps (recommend starting out with 9600 bps)
- 8-bit data bits
- No parity
- 1 stop bit
- Flow control = None
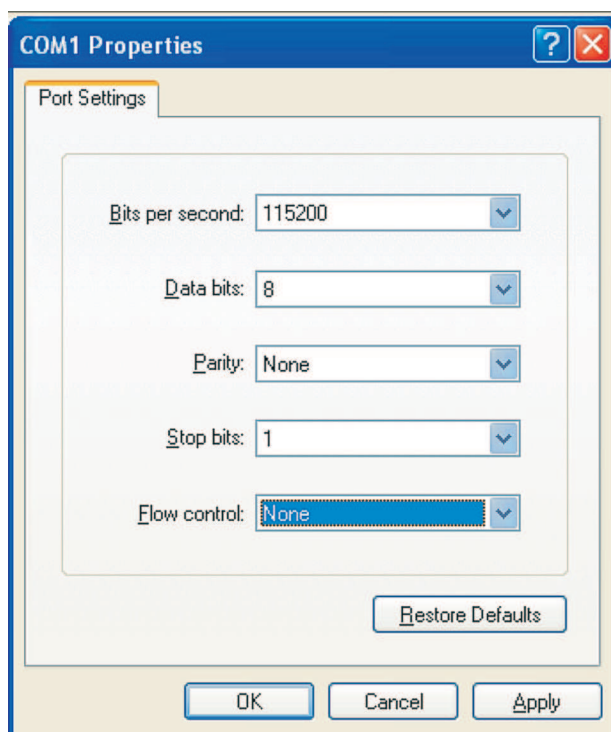
Figure 16. HyperTerminal Communications Configuration

With HyperTerminal configured and ready for transmitting or receiving data, prepare the F281x target by cycling its power (turn the power off and then on). This resets the F281x Boot ROM and causes it to scan the four GPIO boot pins for their configuration. If you have them configured for SCI-A boot mode, the F281x begins executing the SCI_Boot code in Boot ROM.

This code will first configure the SCI-A port's baud rate using the auto-baud feature of this port. To lock in the F281x SCI-A baud-rate to the baud-rate configured for HyperTerminal, type the character 'a' or 'A'. This is the expected character of the auto-baud feature of the SCI and it is used to configure the SCI baud-rate on the DSP. See the flowchart of the Boot ROM code in the *TMS320x281x Boot ROM Reference Guide* (SPRU095). The Boot ROM code echoes the received character back to HyperTerminal to identify that the baud-rate has been successfully configured (see Figure 17). The auto-baud function is documented in the *TMS320x28xx, 28xxx DSP Serial Communication Interface (SCI) Reference Guide* (SPRU051).
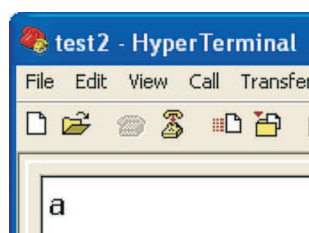
Figure 17. Echoed Character From F2812 SCI Auto-Baud Logic

If the entered character is not echoed back to HyperTerminal, verify the following conditions:

- Hardware connections are in place
- F2812 target board power was cycled to force a reset
- F2812 GPIO boot pins (GPIOF4, F12, F3, and F2) are configured for SCI boot mode
- Verify HyperTerminal settings are correct

If the above conditions are verified, then the baud-rate of HyperTerminal needs to be reduced and the auto-baud locking process restarted.

3.4 Flash Programming Procedure

Once the application and CKFA software are prepared for transfer, and serial communication has been established with the Boot-ROM SCI code, you are now ready to start the Flash programming process. This consists of the serial transfer of the CKFA software, unlocking the CSM, transferring the application code, and verification that the application code was programmed correctly.

3.4.1 Transfer CKFA Software

The target baud-rate is set and ready to transfer the CKFA. Within HyperTerminal, select Send Text File from the Transfer menu. Then, select the CKFA.bin file in the code→CKFA→Debug folder.

The transferred characters are echoed to the HyperTerminal screen by the SCI Boot ROM code. These echoed characters can be used to verify that each character sent has been received correctly. This application report does not verify each transmitted character. Verification is achieved by calculating a checksum on the Flash after the application code is programmed.

Figure 18 shows that the CKFA transfer to RAM is complete and executing, with the processor unlocked, when the HyperTerminal display appears.
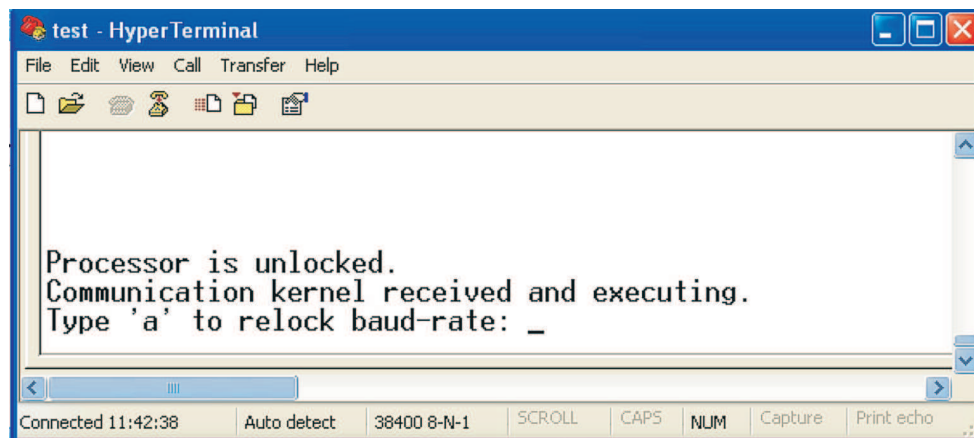
Figure 18. HyperTerminal - CKFA Software Ready to Update F281x Baud-Rate

3.4.2 Unlocking the CSM

If the CSM is locked and the incorrect passwords were used in the CKFA software, then your HyperTerminal screen will look like Figure 19. Correct the password values used in Example_Flash281x_CsmKeys.asm, rebuild the CKFA software, reset the DSP, and attempt the CKFA transfer again.
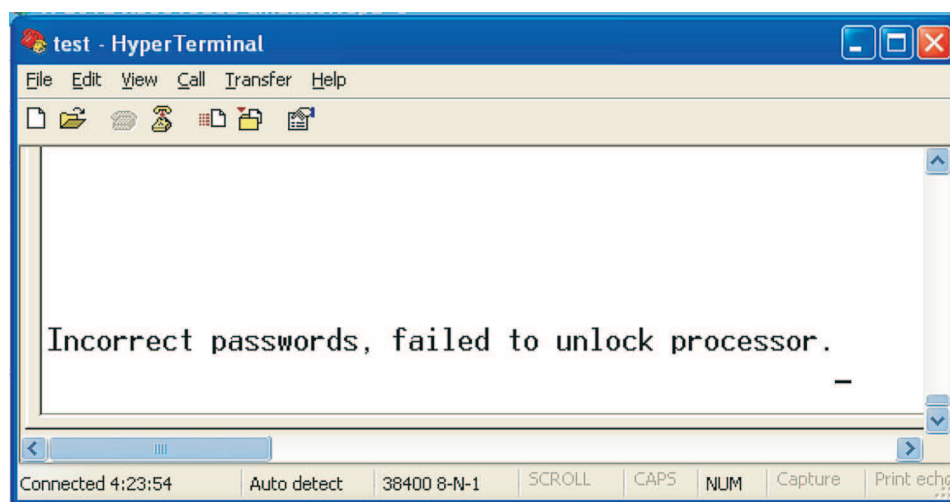
Figure 19. HyperTerminal - CKFA Transfer Failure Due to a Locked F281x

3.4.3 Prep Target Using CKFA Interface

Once the processor is unlocked and the CKFA software is executing, the CKFA software enables and configures the PLL. For this application report, the CPU is configured to run at 150 MHz. With the new CPU rate, the CKFA software needs to update the SCI-A baud rate. Type 'a' or 'A' to relock the auto baud logic to the HyperTerminal baud rate, as shown in Figure 20. If the baud-rate is unable to relock after typing 'a' or 'A' several times, reduce the HyperTerminal's baud rate and start the process over.

Once the baud-rate is updated, the CKFA software calculates a checksum on the Flash and transmits this using the SCI-A. You can use this data to determine the following:

- If the Flash is already erased, the checksum is 0×0000. The erase step can be skipped and the Flash programmed.
- If the checksum is equal to the value of expected of the application software, then the device is already programmed as intended and the entire process can end.
- If the checksum is none of these cases, then the Flash has been programmed with data other than the intended application code and must be erased. It is critical that you answer yes ('y') to the *Erase FLASH?* prompt (Figure 20).

TI ships the F281x Flash completely erased so you can immediately program the Flash.
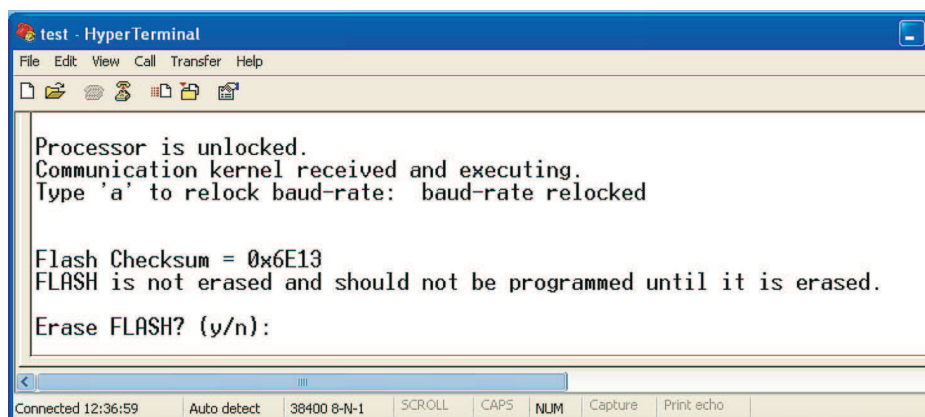
Figure 20. HyperTerminal - CKFA Checksum Determines Flash is Not Erased

While the device is erasing, it is important not to interrupt the process. Wait for the erase to complete. Once the Flash is erased, you will receive a status update of *erasing done*. At this point, the CKFA software is ready to receive the application code (see Figure 21).
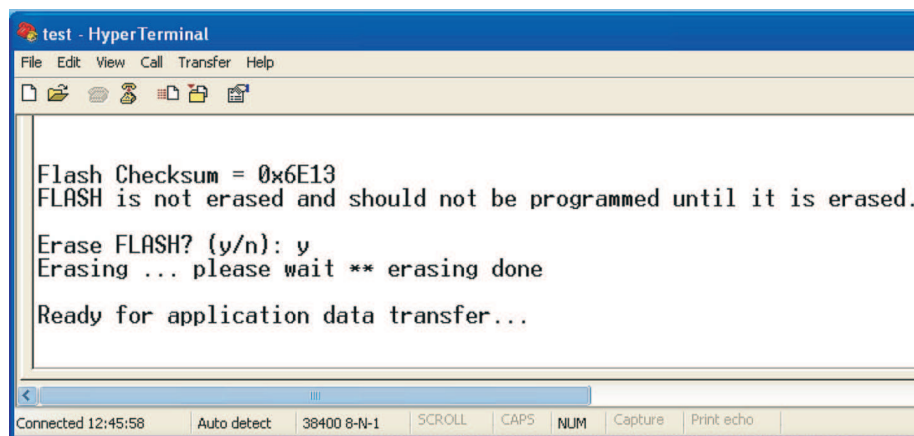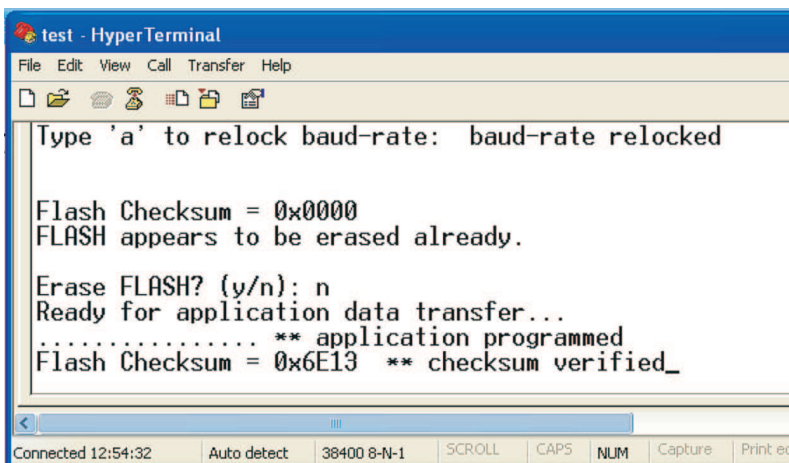
Figure 21. HyperTerminal - CKFA Software is Ready to Transfer and Program Application Code

3.4.4 Transfer AppCode

To transfer the application software using HyperTerminal, select Send Text File from the Transfer menu. Then, select the AppCode.bin file from the code→AppCode→Debug folder. Once the application is received and programmed into Flash, the CKFA software calculates a checksum on the Flash. This value is transmitted using SCI-A and compared against a value stored in Example_Flash281x_API.c. If the calculated checksum matches the expected value, this is communicated using SCI-A, showing that the checksum was verified (Figure 22).

Figure 22. HyperTerminal - CKFA Software Has Transferred and Programmed Application Code

3.4.5 Verify AppCode Execution

The F281x is now ready to execute the application code out of Flash.

1. Remove power from the F281x target board.
2. Change the GPIO boot mode pins from SCI boot mode to *Jump to Flash* mode. (On the F2812 eZdsp, this is easily done by moving JP7 from position 2-3 to 1-2.)
3. Apply power to the F281x target board. The application code will execute out of Flash. (If you are using the F2812 eZdsp, the DS2 LED will be blinking.)

4 Flash Program Timing Results

As reducing programming time is a goal of this application report, the document methodology takes that into consideration by:

- Continuous transfer of application code into two 4 KW buffers
- Eliminating any overhead (non-data) from AppCode binary file
- Checking Flash condition before programming to optionally skip erase step
- Filling all unused memory to 0xFFFF (erased state)
- Maximizing SCI-A baud-rate with the CKFA code setting PLL

Typical program times are listed in the *TMS320F2810, TMS320F2811, TMS320F2812, TMS320C2810, TMS320C2811, TMS320C2812 Data Manual* (SPRS174) with 500 ms and 250 ms listed for 16 KW and 8 KW sectors, respectively. The F2810 has three 16 KW and two 8 KW sectors; therefore, the typical programming time for the entire 64 KW of Flash is 2s.

Maximizing baud rate reduces the programming time significantly. Table 3 shows Flash programming times referenced from the *TMS320F2810, TMS320F2811, TMS320F2812, TMS320C2810, TMS320C2811, TMS320C2812 Data Manual* (SPRS174).

Table 3. Flash Parameters at 150-MHz SYSCLOUT [1]

PARAMETERS				MIN	TYP	MAX	UNIT
Program time	16-bit word	Using Flash API v1			35		µs
		Using Flash API v2.10			50		µs
	8K sector	Using Flash API v1			170		ms
		Using Flash API v2.10			250		ms
	16K sector	Using Flash API v1			320		ms
		Using Flash API v2.10			500		ms
Erase time	8K sector				10		s
	16K sector				11		s
$I_{DD3VFLP}$	I_{DD3VFL}		Erase		75		mA
			Program		35		mA
I_{DDP}	V_{DD} current consumption during erase/program cycle				140		mA
I_{DDIOP}	V_{DDIO} current consumption during erase/program cycle				20		mA

[1] Typical parameters, as seen at room temperatures, including function call overhead.

4.1 PC to F281x Target Board

Using the timer in HyperTerminal, the application code takes approximately 37 seconds to program the AppCode configured for 64 KW with the RS-232 baud-rate set for 38400 bps. At 57600 bps, the programming is reduced to 24 seconds.

4.2 ICT to F281x Target Board

Baud-rate can be greatly increased with direct connections to the F281x; SCI-A receive and transmit is possible. RS-232 transceiver bandwidths are limited and significantly increase programming time. The AppCode configured for 64 KW Flash programs in 1.4 seconds using the emulated ICT (EICT) hardware used in this application report.

4.2.1 Methodology

The PC is used to transfer the CKFA and AppCode binary files to RAM on the EICT. The F2812 eZdsp represents the EICT. The CKFA binary file is stored in the eZdsp's internal RAM. The AppCode binary file is stored in the F2812 eZdsp; it has 64 KW of external RAM.

The PC to EICT transfer is done using RS-232 and HyperTerminal, a relatively slow data transfer.
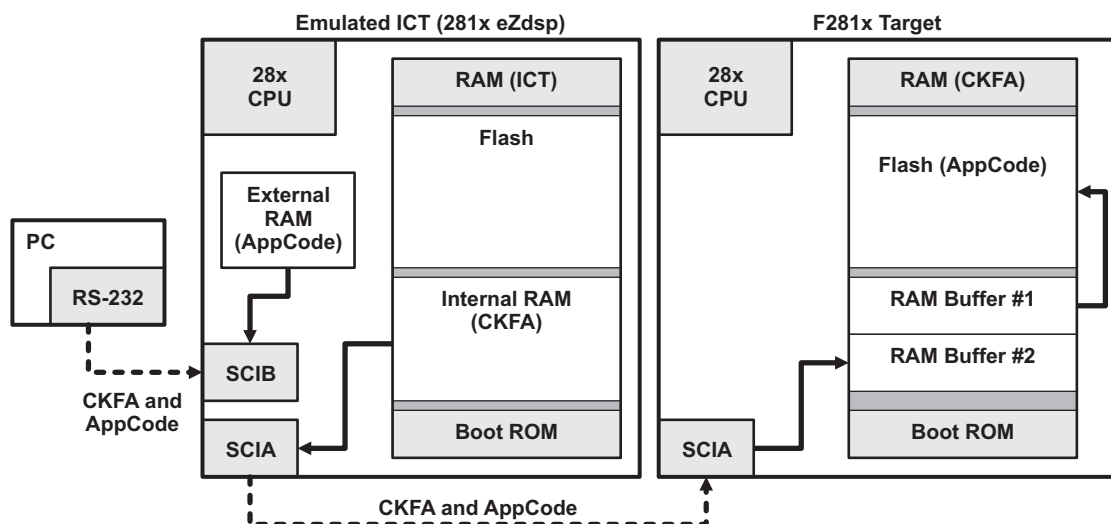
Figure 23. Block Diagram of Flash Programming From ICT to F2810 Target Board

The EICT to F281x target is fast because it does not use RS-232 transceivers. The SCI pins are directly connected between the EICT and eZdsp, just as they would be with an ICT on a production line. The CKFA is controlling the F281x target and its PLL setting. Together this allows the EICT to transfer to the F281x target at a baud-rate of 1.875 Mbps.

Hardware:
• (2) F2812 eZdsp From Spectrum Digital
• RS-232 Interface From Link Research
• JTAG Emulator From Spectrum Digital

Software:
• PC = HyperTerminal
• EICT = Application Report EICT SW
• Target = Application Report CKFA SW
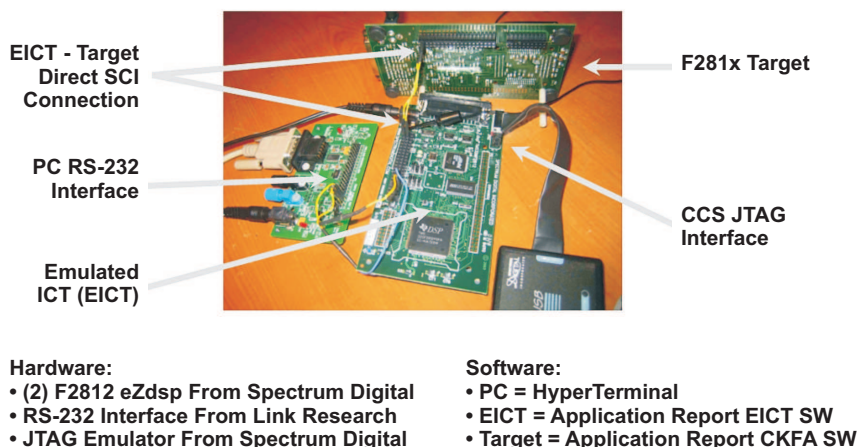
Figure 24. Photo of Emulated ICT to F281x Target Board

4.2.2 Flash Timing Results

With the Flash programming process understood, look at the amount of time it takes to program the Flash using the technique presented in this application report. Timing is directly related to the baud-rate for the serial transfer of the 64 kW or 128 kW application code.

4.2.2.1 Baud-Rate Settings for CKFA Transfer From EICT to F281x Target

The target Boot ROM code determines the baud rate for the CKFA transfer from EICT to the F281x because the Boot ROM does not enable the PLL at reset. The resulting CPUCLK for the target is then based on the input oscillator frequency, which is 30 MHz on the F281x eZdsp.

At reset, the PLL is disabled. The resulting CPUCLK is based on the oscillator frequency. This results in a CPUCLK = 30 MHz.

At reset, the low-speed peripheral clock (LSPCLK) is configured as CPUCLK/4, which is not changed by the Boot ROM code. This results in a LSPCLK of 7.5 MHz.

The minimum value for the SCI baud-rate register is 1. This results in a max baud-rate of 468 Kbps for the F281x Target at reset.

Transferring the CKFA at 468 Kbps is much faster than the RS-232 typical PC (HyperTerminal) baud rate of 38 or 56 Kbps. In addition, the CKFA binary file size is small (6.4KB), compared to the AppCode size of 128 or 256KB. Maximizing the baud rate for the AppCode transfer is critical.

EICT:

CPUCLK = 150 MHZ

LSPCLK = 150 MHz/2 = 75 MHz

BRR = 19

Baud-Rate = LSPCLK/((BRR+1)*8) = 468750 bps

Target:

CPUCLK = 30 MHZ   PLL bypassed at reset and not enabled by Boot ROM]

LSPCLK = 30 MHz/4 = 7.5 MHz   [/4 default at reset]

BRR = 1   [Set by SCI-Autobaud]

Baud-Rate = LSPCLK/((BRR+1)*8) =
468750 bps

4.2.2.2   Baud-Rate Settings for AppCode Transfer From EICT to F281x Target

With the CKFA code controlling the F281x target, the SCI baud-rate can be configured to the maximum rate supported by external hardware connections. The maximum baud-rate supported by the F281x SCI is 20 Mbps, limited by the F281x IO buffer speed.

The PLL is configured by the CKFA software as *x5*. This results in a CPUCLK of 150 MHz.

The LSPCLK is configured by the CKFA software as CPUCLK/2. This results in a LSPCLK of 75 MHz.

The minimum value for the SCI baud-rate register supported by the hardware used in this application report is 4. This results in a maximum baud-rate of 1.875 Mbps for the F281x eZdsp target. A BRR setting of 3 was tested (2.34 Mbps), but this resulted in serial communications errors.

EICT:

CPUCLK = 150 MHZ

LSPCLK = 150 MHz/2 = 75 MHz

BRR = 4

Baud-Rate = LSPCLK/((BRR+1)*8) = 1.875 Mbps

Target:

CPUCLK = 150 MHZ   [PLL enabled by CKFA]

LSPCLK = 150 MHz/2 = 75 MHz   [LSPCLK divider set by CKFA]

BRR = 4   [Set by SCI-Autobaud]

Baud-Rate = LSPCLK/((BRR+1)*8) = 1.875 Mbps

4.2.3   ICT Flash Programming Procedure

Using an ICT allows for much faster serial port baud-rates. Connecting directly to the serial pins of the processor eliminates the speed-limiting use of an RS-232 transceiver, as used in the previous section when a PC was used for Flash programming. Achievable baud-rates for an ICT based system are > 2 Mbps. In this application report, 1.875 Mbps was achieved using an F2812 eZdsp to emulate an ICT.

4.2.3.1   Connecting PC to Emulated ICT

You must connect an RS-232 cable between the PC and the F2812 eZdsp representing the ICT. Since the F2812 eZdsp does not have an RS-232 transceiver, the RS-232 interface from Link-Research was used. Complete product information for Link Research model LR-F2812COM-2 can be downloaded from the following URL: http://www.link-research.com/.

As the Link Research board is designed to match the Spectrum Digital eZdsp headers, the connections point to the same header positions (Table 4).

Table 4. Link Research RS-232 Board Connections to EICT (F2812 eZdsp)

	Link-Research RS-232 Board	Spectrum Digital F2812 eZdsp
SCIRXDB	P4-19	P4-19
SCITXDB	P4-20	P4-20

4.2.3.2 Connecting Emulated ICT to F281x Target Board

You must connect SCI-A from the EICT (F2812 eZdsp) to the SCI-A of the target board (F2812 eZdsp). The eZdsp header pins P8-2 and P8-3 are used. Be sure to swap the transmit and receive connections between the boards (Table 5).

Table 5. Link Research RS-232 Board Connections to EICT (F2812 eZdsp)

EICT	Target
P4-20	P4-20
P8-3 SCITXDA	P8-4 SCIRXDA
P8-39 GND	P8-39 GND

4.2.3.3 Prepare Emulated ICT Software

The following steps show how to connect the IEEE Std. 1149.1-1990 (JTAG) emulator to the EICT.

1. Connect the IEEE Std. 1149.1-1990 (JTAG) emulator to EICT.
2. Open the Code Composer Studio workspace *SCI_FLASH_AppReport.wks*.
3. Reset the CPU.
4. Load the code.
5. Run in real-time mode
6. Set the Watch and Memory windows to continuously update in real-time mode.

4.2.3.4 Lock Baud Rate Between PC and Emulated ICT

The EICT software first configures the EICT hardware's baud rate to match the baud rate set for the PC (HyperTerminal). With the EICT software running, under Code Composer Studio control, enter the autobaud character ('a' or 'A') into the HyperTerminal window. (Be sure to include the quotation as part of your entry.) In response, the EICT software will confirm that the EICT baud rate is locked and ready for CKFA binary file transfer (see Figure 25).
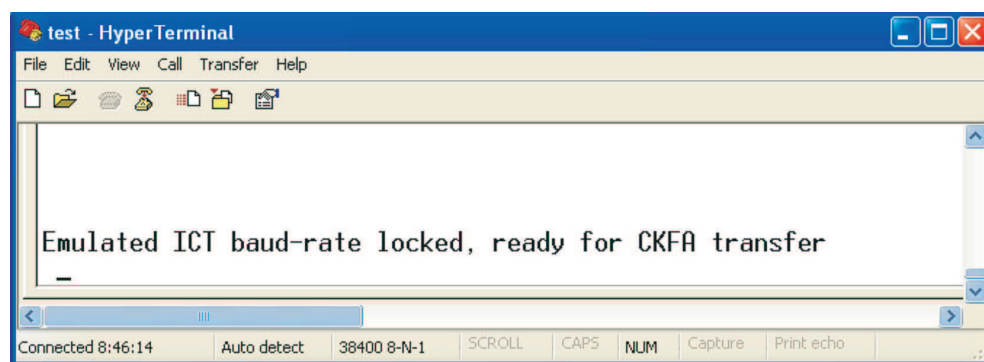
Figure 25. Emulated ICT Ready for CKFA Transfer From PC

4.2.3.5 Transfer CKFA and AppCode From PC to Emulated ICT RAM

Transfer the CKFA software from HyperTerminal using the same procedure as described in Section 3.4.1. Within HyperTerminal, select Send Text File from the Transfer menu. Then, select the CKFA.bin file from the code→CKFA→Debug folder.
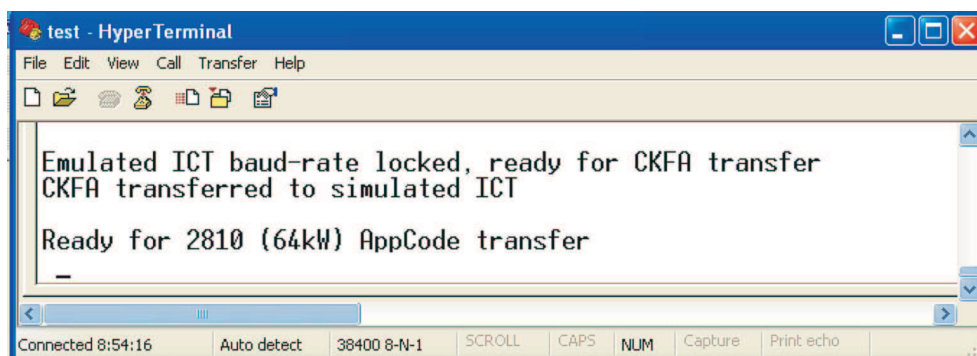
Figure 26. Emulated ICT Ready for AppCode Transfer From PC

Transfer AppCode software from HyperTerminal using the same procedure as described in Section 3.4.4. Within HyperTerminal, select Send Text File from the Transfer menu. Then, the AppCode.bin file from the code→AppCode→Debug folder.

4.2.3.6 Lock Baud Rate Between Emulated ICT and F281x Target Board's Boot ROM Code

The EICT now has both the CKFA and AppCode binary files loaded into its RAM. The EICT is ready to follow the standard procedure used earlier, but instead of using the slow HyperTerminal RS-232 transfer, you use the fast direct connection of the EICT.

The EICT transmits the procedures to follow to HyperTerminal (Figure 27). The PC is connected to the EICT's SCI-B, this does not disturb the SCI-A communication between the EICT and the target.
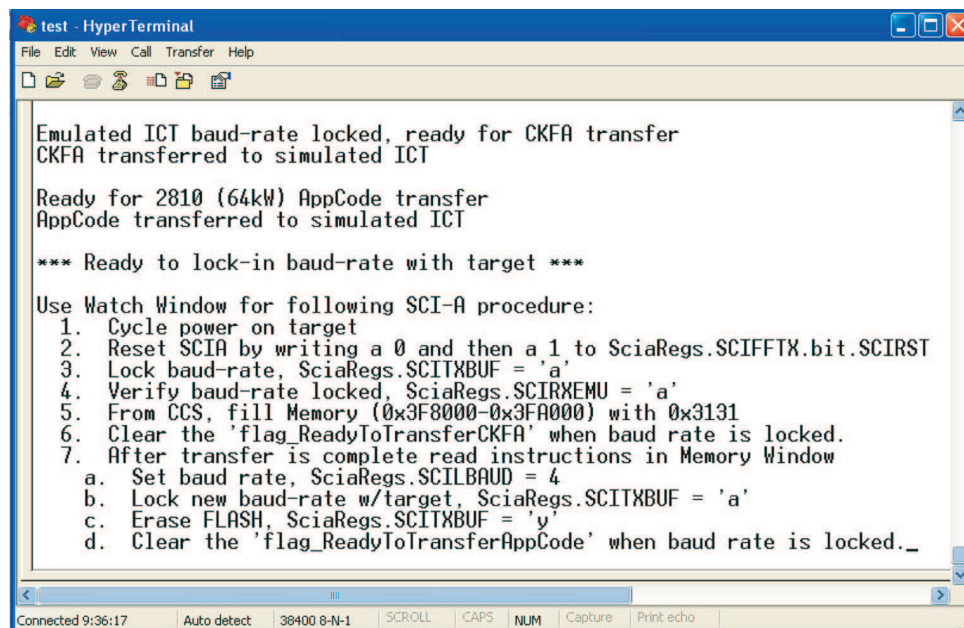
Figure 27. Emulated ICT Ready to Start F281x Target Procedure

4.2.3.7 Transfer CKFA From Emulated ICT to F281x Target Board

With Code Composer Studio running in real-time mode and the Watch and Memory Windows configured for continuous refresh, cycle the power on the F281x target board. Leave the second Watch Window tab open. For the F2812 eZdsp, you must remove the 5 V power supply and then reapply it.

Notice that the EICT's SCIA has a receive error; typically, SciaRegs.SCIRXST.all is 178. As instructed in HyperTerminal, reset EICT's SCIA by writing a 0 and then a 1 to SciaRegs.SCIFFTX.bit.SCIRST. Note that SciaRegs.SCIRXST.all now equals 0.

Set the target board's baud rate. Enter the autobaud character ('a' or 'A') into the Watch Window, SciaRegs.SCITXBUF = 'a'. Be sure to include the quotations as part of your entry. In response, the target board sends the autobaud character back and you should see it in the Watch Window variable SciaRegs.SCIRXEMU (97 = 'a').

The target board is not ready to receive the CKFA transfer. Before you start the transfer, prepare the Code Composer Studio Memory Window to receive messages from the target board. Select Memory→Fill from the Code Composer Studio Edit Menu. Fill memory at 0×3F8000 (length 0×2000) with 0×3131 ("11") (Figure 28).
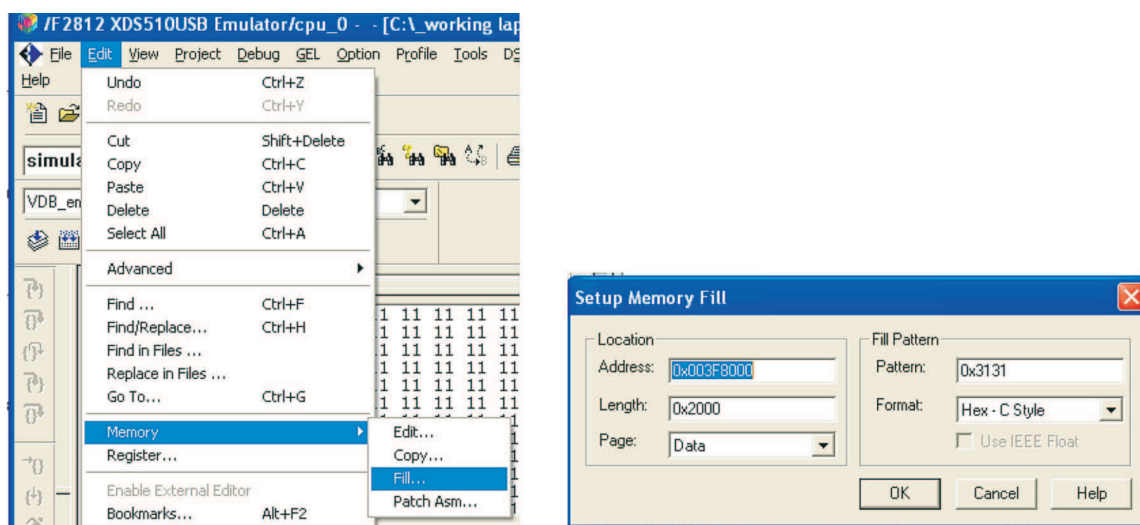
Figure 28. Preparing the Memory Window for Target Board Response Messages

Start the CKFA transfer by setting the *flag_ReadyToTransferCKFA* to 0 in the Watch Window. The transfer is complete when the Memory Window displays this message from the CKFA software running on the target board, *Processor is unlocked. Communication kernel received and executing. Type 'a' to relock baud-rate:* (see Figure 29). The next step is to increase the EICT baud-rate.
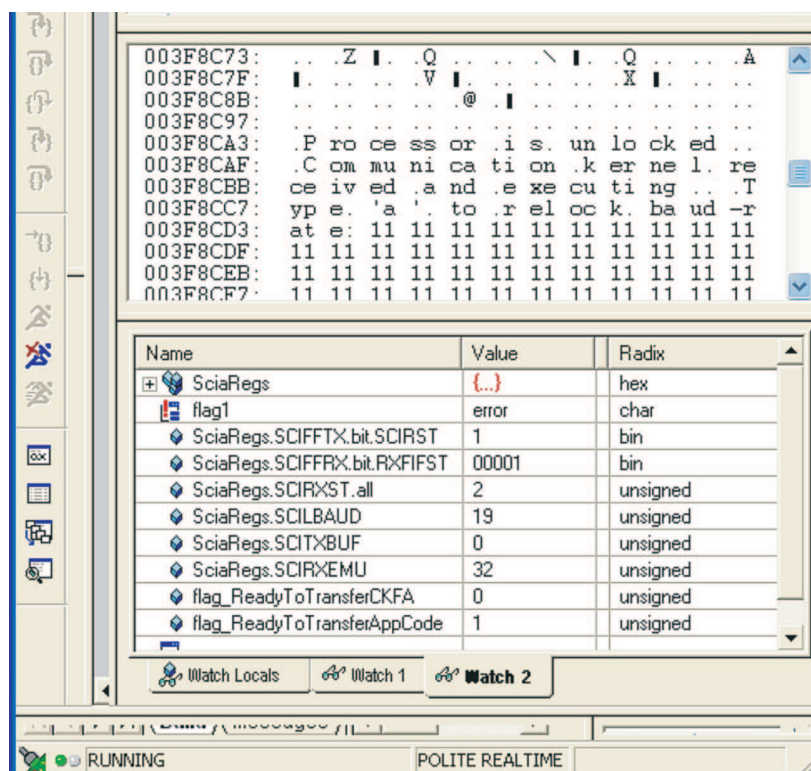
Figure 29. CKFA Transfer From Emulated ICT to Target Board Successful

4.2.3.8   Lock in New Maximum Baud-Rate

Set the EICT baud rate register to 4 in the Watch Window, SciaRegs.SCILBAUD = 4. The CKFA software, running on the target board, has enabled its SCI-A autobaud logic to relock its baud rate to the new EICT rate.

Enter the autobaud character ('a' or 'A') into the Watch Window, SciaRegs.SCITXBUF = 'a'. (Be sure to include the quotations as part of your entry.) In response, the target board sends the autobaud character back and you should see it in the Watch Window variable SciaRegs.SCIRXEMU (97 = 'a') (Figure 30).
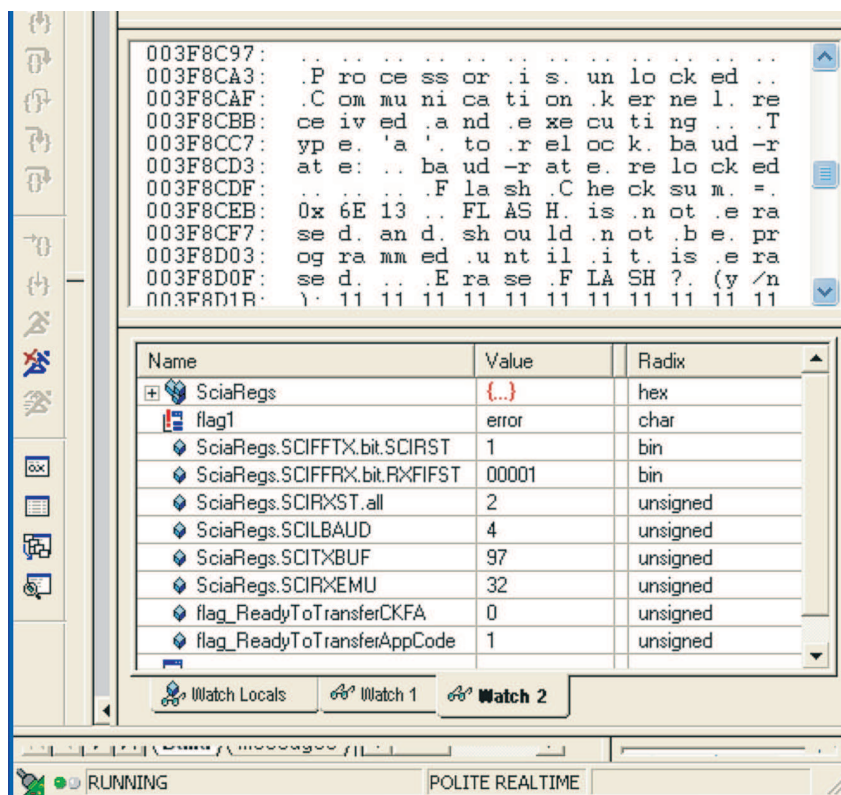
Figure 30. CKFA Baud-Rate Relocked to Emulated ICT at 1.875 Mbps

In response to the target board's message, *Erase Flash?*, enter the character 'y' into the Watch Window (SciaRegs.SCITXBUF = 'y'). (Be sure to include the quotations as part of your entry.) This instructs the CKFA software to erase the target board's Flash.

The target board sends the message, *Erasing … please wait*, which can be seen in the Memory Window.

4.2.3.9   *Transfer AppCode From Emulated ICT to F281x Target Board*

Start the AppCode transfer by setting *flag_ ReadyToTransferAppCode* to 0 in the Watch Window. The transfer is complete when the Memory Window displays this message from the CKFA software running on the target board, *** erasing done. Ready for application data transfer* (see Figure 31).
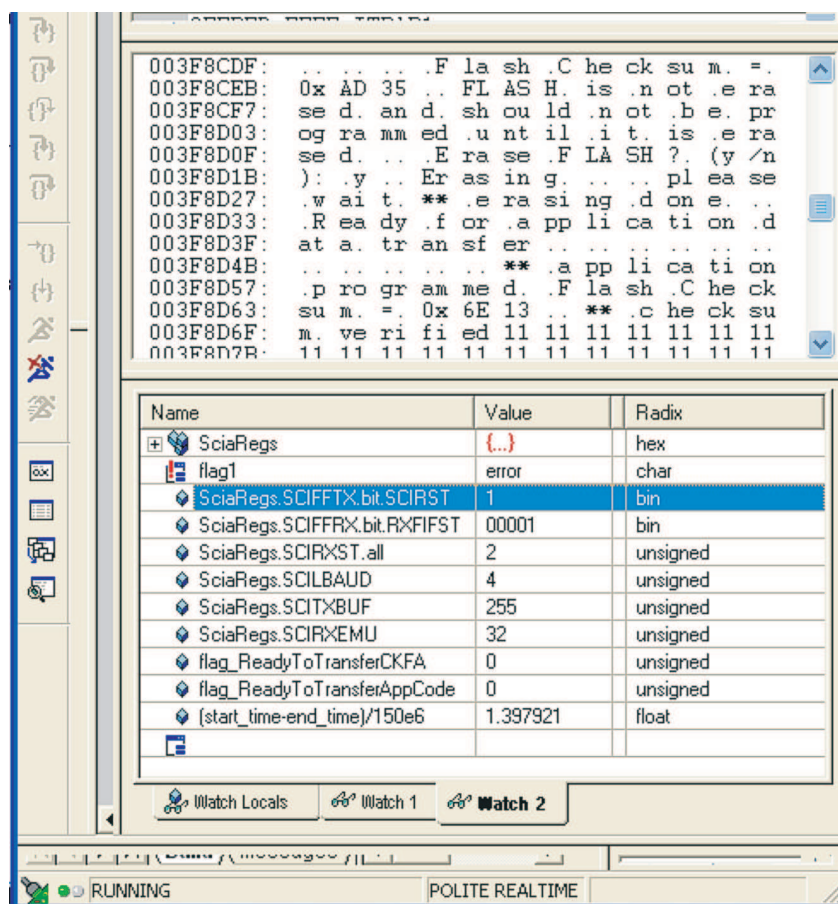
Figure 31. AppCode Flash Programming on Target Board Successful

The AppCode transfer and Flash programming is complete when the Memory Window displays this message from the CKFA software running on the target board, ** *application programmed. Flash Checksum = 0x6E13.* ** *checksum verified*. This is the same message that you received when using HyperTerminal to program AppCode into the target board's Flash.

The programming time is listed in the Watch Window:

(start_time – end_time)/150e6 = 1.398 seconds

This is the benchmark timing using the F28x CPU count-down timer.

5 References

- *Download TMS320F2810, TMS320F2811 and TMS320F2812 Flash API* (http://www-s.ti.com/sc/techlit/sprc125.zip)
- *TMS320x281x DSP System Control and Interrupts Reference Guide* (SPRU078)
- *TMS320x281x Boot ROM Reference Guide* (SPRU095)
- *TMS320x28xx, 28xxx DSP Serial Communication Interface (SCI) Reference Guide* (SPRU051)
- *TMS320C28x DSP CPU and Instruction Set Reference Guide* (SPRU430)
- *TMS320F2810, TMS320F2811, TMS320F2812, TMS320C2810, TMS320C2811, TMS320C2812 Data Manual* (SPRS174)
- *Download: C281x C/C++ Header Files and Peripheral Examples - SPRC097* (http://www-s.ti.com/sc/techlit/sprc097.zip)
- *TMS320C28x Assembly Language Tools User's Guide* (SPRU513)
- eZdsp 2812 Technical Reference (http://c2000.spectrumdigital.com/ezf2812/docs/ezf2812_techref.pdf)
- SDFlash Utility (www.spectrumdigital.com/)
- HEX2BIN, Intel Hex to Binary Converter (http://gnuwin32.sourceforge.net/)
- Link Research (www.link-research.com)

Appendix A TMS320F281x Memory Maps

A.1 *F2812 Memory Map*

Figure A-1 shows the F2812 memory map.
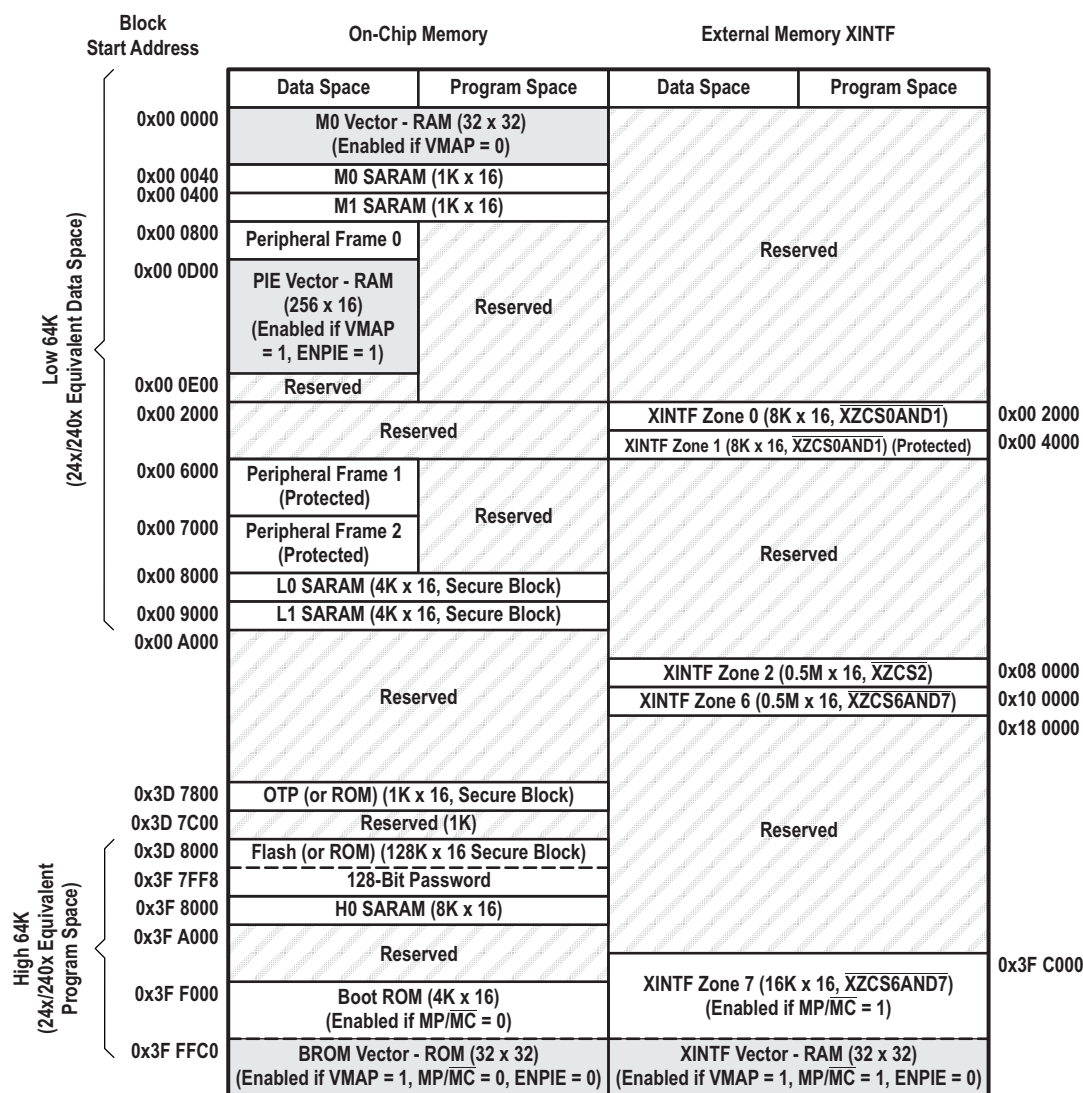
Legend:
Only one of these vector maps - M0 vector, PIE vector, BROM vector, XINTF vector - should be enabled at a time.

A   The memory blocks are not to scale.

B   Reserved locations are reserved for future expansion. The application should not access these areas.

C   Boot ROM and Zone 7 memory maps are active either in on-chip or XINTF zone depending on MP/$\overline{\text{MC}}$, not in both.

D   Peripheral frame 0, peripheral frame 1, and peripheral frame 2 memory maps are restricted to data memory only. The program cannot access these memory maps in program space.

E   *Protected* means that the order of Write followed by Read operations is preserved rather than the pipeline order.

F   Certain memory ranges are EALLOW protected against spurious writes after configuration.

G   Zones 0 and 1 and Zones 6 and 7 share the same chip select; therefore, these memory blocks have mirrored locations.

Figure A-1. F2812 Memory Map

A.2 F2810 Memory Map

Figure A-2 shows the F2810 memory map.

Block Start Address — **On-Chip Memory**

Block Start Address	Data Space	Program Space
0x00 0000	M0 Vector - RAM (32 x 32) (Enabled if VMAP - 0)	
0x00 0040	M0 SARAM (1K x 16)	
0x00 0400	M1 SARAM (1K x 16)	
0x00 0800	Peripheral Frame 0	Reserved
0x00 0D00	PIE Vector - RAM (256 x 16) (Enabled if VMAP = 1, ENPIE = 1)	
0x00 0E00	Reserved	
0x00 2000	Reserved	
0x00 6000	Peripheral Frame 1 (Protected)	Reserved
0x00 7000	Peripheral Frame 2 (Protected)	
0x00 8000	L0 SARAM (4K x 16, Secure Block)	
0x00 9000	L1 SARAM (4K x 16, Secure Block)	
0x00 A000	Reserved	
0x3D 7800	OTP (or ROM) (1K x 16, Secure Block)	
0x3D 7C00	Reserved	
0x3D 8000	Flash (or ROM) (64K x 16 Secure Block)	
0x3F 7FF8	128-Bit Password	
0x3F 8000	H0 SARAM (8K x 16)	
0x3F A000	Reserved	
0x3F F000	Boot ROM (4K x 16) (Enabled if MP/$\overline{MC}$ = 0)	
0x3F FFC0	BROM Vector - ROM (32 x 32) (Enabled if VMAP = 1, MP/$\overline{MC}$ = 0, ENPIE = 0)	

Low 64K (24x/240x Equivalent Data Space)

High 64K (24x/240x Equivalent Program Space)

Legend:
▢ Only one of these vector maps - M0 vector, PIE vector, BROM vector - should be enabled at a time.

A   The memory blocks are not to scale.

B   Reserved locations are reserved for future expansion. The application should not access these areas.

C   Peripheral frame 0, peripheral frame 1, and peripheral frame 2 memory maps are restricted to data memory only. The program cannot access these memory maps in program space.

D   *Protected* means that the order of Write followed by Read operations is preserved rather than the pipeline order.

E   Certain memory ranges are EALLOW protected against spurious writes after configuration.

Figure A-2. F2810 Memory Map

Appendix B TMS320F281x Flash Sectors

B.1 *Addresses of Flash Sectors in F2812 and F2811*

The following F2812 and F2811 Flash sectors are programmed by the technique used in this application report.

Table B-1. F2812 and F2811 Flash Sector Addresses

Address Range	Program and Data Space
0×3D 8000 0×3D 9FFF	Sector J, 8K × 16
0×3D A000 0×3D BFFF	Sector I, 8K × 16
0×3D C000 0×3D FFFF	Sector H, 16K × 16
0×3E 0000 0×3E 3FFF	Sector G, 16K × 16
0×3E 4000 0×3E 7FFF	Sector F, 16K × 16
0×3E 8000 0×3E BFFF	Sector E, 16K × 16
0×3E C000 0×3E FFFF	Sector D, 16K × 16
0×3F 0000 0×3F 3FFF	Sector C, 16K × 16
0×3F 4000 0×3F 5FFF	Sector B, 8K × 16
0×3F 6000	Sector A, 8K × 16
0×3F 7F80 0×3F 7FF5	Program to 0×0000 when using the code security mode
0×3F 7FF6 0×3F 7FF7	Boot-to-Flash (or ROM) entry point (program branch instruction here)
0×3F 7FF8 0×3F 7FFF	Security password (128 bit) (Do not program to all zeroes)

B.2 *Addresses of Flash Sectors in F2810*

The following F2810 Flash sectors are programmed by the technique used in this report.

Table B-2. F2810 Flash Sector Addresses

Address Range	Program and Data Space
0×3E 8000 0×3E 9FFF	Sector E, 16K × 16
0×3E C000 0×3E FFFF	Sector D,16K × 16
0×3D C000 0×3D FFFF	Sector H, 16K × 16
0×3F 4000 0×3F 5FFF	Sector B, 8K × 16
0×3F 6000	Sector A, 8K × 16
0×3F 7F80 0×3F 7FF5	Program to 0×0000 when using the code security mode
0×3F 7FF6 0×3F 7FF7	Boot-to-Flash (or ROM) entry point (program branch instruction here)
0×3F 7FF8 0×3F 7FFF	Security password (128 bit) (Do not program to all zeroes)

Appendix C 8-Bit Data Stream Expected by Boot ROM SCI-A

C.1 TMS320C28x Assembly Language Tools User's Guide

The following data stream format is used by the Boot-ROM SCI-A boot option code. The software used in this application report to transfer the CKFA software adheres to this format.

Table C-1. LSB/MSB Loading Sequence in 8-Bit Data Stream

Byte	Contents
1	LSB = AA (KeyValue for memory width = 8 bits)
2	MSB = 08h (KeyValue for memory width = 8 bits)
3	LSB = Register initialization value or reserved for future use
4	MSB = Register initialization value or reserved for future use
…	…
17	LSB = Reserved for future use
18	MSB = Reserved for future use
19	LSB = Upper half of entry point (PC[23:16]
20	MSB = Upper half of entry point (PC[31:24]
21	LSB = Lower half of entry point PC[7:0]
22	MSB = Lower half of entry point PC[15:8]
23	LSB = Block size in words of the first block to load. If the block size is 0, this indicates the end of the source program. Otherwise, another block follows. For example, a block size of 0×000A would indicate 10 words of 20 bytes in the block.
24	MSB = Block size
25	LSB = Upper half of destination address of first block Addr[23:16]
26	MSB = Upper half of destination address of first block Addr[31:24]
27	LSB = Lower half of destination address of first block Addr[7:0]
28	MSB = Lower half of destination address of first block Addr[15:8]
29	LSB = First word of the first block being loaded
30	MSB = First word of the first block being loaded
	…
	…
	LSB = Last word of the first block of the source being loaded
	MSB = Last word of the first block of the source being loaded
	LSB = Block size of the second block
	MSB = Block size of the second block
	LSB = Upper half of destination address of second block Addr[23:16]
	MSB = Upper half of destination address of second block Addr[31:24]
	LSB = Lower half of destination address of second block Addr[15:8]
	MSB = Lower half of destination address of second block Addr[8:0]
	LSB = First word of the second block being loaded
	MSB = First word of the second block being loaded
	…
	…
	LSB = Last word of the second block of the source being loaded
	MSB = Last word of the second block of the source being loaded
	…
	…
	…

Table C-1. LSB/MSB Loading Sequence in 8-Bit Data Stream (continued)

Byte	Contents
	LSB = Block size of the last block
	MSB = Block size of the last block
	LSB = Upper half of the destination of last block Addr[23:16]
	MSB = Upper half of destination address of second block Addr[31:24]
	LSB = Lower half of destination address of second block Addr[15:8]
	MSB = Lower half of destination address of second block Addr[8:0]
	LSB = First word of the last block being loaded
	MSB = First word of the last block being loaded
	…
	…
	LSB = Last word of the last block being loaded
	MSB = Last word of the last block being loaded
n	LSB = 00h
n + 1	MSB = 00h - indicates the end of the source

Appendix D Hex-Conversion File Formats

D.1 *Intel*

An Intel® formatted hex file is used for the CKFA file conversion.
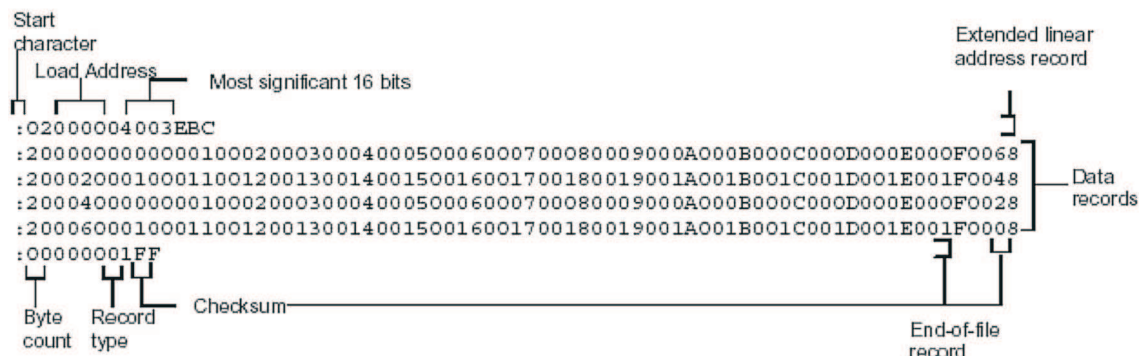
Figure D-1. Intel MCS86 Hexadecimal Object Format

Record Type	Description
00	Data Record
01	End-of-file record
04	Extended linear address record

D.2 *Motorola*

A Motorola® formatted hex file is used for the AppCode file conversion.
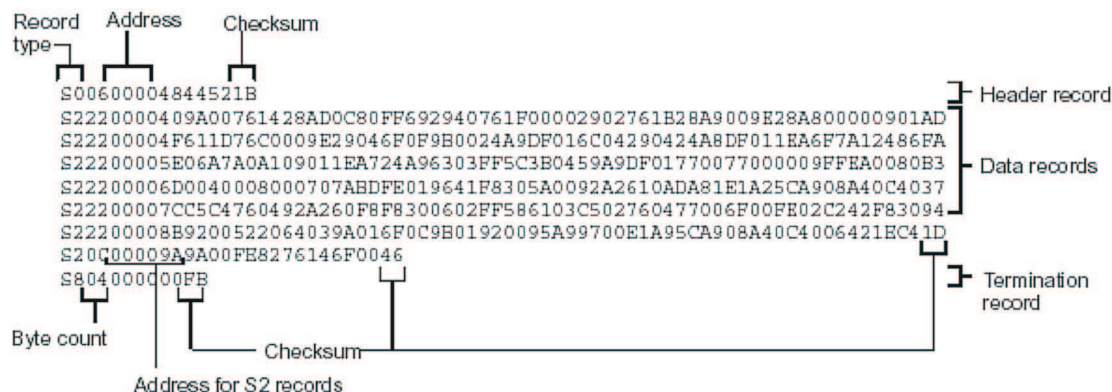
Figure D-2. Motorola-S Object Format

Record Type	Description
S0	Header Record
S2	Code/data record
S4	Termination record

Appendix E Software Flowcharts

E.1 CKFA

The following flowchart illustrates the CKFA execution order preparing the processor for the AppCode transfer.
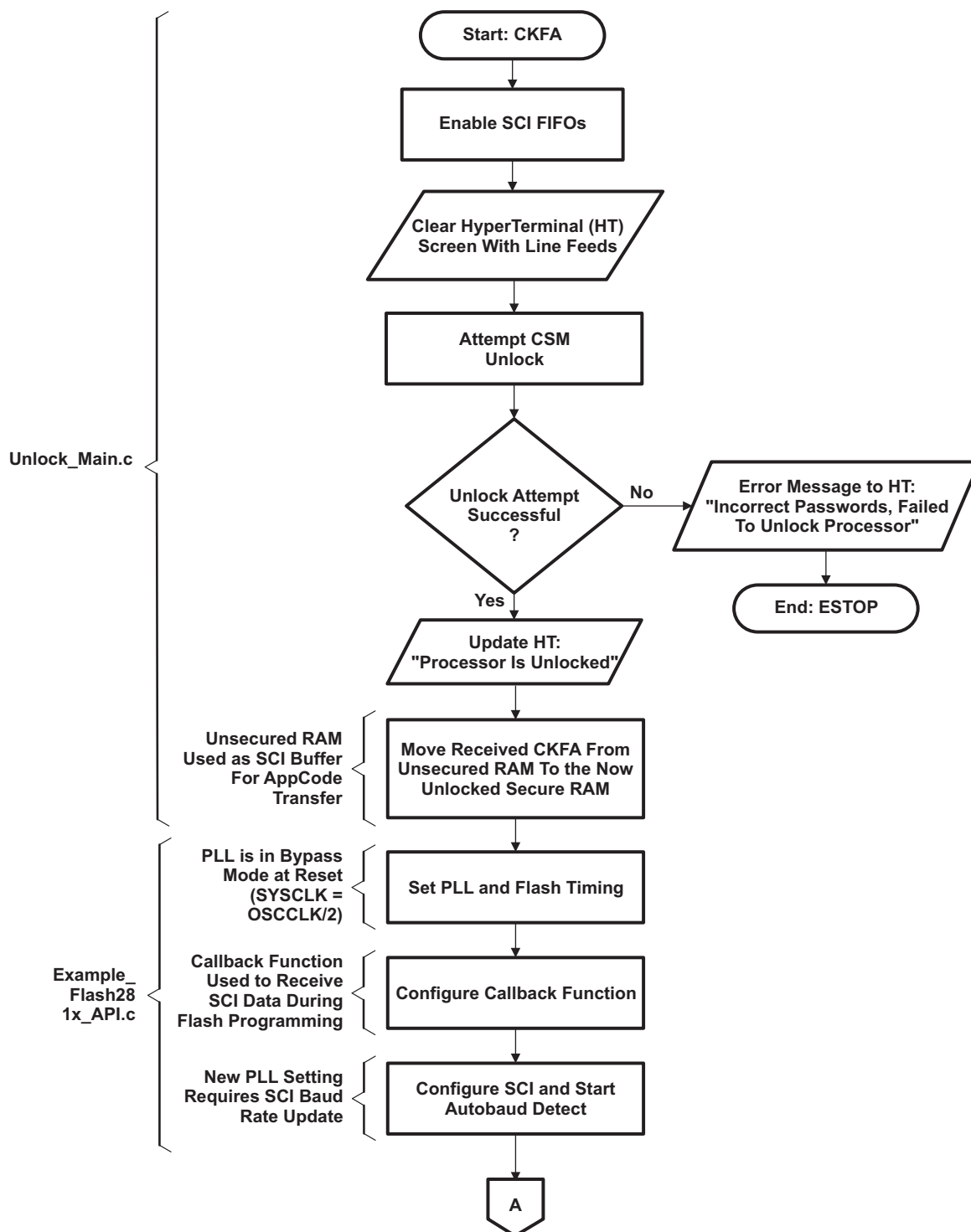
Figure E-1. CKFA Flowchart A
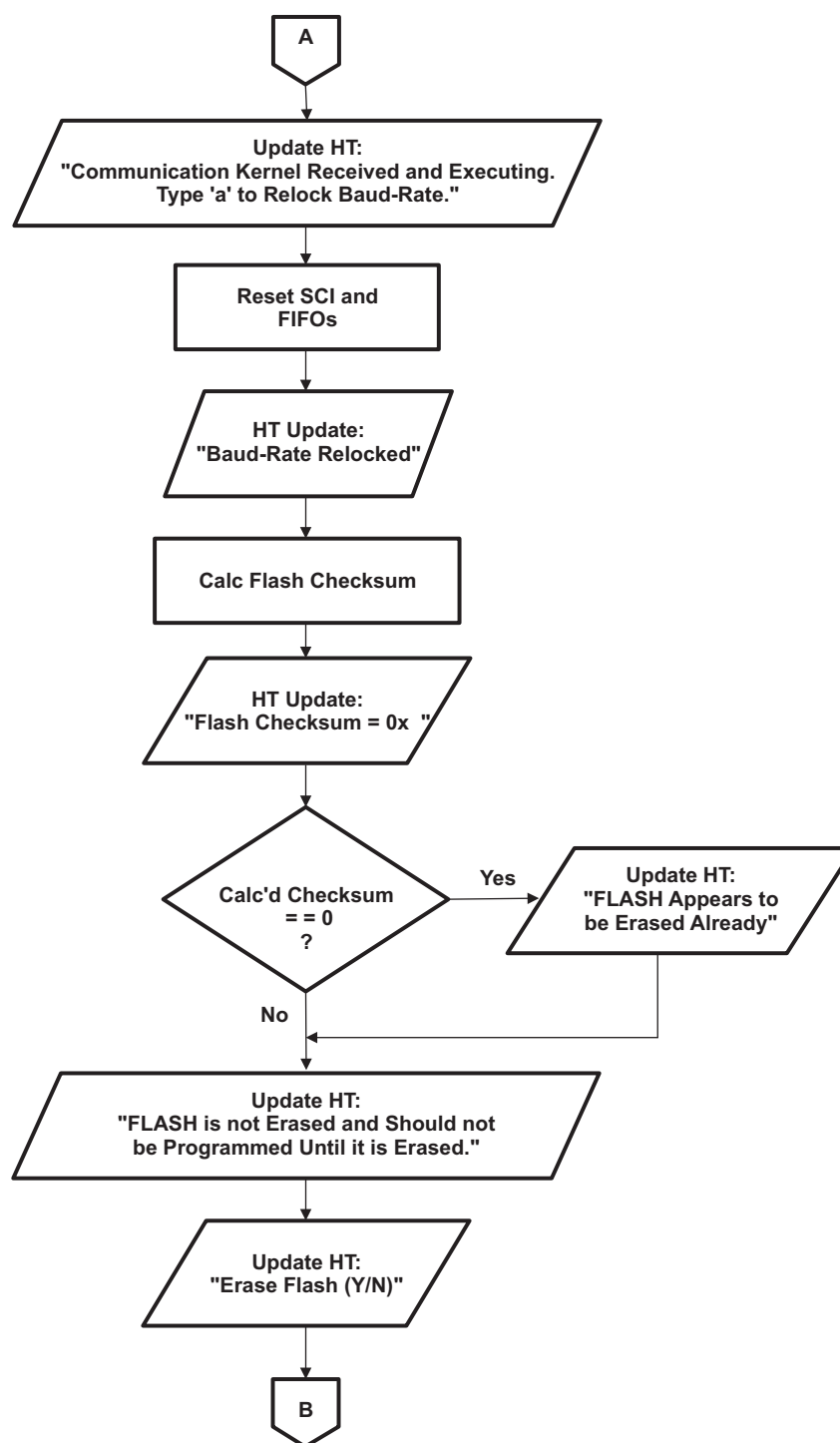
Figure E-2. CKFA Flowchart B
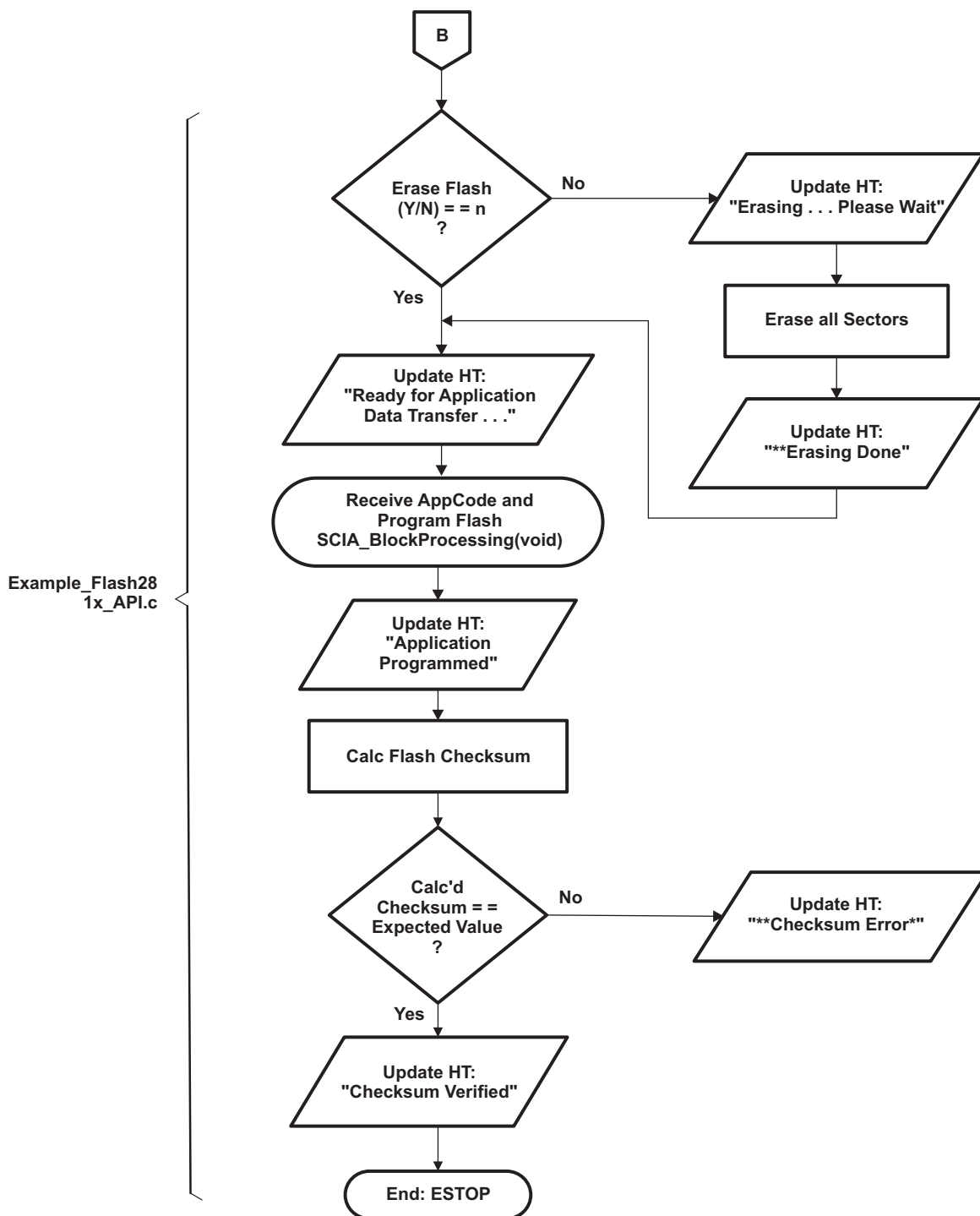
Figure E-3. CKFA Flowchart Through ESTOP

E.2 SCI Block Processing

The following flowchart illustrates the portion of the CKFA software that processes blocks of AppCode data transfers. Blocks are received and then programmed into Flash.
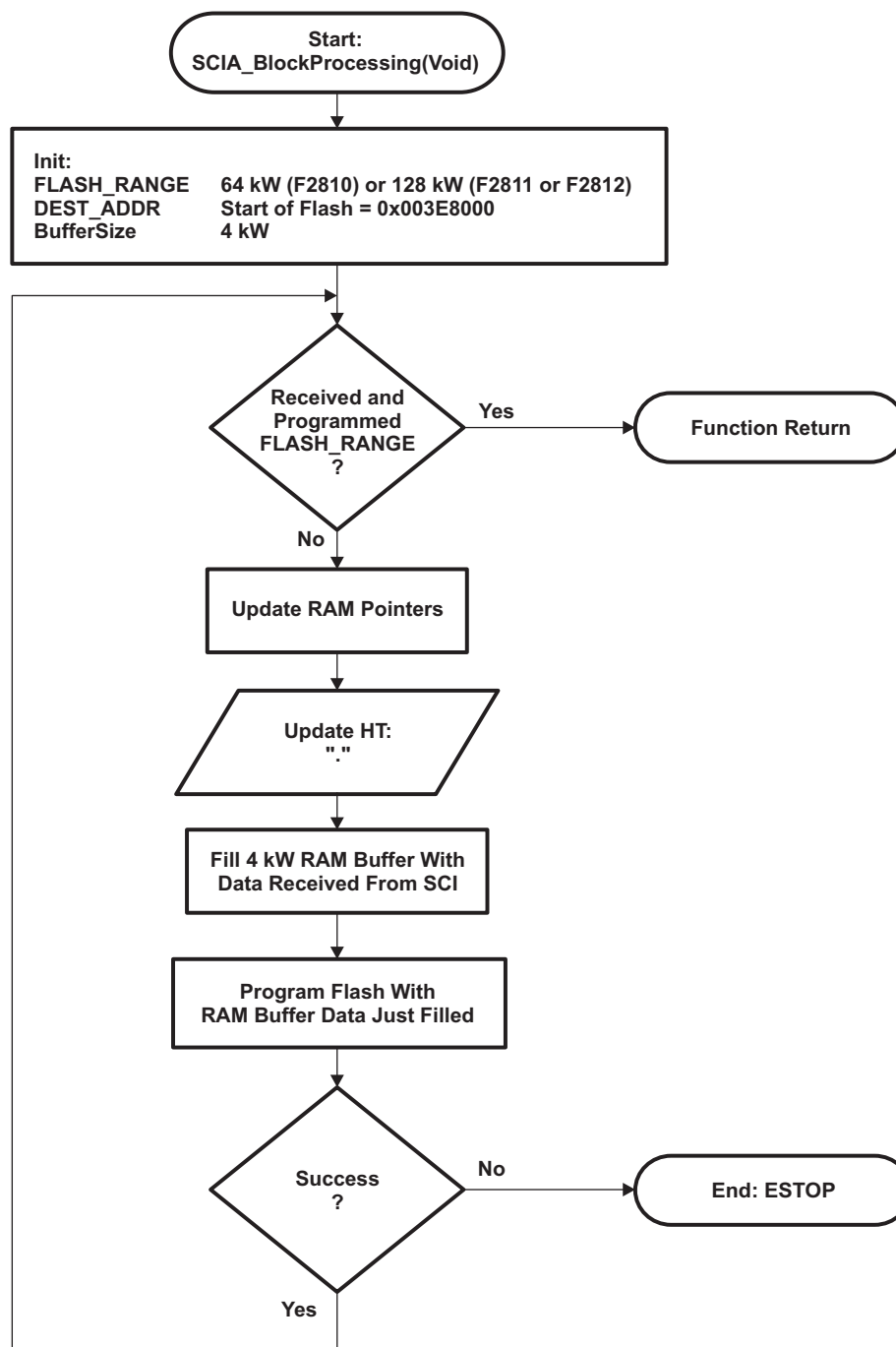
Figure E-4. SCI Block Processing Flowchart

Appendix F CKFA Linker and HEX2000 MAP Files

F.1 CKFA Linker Command File

The CKFA linker command file is crucial to this application report. Note the distinctions between unsecured and secured memory for the CKFA code (.text). The unsecured memory executes first, allowing the CKFA to unlock the processor. Note also the overlay of H0RAM. This memory range is used for two purposes: holding the CKFA software until the processor is unlocked and as a buffer for AppCode transfers.

F.1.1 File Description – CKFA.cmd

The linker command file for the CKFA software plays a crucial role. The initial transfer of the entire CKFA must assign addresses to unsecured memory to allow the SCI-A Boot ROM code to transfer the CKFA software to RAM even if the device is locked.

Once the CKFA is transferred, the CSM is unlocked with the passwords used in Example_Flash281x_CsmKeys.asm. After unlocking the CSM, the main CKFA software is transferred to secured RAM and the large unsecured RAM is used for SCI communications buffers when transferring the application software.

In summary, the CKFA.cmd does several tasks:

- Assigns load and run addresses to unsecured memory
- Assigns load addresses to unsecured memory and run addresses to secured memory
- Overlays load addresses of unsecured memory

Example F-1. CKFA.cmd

```
PAGE 1 :
   SCIA        : origin = 0x007050, length = 0x000010    /* SCI-A registers */
   RAML0L1     : origin = 0x008000, length = 0x002000    /* on-chip RAM block L0 L1 */
   RAMM0M1     : origin = 0x000200, length = 0x000600    /* on-chip RAM block M0 */
   RAMH0_1     : origin = 0x3F8000, length = 0x001000    /* on-chip RAM block H0 */
   RAMH0_2     : origin = 0x3F9000, length = 0x001000    /* on-chip RAM block H0 */

PAGE 2 :
   RAMH0_1     : origin = 0x3F8000, length = 0x001000    /* on-chip RAM block H0 */
   RAMH0_2     : origin = 0x3F9000, length = 0x001000    /* on-chip RAM block H0 */

SECTIONS
{

    .text_unsecured    : > RAMM0M1    PAGE = 0
    {
        unlock_main.obj (.text)
        DSP281x_CodeStartBranch.obj (.text)
        Example_Flash281x_CsmKeys.obj (.text)
        DSP281x_MemCopy.obj (.text)
        rts2800_ml.lib (.text)
    }
    econst_unsecured    : > RAMM0M1    PAGE = 0
    {
        unlock_main.obj (.econst)
    }


    .text              : LOAD = RAMH0_1,
                         RUN = RAML0L1,
                         LOAD_START(_textLoadStart),
                         LOAD_END(_textLoadEnd),
                         RUN_START(_textRunStart),
                         PAGE = 1

    .econst            : LOAD = RAMH0_1,
                         RUN = RAML0L1,
                         LOAD_START(_econstLoadStart),
```

Example F-1. CKFA.cmd (continued)

```
                                LOAD_END(_econstLoadEnd),
                                RUN_START(_econstRunStart),
                                PAGE = 1

        .cinit                  : LOAD = RAMH0_1,
                                RUN = RAML0L1,
                                LOAD_START(_cinitLoadStart),
                                LOAD_END(_cinitLoadEnd),
                                RUN_START(_cinitRunStart),
                                PAGE = 1


        /* User Defined Sections   */
        codestart               : > RAMM0M1         PAGE = 0
        BlockTransferBuffer1   : > RAMH0_1         PAGE = 2
        BlockTransferBuffer2   : > RAMH0_2         PAGE = 2
        csmpasswds              : > CSM_PWL         PAGE = 0
        csm_rsvd                : > CSM_RSVD        PAGE = 0
```

F.2 CKFA Linker MAP File – RAM Overlay

```
*******************************************************************************
        TMS320C2000 COFF Linker PC v4.1.0
*******************************************************************************

OUTPUT FILE NAME:   <./Debug/CKFA.out>
ENTRY POINT SYMBOL: "code_start"  address: 000002f2


MEMORY CONFIGURATION

                  name          origin    length    used     attr   fill
               ----------------- -------- --------- -------- ---- --------
PAGE  0: RAMM0M1               00000100  00000200  000001f3  RWIX
         OTP                   003d7800  00000800  00000000  RWIX
         FLASHJ                003d8000  00002000  00000000  RWIX
         FLASHI                003da000  00002000  00000000  RWIX
         FLASHH                003dc000  00004000  00000000  RWIX
         FLASHG                003e0000  00004000  00000000  RWIX
         FLASHF                003e4000  00004000  00000000  RWIX
         FLASHE                003e8000  00004000  00000000  RWIX
         FLASHD                003ec000  00004000  00000000  RWIX
         FLASHC                003f0000  00004000  00000000  RWIX
         FLASHB                003f4000  00002000  00000000  RWIX
         FLASHA                003f6000  00001f80  00000000  RWIX
         CSM_RSVD              003f7f80  00000076  00000000  RWIX
         BEGIN                 003f7ff6  00000002  00000000  RWIX
         CSM_PWL               003f7ff8  00000008  00000000  RWIX
         ROM                   003ff000  00000fc0  00000000  RWIX
         RESET                 003fffc0  00000002  00000000  RWIX
         VECTORS               003fffc2  0000003e  00000000  RWIX

PAGE  1: RAMM0M1               00000300  00000500  00000400  RWIX
         SCIA                  00007050  00000010  00000010  RWIX
         RAML0L1               00008000  00002000  00000a9c  RWIX
         RAMH0_1               003f8000  00001000  000009fe  RWIX
         RAMH0_2               003f9000  00001000  00000000  RWIX

PAGE  2: RAMH0_1               003f8000  00001000  00001000  RWIX
         RAMH0_2               003f9000  00001000  00001000  RWIX
```

The CKFA software uses the linker to overlay RAM. Note that RAMH0_1 and RAMH0_2 are used on both PAGE 1 and PAGE 2. During the Boot ROM SCI-A transfer of the CKFA software at reset, the DSP could be locked so the CKFA software is transferred to unsecured RAM (H0). Once the DSP is unlocked, the CKFA software is transferred to the run address of 0×8000. This allows H0RAM to be used as SCI RAM buffers when transferring the AppCode software.

```
        .text    1   003f8000   000007ca    RUN ADDR = 00008000
```

```
                       003f8000    000001d2    Example_Flash281x_API.obj (.text)
                       003f81d2    00000047    HexToASCII.obj (.text)
                       003f8219    000000c6    SCI.obj (.text)
                       003f82df    00000091    Flash2810_API_V210.lib : (.text)

.econst    1   003f87ca    00000234    RUN ADDR = 000087ca
                       003f87ca    00000201    Example_Flash281x_API.obj (.econst)
                       003f89cb    00000001    --HOLE-- [fill = 0]
                       003f89cc    00000032    Flash2810_API_V210.lib : (.econst)

BlockTransferBuffer1
*          2   003f8000    00001000    UNINITIALIZED
                       003f8000    00001000     Example_Flash281x_API.obj (BlockTransferBuffer1)

BlockTransferBuffer2
*          2   003f9000    00001000    UNINITIALIZED
                       003f9000    00001000     Example_Flash281x_API.obj (BlockTransferBuffer2)
```

F.3   CKFA HEX2000 MAP File – 8-Bit SCI Boot Format

The CKFA software is converted into a format suitable for SCI transfer by the code in Boot ROM. This requires an 8-bit format, header information, and each data block listed with address and size information. The Appcode software does not require boot-loading format because its transfer is controlled by the CKFA software.

```
********************************************************************************
TMS320C2000 COFF/Hex Converter                                            v4.3.0
********************************************************************************

INPUT FILE NAME: <CKFA.out>
OUTPUT FORMAT:   Intel

PHYSICAL MEMORY PARAMETERS
   Default data width   :  16
   Default memory width :   8 (LS-->MS)
   Default output width :   8

BOOT LOADER PARAMETERS
   Table Type:                     SERIAL PORT (SCI 8 bit Mode)
   Entry Point:                     0x000002f2


OUTPUT TRANSLATION MAP
------------------------------------------------------------------------------
00000000..003fffff  Page=0  Memory Width=8  ROM Width=8
------------------------------------------------------------------------------
   OUTPUT FILES: CKFA.hex [b0..b7]

   CONTENTS: 00000000..0000181d   BOOT TABLE
                   .text_unsecured : dest=00000100  size=0000016b  width=00000002
                 .econst_unsecured : dest=0000026c  size=0000005f  width=00000002
                             .cinit : dest=000002cb  size=00000027  width=00000002
                          codestart : dest=000002f2  size=00000002  width=00000002
                              .text : dest=003f8000  size=000007ca  width=00000002
                            .econst : dest=003f87ca  size=00000234  width=00000002


------------------------------------------------------------------------------
00000000..003fffff  Page=1  Memory Width=8  ROM Width=8  "*DEFAULT PAGE 1*"
------------------------------------------------------------------------------
   NO CONTENTS
```

Appendix G Example Software – File Listing and Descriptions

The application code used in this report is based on the Flash example that is included in the *Download: C281x C/C++ Header Files and Peripheral Examples* (SPRC097). The only modification that was made was a function to toggle the GPIO to confirm that the application was executing from Flash.

G.1 Directory Structure and File Listing

The application code's Code Composer Studio project consists of the files shown in Table G-1 through Table G-5.

Table G-1. Directory Structure Used in This Application Report

Directory	Contents
C:\CCStudio_v*x*	Code Composer Studio installation directory
\code\Appcode	Application software
\code\EICT	Emulated ICT software
\code\FileIOShell	MOT2BIN software

Table G-2. CKFA Files Used in This Application Report

Filename	Contents
\code\CKFA\CKFA.pjt	Code Composer Studio project for CKFA software
\code\CKFA\CKFA_COFF2BIN.bat	Batch file used in final Code Composer Studio build steps for CKFA code – converts COFF to binary format
\code\CKFA\CKFA.cmd	CKFA linker command file
\code\CKFA\source\Example_Flash281x_API.c	Main CKFA source file
\code\CKFA\source\Unlock_main.c	Source code that unlocks CSM
\code\CKFA\source\SCI.c	Miscellaneous SCI functions
\code\CKFA\source HexToASCII.c	Conversion code to display checksum in ASCII format
\code\CKFA\source\DSP281x_MemCopy.c	Memory copy utility
\code\CKFA\source\Example_Flash281x_CsmKeys.asm	Passwords used to unlock CSM
\code\CKFA\source\DSP281x_CodeStartBranch.asm	CKFA code entry point
\code\CKFA\API Libraries\Flash2810_API_V210.lib	F2810 Flash API Library
\code\CKFA\API Libraries\Flash2811_API_V210.lib	F2811 FLASH API Library
\code\CKFA\API Libraries\Flash2812_API_V210.lib	F2812 Flash API Library
\code\CKFA\Debug\CKFA.out	CKFA COFF executable
\code\CKFA\Debug\CKFA.map	CKFA COFF executable MAP file
\code\CKFA\Debug\CKFA_hex.cmd	Command line input to hex2000.exe
\code\CKFA\Debug\CKFA.hex	HEX file output from hex2000.exe (input to hex2bin.exe)
\code\CKFA\Debug\hex2bin.exe	Intel HEX to binary converter
\code\CKFA\Debug\CKFA.bin	Binary file output from hex2bin.exe

Table G-3. AppCode Files Used in This Application Report

Filename	Contents
\code\AppCode\AppCode.pjt	Code Composer Studio project for application software
\code\AppCode\AppCode_COFF2BIN_2810.bat	F2810 batch file used in final Code Composer Studio build steps for application code – converts COFF to custom binary format
\code\AppCode\AppCode_COFF2BIN_2812.bat	F2811/F2812 batch file used in final Code Composer Studio build steps for application code – converts COFF to custom binary format
\code\AppCode\Debug\AppCode.out	Application code's COFF executable
\code\AppCode\Debug\ AppCode.map	Application code's COFF executable MAP file
\code\AppCode\Debug\AppCode_hex_2810.cmd	F2810 command line input to hex2000.exe
\code\AppCode\Debug\AppCode_hex_2812.cmd	F2811/F2812 command line input to hex2000.exe
\code\AppCode\Debug\AppCode.hex	HEX file output from hex2000.exe (input to FileIOShell.exe)
\code\AppCode\Debug\FileIOShell.exe	Motorola-S record to binary converter
\code\AppCode\Debug\ AppCode.bin	Binary file output from FileIOShell.exe
\code\AppCode\64kW Hex file \AppCode.bin	Pre-configured 64 KW binary file for report's example application code
\code\AppCode\64kW Hex file \AppCode.hex	Pre-configured 64 KW hex file for report's example application code
\code\AppCode\128kW Hex file \AppCode.bin	Pre-configured 128 KW binary file for report's example application code
\code\AppCode\128kW Hex file \AppCode.hex	Pre-configured 128 KW hex file for report's example application code

Table G-4. EICT Files Used in This Application Report

Filename	Contents
\code\EICT\EICT.pjt	Code Composer Studio project for EICT software
\code\EICT\max_baud_rate.c	EICT software source code

Table G-5. FileIOShell Files Used in This Application Report

Filename	Contents
\code\FileIOShell\FileIOShell.cpp	FileIOShell source code
\code\FileIOShell\FileLibrary.cpp	FileIOShell source code

IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI's terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of TI information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation. Information of third parties may be subject to additional restrictions.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

TI products are not authorized for use in safety-critical applications (such as life support) where a failure of the TI product would reasonably be expected to cause severe personal injury or death, unless officers of the parties have executed an agreement specifically governing such use. Buyers represent that they have all necessary expertise in the safety and regulatory ramifications of their applications, and acknowledge and agree that they are solely responsible for all legal, regulatory and safety-related requirements concerning their products and any use of TI products in such safety-critical applications, notwithstanding any applications-related information or support that may be provided by TI. Further, Buyers must fully indemnify TI and its representatives against any damages arising out of the use of TI products in such safety-critical applications.

TI products are neither designed nor intended for use in military/aerospace applications or environments unless the TI products are specifically designated by TI as military-grade or "enhanced plastic." Only products designated by TI as military-grade meet military specifications. Buyers acknowledge and agree that any such use of TI products which TI has not designated as military-grade is solely at the Buyer's risk, and that they are solely responsible for compliance with all legal and regulatory requirements in connection with such use.

TI products are neither designed nor intended for use in automotive applications or environments unless the specific TI products are designated by TI as compliant with ISO/TS 16949 requirements. Buyers acknowledge and agree that, if they use any non-designated products in automotive applications, TI will not be responsible for any failure to meet such requirements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

Products		Applications	
Amplifiers	amplifier.ti.com	Audio	www.ti.com/audio
Data Converters	dataconverter.ti.com	Automotive	www.ti.com/automotive
DSP	dsp.ti.com	Broadband	www.ti.com/broadband
Interface	interface.ti.com	Digital Control	www.ti.com/digitalcontrol
Logic	logic.ti.com	Military	www.ti.com/military
Power Mgmt	power.ti.com	Optical Networking	www.ti.com/opticalnetwork
Microcontrollers	microcontroller.ti.com	Security	www.ti.com/security
RFID	www.ti-rfid.com	Telephony	www.ti.com/telephony
Low Power Wireless	www.ti.com/lpw	Video & Imaging	www.ti.com/video
		Wireless	www.ti.com/wireless

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265
Copyright © 2007, Texas Instruments Incorporated