TEXAS INSTRUMENTS

Application Report
SPRAB60—November 2009

Creating a DSP Boot Image for Host Boot

High-Performance and Multicore Processors

Brighton Feng
Shenzhen, China

Abstract

This application note describes how to create a DSP boot image from a COFF file (.out file), which can be used for host boot, such as HPI/PCI boot, RapidIO boot, etc. The implementation of the DSP Boot Assist Tool is introduced and sample source code is provided with this application note.

Contents

Tables

Figures

1 Introduction

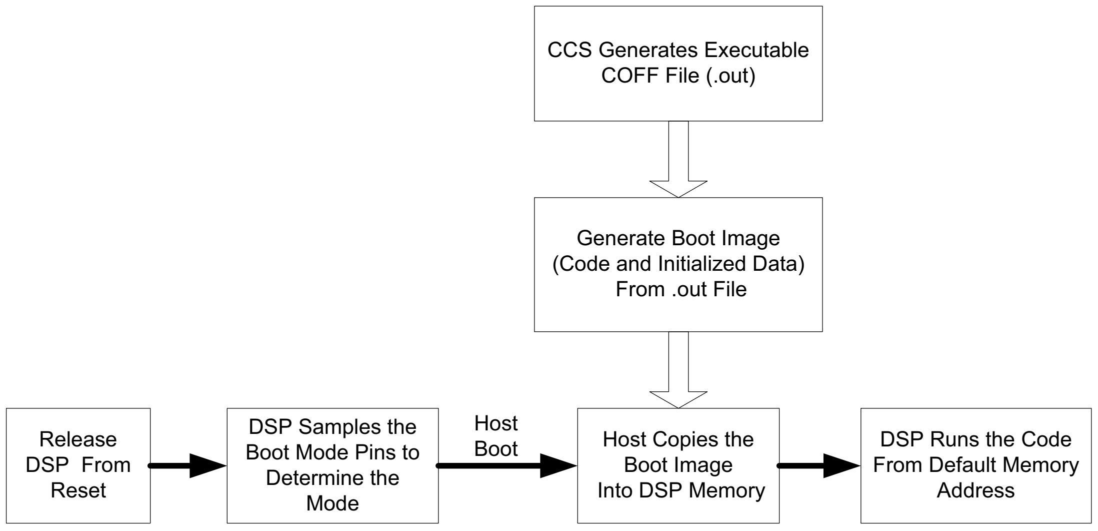
A host processor can directly access the DSP address space through a host port including HPI/PCI, RapidIO, etc. These peripherals allow a host processor to exchange information with the DSP. They can also be used by the host to initialize and load boot code in the DSP.

The Host boot configuration is selected by the external BOOTMODE pins. During reset, DSP samples the voltage level on these pins to determine the boot mode. When the Host boot process is selected, the DSP core is held in reset while the remainder of the device awakens from reset. At that time, a host processor (connected to the DSP through the host port) can access all of the DSP memory space, including internal, external, and on-chip peripheral registers.

After loading the code into the DSP, the host informs DSP to begin execution. For HPI boot, the host writes a 1 to the DSPINT bit in the HPI control register (HPIC). For RapidIO boot, the host sends a DOORBELL message. The DSP then starts the program execution from a default address.

Figure 1 shows the host boot process of C6000 DSP.

Figure 1 Host Boot Process


CCS Generates Executable COFF File (.out)
Generate Boot Image (Code and Initialized Data) From .out File
Release DSP From Reset
DSP Samples the Boot Mode Pins to Determine the Mode
Host Boot
Host Copies the Boot Image Into DSP Memory
DSP Runs the Code From Default Memory Address

The problem is how to load the code into DSP memory. The .out file (COFF format) generated by Code Composer Studio (CCS) cannot be loaded into DSP memory directly, because it contains application code and initialized data (which is needed to run the DSP standalone) along with information for debugging and linking.

To solve this problem, the application code and initialized data will be extracted from the .out file to build a simple file (or table). Then the host can copy the code/data to the DSP. This file (or table) is called boot image or boot table.

TI provides a tool named OFD (object file display) that is based on XML and Perl. The OFD utility processes common object file format (COFF) files and converts them to XML format. The TI application note *Using OFD Utility to Create a DSP Boot Image* (SPRAA64) describes how to create a DSP boot image using COFF and XML files with a simple Perl script. The resulting image is a C source file that can be included in the host processor's application program and downloaded to the DSP through the HPI or PCI interfaces.

If you are familiar with Perl and XML, the OFD tool is a good choice. If not, this application note introduces another more straightforward and flexible way to directly generate boot image from COFF (.out file).

2 .out File Format -- COFF (Common Object File Format)

The TI DSP assembler and linker creates object files in common object file format (COFF). COFF is an implementation of an object file format that was developed by AT&T for use on UNIX-based systems. This format is used because it encourages modular programming and provides powerful and flexible methods for managing code segments and target system memory.

2.1 COFF File Structure

The elements of a COFF object file describe its sections and symbolic debugging information. These elements include:

- A file header
- Optional header information
- A table of section headers
- Raw data for each initialized section
- Relocation information for each initialized section
- A symbol table
- A string table

The assembler and linker produce object files with the same COFF structure; however, a program that is linked for the final time does not usually contain relocation entries. Figure 2 illustrates the object file structure.

Figure 2 ***COFF File Structure***

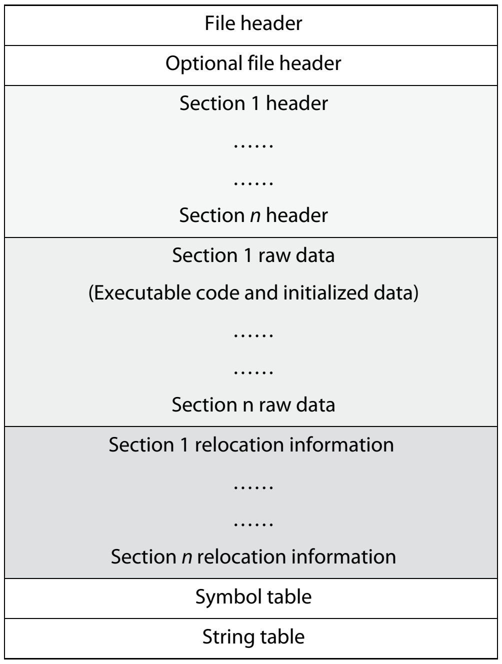


Figure 3 shows a typical example of a COFF object file that contains the three default sections .text, .data, and .bss, and a named section, referred to as <named>. By default, the tools place sections into the object file in the following order:

1. .text
2. .data

3. Initialized named sections
4. .bss
5. Uninitialized named sections

Although uninitialized sections have section headers, notice that they have no raw data, relocation information, or line number entries. This is because the .bss and .usect directives simply reserve space for uninitialized data. Uninitialized sections contain no actual code.

Figure 3 Sample COFF Object File

File header
.text section header .data section header .bss section header *<named> section* header
.text raw data .data raw data *<named> section* raw data
.text relocation information .data relocation information *<named> section* relocation information
Symbol table
String table

To create a boot image, extract raw data (executable code and initialized data) according the information from the file headers and section headers.

2.2 File Header Structure

The file header contains 22 bytes of information that describe the general format of an object file. Table 1 shows the structure of the COFF file header.

Table 1 File Header Contents

Byte #	Type	Description
0-1	Unsigned short	Version ID – indicates version of COFF file structure. Currently it should always be 0xC2
23	Unsigned short	Number of section headers
47	Integer	Time and date stamp – indicates when the file was created
811	Integer	File pointer – contains the symbol table's starting address (bytes offset from the beginning of the file)
1215	Integer	Number of entries in the symbol table
1617	Unsigned short	Number of bytes in the optional header – this field is either 0 or 28. If it is 0, there is no optional file header.
1819	Unsigned short	Flags (see Table 2)
2021	Unsigned short	Target ID – indicates the file can be executed in what kind of DSP 0x98 for C54x 0x99 for C6000 0x9c for C55x

Table 2 lists the flags that can appear in bytes 18 and 19 of the file header. Any number and combination of these flags can be set at the same time. For example, if bytes 18 and 19 are set to 0003h, both F_RELFLG and F_EXEC are set.

Table 2 File Header Flags (Bytes 18 and 19)

Mnemonic	Flag	Description
F_RELFLG	0001h	Relocation information was stripped from the file.
F_EXEC	0002h	The file is relocatable (it contains no resolved external references).
	0004h	Reserved
F_LSYMS	0008h	Local symbols were stripped from the file.
F_LITTLE	0100h	The target is a little-endian device.
F_BIG	0200h	The target is a big-endian device.

The following pseudo code shows how to use F_LITTLE to determine the endian mode of a DSP program. The tCoffHeader in the code is a structure including file header contents. See 3.2.1 "Defining a Header File" for more information.

Example 1 Pseudo Code to Determine the Endian Mode

```
if((tCoffHeader.uiFlags)& F_LITTLE)
    The target is a little-endian device;
```

End of Example 1

2.3 Optional File Header Format

The linker creates the optional file header and uses it to perform relocation at download time. Partially linked files do not contain optional file headers. Table 3 illustrates the optional file header format.

Table 3 Optional File Header Contents

Byte #	Type	Description
01	Short	Optional file header magic number (0108h for C6000)
23	Short	Version stamp
47	Integer	Size (in bytes) of .text section
811	Integer	Size (in bytes) of .data section
1215	Integer	Size (in bytes) of .bss section
1619	Integer	Entry point
2023	Integer	Beginning address of executable code (.text)
2427	Integer	Beginning address of initialized data (.data)

2.4 Section Header Structure

COFF object files contain a table of section headers that define where each section begins in the object file. Each section has its own section header. Table 4 shows the structure of each section header.

Table 4 Section Header Contents

Byte #	Type	Description
07	Character	This field contains one of the following: • An 8-character section name padded with nulls • Bytes 4 -7 are a pointer into the string table if the symbol name is longer than eight characters. Bytes 0 - 3 are padded with nulls
811	Integer	Section's run address — should be same as Load address in most cases, except you have overlapped code sections.
1215	Integer	Section's load address — the address in the DSP where this section should be copied to during boot.
1619	Integer	Section size in bytes
2023	Integer	File pointer to raw data — byte offset of the raw data of this section from the beginning of this .out file
2427	Integer	File pointer to relocation entries
2831	Integer	Reserved
3235	Unsigned integer	Number of relocation entries
3639	Unsigned integer	Reserved
4043	Unsigned integer	Flags (see Table 5)
4445	Unsigned short	Reserved
4647	Unsigned short	Memory page number

Actually, to boot is to copy raw data (codes and initialized data) in the .out file to DSP memory. This table includes main information for copying.

- Source Address — File pointer to raw data
- Destination Address — Section's load address
- Byte Count — Section size

Table 5 lists the flags that can appear in bytes 40 through 43of the section header.

Table 5 Section Header Flags (Bytes 40 Through 43) (Part 1 of 2)

Mnemonic	Flag	Description
STYP_REG	0000 0000h	Regular section (allocated, relocated, loaded)
STYP_DSECT	0000 0001h	Dummy section (relocated, not allocated, not loaded)
STYP_NOLOAD	0000 0002h	Noload section (allocated, relocated, not loaded)
STYP_COPY	0000 0010h	Copy section (relocated, loaded, but not allocated; relocation entries are processed normally)
STYP_TEXT	0000 0020h	Section contains executable code
STYP_DATA	0000 0040h	Section contains initialized data
STYP_BSS	0000 0080h	Section contains uninitialized data
STYP_BLOCK	0000 1000h	Alignment used as a blocking factor

Table 5 Section Header Flags (Bytes 40 Through 43) (Part 2 of 2)

Mnemonic	Flag	Description
STYP_PASS	0000 2000h	Section should pass through unchanged
STYP_CLINK	0000 4000h	Section requires conditional linking
STYP_VECTOR	0000 8000h	Section contains vector table
STYP_PADDED	00010000h	Section has been padded

The flags listed in Table 5 can be combined. For example, if the flag's word is set to 060h, both STYP_DATA and STYP_TEXT are set.

Bits 8-11 of the section header flags are used for defining the alignment. The alignment is defined to be $2^{(\text{value of bits 8-11})}$. For example if bits 8-11 are 0101b (decimal integer 5), then the alignment is 32 (2^5).

Uninitialized sections (created with the .bss and .usect directives) vary from this format. Although uninitialized sections have section headers, they have no raw data or relocation information. They occupy no actual space in the object file. Therefore, the number of relocation entries, the number of line number entries, and the file pointers are 0 for an uninitialized section. The header of an uninitialized section simply tells the linker how much space it should reserve in the memory map for variables.

To boot up the DSP, only the code and initialized data section need to be copied to DSP memory. Example 2 provides pseudo code to determine which section should be added to the boot image.

Example 2 Pseudo Code for Determining Sections to Copy

```
if(tSectionHeader.uiFlags&(SECTION_TYPE_DSECT|SECTION_TYPE_NOLOAD|
    SECTION_TYPE_COPY))
    continue; //skip useless sections


if((tSectionHeader.uiFlags&(STYPE_TEXT|STYPE_VECTOR|STYPE_DATA))
    Add this section into boot image;
```

End of Example 2

2.5 String Table Structure

Symbol names that are longer than eight characters are stored in the string table. For example, if a section name is longer than 8 characters, it will be stored in the string table. Names are stored contiguously in the string table, delimited by a null byte. The first four bytes of the string table contain the size of the string table in bytes. Thus offsets into the string table are greater than or equal to 4.

Figure 4 is a string table that contains two symbol names, Adaptive-Filter and Fourier-Transform. The index in the string table is 4 for Adaptive-Filter and 20 for Fourier-Transform.

Figure 4 String Table Entries for Sample Symbol Names

38 bytes

4 bytes			
'A'	'd'	'a'	'p'
't'	'i'	'v'	'e'
'-'	'F'	'i'	'l'
't'	'e'	'r'	'\0'
'F'	'o'	'u'	'r'
'i'	'e'	'r'	'-'
'T'	'r'	'a'	'n'
's'	'f'	'o'	'r'
'm'	'\0'		

Because the string table follows the symbol table, the start address of the string table is dependent on the symbol table offset and the symbol table size. The pseudo codes in Example 3 calculate the location of a section name, whose length is longer than 8 bytes and stored in the string table.

Example 3

```
iSymbolTableSize=SYMBOL_ENTRY_SIZE* tCoffHeader.uiSymbolEntryNumber;
cpStringTableAddress=  tCoffHeader.uiSymbolPointer+ iSymbolTableSize;

cpSectionNameAddress=
    cpStringTableAddress+tSectionHeader.SectionNameOffsetInStringTable;
```

End of Example 3

Relocation information and the symbol table used by the linker and debugger are not related to the boot process, so they are not introduced here. For detailed information, please refer to *Assembly Language Tools User's Guide v6.1* (SPRU186)

3 Creating a Boot Image

This section discusses how to create a DSP boot image that can be downloaded from the host through the HPI.

3.1 Boot Image Format

First, the boot image format needs to be defined. The boot image should include enough information for boot loading while minimizing image size to reduce boot time and save host memory. Figure 5 shows a recommend format, however you can define your own format according your application.

Figure 5 Boot Image Format

Entry point (4 bytes)
Section 1 size (4 bytes) Section 1 load address (4 bytes) Section 1 run address (4 bytes) Section 1 raw data (4 × n bytes)
Section 2 Size (4 bytes) Section 2 load address (4 bytes) Section 2 run address (4 bytes) Section 2 raw data (4 × n bytes)
……
Section N size (4 bytes) Section N load address (4 bytes) Section N run address (4 bytes) Section N raw data (4 × n bytes)
0x00000000 (end flag)

NOTE—The section size in a COFF file may not be a multiple of 4 bytes, but here we align every entry in the table to 4 byte boundary. If the section size is not a multiple of 4 bytes, padding will be added to the end of the raw data. The padding does not need to be copied to DSP memory.

The table can be saved to a binary file or saved as a C header file. The binary file is suitable for a host with a file system. During start up, the host will read the binary file and copy the contents of the file into DSP memory. An array embedded in a C Header file is suitable when the host does not support a file system. Example 4 illustrates an embedded array.

Example 4 Boot Table Array in a C Header File

```
#ifndef   BootTable_H

#define  BootTable_H


const char BootTable[]={

 /*Program Entry point*/
 0x60, 0xae, 0x00, 0x00,

 /*Section .hwi_vec begin*/

 0x00, 0x02, 0x00, 0x00, /*Size in bytes*/

 0x00, 0x00, 0x00, 0x00, /*load address*/

0x00, 0x00, 0x00, 0x00, /*run address*/

 /*Raw section Data*/

    0x2a, 0x30, 0x57, 0x00, 0x6a, 0x00, 0x00, 0x00,
    0x62, 0x03, 0x00, 0x00, 0x5a, 0xa3, 0x00, 0x00,

    0xa2, 0x03, 0x00, 0x02, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,

......

/*Section .sysinit begin*/

 0xa0, 0x03, 0x00, 0x00, /*Size in bytes*/

 0x80, 0xab, 0x00, 0x00, /*load address*/

0x80, 0xab, 0x00, 0x00, /*run address*/

 /*Raw section Data*/

    0xf6, 0x54, 0xbc, 0x01, 0x2a, 0xd0, 0x98, 0x01,
    0x6a, 0x01, 0x80, 0x01, 0x62, 0x03, 0x0c, 0x00,

    0x2a, 0xce, 0xd5, 0x01, 0x6a, 0x00, 0x80, 0x01,
    0x00, 0x40, 0x00, 0x00, 0x28, 0x48, 0x82, 0x00,

......

/*End Flag*/

 0x00, 0x00, 0x00, 0x00

};


#endif
```

End of Example 4

The C header file should be added into host projects so that the host boot code can read the boot image from the BootTable array, and copy the image into DSP memory.

3.2 Developing a Tool to Convert a COFF File to a Boot Image

The following sections use the DSP Boot Assist Tool as an example of how to implement a tool to convert from a COFF file to a boot image.

3.2.1 Defining a Header File

The DSP Boot Assist Tool is built with Turbo C++ Explorer. To easily parse the COFF file format, you will need to begin by defining a header file that describes the format of COFF file. Example 5 shows the definition for the DSP Assist Boot Tool.

Example 5

```
REMOVE

/*Define TI DSP COFF (.out) file format structure*/

#ifndef   _COFF_H
#define   _COFF_H
```

```
#define HEADER_SIZE  22
#define OPTIONAL_HEADER_SIZE  28
#define SECTION_HEADER_SIZE  48

//define of Header flag
//Relocation information was stripped from the file.
#define HEADER_FLAG_RELFLG  0x001

//The file is relocatable (it contains no unresolved external references).
#define HEADER_FLAG_EXEC  0x002

//Local symbols were stripped from the file.
#define HEADER_FLAG_LSYMS  0x008

//The target is a little-endian device.
#define HEADER_FLAG_LITTLE  0x100

//The target is a big-endian device.
#define HEADER_FLAG_BIG  0x200

//define of section type
//Regular section (allocated, relocated, loaded)
#define SECTION_TYPE_REG  0x0000000

//Dummy section (relocated, not allocated, not loaded)
#define SECTION_TYPE_DSECT  0x0000001

//Noload section (allocated, relocated, not loaded)
#define SECTION_TYPE_NOLOAD  0x0000002

/*Copy section (relocated, loaded, but not allocated;
relocation entries are processed normally)*/
#define SECTION_TYPE_COPY  0x0000010

//Section contains executable code
#define SECTION_TYPE_TEXT  0x0000020

//Section contains initialized data
#define SECTION_TYPE_DATA  0x0000040

//Section contains uninitialized data
#define SECTION_TYPE_BSS  0x0000080

//Alignment used as a blocking factor
#define SECTION_TYPE_BLOCK  0x0001000

//Section should pass through uncxhanged
#define SECTION_TYPE_PASS  0x0002000

//Section requires conditional linking
#define SECTION_TYPE_CLINK 00004000

//Section contains vector table
#define SECTION_TYPE_VECTOR  0x0008000

//Section has been padded
#define SECTION_TYPE_PADDED  0x0010000

typedef struct _tCoffHeader
{
  unsigned short usVersionID;   //indicates version of COFF file structure
  unsigned short usSectionNumber;   //Number of section headers
  int iTimeStamp;   //indicates when the file was created
  unsigned int uiSymbolPointer;   //contains the symbol table's starting address
  unsigned int uiSymbolEntryNumber;   //Number of entries in the symbol table

  //This field is either 0 or 28; if it is 0, there is no optional file header.
  unsigned short uiOptionalHeaderBytes;  //Number of bytes in the optional header

  unsigned short uiFlags;

  //magic number (0099h) indicates the file can be executed in a C6000 system
  unsigned short uiTargetID;
}TCoffHeader;

typedef struct _tCoffOptionalHeader
{
  //Optional file header magic number (0108h) indicates C6000
  unsigned short usOptionalHeaderID;
  unsigned short usVersionStamp;
  unsigned int uiTextSize;   //Integer Size (in bytes) of .text section
  unsigned  int uiDataSize;   //Integer Size (in bytes) of .data section
  unsigned  int uiBssSize;   //Integer Size (in bytes) of .bss section
  unsigned  int uiEntryPoint;
  unsigned int uiTextAddress;   //Integer Beginning address of .text section
  unsigned  int uiDataAddress; //Integer Beginning address of .data section
}TCoffOptionalHeader;

typedef struct _tSectionHeader
{
  union
  {
    char sName[8];  //An 8-character section name padded with nulls

    //A pointer into the string table if the symbol name >8bytes
    unsigned int uiPointer[2];
  }SectionName;
  unsigned int uiPysicalAddress;  //Section's physical address (Run Address)
  unsigned int uiVirtalAddress;   //Section's virtual address (Load Address)
  unsigned int uiSectionSize;   //Section size in bytes
  unsigned int uiRawDataPointer;   //File pointer to raw data
  unsigned int uiRelocationEntryPointer;  //File pointer to relocation entries
  unsigned int uiReserved0;
  unsigned int uiRelocationEntryNumber;  //Number of relocation entries
  unsigned int uiReserved1;

  /*Type of the section*/
  unsigned int uiFlags;
  unsigned short usReserved;
  unsigned short usMemoryPageNumber;
}TSectionHeader;

#endif
```

End of Example 5

3.2.2 Parsing the COFF File

The C++ class named TCoffParser parses the COFF file and generates a boot image. TCoffParser is defined in Example 6.

Example 6 Parsing the COFF File with TCoffParser

```
#ifndef   _COFFPARSER_H
#define   _COFFPARSER_H


#include "vcl.h"
#include "Coff.h"


class TCoffParser
{

public:
    TCoffParser();
    ~TCoffParser();



  //Parser COFF file, get information, prepare for generating boot miage
    int Parser(AnsiString CoffFileName);
    //generate binary boot image;
    int GenerateBinBootFile(AnsiString BinFileName);
    //generate C header file includeing boot image
    int GenerateCBootFile(AnsiString CFileName);

  //API for providing statistics information of the program
    //Total memery size should be consumed by the program

  unsigned int GetTotalSize() {return
uiCodeSize+uiInitializedDataSize+uiUninitializedDataSize;}
//Initialized sections, including codes and initialized data, which should be copy
during boot
```

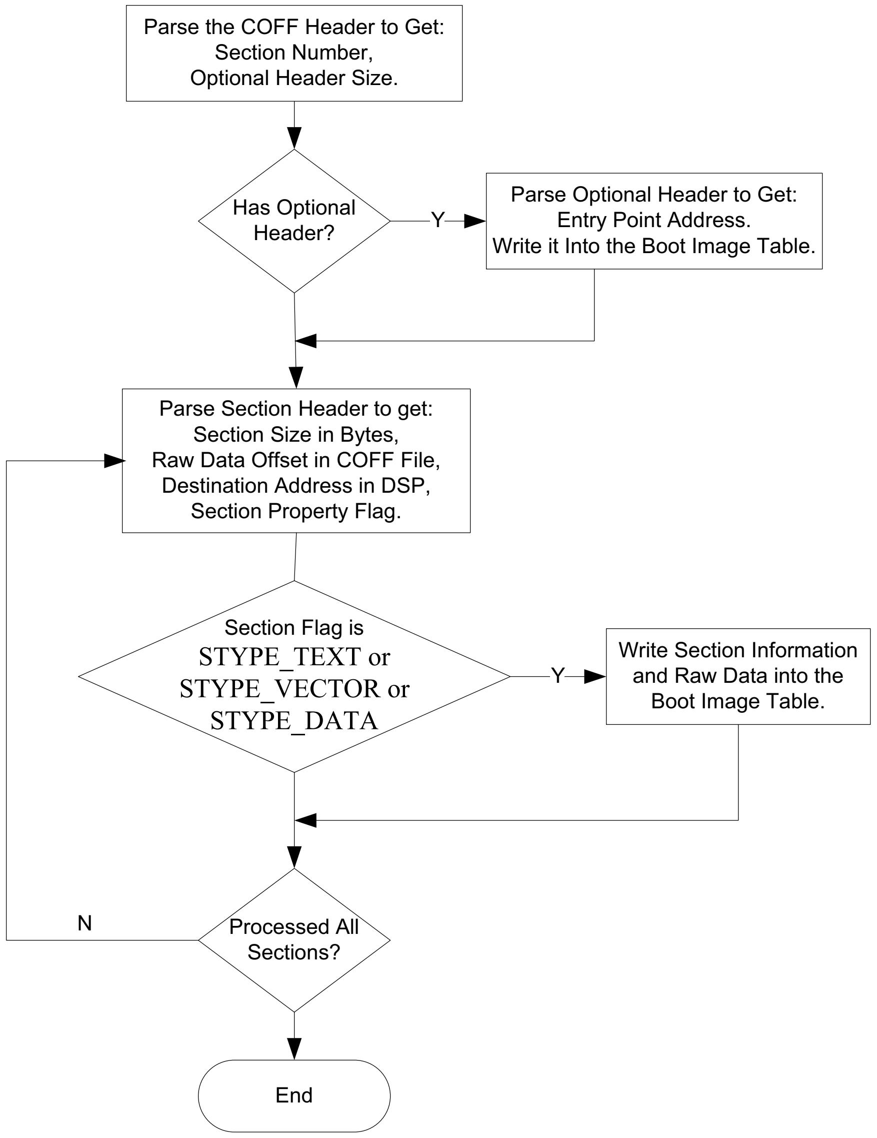
```
  unsigned int GetInitializedSectionSize() {return
uiCodeSize+uiInitializedDataSize;}
    unsigned int GetCodeSize() {return uiCodeSize;}
    unsigned int GetInitializedDataSize() {return uiInitializedDataSize;}
    unsigned int GetUninitializedDataSize() {return uiUninitializedDataSize;}
    unsigned int GetInitializedSectionNumber()
    {returnuiCodeSectionNumber+uiInitializedDataSectionNumber;}
    unsigned int GetCodeSectionNumber() {return uiCodeSectionNumber;}
    unsigned int GetInitializedDataSectionNumber() {return
    uiInitializedDataSectionNumber;} unsigned int
    GetUninitializedDataSectionNumber() {return uiUninitializedDataSectionNumber;}

    TCoffHeader tCoffHeader;
    TCoffOptionalHeader tCoffOptionalHeader;
    ......

private:

    TSectionHeader tSectionHeader;
    unsigned int uiCodeSize;   //Total codes size in bytes
    unsigned int uiInitializedDataSize;   //Total Initialized Data Size
    unsigned int uiUninitializedDataSize;   //Totol uninitialized data size
    unsigned int uiCodeSectionNumber;
    unsigned int uiInitializedDataSectionNumber;
    unsigned int uiUninitializedDataSectionNumber;

    ......

};


#endif
```

End of Example 6

3.2.3 Boot Image Generation Process

The boot image generation process is shown in Figure 6.

Figure 6 Boot Image Generation Process


Parse the COFF Header to Get: Section Number, Optional Header Size.
Has Optional Header?
Y
Parse Optional Header to Get: Entry Point Address. Write it Into the Boot Image Table.
Parse Section Header to get: Section Size in Bytes, Raw Data Offset in COFF File, Destination Address in DSP, Section Property Flag.
Section Flag is STYPE_TEXT or STYPE_VECTOR or STYPE_DATA
Y
Write Section Information and Raw Data into the Boot Image Table.
Processed All Sections?
N
End

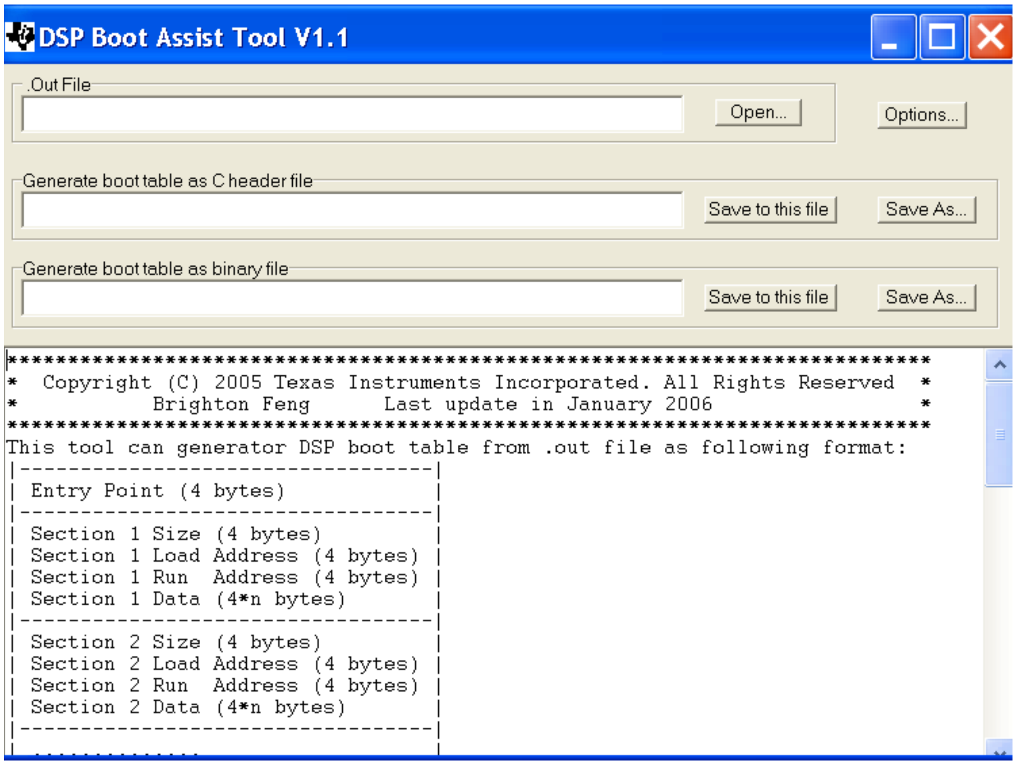
NOTE—To reduce boot time and save host memory, only copy sections with the STYPE_TEXT, STYPE_VECTOR, or STYPE_DATA flag into the boot image.

For implementation details, please refer to the source code bundled with this application note. It can be downloaded from the following URL http://www-s.ti.com/sc/techlit/SPRAB60.zip.

4 How to use the DSP Boot Assist Tool for Host Boot

Using the Windows DSP Boot Assist Tool is very simple. Figure 7 shows the interface of the tool. The tool is tested on the C6000 DSP, but it may be used on the C5000 DSP because the COFF format is same.

Figure 7 DSP Boot Assist Tool


DSP Boot Assist Tool V1.1
.Out File
Open...
Options...
Generate boot table as C header file
Save to this file
Save As...
Generate boot table as binary file
Save to this file
Save As...

```
******************************************************************************
*  Copyright (C) 2005 Texas Instruments Incorporated. All Rights Reserved  *
*           Brighton Feng      Last update in January 2006                 *
******************************************************************************
This tool can generator DSP boot table from .out file as following format:
|-----------------------------------|
| Entry Point (4 bytes)             |
|-----------------------------------|
| Section 1 Size (4 bytes)          |
| Section 1 Load Address (4 bytes)  |
| Section 1 Run  Address (4 bytes)  |
| Section 1 Data (4*n bytes)        |
|-----------------------------------|
| Section 2 Size (4 bytes)          |
| Section 2 Load Address (4 bytes)  |
| Section 2 Run  Address (4 bytes)  |
| Section 2 Data (4*n bytes)        |
|-----------------------------------|
| ..............                    |
```

4.1 Creating a Boot Image With the DSP Boot Assist Tool

Procedure 1 describes how to use the DSP Boot Assist Tool

Procedure 1

Step – Action

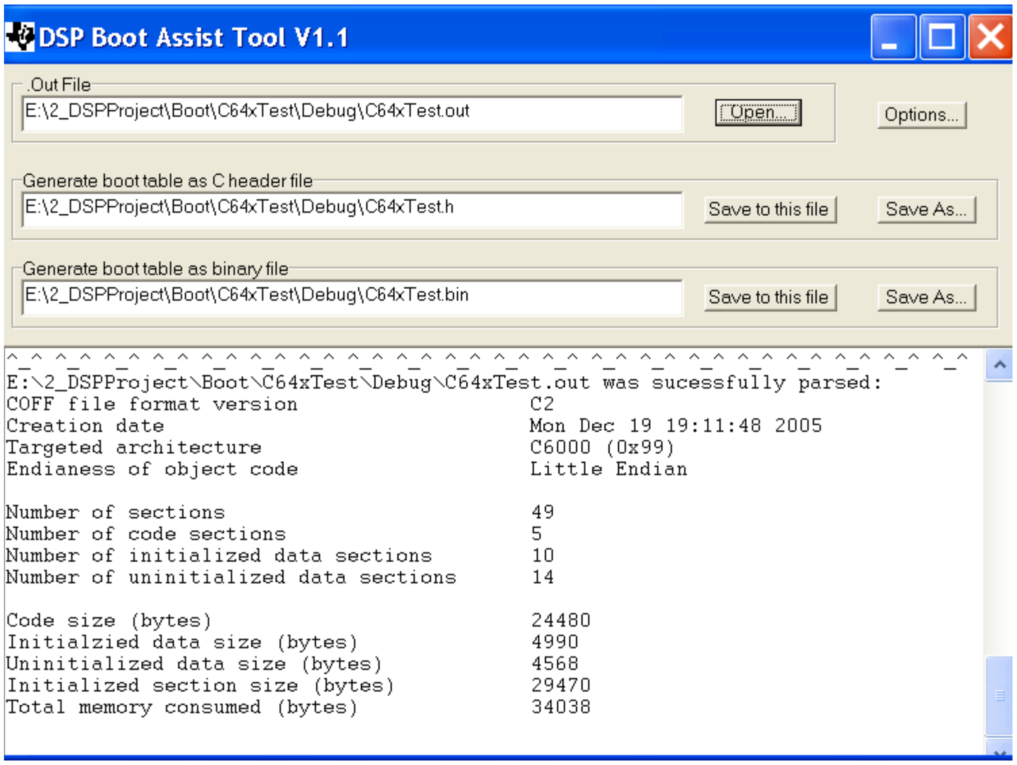
1 Click the **Open** button and choose your .out file for processing. Figure 8 shows the status when you open a .out file. Statistics information including the code size, section number, etc. display in the bottom window.

2 In the appropriate text box, save the file **as a C header file** or **Generate a boot table as a binary file**.

- Click the **Save to this file** button to generate a boot image and save to a relative file. The default directory for the generated file is the same as .out file.
- Click the **Save as** button to choose the directory or change the file name for the generated file.

End of Procedure 1

Figure 8 Open a .out File in DSP Boot Assist Tool

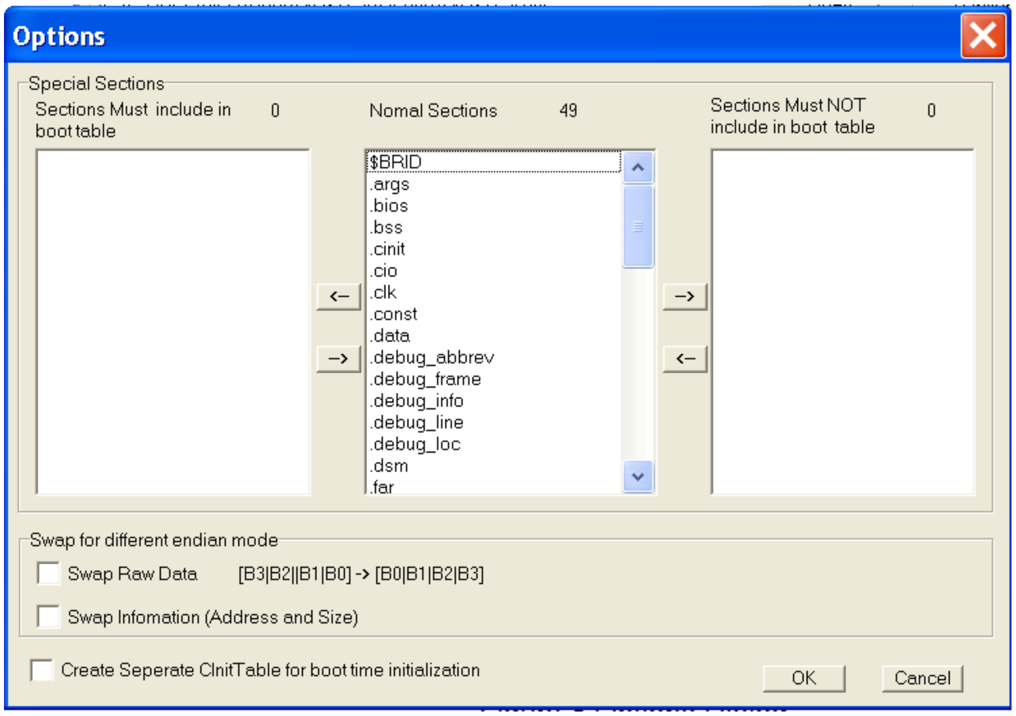

DSP Boot Assist Tool V1.1
.Out File
E:\2_DSPProject\Boot\C64xTest\Debug\C64xTest.out
Open...
Options...
Generate boot table as C header file
E:\2_DSPProject\Boot\C64xTest\Debug\C64xTest.h
Save to this file
Save As...
Generate boot table as binary file
E:\2_DSPProject\Boot\C64xTest\Debug\C64xTest.bin
Save to this file
Save As...

```
^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^ ^_^
E:\2_DSPProject\Boot\C64xTest\Debug\C64xTest.out was sucessfully parsed:
COFF file format version                  C2
Creation date                             Mon Dec 19 19:11:48 2005
Targeted architecture                     C6000 (0x99)
Endianess of object code                  Little Endian

Number of sections                        49
Number of code sections                   5
Number of initialized data sections       10
Number of uninitialized data sections     14

Code size (bytes)                         24480
Initialzied data size (bytes)             4990
Uninitialized data size (bytes)           4568
Initialized section size (bytes)          29470
Total memory consumed (bytes)             34038
```

4.2 Boot Image Options

The DSP Boot Assist Tool provides advanced options if you want to add or remove sections from the boot image. The **Options** button displays the Options dialog shown in Figure 9.

Figure 9 Options Dialog


Options

Special Sections

Sections Must include in boot table 0

Nomal Sections 49

Sections Must NOT include in boot table 0

$BRID
.args
.bios
.bss
.cinit
.cio
.clk
.const
.data
.debug_abbrev
.debug_frame
.debug_info
.debug_line
.debug_loc
.dsm
.far

Swap for different endian mode

☐ Swap Raw Data [B3|B2||B1|B0] -> [B0|B1|B2|B3]

☐ Swap Infomation (Address and Size)

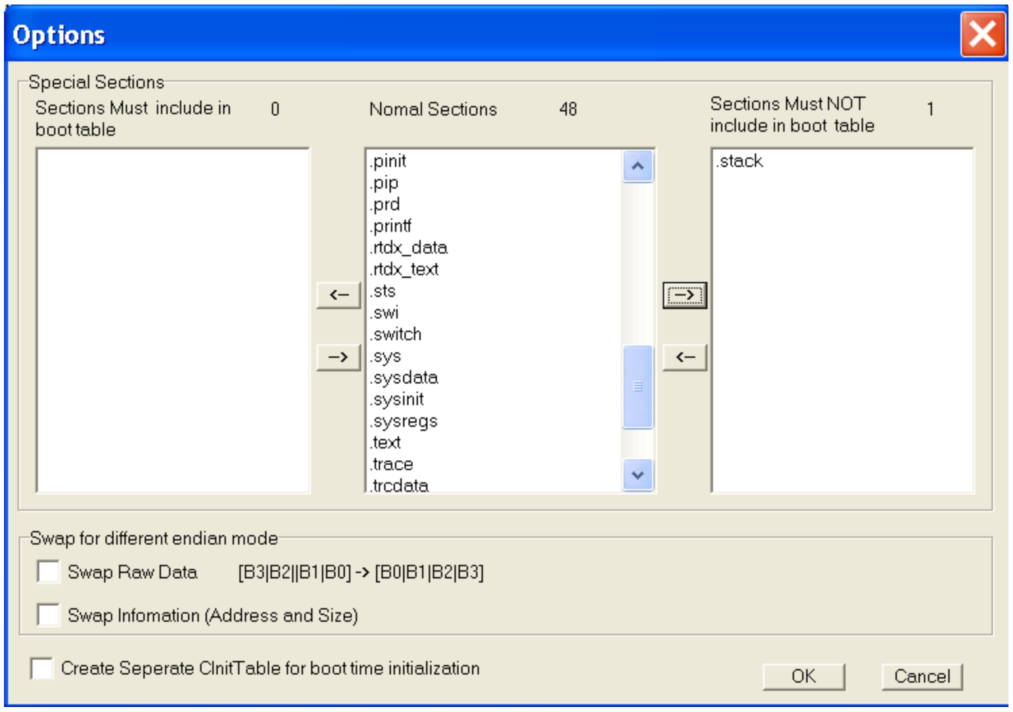
☐ Create Seperate CInitTable for boot time initialization

All the available sections are listed in the **Normal Sections** box. Select the desired section and click the left or right arrow button to add or remove the section from the boot image. These settings will override the rules about boot image section inclusion that were shown in Figure 6.

For example, The stack does not need to be included into the boot image because it does not need to be initialized, but some compilers will automatically initialize the stack with default values for debug purposes. In this case, the .stack section type should be STYPE_DATA, which means initialized section. According the rules shown in Figure 6, the section should be included in boot image. Through the Options dialog shown in Figure 9, you can manually remove the section from the boot image to reduce the boot time and save host memory.

The Figure 10 dialog box shows the .stack will NOT be included into the boot image regardless of how the section flag is set.

Figure 10 Exclude .stack Section From Boot Image


Options
Special Sections
Sections Must include in boot table 0
Nomal Sections 48
Sections Must NOT include in boot table 1
.pinit
.pip
.prd
.printf
.rtdx_data
.rtdx_text
.sts
.swi
.switch
.sys
.sysdata
.sysinit
.sysregs
.text
.trace
.trcdata
.stack
<-
->
Swap for different endian mode
Swap Raw Data [B3|B2||B1|B0] -> [B0|B1|B2|B3]
Swap Infomation (Address and Size)
Create Seperate CInitTable for boot time initialization
OK
Cancel

The DSP Boot Assist Tool provides options to support different endian mode. The DSP can support both little and big endian. Generally speaking, Intel CPUs are little endian and Motorola (Freescale) CPUs are big endian. If the host endian mode is not the same as the DSP, you can swap the data in the boot image to match it.

Since the boot image file is generated by a Windows PC, the address and size information should always be little endian; raw data endian is determined by the target DSP endian mode. If your host endian mode is different from the target DSP, you should swap the raw data; if your host endian mode is big endian (different from Windows PC) you should swap other information including the address and size

If you check **Swap Raw Data**, the raw section data will be swapped. If you check **Swap Information**, the address and size will be swapped. Swapping is done for every four bytes, it will change the byte order from [B3|B2|B1|B0] to [B0|B1|B2|B3].

The option **Create Separate CinitTable for boot time initialization** can support autoinitialization of variables at boot. When selected, the .cinit section will be saved separately at the end of the boot image file as shown in Figure 11.

Figure 11 Boot Image With Separate Cinit Table

Entry point (4 bytes)
Section 1 size (4 bytes) Section 1 load address (4 bytes)

Figure 11 Boot Image With Separate Cinit Table

Section 1 run address (4 bytes) Section 1 raw data (4 × n bytes)
……
Section N size (4 bytes) Section N load address (4 bytes) Section N run address (4 bytes) Section N raw sata (4 × n bytes)
0x00000000 (end flag)
.cinit section size (4 bytes)
.cinit section load address (4 bytes, useless)
.cinit section run address (4 bytes, useless)
.cinit section raw data (4 × n bytes)
0x00000000 (end flag)

If you generate a C header file, the .cinit section will generate a separate array at the end of the file.

For more details about load time auto initialization, Please refer to:

- *TMS320C6000 Optimizing Compiler User's Guide* (spru187)
- *TMS320C6000 Assembly Language Tools User's Guide* (spru186)

4.3 Copying the Boot Image into DSP memory

If the boot image table is saved as a C header file, you should include the header file into your host project where you should add code to load the boot image into DSP memory. Example 7 shows loading the boot image through HPI.

Example 7 HPI Boot Sample Code

```
#include "BootTable.h"   //Include the header file generated by the tool


void LoadDSPCodes()

{

    unsigned int uiSectionSize, uiEntryPoint;
    //data pointer for the array, which contains the boot image
    unsigned int *ipBootTable= (unsigned int *)BootTable;

    uiEntryPoint=*ipBootTable++;
    uiSectionSize=*ipBootTable++;
    while(uiSectionSize)   //0 is the end flag
{

    HPIA= *ipBootTable++;  //Write HPI transfer target address to HPIA
    ipBootTable++; //Skip the run address
    //Copy a section into DSP Memory
    for(int i=0; i< uiSectionSize; i+=4)
        HPID= *ipBootTable++; //Write raw section data to HPID register
    uiSectionSize=*ipBootTable++; //Next section
 }

}
```

End of Example 7

If the boot image is saved as a binary file, you should store the file on the host's file system. With HPI boot, shown in Example 7, code is copied to DSP memory as an array, but when loading from a file, you should modify the code to copy the data from the binary file to DSP memory.

An efficient way to do this is to load the binary file into an array and name the array BootTable. Then you can use the code in Example 7 without modification.

After loading the boot image into DSP memory, notify the DSP to begin running the program. For HPI boot, the host writes a 1 to the DSPINT bit in the HPI control register (HPIC). For RapidIO boot, the host sends a DOORBELL message. The DSP will begin running from the predefined address. For C6000 DSP, the "Entry Point" in the .out file is useless, because the C6000 DSP will always run from the default address after host boot. For C62x and C64x, the default address is 0. For C64x+, the default address is not 0. Refer to device specific documentation for the default address.

5 Another Use of the DSP Boot Assist Tool: Getting Statistics From a DSP Program

Even if you don’t need to generate a boot image for your application, the DSP Boot Tool is still useful. You can use it to gather statistical information from a DSP program. To retrieve statistics, open a .out file in the DSP Boot Tool. The information will be shown in the output window from Figure 8.

This information is helpful when:

- You want to know how much memory is consumed on a DSP.
- You want to compare memory consumption when you are optimizing a program with different methods.

6 Summary and Conclusion

This application note shows how to create a DSP boot image directly from a .out file. This approach eliminates the need to use the HEX utility which requires reformatting the intermediate output of these tools. Thus, the development flow becomes simpler, more robust and more flexible.

This application note only introduces how to create boot image for host boot, but the tool can easily be modified to accommodate various boot modes.

TI provides another tool named OFD, which is based on XML and Perl script. If you are familiar with Perl and XML, it is a good choice. See *Using OFD Utility to Create a DSP Boot Image* (SPRAA64) for more details.

7 References

- *TMS320C6000 Assembly Language Tools User's Guide* (SPRU186)
- *Implementing the TMS320C6201/C6701/C6211 HPI Boot Process* (SPRA512)
- *TMS320C6000 Optimizing Compiler User's Guide* (SPRU187)
- *Using OFD Utility to Create a DSP Boot Image* (SPRAA64)