Developing a CCStudio 2.0 DSP/BIOS Application for FLASH Booting on the TMS320C5402 DSK

Xiaozhen Zhang *Software Development Systems*

ABSTRACT

This application report describes the process of implementing a DSP/BIOS application in FLASH ROM with Code Composer Studio™ IDE 2.0, and notes the specific issues encountered during this process. The first half of the note will mainly be about creating a small (less than 48K words) DSP/BIOS application; the second half of the note is about building a secondary boot loader to support large applications. The reader will learn how to create a secondary boot loader, link the program appropriately, and prepare the ROM image. Examples are given for using FLASH ROM on the TMS320VC5402 DSK development board.

Contents

List of Figures

List of Tables

1 Introduction

Building a Code Composer Studio (CCStudio) 2.0 DSP/BIOS application for FLASH booting on the TMS320VC5402 DSK (C5402 DSK) involves only a few simple steps:

1. Create and build a project using CCStudio 2.0 build.
2. Use hex500 utility to create a boot table from project.out.
3. Use the FLASH programmer to program the boot table onto the FLASH.
4. Use the boot loader to transfer user application from FLASH to RAM.

In the following pages, the steps of how to accomplish the above procedures will be described in detail. For the terminology used in this application report, a list of related terms is provided in Section 8.

2 ROM Boot Overview

The TMS320UC5402 and TMS320VC5402 (hereafter referred to as C5402) digital signal processors have a built-in boot loader which resides in the device’s on-chip ROM. The function of the boot loader is to transfer user code from an external source to the program memory upon power up. This allows code to reside in slow non-volatile memory externally, and be transferred to high-speed memory when executed.

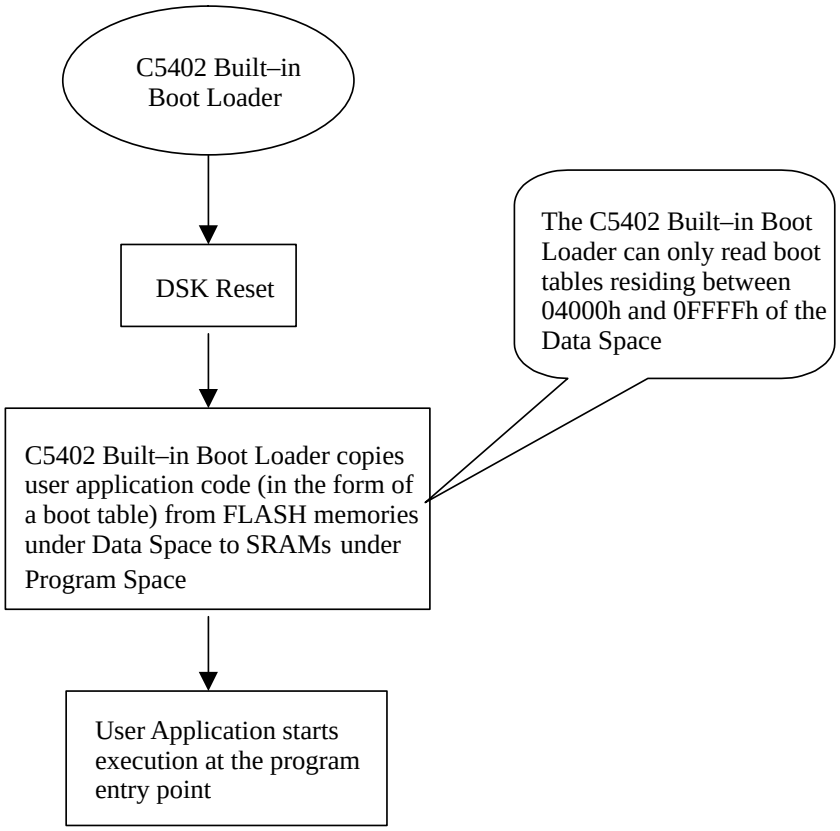

Figure 1. Generic C5402 Boot Loading Sequence in Parallel Boot Mode

Figure 1 illustrates the major steps involved in boot loading a small (less than 48K 16-bit words) user application with the C5402 built-in boot loader. The following paragraphs will detail the key assumption that went into Figure 1, the major components of the figure, and an analysis of the limitations of the approach.

Key Assumption:

Upon power-up, the C5402 boot loader automatically goes through a set of procedures to determine the valid boot mode. The C5402 boot loader provides a variety of different ways to download code to accommodate different system requirements. Boot loading on the C5402 DSK is supported via the parallel boot mode in both 8-bit byte and 16-bit word. Please note that in order to better illustrate the relationship between the boot loader and the user application code, Figure 1 has intentionally left out the steps that the C5402 boot loader uses to determine the appropriate boot mode. This figure assumes that a valid parallel mode condition is met.

Key Components:

- A special table (boot table) is used to store the user code and related header information in hex format onto the external non-volatile memory, such as the FLASH memories on the C5402 DSK.
- The boot table must be stored as a hex image on the FLASH memories because EPROMs can not store object code directly.
- Since the output file created by CCStudio IDE 2.0 is in object code format, a hex conversion utility must be used to build the boot table for the boot loader.

If the boot table has been created correctly by the user, the C5402 boot loader automatically copies all sections in FLASH memory mapped under the Data Page to program memory. The boot table should provide all the information necessary for the transfer. Header information gets stripped away while the transfer is being done. Program starts execution from the entry point at the end of the transfer.

Key Limitation:

This boot loading method has an important limitation that limits the size of the user application to 48K 16-bit words or smaller. The limitation that the application size must be smaller than 48K words comes inherently from the parallel boot mode itself. In the parallel boot mode, the C5402 boot loader reads the boot table from the data space. The boot table can only be located at data space addresses between 4000h and FFFFh. Since there is only one page of data memory on the C5402, this limits the application size to be less than 48K words.

Demand for a Secondary Boot Loader:

The C5402 supports eight pages of program memories. For FLASH ROM application with size larger than 48K words, it is certainly desirable to have a boot loader that can read boot tables under the extended program space. Depending on the hardware, of course, this approach can potentially increase the bootable application size to eight times bigger, or 384K words. Letting that be the idea, create another boot loader, calling it the Secondary Boot Loader, that does exactly what has been envisioned.

Figure 2 will illustrate the sequence of events following power-on with the Secondary Boot Loader in place.

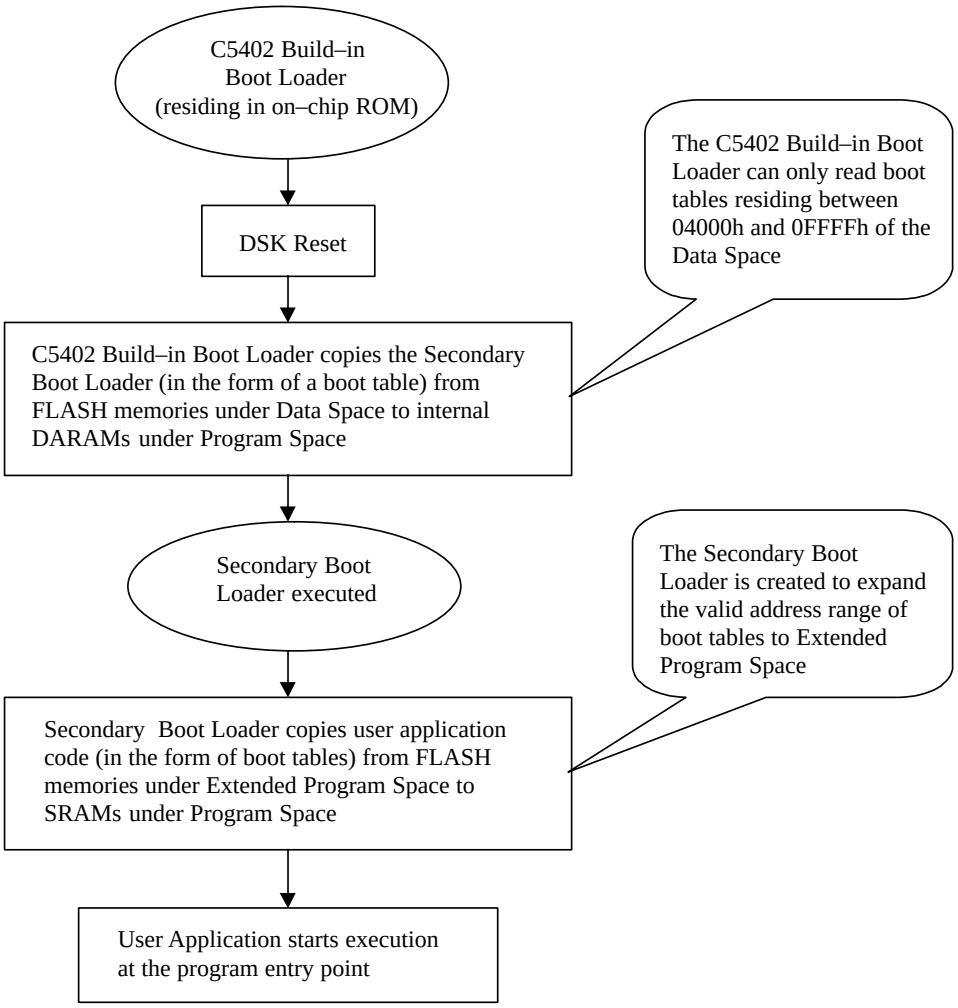

Figure 2. Boot Loading in Parallel Boot Mode with the Secondary Boot Loader

Key Components:

- Compared to Figure 1, Figure 2 now shows the secondary boot loader between the C5402 boot loader and the original user application. A ROM image (boot table) is first created for the secondary boot loader and placed under the Data Page so that the C5402 built-in boot loader is able to copy it into internal RAM and execute it. Remember, the secondary boot loader itself is a user created code that must be executed through the C5402 built-in boot loader.
- Another ROM image is created for the original user application. Since the original user application is large, multiple boot tables must be created for it. The boot tables should be placed under the extended program space for boot loading. The secondary boot loader reads and copies the user application sections into their appropriate location in SRAM.

If the boot tables have been created correctly by the user, the C5402 boot loader automatically copies the secondary boot loader in FLASH memory mapped under the Data Page to internal RAM and executes it. The secondary boot loader then copies the ROM image for the user application in FLASH memory mapped under the extended program space to SRAM. The boot tables should provide all the information necessary for the transfer. Header information gets stripped away while the transfer is being done. User application starts execution from the entry point at the end of the transfer.

Key Limitation:

The maximum size of each boot table is limited to 32K words. The hex500 utility has a 16-bit byte addressable space; the utility can only build boot tables that are less than or equal to 32K 16-bit words. Therefore, multiple boot tables need to be built for a single application if the size of the application exceeds 32K 16-bit words.

To summarize the ideas that are introduced in this section:

When a DSP/BIOS application is booted from FLASH, depends on the size of the application, one of the two following approaches would take place:

For small application (smaller than 48K words):

- Upon reset, the C5402 built-in boot loader will perform section copying to initialize program memory as needed (copy user application from FLASH into RAM).
- DSP/BIOS startup procedures will initialize C variables in the .bss section with data contained in the .cinit section, as well as perform general DSP/BIOS setup.
- The application will begin execution at the entry point, generally c_int00.

For large application (larger than 48K words):

- A secondary boot loader must be created to extend the capability of the TMS3205402 built-in boot loader.
- Upon reset, the C5402 boot loader will load and execute the secondary boot loader as described above.
- The secondary boot loader will perform additional section copying into extended program space that is not reachable by the TMS320C5402 built-in boot loader and to initialize program and data memory as needed.
- DSP/BIOS startup procedures will initialize C variables in the .bss section with data contained in the .cinit section, as well as perform general DSP/BIOS setup.
- The application will begin execution at the entry point, generally c_int00.

The first half of the application report will detail on how to build a small FLASH ROM application. The second half of the report will describe how to create a secondary boot loader to build a large FLASH ROM application. Please note that many of the aspects discussed in building a small FLASH ROM application is also applicable for creating a large FLASH ROM application. Therefore, only the difference will be described in the second half of the application report, rather than trying to repeat describing the entire procedure.

Before starting, the C5402 DSK needs to be set up properly. The same board configuration is used for both approaches.

3 Set Up the TMS320C5402 DSK

When using the C5402 DSK connected through the parallel port (the printer port), make sure the DPI switch settings are as of the following:

Table 1. C5402 DSK DPI Switch Settings for the Parallel Port

DPI Switch #	Settings
1	On (down)
2	On (down)
3	On (down)
4	On (down)
5	Off (up)
6	Off (up)
7	On (down)
8	On (down)

If your C5402 DSK is connected to the PC through XDS510, make sure that DPI Switch #1 is set to Off (up) with everything else in the above table staying the same.

4 Project Configuration for the Small ROM Approach

Before building your project, there are a few basic settings to configure so that Code Composer Studio IDE will generate code suitable for booting a small application from FLASH.

4.1 Memory Configuration

The DSP needs to be set in Microcomputer Mode (MP/MC=0), the default reset mode, in order to see the boot code in the on-chip ROM of the Program space. In addition, OVLY bit is set to 1 to allow the mapping of the internal DARAMs into both program/data spaces.

Figure 3 shows the DSK memory map with MP/MC=0 and OVLY=1. Note that only Page 0 Program is shown due to the limited memory space applicable for this method.

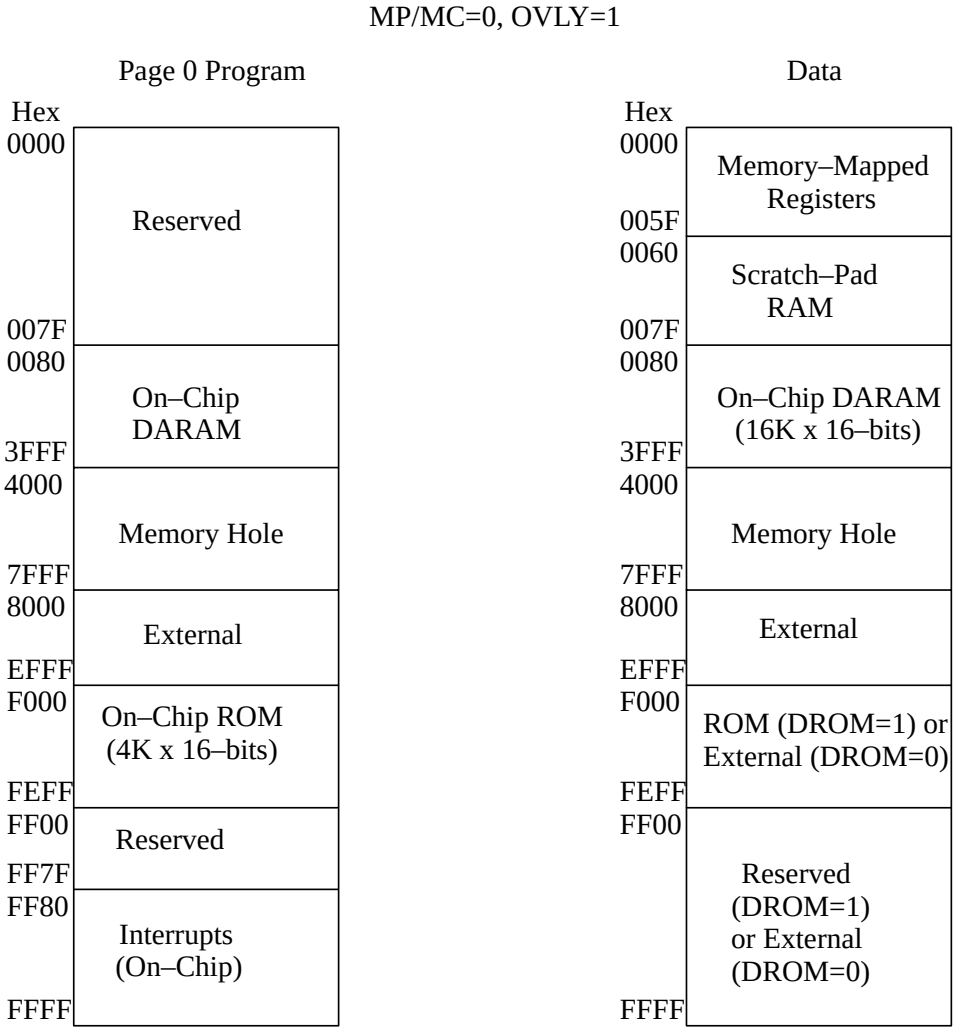

Figure 3. Page 0 Memory Map for the C5402 DSK with MP/MC=0 and OVLY=1

The C5402 has on-chip RAM between 0x0080 and 0x4000. The memory is mapped to external memory for addresses above 0x4000. The C5402 DSK has a memory hole between 0x4000 and 0x8000 for both the Program Page and the Data Page. Therefore, when using the C5402 DSK users should be cautious not to put valid data into this memory segment. In addition, it is important to note that sections should not be linked into address ranges where no RAM is actually present in the system. For example, no sections should be linked to address ranges 0xF000h-0xFFFFh, because this range of program space is occupied by the on-chip ROM and cannot be written to by the boot loader.

The user must place all code and data sections below 0x4000 if the code is in the internal RAM. The default configuration for MEM includes the following memory sections: USERREGS, BIOSREGS, VECT, IDATA, IPROG, EDATA, EPROG. The default configuration in CCStudio IDE 2.0 needs to be modified for the DSK. Below is a sample memory configuration used for the audio program in CCStudio 2.0 tutorial after such a modification.

Table 2. Sample Memory Configuration for dsk5402.cdb

name	page	origin	length
USERREGS	PAGE 1	00000060	0000001C
BIOSREGS	PAGE 1	0000007C	00000004
VECT	PAGE 0	00003F80	00000080
IDATA	PAGE 1	00000080	00001F80
IPROG	PAGE 0	00002000	00001F80
EDATA	PAGE 1	00008000	00008000
EPROG	PAGE 0	00008000	00007000

NOTE: External Memory (EDATA, EPROG) can either be FLASH or SRAM depending on the setting of control register bit FLASHENB. If FLASHENB = 1 (power-on default), this will be FLASH (seven-wait states); if FLASHENB = 0, this will be SRAM (one-wait state).

4.2 Build Options

In the Code Composer Studio menu bar, select Project -> Build Options

- Click on the Compiler tab. **For Processor Version, type in 548.** This compiles the project using the -v548 compiler option. This option marks the object files produced by the assembler specifically for the devices with enhanced boot loader functions such as the C5402. The hex conversion utility uses this information to generate the correct format for the boot table. If this option is not included during assembly, the hex conversion utility may produce a version of the boot table intended for earlier TMS320CC54x™ devices without generating warnings or errors.
- Click on the Linker tab. **For Autoinit Model, select Run-time Autoinitialization.** This linker option autoinitializes variables at run time.
- Click on the Linker tab. Specify a map filename so that the linker will generate a map file. This file will contain linking information that is necessary to understand section placement and to write your boot code.

5 Build a Small FLASH ROM Application

Code Composer Studio comes with several utilities that can help you to write your application into FLASH. For both the C54x™ and TMS320C6x™ architectures, a hex conversion utility and a flash programmer are provided. To write an application to a 5402 DSK, follow this procedure.

TMS320C54x, TMS320C6x, and C54x are trademarks of Texas Instruments.

5.1 Build and Debug Your Project to Generate a .out File

Before creating the ROM image for the application, build, load, and run the code under CCStudio IDE to make sure that the program works.

5.2 Create a Boot Table Using the TMS320C5000 Hex Conversion Utility

To use the features of the C5402 boot loader, generate a boot table, which contains all of the data the boot loader needs. The boot table (a ROM image) is generated by the TMS320C5000™ hex conversion utility tool.

This utility operates using command files. After obtaining the .out file from Section 5.1, in the command file, specify the input .out file, the format for the output .hex file, the type and size of FLASH for your board, and what sections to be placed in FLASH. Choose the appropriate options for the desired boot mode and run the hex conversion utility to convert the COFF file produced by the linker into a boot table. An example hex command file for a 5402 DSK is shown in Figure 4.

To invoke the Hex Conversion Utility, type in the command line:

```
hex500 hex.cmd
```

When the Hex Conversion Utility is invoked with the example command file of Figure 4, it creates a Motorola-S1 format hex file called audio.hex, which can be used to run the audio example on the CCStudio 2.0 Tutorial from the C5402 DSK. All of the initialized sections from the input file are placed in the boot table, and the entry point is set to address 0x3705 in this particular example. After building your project, you will have a map file generated by the linker that will contain important section link information. **Examine the .map file to find out about the entry point which is generally shown at the very top of the .map file.** The entry point address in Figure 4 is just an example, the actual number depends on your application.

In addition, each block of the boot table data corresponds to an initialized section in the COFF file. Initialized sections include; .text, .const, and .cinit. Uninitialized sections are for example .bss and .sysmem. You will also need to examine the .map file to find out more information about each section. All uninitialized sections put in the hex.cmd will be ignored by the hex500 utility.

TMS320C5000 is a trademark of Texas Instruments.

```
/* hex.cmd */
audio.out               /* Input COFF file */
–m1                     /* Select Motorola–S1 */
–map audiohex.map       /* Name hex utility map file */
–o audio.hex            /* Name hex output file */
–memwidth 16            /* Set EPROM system memory width */
–romwidth 16            /* Set physical ROM width */
–bootorg PARALLEL       /* Specify the source of the boot loader table as the
                           parallel port */
–e 0x3705               /* Entry point for the boot table; this number needs
                           to be edited by observing the .map file */
–swwsr 0x7fff           /* Software Wait State Register Value for PARALLEL/
                           WARM boot mode */
–bscr 0x8806            /* Bank–Switch Control Register Value for PARALLEL/
                           WARM boot mode */
ROMS
{
  PAGE 0:  FLASH: origin = 0x8000, len = 0x8000
}
SECTIONS
{
  .hwi_vec        boot
  .hwi            boot
  .bios           boot
  .gblinit        boot
  .trcdata        boot
  .sysinit        boot
  .const          boot
  .text           boot
  .cinit          boot
  .pinit          boot
  .switch         boot
  .csldata        boot
  .rtdx_text      boot
}
```

Figure 4. Hex.cmd - Example hex500.exe Command File

5.3 The Bootable COFF Sections

As a quick guide to what sections need to be specified for boot in hex.cmd, Table 3 contains the initialized sections that need to be booted from the FLASH.

Table 3. The Bootable DSP/BIOS and Compiler Sections

Section Name	Section Type	Memory Type	Description / Notes
.hwi_vec	Code	Program	Interrupt table
.hwi	Code	Program	Dispatch code for interrupt service routines
.bios	Code	Program	DSP/BIOS code
.gblinit	Initialized Data	Program	Init tables, used only during startup
.trcdata	Initialized Data	Program	Trace mask section
.sysinit	Code	Program	Init code, run only during startup
.text	Code	Program	Program code
.cinit, .pinit	Initialized Data	Program	C variable and function initialization tables
.switch	Initialized Data	Program	Switch statement jump tables
.rtdx_text	Code	Program	Code sections for the RTDX module
.const	Initialized Data	Data	Constants
.csldata	Initialized Data	Data	Chip support library

WARNING:

The C5402 Built-in Boot Loader copies all sections of the boot table (whether the section is a program section or a data section) into Program Memory. Therefore, this requires that .const and .csldata sections must be loaded into a memory region where Program Memory overlays the Data Memory.

5.4 Program your FLASH

After generating a .hex file, use a FLASH programming utility to write the hex image to the board's FLASH. Flashburn is a new FLASH programmer that is very easy to use and is capable of programming the entire FLASH memories available on the board at specified logical address. Flashburn is not part of the standard CCStudio 2.0 package, but is available through the CCStudio 2.0 Update Advisor.

- Make sure Code Composer Studio IDE is not running when executing the flash programming utility.
- If you are using the C5402 DSK, make sure the jumper setting is set for HPI boot.

Boot Jumper Settings:

  – *HPI boot: JP2 pin 2 and 3 connected.*

  – *Flash boot: remove jumper (or connect pin 1&2).*

- Invoke the Flashburn utility and program your FLASH.
- In a few seconds, your application will be in FLASH and ready to run.

- Then remove the HPI BOOT jumper setting to enable FLASH boot. Press the reset button, the program should boot from FLASH and start running.

5.5 The Small Application Flowchart

Figure 5 is a block diagram summarizing the flow for building a small FLASH ROM application on the C5402 DSK.

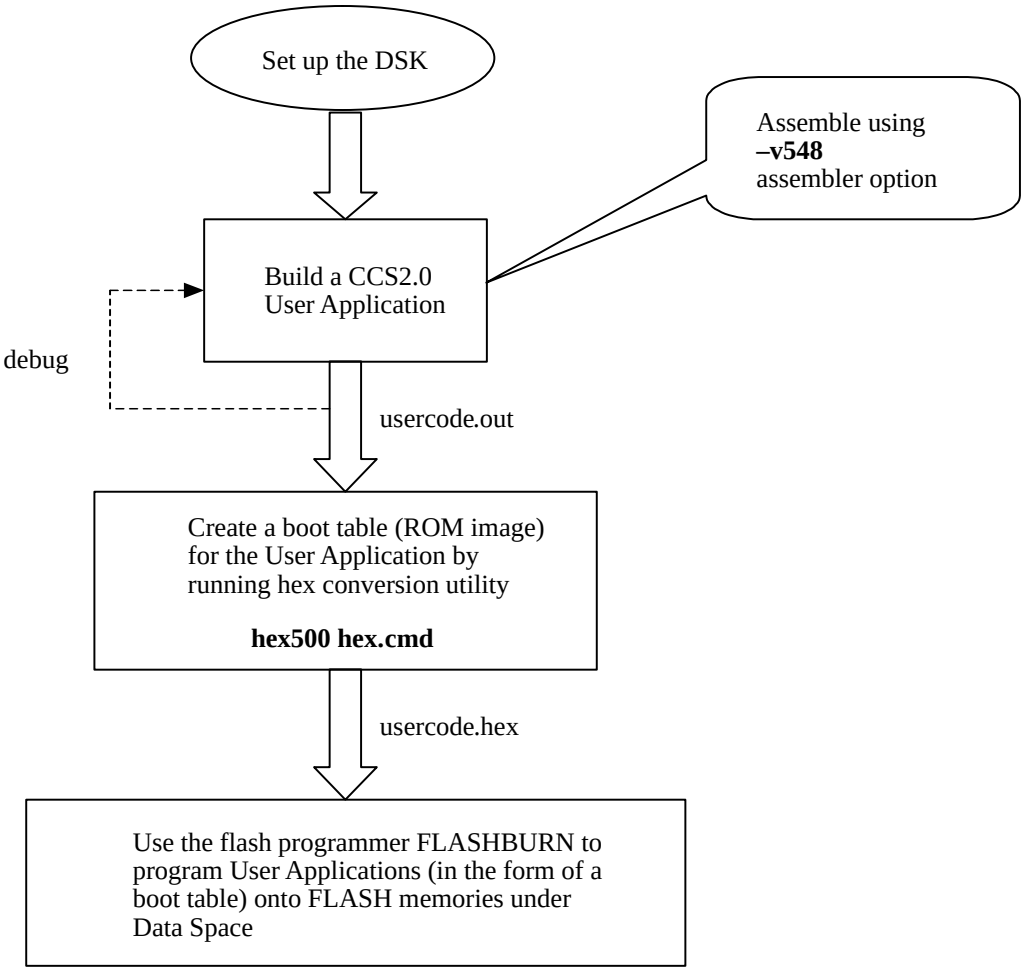

Figure 5. Flowchart for Building a Small FLASH ROM Application

5.6 The Limitations

Parallel boot mode reads the desired boot table from data space via the external parallel interface (external memory interface) and transfers the code to program space. The 5402 built-in boot loader gets the source address from either I/O port 0FFFFh or data memory address 0FFFFh. Since the source address read from either of these locations is 16 bits wide, the boot table can only reside in any valid external address range within the data space of C5402 memory map.

- The valid addresses for external data memory on the C5402 are between 04000h and 0FFFFh (48K 16-bit words).

- With the memory hole on the C5402 DSK, this restricts the valid addresses to be between 08000h and 0FFFFh (32K 16-bit words).

6 Build a Large FLASH ROM Application

If the size of your application is larger than the limits imposed by the C5402 built-in boot loader, you will need to create a secondary boot loader to allow FLASH booting from other memory regions besides those under the data space. The C5402 supports eight extended program spaces; utilizing the extended program spaces for storing the large boot table seems a natural thing to do.

The steps for this approach includes the following:

- Write a secondary boot loader.
- Build a boot table for the secondary boot loader in FLASH memory under Data Space.
- Organize your target memory.
- Provide a PROTECTED memory section in on-chip RAM for the secondary boot loader.
- Build another boot table for your actual application.
- Map the user application boot table to FLASH memory under extended Program Space.

This approach allows the secondary boot loader to be automatically copied into the internal RAM by the C5402 built-in boot loader and to be executed. The secondary boot loader would reach into the extended program space and copy user applications from FLASH into on-chip DARAM or external SRAM.

The above idea can be most easily illustrated in the following two diagrams.

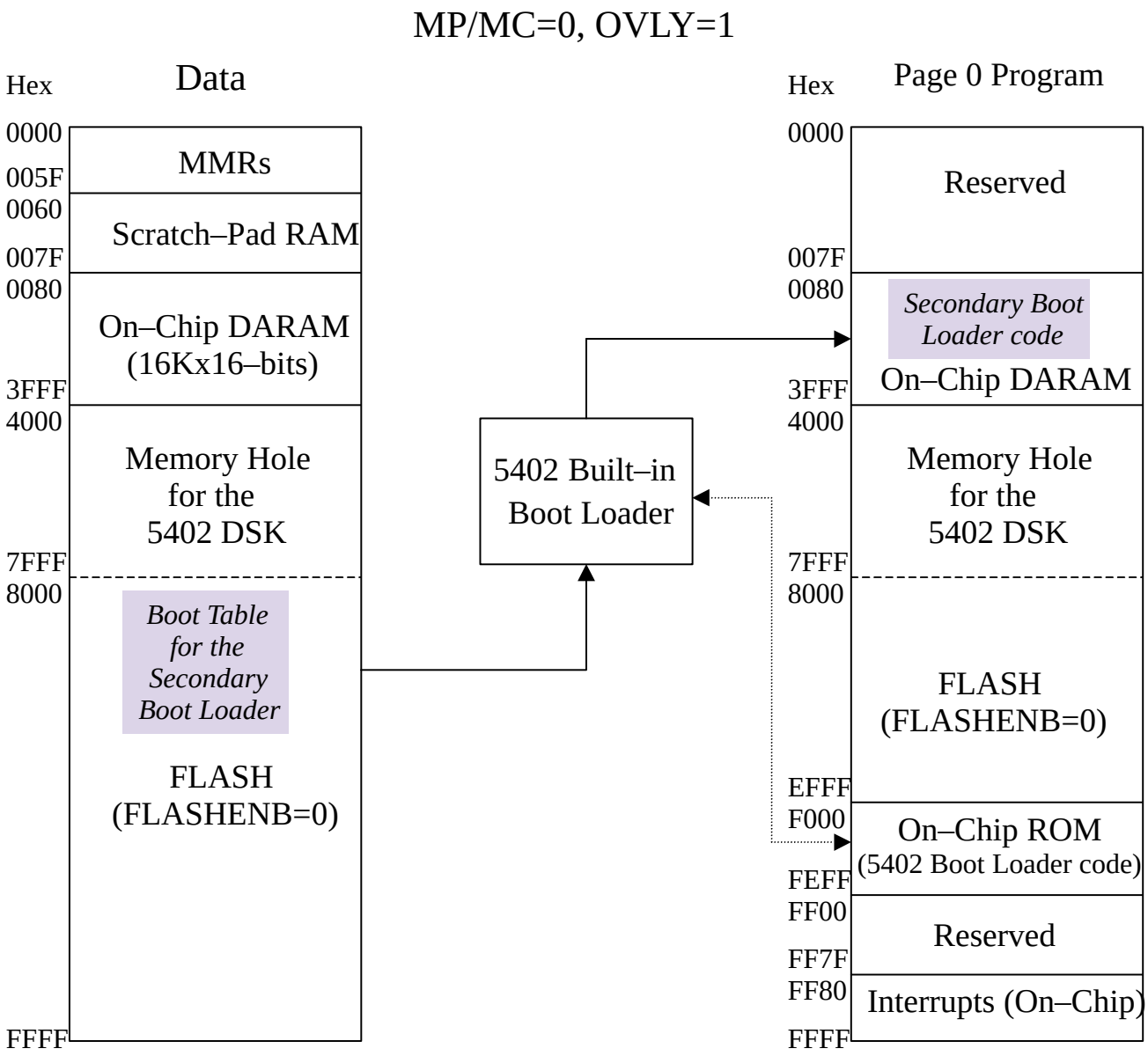

Figure 6. Boot Loading the Secondary Boot Loader

In Figure 6, a boot table for the secondary boot loader has been built and programmed onto the FLASH ROM that is mapped under the Data Page. Upon reset, the C5402 built-in boot loader reads the boot table headers, sets the corresponding registers, and copies the sections of the secondary boot loader into on-chip DARAM. At the completion of the boot loading, program execution starts at the entry point of the secondary boot code.

Figure 7 illustrates the secondary boot loader in action.

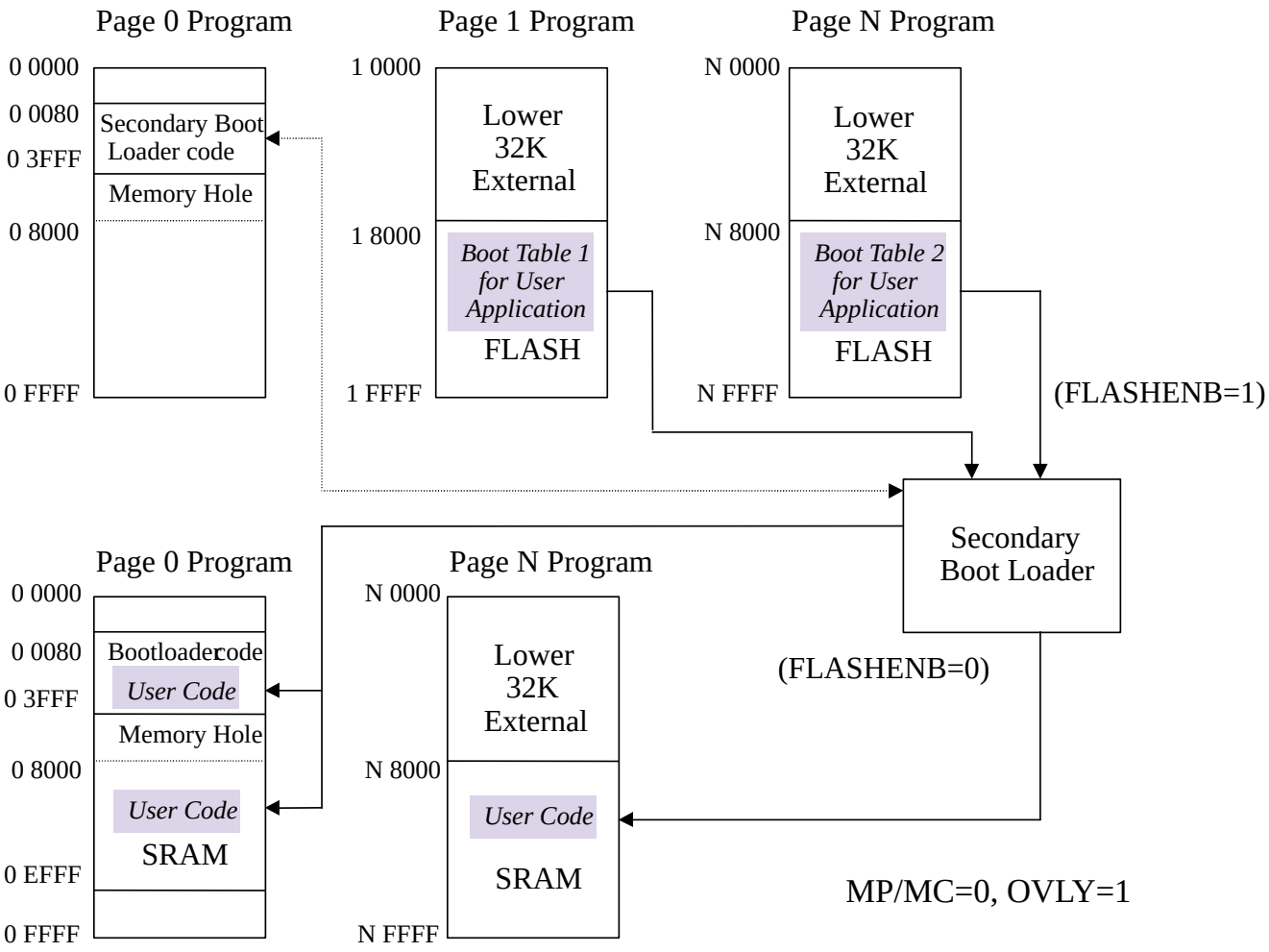

Figure 7. Boot Loading of the User Application by the Secondary Boot Loader

Setting the OVLY bit overlays the internal RAM to the lower 32K of all extended program spaces. A good programming practice for the C54x is to keep programs that reside in the extended program space in the upper 32K so that interrupts can be properly serviced even when the programs are running out of the extended program spaces. This tradition would be followed here. With the lower 32K memory overlaid to the internal RAM, only the FLASH memory that is mapped under the upper 32K memory space of the extended program space can be used for FLASH boot.

In Figure 7, several boot tables are built for the large user application and programmed onto the FLASH ROM which are mapped under the extended program space. The maximum size of each boot table is limited to 32K words. This limitation comes in two parts: First, only 32K memory space is available for boot loading for each program page of the extended program space, as already discussed in the previous paragraph. Second, the hex500 utility has a 16-bit byte addressable space; the utility can only build boot tables that are less than or equal to 32K 16-bit words.

It is an easy task to create multiple boot tables for a single user application. You only need to create several hex command files for the hex500 utility, and in each hex command file, specify only the bootable sections for the boot table.

Going back to Figure 7, the FLASHENB selects between FLASH and SRAM for the external memory. When FLASHENB=1 (power-on default), FLASH memories are mapped under the external memory space; when FLASHENB=0, SRAMs are mapped under the external memory space.

The secondary boot loader, which is running out of the on-chip DARAM on Page 0 Program Page, reads the headers for the user application boot table on FLASH, sets the appropriate registers, and copies the sections into their destinations on the on-chip DARAM or external SRAM.

Two things are important to note here:

- If user sections are to be loaded onto the on-chip DARAMs, make sure that you set a PROTECTED memory region for your secondary boot loader code so that your user application will not overwrite your secondary boot loader. This memory segment can then be used as data memory after boot loading is done.
- If user sections are to be loaded onto the external SRAMs, please note that FLASHENB bit sets the mapping of external memories of all pages to either FLASH or SRAM, but not both. Therefore, the secondary boot loader must enable FLASH for source code reading and then enable SRAM for writing the user code to their destinations.

6.1 Understand the Boot Table

To be able to build the secondary boot loader, it is first necessary to understand the boot table structure. The boot table format is simple. Typically, there is a header record which contains the width of the table and possibly some values for various control registers. Each subsequent block has a header which contains the size and destination address of the block followed by data for the block. Multiple blocks can be entered with a termination block following the last block. Finally, the table can have a footer containing more control register values. Table 4 below illustrates the boot table format for source program data stream in 16-bit mode.

Table 4. General Structure of Boot Table for Source Program in 16-Bit Mode

Word	Contents
1	10AAh (memory width of the source program is 16 bits)
2	Value to set in Register SWWSR
3	Value to set in Register BSCR
4	XPC value of the entry point (least significant 7 bits are used as A23-A16)
5	Entry point (PC) (16 bits are used as A15-A0)
6	Block size of the first section to load
7	XPC value of the destination address of the first section (7 bits)
8	Destination address (PC) of the first section (16 bits)
9	First word of the first section of the source program
.	.
.	Last word of the first section of the source program
R	Block size of the second section to load
R+1	XPC value of the destination address of the second section (7 bits)
.	Destination address (PC) of the second section (16 bits)
.	First word of the second section of the source program
.	.
.	Last word of the second section of the source program
.	.
.	Block size of the last section to load
.	XPC value of the destination address of the last section (7 bits)
.	Destination address (PC) of the last section (16 bits)
.	First word of the last section of the source program
.	.
.	Last word of the last section of the source program
n	0000h – indicates the end of source program

6.2 A Sample Secondary Boot Loader

```
* Xiaozhen Zhang *
* May 30, 2001 *
      .sect ".myboot"
      .mmregs
      .version 548
      .global     myboot
      .global     endboot
      .global     bootend

      .asg  ar0, xentry       ; XPC of entry point
      .asg  ar1, entry        ; entry point
      .asg  ar2, flashpage    ; page number in flash (A22–A16)
      .asg  ar3, flashaddr    ; address in flash (A15–A0)
      .asg  ar4, srampage     ; XPC of entry point/page # in sram (A22–A16)
      .asg  ar5, sramaddr     ; entry point/address in sram (A15–A0)
myboot:
      ssbx  intm              ; disable interrupts
      stm   #0FFFFh, @ifr     ; clear IFR flag
      ld    #0, dp
      rsbx  cpl               ; turns off the compiler mode
      rsbx  sxm               ; disables sign extension mode
      rsbx  ovm               ; clears overflow mode bit
      orm   #060h, @pmst      ; ovly=1, mpmc=1
      ld    #0, arp           ; setting the status bit to reset value
      rsbx  c16               ; setting the status bit to reset value
      rsbx  cmpt              ; setting the status bit to reset value
      rsbx  frct              ; setting the status bit to reset value
      stm   #07fffh, swwsr    ; 7 wait states for P_, D_, and I_ spaces
      stm   #08806h, bscr     ; bank switch control reg value for Parallel/
Warm boot
      portr 04, *(ar7)        ; read the current value of CNTL2 Register
      ld          *(ar7), b
      or          #0x0020, b  ; set FLASHENB=1
      stlm  b, ar7
      portw *(ar7), 04h       ; select Flash for external memory
      stm   #1, flashpage     ; set program page # to 1
      stm   #8000h, flashaddr ; boottable starts at 1:8000
      addm  #1, *(flashaddr)  ; skip checking for 16–bit boot.
      ld    *(flashpage), 16, a
      add   *(flashaddr), a
      reada *(swwsr)          ; read desired value of SWWSR & store
      addm  #1, *(flashaddr)  ; update the boottable pointer
      ld    *(flashpage), 16, a
      add   *(flashaddr), a
      reada *(ar6)            ; read desired value of BSCR
      ld    *(ar6), a
      and   #0FFFEh, a        ; ensure EXIO bit is off
      stlm  a, @bscr          ; store in BSCR
      addm  #1, *(flashaddr)  ; update the boottable pointer
      ld    *(flashpage), 16, a
      add   *(flashaddr), a
      reada *(xentry)
```

```
      addm  #1, *(flashaddr)  ; update the bootable pointer
      ld    *(flashpage), 16, a
      add   *(flashaddr), a
      reada *(entry)
      addm  #1, *(flashaddr)  ; update the boottable pointer
loop: ld    *(flashpage), 16, a
      add   *(flashaddr), a
      reada *(ar6)            ; read the size of the section

      ld    *(ar6), a         ; load the size of section to A
      bcd   endboot, aeq      ; section size=0 indicate boot end
      sub   #1, a, b          ; brc = section size – 1
      stlm  b, brc            ; update BRC
      addm  #1, *(flashaddr)  ; update the bootable pointer
      ld    *(flashpage), 16, a
      add   *(flashaddr), a
      reada *(srampage)       ; get the XPC of dest. (A22–A16) & store in
srampage
      addm  #1, *(flashaddr)  ; update the boottable pointer
      ld    *(flashpage), 16, a
      add   *(flashaddr), a
      reada *(sramaddr)       ; get the PC of dest. (A15–A0) & store in sra-
maddr
      addm  #1, *(flashaddr)  ; update the boottable pointer
      rptb  copy–1
      ld    *(flashpage), 16, a
      add   *(flashaddr), a
      reada *(ar6)            ; read object data
      portr 04, *(ar7)        ; read the current value of CNTL2 Register
      ld    *(ar7), b
      and   #0xffdf, b        ; Set FLASHENB=0
      stlm  b, ar7
      portw *(ar7), 04h       ; select SRAM for external memory
      ld    *(srampage), 16, a
      add   *(sramaddr), a    ; load the destination address
      writa *(ar6)            ; write object data to destination
      addm  #1, *(flashaddr)  ; update the flash (source) pointer
      addm  #1, *(sramaddr)   ; increment the destination address
      portr 04, *(ar7)        ; read the current value of CNTL2 Register
      ld    *(ar7), b
      or    #0x0020, b        ; set FLASHENB=1
      stlm  b, ar7
      portw *(ar7), 04h       ; select Flash for external memory

copy:
      b     loop

endboot:
      portr 04, *(ar7)        ; read the current value of CNTL2 Register
      ld    *(ar7), b
      and   #0xffdf, b        ; Set FLASHENB=0
      stlm  b, ar7
      portw *(ar7), 04h       ; select SRAM for external memory
```

```
        ld    *(xentry), 16, a
        add   *(entry), a
        fbacc a                 ; branch to the entry point
bootend:
        .end
```

6.3 The Large Application Flowchart

Figure 8 is a block diagram summarizing the flow for building a large FLASH ROM application on the C5402 DSK.

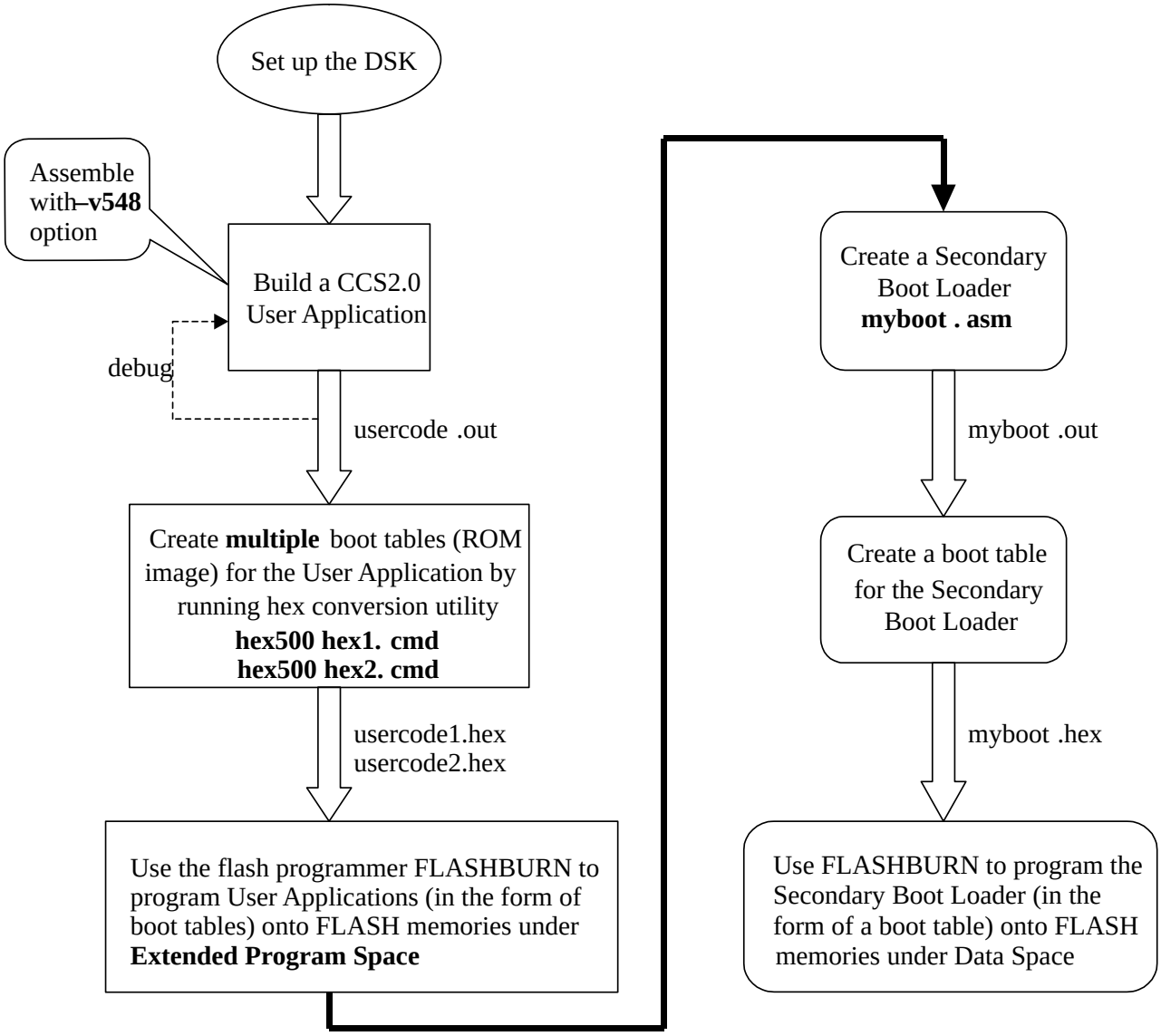

Figure 8. Flowchart for Building a Large FLASH ROM Application

7 Reinitialize the DSK FLASH

If your attempts to program the FLASH have not been successful, your board might be in an undefined state where the CCStudio IDE cannot communicate. Under such a circumstance, you should use your flash programming utility to program the flash.hex example back onto the FLASH. This will bring the board back to a known state when resetting the DSP.

The flash.hex program is located in "`C:\ti\c5400\dsk\examples\dsp\flash`".

8 Terminology

8.1 The TMS320C5402 Built-in Boot Loader

The TMS320UC5402 and TMS320VC5402 (hereafter referred to as C5402) digital signal processors (DSPs) have built-in boot loader residing in the device's on-chip ROM. The ROM is 4K words in size and is located at the 0xF000-0xFFFF address range in program space when the MP/MC input pin is sampled low at reset or if the MP/MC status bit of the Processor Mode Status (PMST) Register is set to zero. The C5402 built-in boot loader (C5402 boot loader) occupies 1K of the on-chip ROM with a base address of 0xF800.

The function of the boot loader is to transfer user code from an external source to the program memory at power up. This allows code to reside in slow, non-volatile memory externally, and be transferred to high-speed memory to be executed.

The C5402 boot loader sets up the CPU status registers before initiating the boot load. Interrupts are globally disabled (INTM = 1) and the internal dual-access RAMs (DARAMs) are mapped into the program/data spaces (OVLY = 1). The DARAM is mapped into program space to enable the boot loading of code by using minimal on-chip memory resources. Seven wait states are initialized for the entire program and data spaces. The boot loader does not change the reset condition of the bank switching control register (BSCR) prior to loading a boot table.

To accommodate different system requirements, the C5402 offers a variety of different boot modes. Parallel boot mode reads the desired boot table from data space via the external parallel interface (external memory interface) and transfers the code to program space. ROM booting on the C5402 DSK utilizes the 16-bit Parallel Boot Mode, which is one of the eight boot modes supported by the C5402 boot loader.

8.2 The Boot Table

The input for a boot loader is the boot table (also known as a *source program*). The boot table contains records that instruct the on-chip loader to copy blocks of data contained in the table to specified destination addresses. Some boot tables also contain values for initializing various processor control registers. The boot table can be stored in memory or read in through a device peripheral. In the case of the C5402 DSK, the boot table will be stored in the FLASH on board.

8.3 The Hex Conversion Utility

A program which accepts COFF object files and converts them into one of several standard ASCII hexadecimal formats suitable for loading into an EPROM programmer. Hex500 is the command that invokes the hex conversion utility.

9 References

1. *TMS320UC5402 and TMS320VC5402 Boot loader* (SPRA618).
2. *Developing a DSP/BIOS Application for ROM on the TMS320C6000 Platform with CCS* (SPRA743).
3. *TMS320C54x Assembly Language Tools User's Guide* (SPRU102).
4. Code Composer Studio IDE 2.0 online help.

IMPORTANT NOTICE

Texas Instruments and its subsidiaries (TI) reserve the right to make changes to their products or to discontinue any product or service without notice, and advise customers to obtain the latest version of relevant information to verify, before placing orders, that information being relied on is current and complete. All products are sold subject to the terms and conditions of sale supplied at the time of order acknowledgment, including those pertaining to warranty, patent infringement, and limitation of liability.

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

Customers are responsible for their applications using TI components.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards must be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance or customer product design. TI does not warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such products or services might be or are used. TI's publication of information regarding any third party's products or services does not constitute TI's approval, license, warranty or endorsement thereof.

Reproduction of 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. Representation or reproduction of this information with alteration voids all warranties provided for an associated TI product or service, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Resale of TI's 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, is an unfair and deceptive business practice, and TI is not responsible nor liable for any such use.

Also see: Standard Terms and Conditions of Sale for Semiconductor Products. www.ti.com/sc/docs/stdterms.htm

Mailing Address:

Texas Instruments
Post Office Box 655303
Dallas, Texas 75265