TEXAS INSTRUMENTS

Application Report

SPRAA69D–September 2009

Using the TMS320C672x Bootloader

Karen Baldwin *DSP EEE*

ABSTRACT

This application report describes the design details of the TMS320C672x bootloader and describes a set of software utilities designed to facilitate formatting of application code for use with the bootloader.

This application report contains a system level patch, the bootloader utilities and project code that can be downloaded from this link: http://www-s.ti.com/sc/techlit/sprc203.zip.

Please note that the system level patch fixes a problem in which the DSP may not be left in a known good state. This patch is required regardless whether any other ROM contents are used by an application.

Contents

List of Figures

List of Tables

1 Introduction

The bootloader resides in the internal ROM of TMS320C672x devices. It starts from the beginning of ROM address space 0x00000000. After reset, the device sets the program counter to the beginning of the ROM address and begins execution of the bootloader.

The following is the list of boot modes supported by the bootloader:

- HPI
- Parallel Flash
- SPI Master
- I2C Master
- SPI Slave
- I2C Slave

When booting in master mode, the bootloader reads the boot information from the slave device if and when required. On the other hand, when booting in slave mode, the bootloader depends on the master device to feed boot information if and when required.

Table 1. Terms Used in This Document

Term	Description
Bootloader	Bootloader Code for TMS320C672x
AIS	Application Image Script
BL	Boot Loader, referring to the bootloader in this text
DSP	Digital Signal Processor, referring to TMS320C672x in this text
I2C	Inter Integrated Circuit
OS	Op-code Synchronization
POS	Ping Op-code Synchronization
ROM	Read Only Memory
SPI	Serial Peripheral Interface
SWS	Start-Word Synchronization

2 Boot Mode Description

The selection of the following boot modes depends upon the status of boot device pins documented below. The device captures the status of these pins on the rising edge of reset into the registers CPGPIN0 and CFGPIN1. The bootloader refers to CFGPIN0 and CFGPIN1 in order to get the status of the boot device pins. For the sake of clarity, this text refers to the boot device pins instead of their corresponding bit positions in one of the CFGPIN registers.

Table 2. CFGPIN0 Register Definition

CFGPIN0 bit	Bit Name	Corresponding Pin
31:8	Reserved	Not implemented
7	PINCAP7	SPI0SOMI/SDA0
6	PINCAP6	SPI0SIMO
5	PINCAP5	SPI0CLK/SCL0
4	PINCAP4	$\overline{\text{SPI0SCS}}$ / SCL1
3	PINCAP3	$\overline{\text{SPI0ENA}}$ / SDA1
2	PINCAP2	SPI1SOMI
1	PINCAP1	SPI1SIMO
0	PINCAP0	SPI1CLK

Table 3. CFGPIN1 Register Definition

CFGPIN1 bit	Bit Name	Corresponding Pin
31:8	Reserved	Not implemented
7	PINCAP15	$\overline{\text{SPI1SCS}}$
6	PINCAP14	$\overline{\text{SPI1ENA}}$
5	PINCAP13	$\overline{\text{HCS}}$
4	PINCAP12	HD[0]
3	PINCAP11	HA[0]
2	PINCAP10	AFSX0
1	PINCAP9	AFSR0
0	PINCAP8	AXR0[0]

Table 4 summarizes boot mode pin configuration.

NOTE: The order of different boot modes in the table is not the same as when they are detected.

Table 4. Boot Device Pin Allocation

Boot Mode Description			Boot Pin Allocation			
Boot Mode Description	**Data Bits**	**Add Bits**	**BL1 boot ($\overline{\text{HCS}}$)**	**SPI0SOMI**	**SPI0SIMO**	**SPI0CLK**
HPI[1]			0	X	X	X
Parallel Flash[2]	-	-	1	0	1	0
SPI0 Master	8	16	1	0	0	1
SPI0 Slave	16	-	1	0	1	1
Reserved	-	-	1	1	0	0
I2C1 Master	8	16	1	1	0	1
Reserved	-	-	1	1	1	0
I2C1 Slave	8	-	1	1	1	1

(1) Additional pins used to configure the boot mode. For details, see the Section 2.1.
(2) First byte in parallel flash gives information on the Data/Address bits. For details, see the Section 2.2.

2.1 HPI Boot

Once selected, the bootloader initializes the “Bootloader HPI Jump Address Register (0x10000714)” and “Bootloader HPI Transfer Done Register (0x10000718)” with zeros. Then, it resets the CSP Bridge and configures the HPI to run in the desired mode depending on the state of the following pins:

Table 5. HPI Configuration Based on Device Pins

Device Pin	Corresponding Bit Name in CFGHPI Register
SPI0SOMI	BYTEAD
SPI0SIMO	FULL
SPI0CLK	NMUX

For more information on these bits, refer to the CFGHPI register description in the official TMS320C672x specification in *TMS320C6727, TMS320C6726, TMS320C6722 Floating-Point Digital Signal Processors* (SPRS268).

After configuring the CFGHPI register based on the logical states of the above mentioned pins, the bootloader enables the HPI peripheral by writing a 1 to the ENA bit in the CFGHPI register.

Then, the bootloader sets the HINT bit of the HPIC register. This causes the HPI_HINTn pin of the device to go low, as an indication to the host that the DSP is ready. The host can clear this interrupt by also writing a 1 to the HINT bit of HPIC. Then the DSP waits while the host places data, via the standard HPI protocol, into its memory. Once complete, the host writes the application entry address into the memory location 0x10000714 and a 1 to address location 0x10000718 to signal the completion of HPI bootloading. The bootloader is continuously polling the address location 0x10000718. As soon as the bootloader finds a non zero value at that address location, it sets the program counter to the value at address 0x10000714 and begins execution of the application.

2.2 Parallel Flash

The first 8-bits on the flash device gives information about the data bits (8/16) that can be accessed from the parallel flash simultaneously.

Figure 1. Parallel Flash Signature Format

7	6	5	4	3	2	1	0
Reserved						Boot Mode	

Table 6. Parallel Flash Boot Mode Field

Boot Mode	Description
00	8-bit parallel flash
01	16-bit parallel flash
10	Reserved
11	Reserved

Once the bootloader detects EMIF boot mode, it reads the first byte from the flash device to determine 8 or 16 bit boot mode. The bootloader then sets the EMIF control registers for accessing 8 or 16 bits according to the mode selected. The first 1024 bytes of data are copied from the FLASH memory into the first 1kBytes of TMS320C672xx internal memory. The bootloader sets the program counter to 0x10000004 (offset of 0x4 in internal memory) and execution of code begins at this address. It is assumed that the first 1024 bytes of code/data contain a user defined secondary bootloader or other user application code that will load/execute the application.

2.3 I2C Master

The bootloader expects data from the I2C to be in AIS format. It first attempts to read the magic word (0x41504954) from the I2C on address 0x50. If the magic word is not detected, the boot mode will fail and the bootloader will execute an infinite while loop. If the bootloader reads the correct magic word, it will expect a valid AIS command as the next data in the stream. The bootloader will continue processing AIS commands and data until an AIS "JUMP_CLOSE" command is encountered.

The bootloader supports devices obeying the standard SPI serial EEPROM protocol established by Motorola. The data burnt into the serial EEPROM must be in AIS format.

2.4 I2C Slave

The DSP I2C peripheral has its own address set of 0x29. The host (master) is required to establish a link with the DSP in the beginning. The host begins transmitting data in application image script (AIS) format. For details about link establishment, see Section 5. For details about, AIS see Section 3.

2.5 SPI Master

The bootloader attempts to read a magic word (0x41504954) through SPI. If the magic word is not obtained during the read call, the detection of this boot mode fails and the bootloader logic ends up in an infinite while loop.

The bootloader supports devices obeying the standard SPI serial EEPROM protocol established by Motorola. The data burnt into the serial EEPROM must be in AIS format.

The bootloader initializes SPI0 of the DSP to operate with following configuration.

- SIMO, SOMI, and CLK are configured as functional SPI pins.
- SCS0 is configured as GPIO pin , with chip select being driven by state of this pin.
- SPI0 Control registers are set as follows:

Table 7. SPI Configuration for Master Boot

Register	Value	Description
SPIPC0	0x00000E00	Selects SIMOFUN, SOMIFUN and CLKFUN to be SPI functional pins, SCSFUN is configured as GPIO
SPIPC1	0x00000001	Sets SCSDIR0 direction as output
SPIGCR1	0x00000003	Sets CLKMOD as internal, places SPI in MASTER mode
SPIFMT0	0x0002FF10	Selects CHARLEN of 16, PRESCALE of 256, and CLK polarity as clock inactive HIGH
SPIDELAY	0x06021030	Sets C2EDELAY of 48, T2EDELAY of 17, and T2CDELAY of 2

2.6 *SPI Slave*

The host (master) is required to establish a link with the DSP in the beginning. The host begins transmitting data in AIS format. For details about link establishment, see the Section 5. For details about AIS, see the Section 3.

When SPI slave boot mode is selected, the on-chip bootloader configures SPI0 as follows:

- SIMO, SOMI, CLK, and SCS are configured as functional SPI pins.
- SPI0 control registers are set as follows:

Table 8. SPI Configuration for Slave Boot

Register	Value	Description
SPIPC0	0x00000E01	Selects SIMOFUN, SOMIFUN, CLKFUN, and SCSFUN are configured as functional SPI pins
SPIGCR1	0x00000000	Sets CLKMOD as external, with slave mode selected
SPIFMT0	0x00020010	Selects CHARLEN of 16, and CLK polarity as clock inactive HIGH

3 Application Image Script

The bootloader accepts boot information in the form of a script, called application image script (AIS). Application image script is a Texas Instruments proprietary application image transfer format. This script is a binary file consisting of a script header followed by various commands that are interpreted and executed by the bootloader. Each command contains an op-code, followed by optional additional data required to execute the op-code. The bootloader supports AIS Version 1.0.

The AIS starts with a magic word (0x41504954), followed by a series of commands shown in Figure 2. Each command, in turn, consists of an op-code followed by optional additional data.

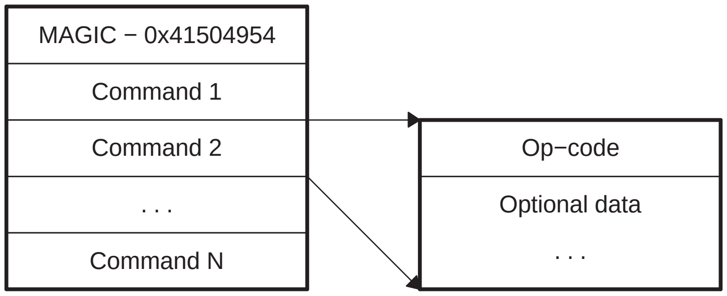


Figure 2. Basic Structure of Application Image Script

The bootloader only accepts data in AIS format for all modes leaving parallel flash and HPI. The following sections define each command with appropriate op-code, structure, and placement in AIS. Table 9 lists the various opcodes that are supported by AIS 1.0:

Table 9. AIS Version 1.0 Supported Opcodes

Opcode	Value
Section Load	0x58535901
Request CRC	0x58535902
Enable CRC	0x58535903
Disable CRC	0x58535904
Jump	0x58535905
Jump_Close	0x58535906
Set	0x58535907
Start Over	0x58535908
Reserved	0x58535909
Section Fill	0x5853590A
Ping	0x5853590B

3.1 SET Command

Set commands are a simple mechanism that enables you to write 8-bit, 16-bit, or 32-bit data to any address in DSP address space. There is a provision to provide delay after the memory write happens. This can be used for memory mapped register write to take effect. Set commands are used to configure various peripherals of DSP which includes PLL and EMIF at minimum and can configure more peripherals, if required.

When the DSP is powered-up, the PLL multiplier is bypassed and PLL Divider D1 is set to divide-by-1. As a result, the CPU is clocked at the same frequency as connected crystal/CLK IN, which is generally very low. This results in slow communication and high boot time. In order to reduce boot time, PLL and EMIF registers should be configured at the very beginning of boot process. For this reason, all set commands are placed at the beginning of AIS as shown in Figure 3.

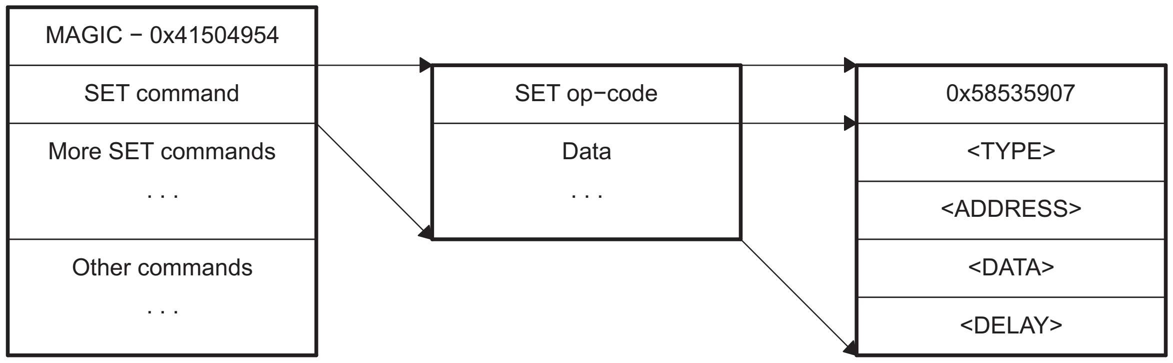


Figure 3. Structure of Set Command

Each set command consists of SET (0x58535907) op-code, followed by four words of additional data as shown. SET entries in AIS are explained using the following representation:

```
<Address> = <Data><Type>: : <Sleep>
```

The above command instructs the bootloader to write <Data> to address <Address> in DSP address space and the sleep for <Sleep> * CPU clocks. The number of CPU cycles specified by <Sleep> is treated as unsigned value. The data-type field <Type> determines the size of the data item such as 8-bit(B), 16-bit(S) or 32-bit(l). Data-type also may be a “field” or “bits”. This allows setting of a particular range of bits within the data at the specified address. For “field” and “bits” data-types, the <Type> field also encodes the “start” and “stop” bit positions that define the field to be modified. Table 10 gives a full list of the data-types that may be used.

Table 10. Data Types

Data Type	Value
8-bit	0
16-bits	1
32-bits	2
Field (1-32bits)	3
Bits (1-32bits)	4

The “field” and “bits” data-types are handled similarly by the bootloader. The difference between these types are that with a specifier of “field” , the bootloader performs a read/modify write operation at the given address. The “bits” data type results in a read of the address, followed by a write of the new value to the address. The <Type> specification is a 32 bit word that contains fields for data type (shown above), “start bit”, and “stop bit”. The “start bit” and “stop bit” fields are required only if a data-type of “field(3)” or (bits(4)” is used. These fields delimit the number of bits that are affected by the instruction. Table 12 shows the encoding of the 32 bit <Type>.

Figure 4. <Type> Field

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Rsvd								Stop bit								Start bit								Data type							

LEGEND: R/W = Read/Write; R = Read only; -*n* = value after reset

Table 11. <Type> Field Descriptions

Bit	Field	Value	Description
31-24	Reserved		
23-16	Stop bit		Stop bit (for "bits" and "fields" data type) last bit position in word that delimits field
15-8	Start bit		Start bit (for "bits" and "fields" data type) first bit position in word that delimits field
7-0	Data type		Data Type (0,1,2,3,4) specifies type of data to write

Table 12. Numeric Formats that can be used in BTEs

Name	Format	Example 1	Example 2	Example 3
Hexadecimal	0[xX][0-9a-fA-F]+	0x1234abCD	0x1000	0X5a
Hexadecimal	[0-9a-fA-F]+[hH]	1234ABCDh	1000H	5ah
Octal	0[0-7]+	02215125715	010000	0132
Decimal	[0-9]+	305441741	4096	90

3.2 Section Load Command

Section load commands are used to load a particular chunk of code/data to DSP memory. All initialized sections (such as .text) are loaded to DSP memory using Section load commands. These commands are placed after all set commands in AIS.

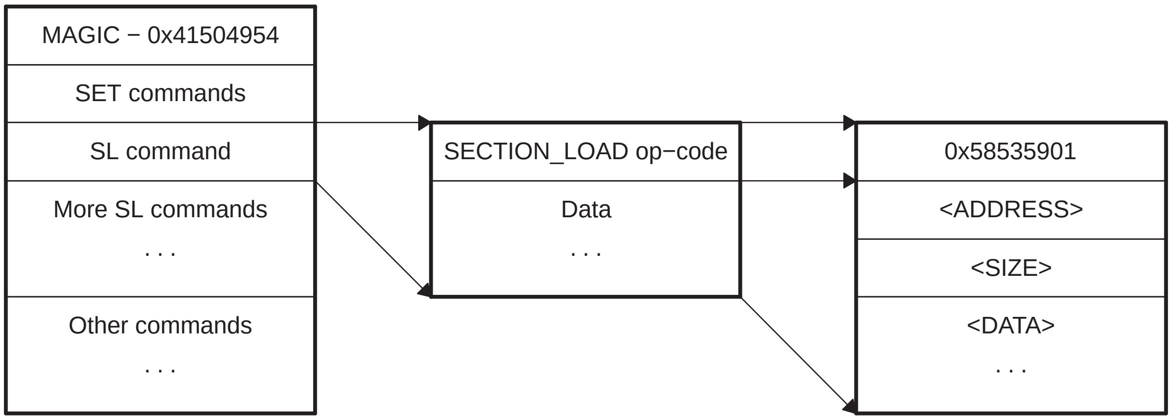


Figure 5. Structure of Section Load Command

Each section load command consists of SECTION_LOAD (0x58535901) op-code, followed by section's start address, size and contents.

3.3 Section Fill Command

Section fill commands are used when a particular section is filled with a specific pattern. For example, a section that contains all zeros is initialized with section fill command. These commands are placed anywhere where a regular section load command occurs.

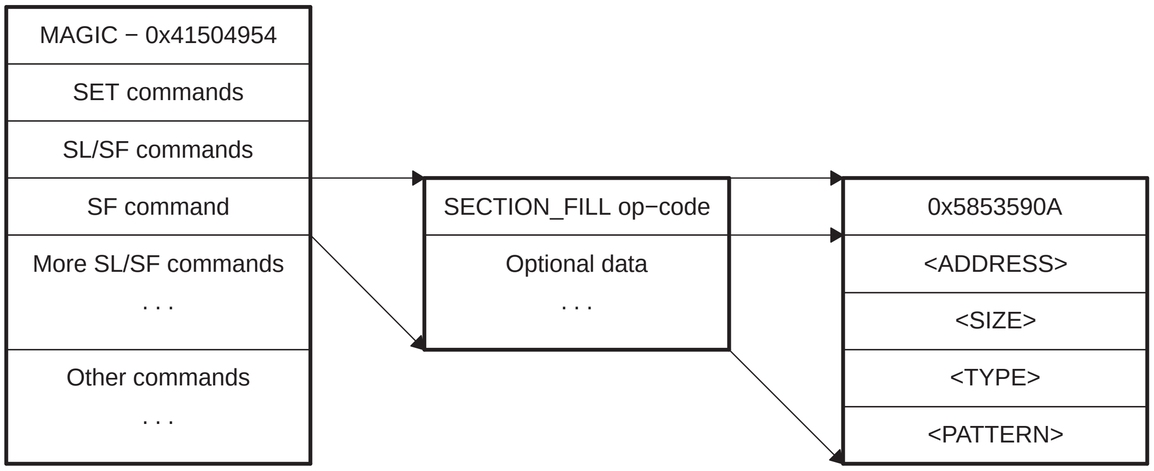


Figure 6. Structure of Section Fill Command

Each section fill command consists of SECTION_FILL (0x5853590A) op-code, followed by section's start address, size, pattern-type (8/16/32-bit) and pattern to be filled.

3.4 Jump Command

This command instructs the DSP to jump to start address of earlier loaded application. It consists of JUMP (0x58535905) op-code, followed by the jump address.

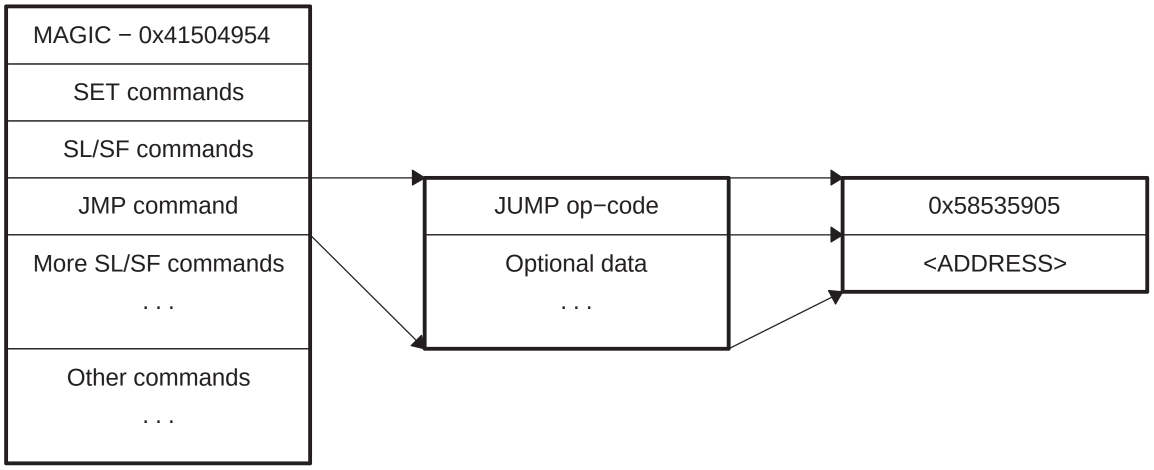


Figure 7. Structure of Jump Command

This command may be used to execute code that has been previously loaded through Section Load and Section Fill commands. It could be used to implement a secondary load process or to execute application code necessary to setup other processes before remainder of code/data is loaded. Once, the JUMP command is issued, execution will begin at the indicated start address. When execution is over, it is the responsibility of the application code to execute a return instruction. This enables return of control to the on-chip bootloader. Normal AIS interpretation and execution will continue at that point.

3.5 Jump_Close Command

This command is used at the end of the boot process to start execution of loaded application. This command instructs the DSP to terminate boot process and jump to start address of loaded application.

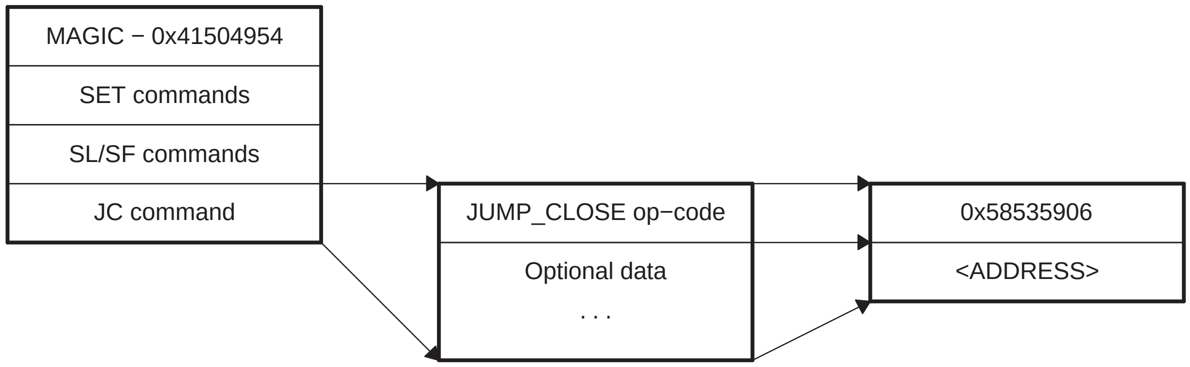


Figure 8. Structure of Jump_Close Command

The command is placed at the end of AIS, after all other commands. It consists of JUMP_CLOSE (0x58535906) op-code, followed by the start address of the application where the bootloader should jump.

3.6 CRC Options

There is a possibility of error in communication when DSP is booting up. Execution of a corrupted application image may result in instability or malfunction. In order to avoid such problems, AIS supports opcodes to verify the validity of data loaded through Section Load / Section Fill commands. A proprietary 32-bit CRC computation algorithm is used for verification. The three options available are:

No CRC

With this option, CRC computation is disabled and there is no way to detect or correct any error.

Single CRC

With this option, single CRC will be computed for all the sections. Verification will be done at the end, just before Jump_Close command. In case of error, all the sections are loaded again. CRC will be recalculated and re-verified again at the end.

Section-Wise CRC

With this option, CRC is computed for each section. Verification is done at the end of each section. The section is reloaded in case of error.

3.6.1 Enable/Disable CRC Commands

These commands are used to enable/disable computation of CRC for sections loaded through Section Load / Section Fill commands.

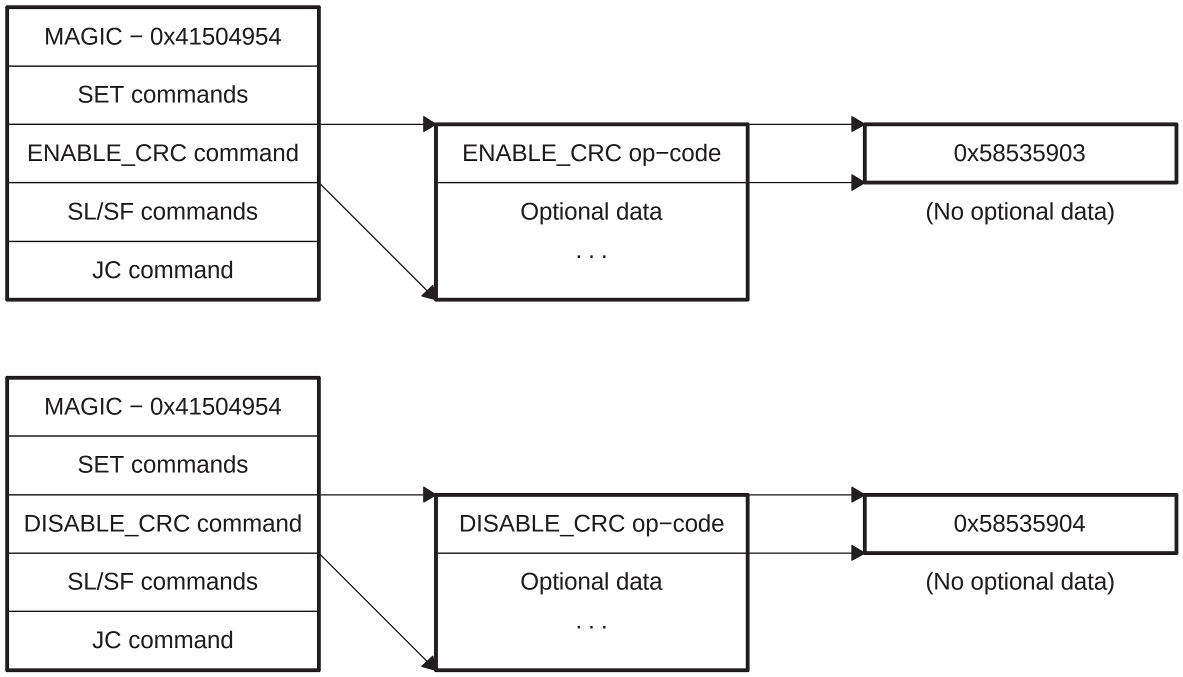


Figure 9. Structure of Enable CRC / Disable CRC Commands

These commands consist of only a single ENABLE_CRC (0x58535903) or DISABLE_CRC (0x58535904) op-code. There is no additional data required.

3.6.2 Request CRC Command

This command is used to request and validate current value of CRC computed by DSP. Using this command requires that the Enable CRC command is issued earlier in the AIS stream. This command consists of REQUEST_CRC(0x58535902) op-code, followed by the expected CRC value and a seek-value. The CRC of loaded/filled section(s) are compared with the expected CRC value. If the CRC is correct, seek-value is ignored and execution shall continue from next command.

A mismatch in CRC indicates that the data loaded to the DSP memory using earlier Section Load/ Section Fill commands is corrupted. In order to load data again, AIS has to be re-executed from the last error-free point (i.e last valid command). The seek-value is expressed in bytes, and is a negative number that is added to the current address in AIS. By adding the seek-value, the AIS stream is then pointed back to the last known good state and AIS interpretation continues from this address.

When operating in master mode, CRC read/compare/seek adjustment are performed automatically by the bootloader. In slave mode operation, a REQUEST_CRC command results in the bootloader transmitting the current CRC value calculated by the DSP to the Host. The Host may then send a Start-Over command as described in next section. On receiving the Start-Over command the DSP knows that CRC error has occurred. It resets its CRC computation and becomes ready to accept more commands from the host. The next command expected by the DSP is a PING command, followed by Host/slave mode exchange (XMT_START/RECV_START).

Please refer to Appendix A for code used to calculate CRC values.

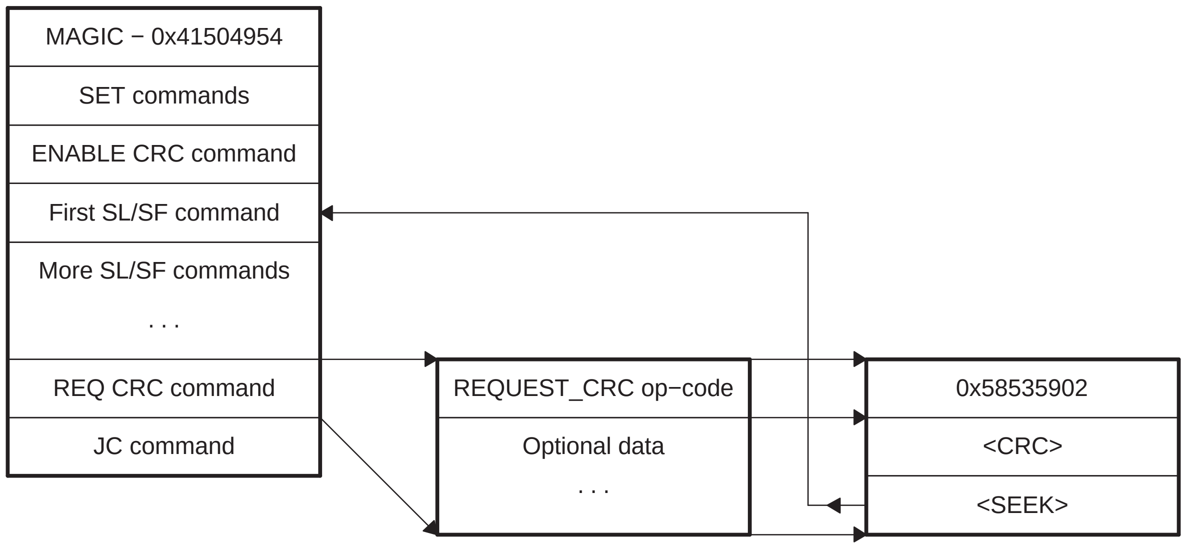


Figure 10. Structure of Request CRC Command for Single CRC Option

For single CRC option, this command appears only once in AIS after the last Section Load / Section Fill command. The seek value is interpreted as a negative number, which when added to the current offset in AIS, will make offset point to start of the first Section Load / Section Fill command as shown.

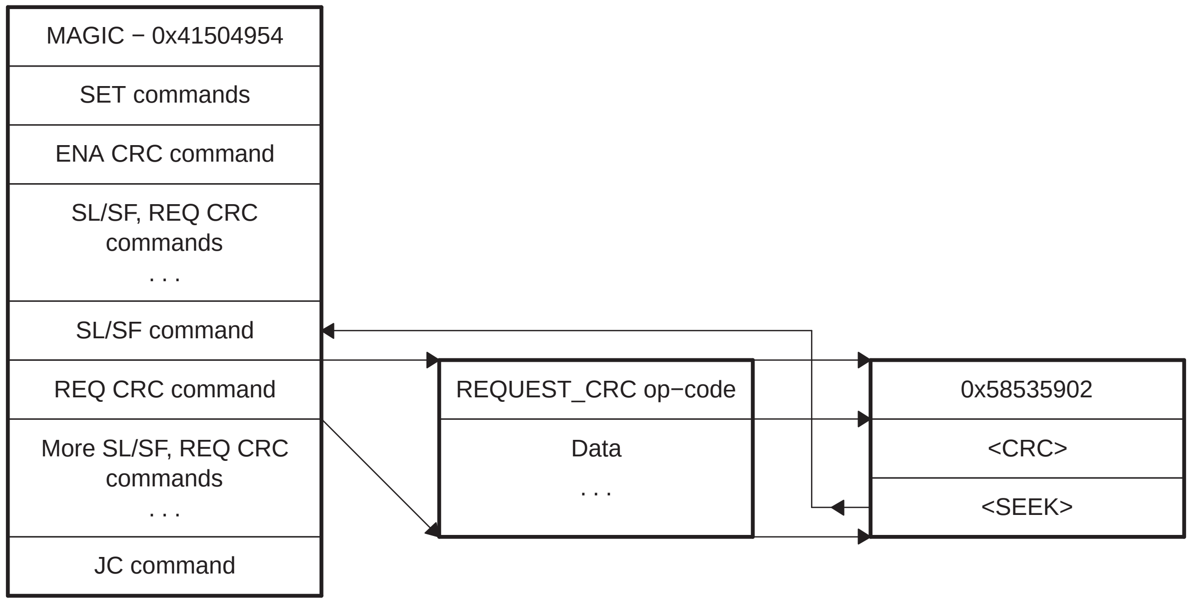


Figure 11. Structure of Request CRC Command for Section-wise CRC Option

For section-wise CRC option, this command appears after each Section Load / Section Fill commands. The seek value is interpreted as a negative number, which when added to current offset in AIS, will make offset point to start of the previous Section Load / Section Fill command as shown.

3.6.3 Start-over Command

The start-over command consists of a STARTOVER (0x58535908) op-code with no additional data. This instructs the bootloader to reset its computed CRC value to 0.

It has to be issued by the host on its own when it detects a CRC mismatch for slave modes. For master modes, this is handled by the bootloader state machine.

4 External Serial EEPROM Boot

The bootloader contains the AIS interpreter for parsing the data read from the serial EEPROM. After parsing the data retrieved, the bootloader takes appropriate actions in order to execute the opcode.

5 External Host Processor Boot

When booting from the external host processor, the host processor acts as a boot master and the DSP acts as slave. Since the DSP does not have direct access to AIS, the host processor has to transfer it to the DSP through a well-defined protocol explained in following sections. An AIS interpreter is required on the host processor to control this transfer.

5.1 AIS Interpreter on the Host

The AIS interpreter on the host is responsible for transferring the AIS to the DSP. For this, it has to understand the transfer protocol and implement the required handshake mechanism. The AIS interpreter on host directly interacts with the bootloader on the DSP.

NOTE: For the sake of simplicity, AIS interpreter on host will be simply referred to as ‘host’ and bootloader on TMS320C672x devices as ‘DSP’ in this section.

It is important to have a successful link establishment between the DSP and the host before starting transfer of AIS. Once the link is established, AIS is transferred to DSP. The whole process is divided into three phases:

- start-word synchronization (SWS)
- ping op-code synchronization (POS)
- op-code synchronization (OS)

5.2 Start-Word Synchronization

Start-word synchronization (SWS) is the default power-up state and is responsible for initiating communication between the DSP and the host.

The bootloader tries to read the transmit start-word (XMT_START) from the host. After receiving it, the DSP acknowledges by sending receiver-start-word (RECV_START) to the host.

The XMT_START and RECV_START can be 8-bit, 16-bit, or 32-bit depending on the boot mode used. The following table shows corresponding values for different boot modes. Please note that in all cases the bootloader expects data to be transmitted most significant bit (MSB) first.

Boot Mode	XMT_START	RECV_START
8-bit	0x58	0x52
16-bit	0x5853	0x5253
32-bit	0x58535441	0x52535454

The host must keep on sending XMT_START until it receives RECV_START from the DSP. This process initiates communication between the DSP and the host. Figure 12 shows the flowchart of how SWS is implemented on the host.

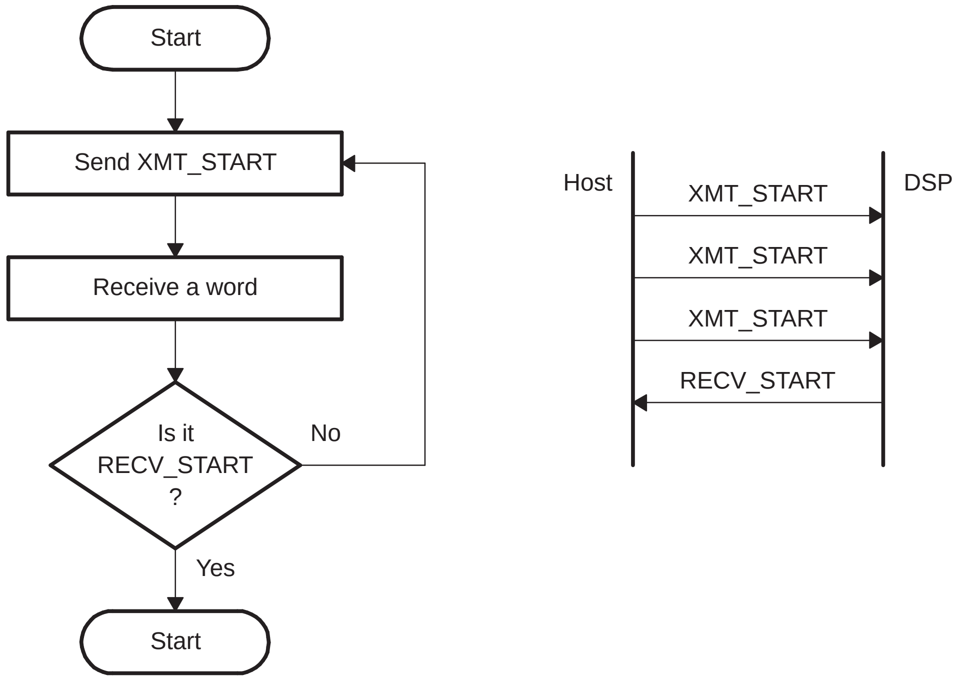


Figure 12. Flowchart: Start-Word Synchronization

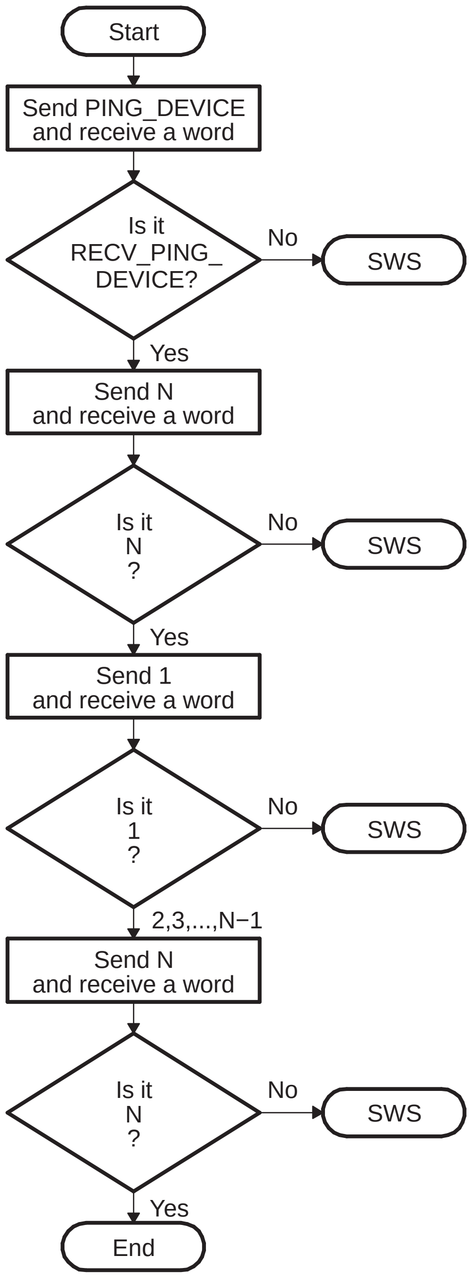
5.3 *Ping Op-code Synchronization*

Ping Op-code Synchronization (POS) is used to make sure that the boot mode selected is correct and the communication link between the host and the DSP is reliable.

After successful SWS,

- DSP waits for the host to send PING_DEVICE (0x5853590B) op-code. On receiving it, DSP acknowledges it by sending RECV_PING_DEVICE (0x5253590B) to the host.
- The host then sends a number N to the DSP and gets back the same number from the DSP as acknowledgment.
- The host shall then start sending numbers 1 to N to DSP and will receive the same sequence as acknowledgment.

POS is implemented as a simple bootloader command and it can be issued any time during AIS transfer to check reliable communication. If POS fails at any point, the DSP and the host are required to start all over again from SWS. Figure 13 shows the flowchart of how POS is implemented on the host.


Start
Send PING_DEVICE and receive a word
Is it RECV_PING_ DEVICE?
No
SWS
Yes
Send N and receive a word
Is it N ?
No
SWS
Yes
Send 1 and receive a word
Is it 1 ?
No
SWS
2,3,...,N−1
Send N and receive a word
Is it N ?
No
SWS
Yes
End

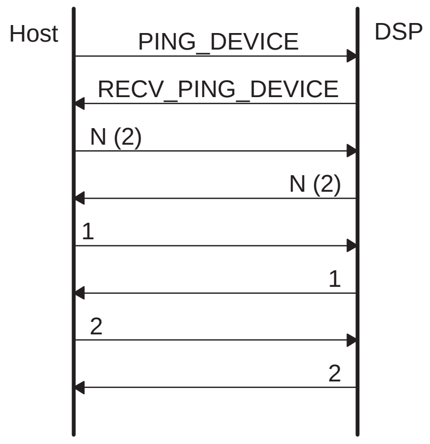


Figure 13. Flowchart: Ping Op-code Synchronization

5.4 Op-code Synchronization (OS) for Serial Slave Modes

After a successful link establishment, the DSP and the host are ready to transfer AIS commands. Since all AIS commands start with an op-code, the DSP waits to receive one of valid op-codes from the host. For serial slave modes on receiving an opcode, the DSP acknowledges by sending corresponding RECV opcode. This process is referred to as opcode synchronization.

All opcodes (including PING_DEVICE) transmitted by the host to the DSP are of the form 0x585359##, where ## varies for individual op-codes. DSP acknowledges each op-code by corresponding RECV op-code. RECV op-codes are generated from the original op-codes by changing the most significant byte to 0x52. Thus, they are of the form 0x525359##.

Not getting a correct response (RECV op-code) from DSP means that the DSP is busy executing an earlier op-code. The host should continue sending the op-code until successfully acknowledged by the DSP. Figure 14 shows the flowchart of how OS is implemented on the host.

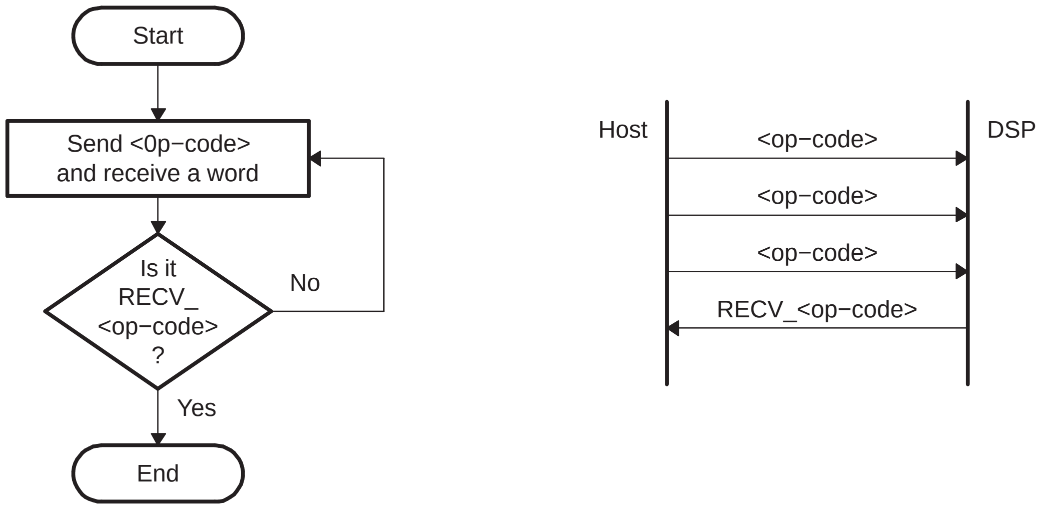


Figure 14. Flowchart: Op-code Synchronization

DSP starts executing the op-code only after the OS is finished. If more information is required in order to execute the op-code, the DSP gets it from the host before starting execution. The host is required to understand each op-code and supply required data to the DSP from the AIS.

The DSP keeps on executing commands from the host until it gets a Jump_Close command, using Op-code synchronization at the beginning of each command. On getting JNC command, the DSP closes the peripheral used for booting, terminates the boot process and jumps to the address specified along with the op-code.

6 Bootloader Utilities

Two software utilities, genBootCfg and genAIS, have been developed to assist in creating AIS data streams for booting in SPI or I2C boot modes. genAIS, may also be used with output from genBootCfg to generate a template for a secondary bootloader that can be used for booting from Parallel Flash. genBootCfg is a Perl TK GUI that assists you in creating configuration data for PLL, EMIF SDRAM and ASYNC RAM, and GPIO pin configuration for GPIO pins that may be used as address pins for extending the addressing for Parallel Flash boot. genBootCfg produces two output files, *.cfg and *cfg.c. The .c file contains code to perform peripheral initialization for PLL, SDRAM, ASYNC RAM according to options selected when running genBootCfg. If the secondary bootloader is not used, this code could be incorporated in any source code to initialize PLL or EMIF.

The SPI, I2C and secondary bootloader require AIS data streams for boot load. genAIS, creates the necessary AIS boot format needed for each of these boot modes. This utility expects as input an application object (*.out) and an optional *.cfg file saved from invocation of genBootCfg, and produces an AIS boot format file in either ASCII, binary, or C672x assembly format. The AIS data stream produced can be programmed onto an I2C or SPI EEPROM or Flash. When using C672x assembly format as output, this assembly can be assembled and linked and then input to the TMS320C6000 Hex Conversion Utility (hex6x, see Code Gen Tools UG).

NOTE: The .cfg file is optional for SPI and I2C boot, but is required when generating secondary bootloader for Parallel Flash boot.

genAIS can be invoked from a command line or within a makefile, so that it may be included as part of an application build script.

6.1 Installing Bootloader Utilities

If you do not currently have Perl 5.8.4 or 5.8.6 installed on your machine, please download and install the latest version for your platform from any available site. Versions later than 5.8.6 have been reported to cause problems.

Once installation is complete, please add the following variables to your environment:

```
PERL5LIB=myUtilInstalDirl\lib
PATHEXT = %PATHEXT%, .PL
PATH = %PATH%; myUtilInstallDir\bin
```

From any command window, genBootCfg or genAIS, may now be invoked.

6.2 Using genBootCfg

genBootCfg is a simple Perl Tk GUI. No arguments are needed when invoking the utility. To start the GUI, type ‘genBootCfg’ on the command line. The main GUI interface will be displayed.

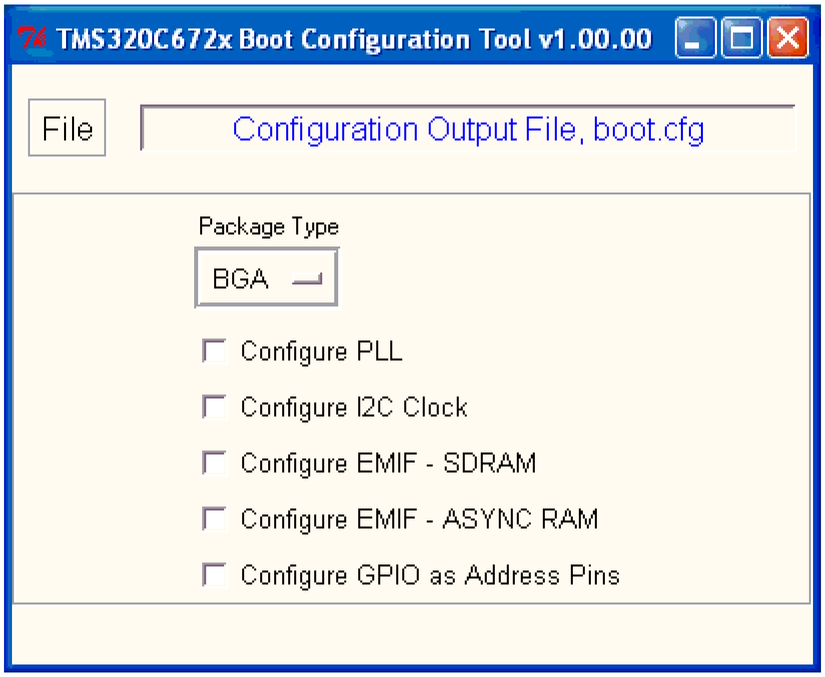

TMS320C672x Boot Configuration Tool v1.00.00
File
Configuration Output File, boot.cfg
Package Type
BGA
Configure PLL
Configure I2C Clock
Configure EMIF - SDRAM
Configure EMIF - ASYNC RAM
Configure GPIO as Address Pins

Figure 15. genBootCfg Main Menu Window

There are a few generic operations that will apply to all configuration windows:

- To save a configuration, click **OK** provided in each configuration window.
- To clear a current set of choices, click **CLEAR** provided in the configuration window. This will remove all choices back to the power on default settings.
- To cancel any configuration operation, click **CANCEL**. No configuration will be performed for any configuration window that has been cancelled.

6.2.1 File Operations

To perform any file operation such as open or save, choose the appropriate item from the File pulldown menu. The following set of options is supported.

- **New** - Creates a new set of configuration files. The root filename for the configuration file will be used in naming the corresponding C source file.
- **Open** - Opens a previously saved configuration file.
- **Save** - Saves configuration to the current selected file.
- **Save as** - Saves configuration to another file.
- **Exit** - Exits configuration tool.

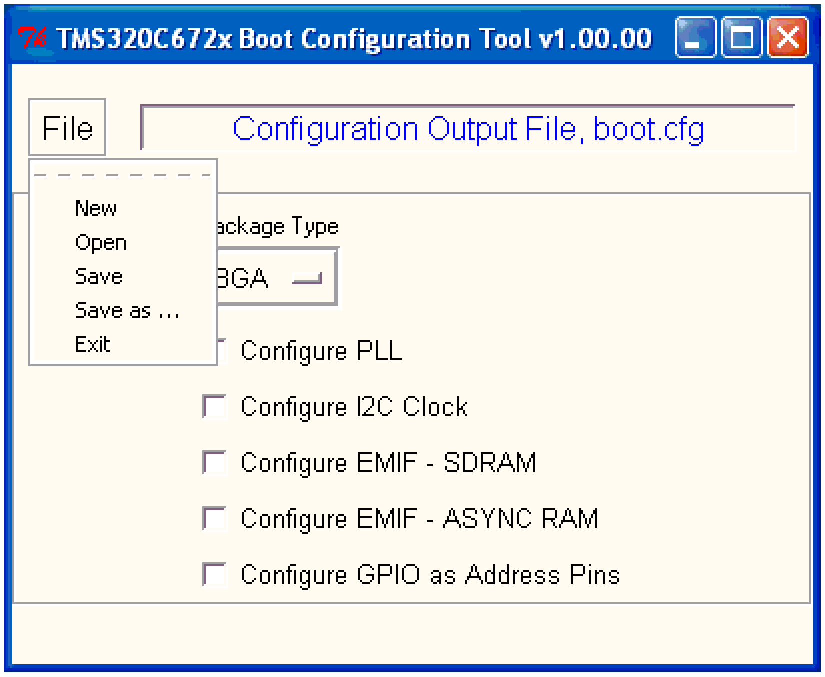

TMS320C672x Boot Configuration Tool v1.00.00
File
Configuration Output File, boot.cfg
New
Open
Save
Save as ...
Exit
ackage Type
3GA
Configure PLL
Configure I2C Clock
Configure EMIF - SDRAM
Configure EMIF - ASYNC RAM
Configure GPIO as Address Pins

Figure 16. File Pulldown Menu

By default, the script will produce output files, boot.cfg and bootcfg.c. To change the output file name, use the File pulldown menu and select **New** or use the **Save as** feature to save the configuration in a different file.

6.2.2 Selecting Device Package Type

From the main menu you should first choose the device type for which you want to perform configuration. This is either “BGA” or “TQFP”. Choosing the package type affects the number of address pins which may be configured and the number of GPIO pins available for configuration when extending the EMIF address range for boot from Parallel Flash.

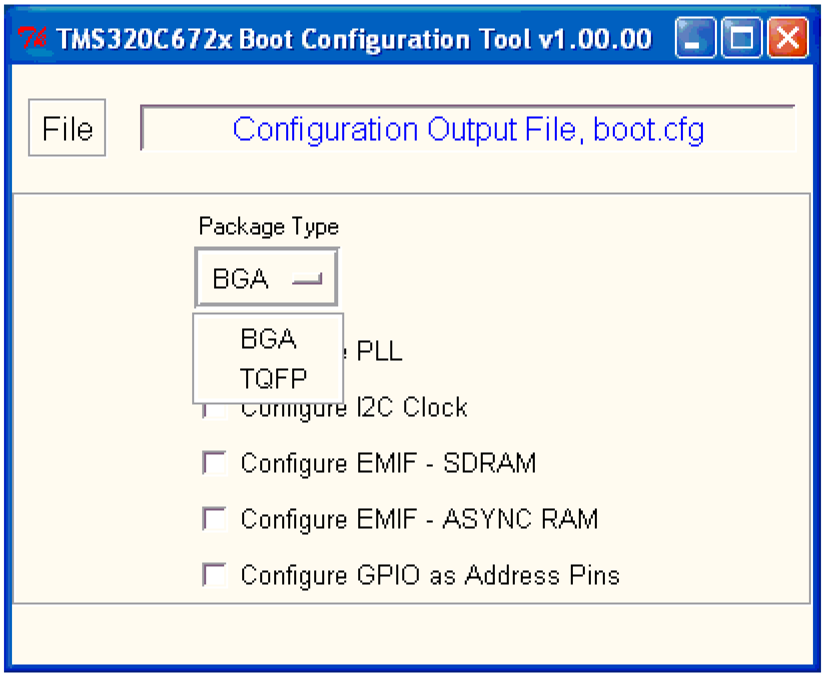
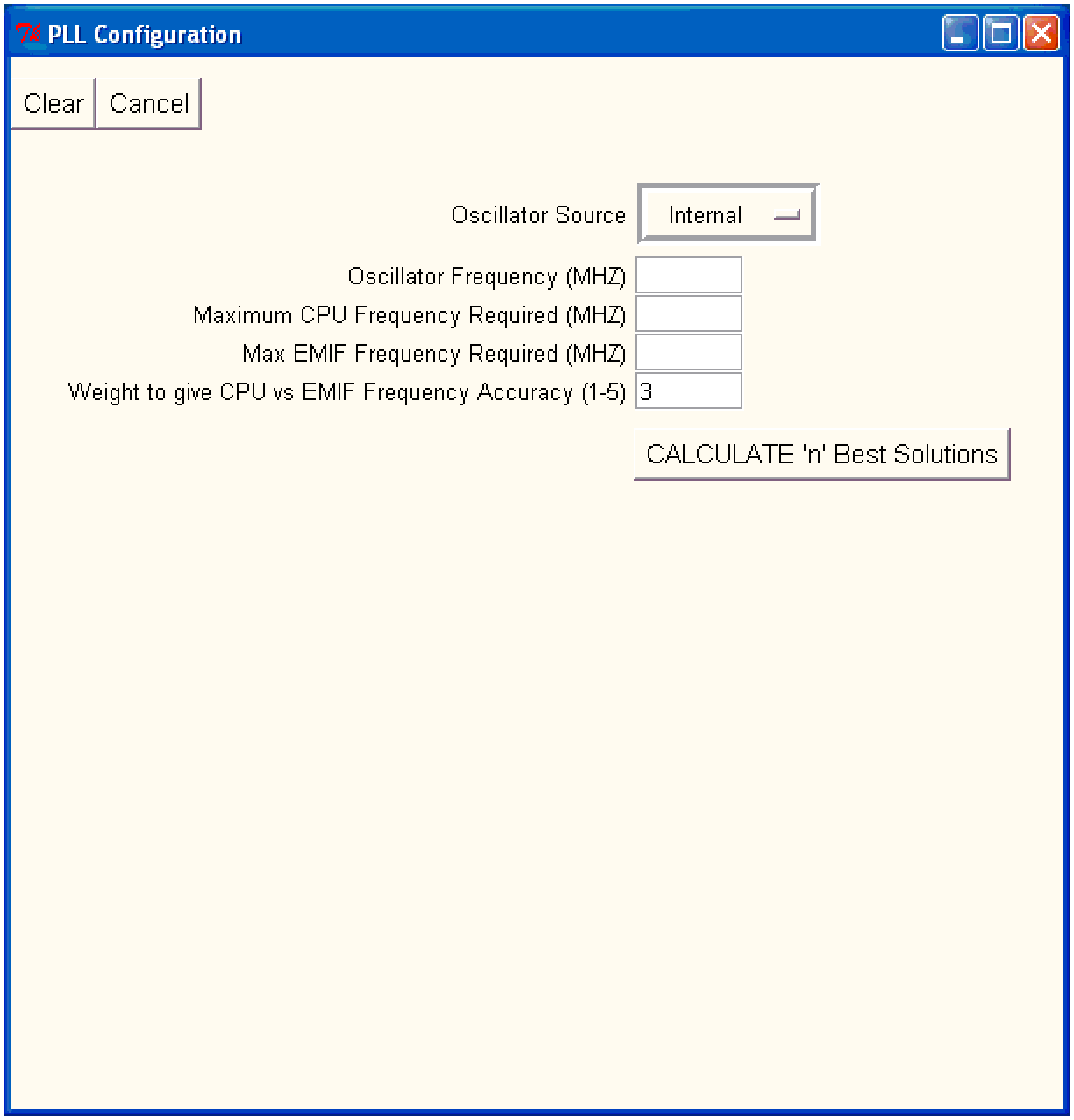

TMS320C672x Boot Configuration Tool v1.00.00
File
Configuration Output File, boot.cfg
Package Type
BGA
BGA
TQFP
PLL
Configure I2C Clock
Configure EMIF - SDRAM
Configure EMIF - ASYNC RAM
Configure GPIO as Address Pins

Figure 17. Package Type Pulldown Menu

6.2.3 Configuring the PLL

From the main menu window, click the “Configure PLL” checkbox. A new window will pop-up. Configuring the PLL requires the following input parameters:

- **Oscillator source -** Choose either the "internal" or "external"
- **Oscillator frequency** - Input oscillator frequency
- **Maximum CPU frequency** - Maximum CPU frequency required by application
- **Maximum EMIF frequency** - Maximum EMIF frequency required by application
- **CPU versus EMIF weight** - Weight given to approximation method when determining PLLM, and PLLDIV values needed to generate the requested CPU and EMIF frequencies.



PLL Configuration

Clear Cancel

Oscillator Source: Internal

Oscillator Frequency (MHZ):

Maximum CPU Frequency Required (MHZ):

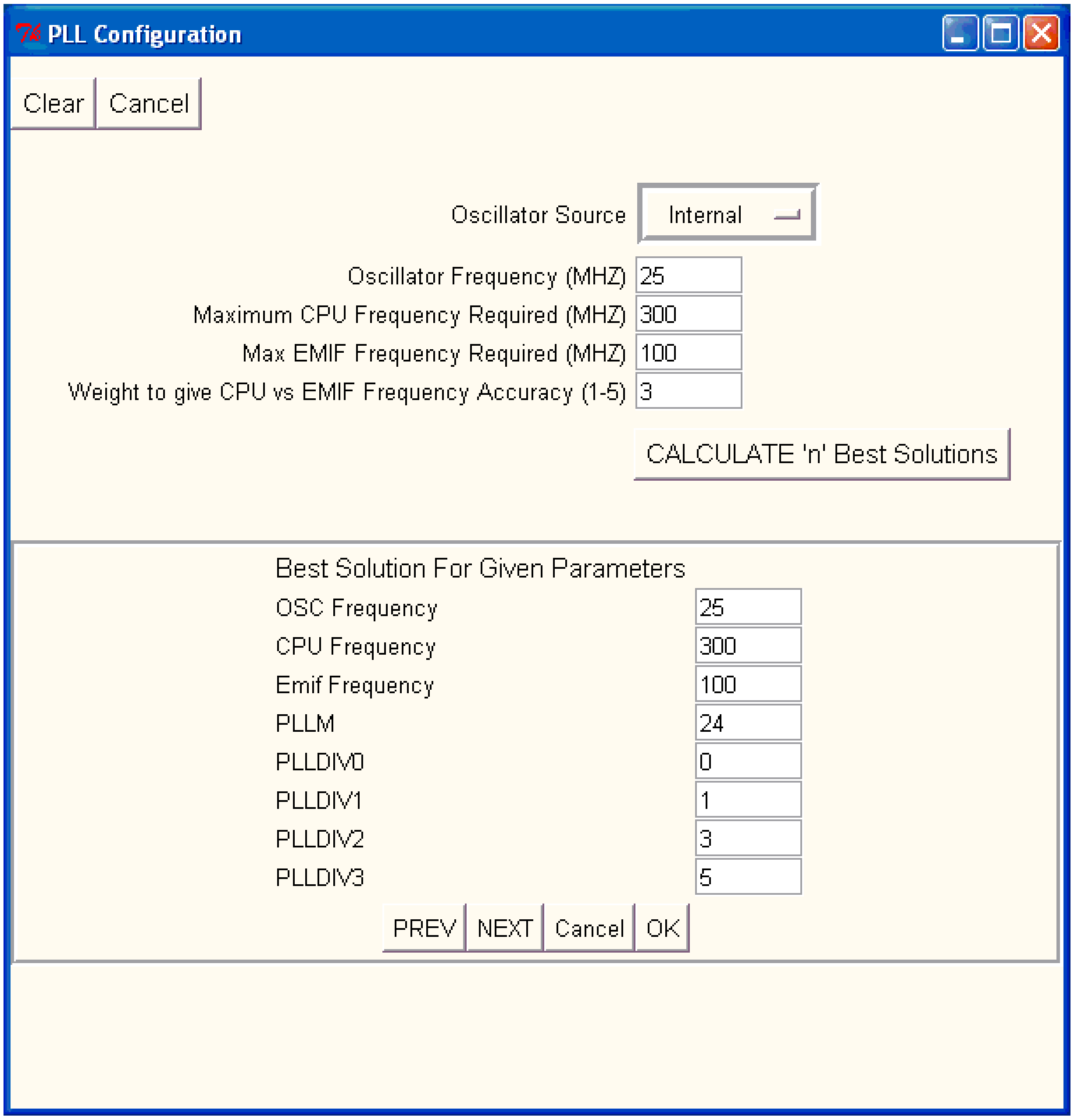
Max EMIF Frequency Required (MHZ):

Weight to give CPU vs EMIF Frequency Accuracy (1-5): 3

CALCULATE 'n' Best Solutions

Figure 18. PLL Configuration Window

To have the best-fit algorithm calculate a set of PLL settings, click “Calculate ‘n’ Best Solutions” button in the PLL configuration window. The best-fit algorithm will generate a number of possible combinations of PLLM, and PLLDIV0, PLLDIV1, PLLDIV2 , and PLLDIV3 values that will approximate the CPU and EMIF frequencies. The solutions will be displayed in a separate frame which is generated once calculations are complete. The best fit solution will be highlighted. To see other possible solutions, use the “prev” and “next” buttons to browse through the solution data base.



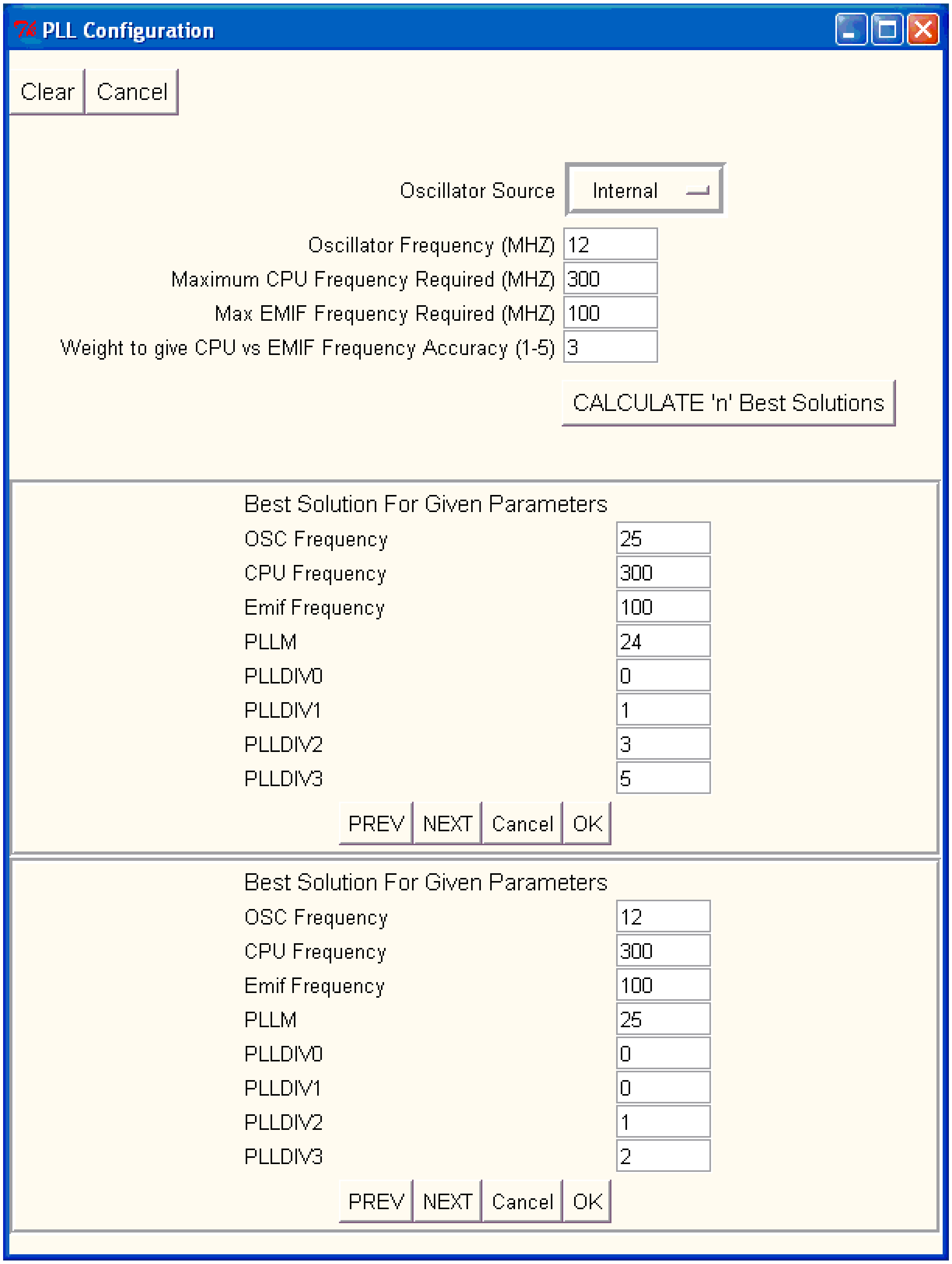
Oscillator Source	Internal
Oscillator Frequency (MHZ)	25
Maximum CPU Frequency Required (MHZ)	300
Max EMIF Frequency Required (MHZ)	100
Weight to give CPU vs EMIF Frequency Accuracy (1-5)	3

Best Solution For Given Parameters

OSC Frequency	25
CPU Frequency	300
Emif Frequency	100
PLLM	24
PLLDIV0	0
PLLDIV1	1
PLLDIV2	3
PLLDIV3	5

Figure 19. PLL Configuration Solutions Frame

To save a particular set of PLL configurations, simply click **OK** while that solution is being displayed. The PLL configuration is capable of providing solutions for more than one set of PLL oscillator, CPU and EMIF frequency inputs, if comparison between different PLL input settings is required. Multiple solution frames will be generated.



Oscillator Source	Internal
Oscillator Frequency (MHZ)	12
Maximum CPU Frequency Required (MHZ)	300
Max EMIF Frequency Required (MHZ)	100
Weight to give CPU vs EMIF Frequency Accuracy (1-5)	3

Best Solution For Given Parameters

OSC Frequency	25
CPU Frequency	300
Emif Frequency	100
PLLM	24
PLLDIV0	0
PLLDIV1	1
PLLDIV2	3
PLLDIV3	5

Best Solution For Given Parameters

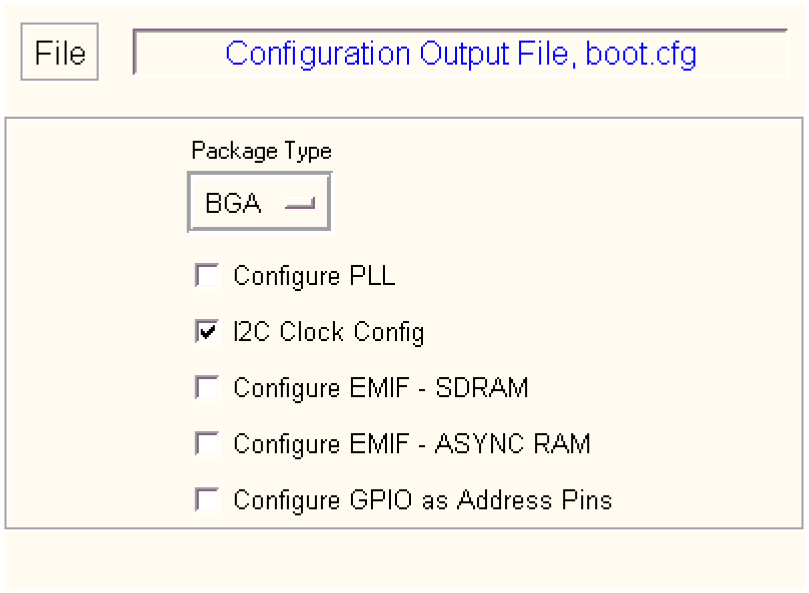
OSC Frequency	12
CPU Frequency	300
Emif Frequency	100
PLLM	25
PLLDIV0	0
PLLDIV1	0
PLLDIV2	1
PLLDIV3	2

Figure 20. Multiple PLL Solution Windows

To save the PLL settings, simply click **OK** in the frame where the needed solution is currently being displayed.

6.2.4 I2C Clock Configuration

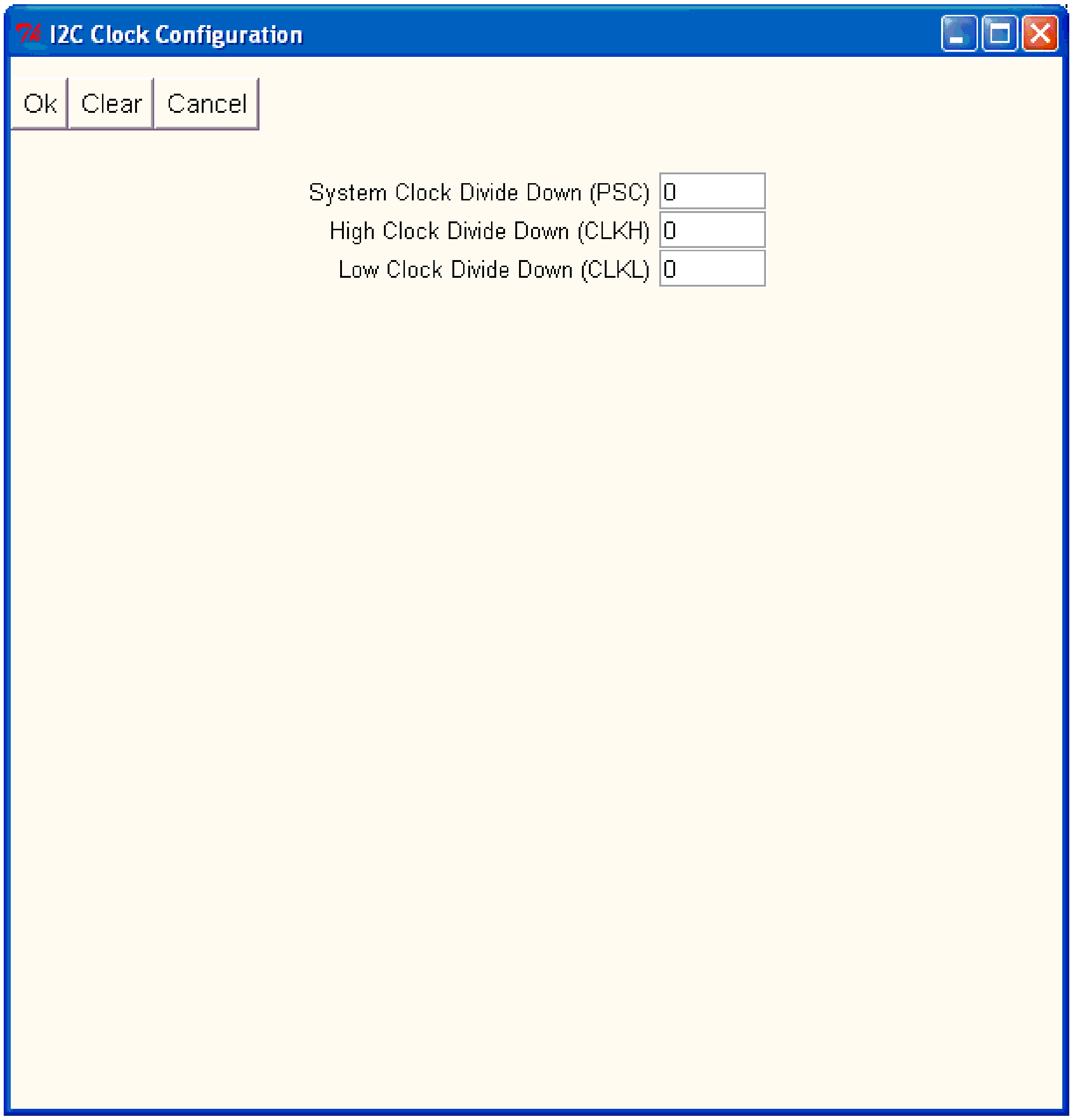
The I2C clock may be configured by selecting the I2C Clock Config checkbox on the genBootCfg main menu.



- [ ] Configure PLL
- [x] I2C Clock Config
- [ ] Configure EMIF - SDRAM
- [ ] Configure EMIF - ASYNC RAM
- [ ] Configure GPIO as Address Pins

Figure 21. I2C Clock Configuration Selection from Main Menu

A configuration window will then open that allows you to specify the clock dividers, PSC, CLKH, and CLKL.



System Clock Divide Down (PSC) 0
High Clock Divide Down (CLKH) 0
Low Clock Divide Down (CLKL) 0

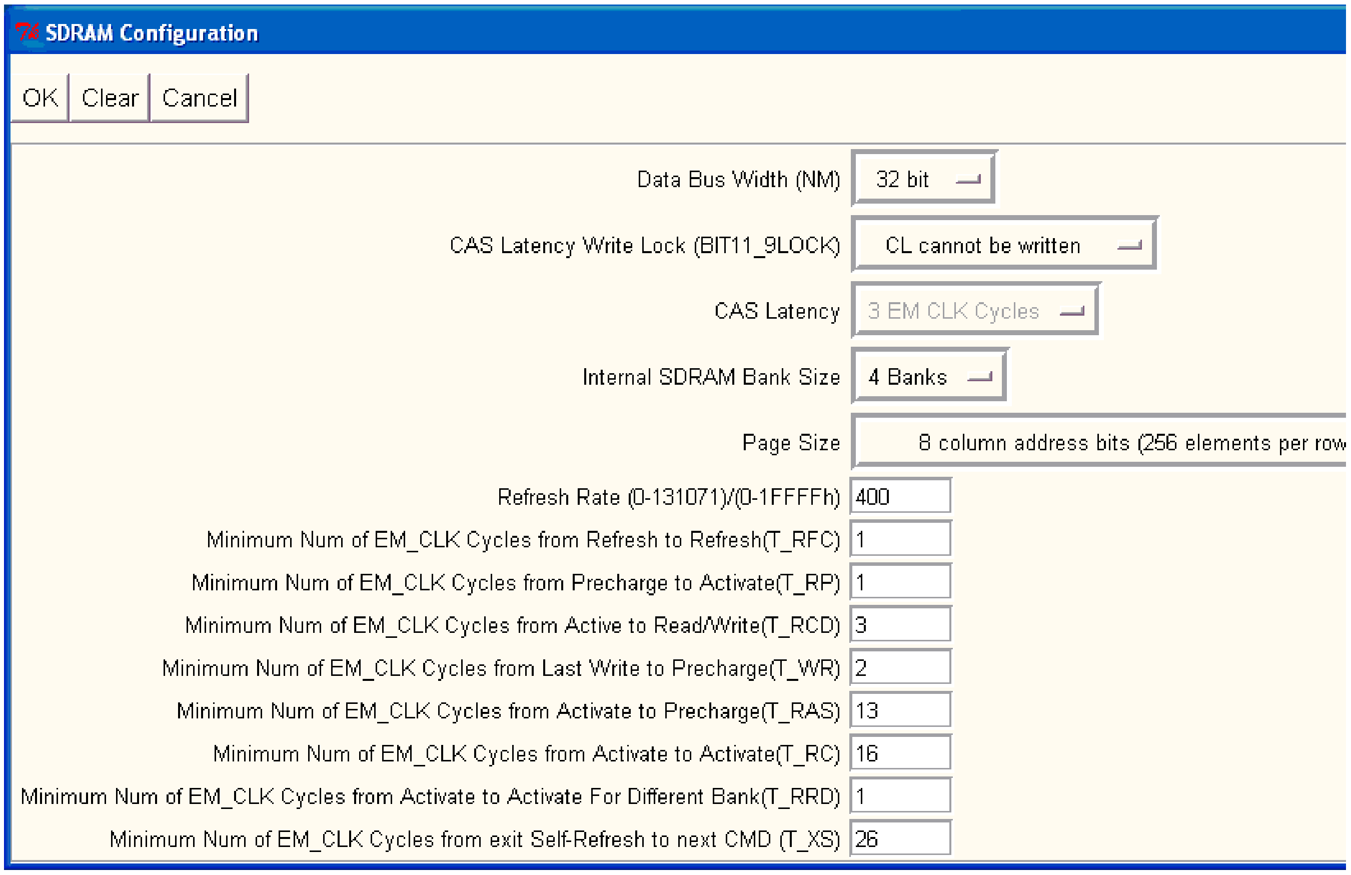
Figure 22. I2C Clock Configuration Window

Please consult the specification for the I2C serial EEPROM for your device to determine appropriate values for PSC, CLKH, CLKL.

6.2.5 SDRAM Configuration

SDRAM configuration is accomplished by clicking the check box labeled “Configure SDRAM”, in the main GUI window.

Once the “Configure SDRAM” checkbox is selected, a new pop-up window appears, that allows setting parameters for SDRAM.



Data Bus Width (NM)	32 bit
CAS Latency Write Lock (BIT11_9LOCK)	CL cannot be written
CAS Latency	3 EM CLK Cycles
Internal SDRAM Bank Size	4 Banks
Page Size	8 column address bits (256 elements per row
Refresh Rate (0-131071)/(0-1FFFFh)	400
Minimum Num of EM_CLK Cycles from Refresh to Refresh(T_RFC)	1
Minimum Num of EM_CLK Cycles from Precharge to Activate(T_RP)	1
Minimum Num of EM_CLK Cycles from Active to Read/Write(T_RCD)	3
Minimum Num of EM_CLK Cycles from Last Write to Precharge(T_WR)	2
Minimum Num of EM_CLK Cycles from Activate to Precharge(T_RAS)	13
Minimum Num of EM_CLK Cycles from Activate to Activate(T_RC)	16
Minimum Num of EM_CLK Cycles from Activate to Activate For Different Bank(T_RRD)	1
Minimum Num of EM_CLK Cycles from exit Self-Refresh to next CMD (T_XS)	26

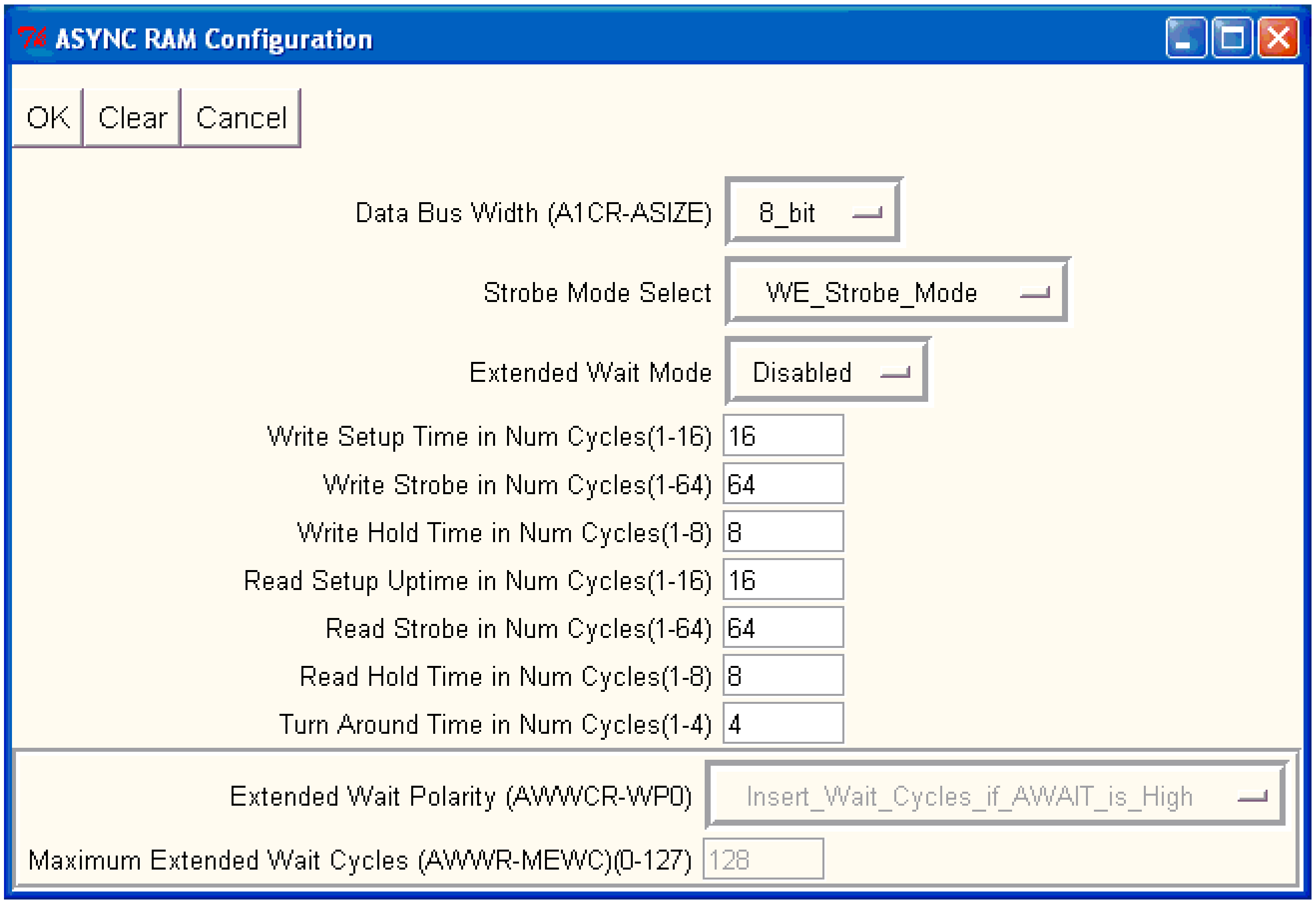
Figure 23. SDRAM Configuration Window

The input parameters in the SDRAM configuration window, map to the available fields in the EMIF configuration registers for SDRAM control, and refresh timing setup. For detailed information regarding these parameters and an example of how to set these parameters for a particular memory, please refer to the *TMS320C672x DSP External Memory Interface (EMIF) User's Guide* (SPRU711). Please note that many of the EMIF register fields encode the number of required cycles as “number of cycles – 1”. The genBootCfg utility automatically subtracts one from the number of cycles specified in the GUI input to properly encode the register fields. When inputting the number of cycles, please input the number required and not the value to encode in the register field.

6.2.6 ASYNC Ram Configuration

To configure EMIF control registers for Asynchronous RAM, simply click the check button labeled “Configure EMIF – ASYNC RAM”, in the genBootCfg main menu window. When this is selected, the ASYNC Ram configuration window pops-up to allow setup of parameters for configuring EMIF connection.

The input parameters for ASYNC RAM configuration correspond the fields that are described in the EMIF control registers for Asynchronous memory interface. For detailed descriptions of these fields and their meaning, please refer to the *TMS320C672x External Memory Interface User's Guide* (SPRU711). Please note that as with the case of SDRAM configuration, many of the control register fields encode cycle timing requirements as “number of cycles – 1”. The genBootCfg utility will automatically subtract one from those fields that meet this requirement when encoding the register values. Therefore, when entering cycle counts, please enter the number of cycles required and not the value expected in the register field.


ASYNC RAM Configuration

Data Bus Width (A1CR-ASIZE)	8_bit
Strobe Mode Select	WE_Strobe_Mode
Extended Wait Mode	Disabled
Write Setup Time in Num Cycles(1-16)	16
Write Strobe in Num Cycles(1-64)	64
Write Hold Time in Num Cycles(1-8)	8
Read Setup Uptime in Num Cycles(1-16)	16
Read Strobe in Num Cycles(1-64)	64
Read Hold Time in Num Cycles(1-8)	8
Turn Around Time in Num Cycles(1-4)	4
Extended Wait Polarity (AWWCR-WP0)	Insert_Wait_Cycles_if_AWAIT_is_High
Maximum Extended Wait Cycles (AWWR-MEWC)(0-127)	128

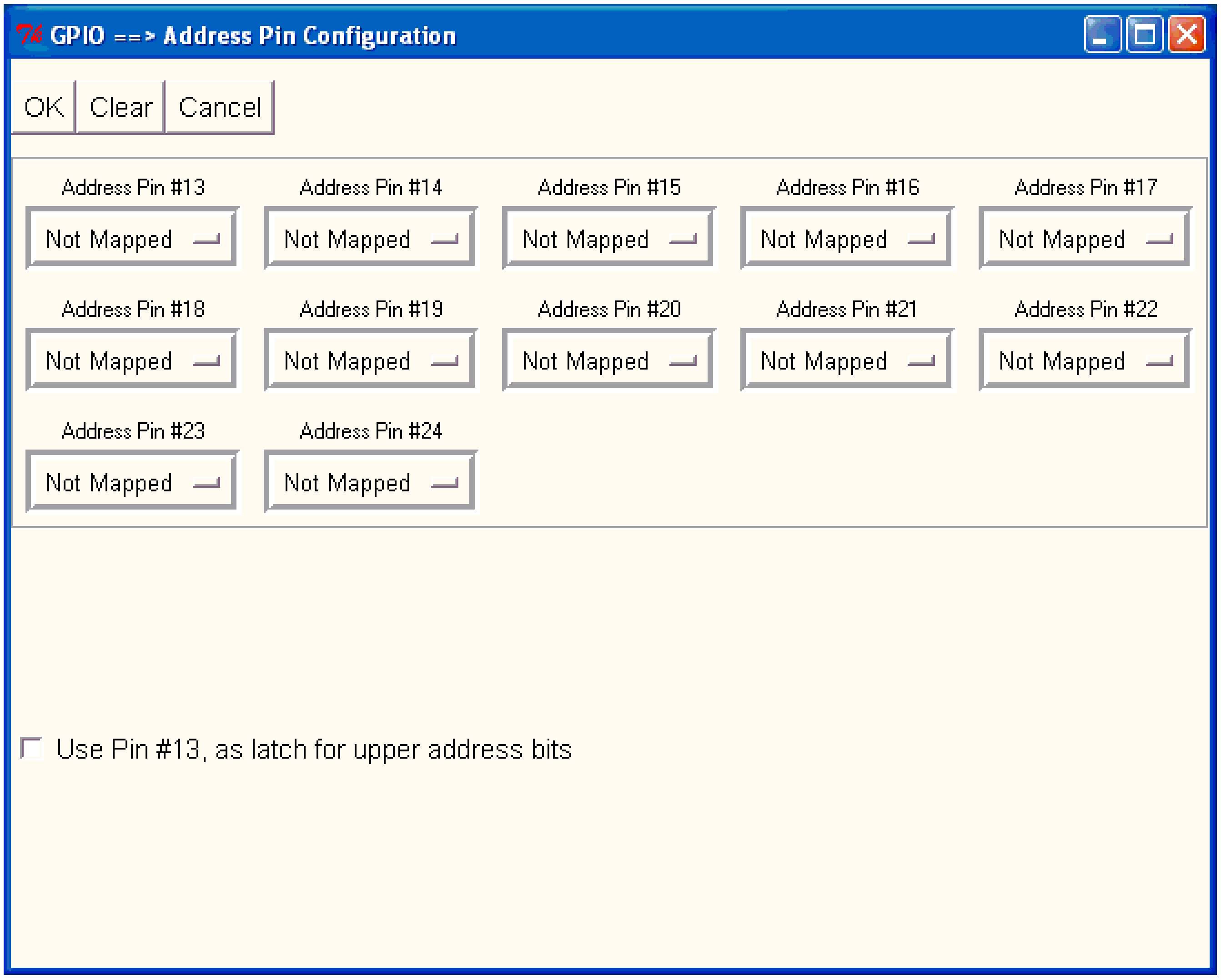
Figure 24. ASYNC RAM Configuration Menu

6.2.7 Configuring GPIO Pins as Address Pins

The TMS320C672x EMIF supports 12 (TQFP packaging) or 13 (BGA packaging) address pins on the EMIF to address Parallel Flash memories. This limits the maximum number of elements in the flash which may be addressed to 8K for BGA package. For applications booting from Parallel Flash that require an address range beyond this limit, GPIO pins may be used to extend the addressing range. The on-chip bootloader does not currently support extending the address range using this method. It will copy the first 1024 bytes of code/data from the Parallel Flash and begin executing from location 0x10000004 in internal memory. It is assumed that this first 1024 bytes will contain code/data to complete boot of the remaining application code/data. A secondary boot loader is therefore required for any sizeable application that must boot from Parallel Flash.

A sample secondary bootloader is provided with the bootloader utilities, genBootCfg and genAIS. This sample code utilizes the pin configuration information generated by genBootCfg to properly set the GPIO pins to access extended FLASH addresses when loading application code/data to the DSP memory.

To choose GPIO pins to configure as address pins, select the check box from the main genBootCfg window, “Configure GPIO pins as address pins”. A configuration window for GPIO – to- Address Pin mapping will pop-up.



Address Pin #13	Address Pin #14	Address Pin #15	Address Pin #16	Address Pin #17
Not Mapped	Not Mapped	Not Mapped	Not Mapped	Not Mapped
Address Pin #18	Address Pin #19	Address Pin #20	Address Pin #21	Address Pin #22
Not Mapped	Not Mapped	Not Mapped	Not Mapped	Not Mapped
Address Pin #23	Address Pin #24			
Not Mapped	Not Mapped			

☐ Use Pin #13, as latch for upper address bits

Figure 25. GPIO to Address Pin Configuration Window

When the GUI is first invoked, all address pins are shown as “Not Mapped”. To configure a particular pin, simply click on the pull-down menu that corresponds to the address pin that needs to be mapped. To choose a GPIO pin, highlight the pin name in the list and then click to select.

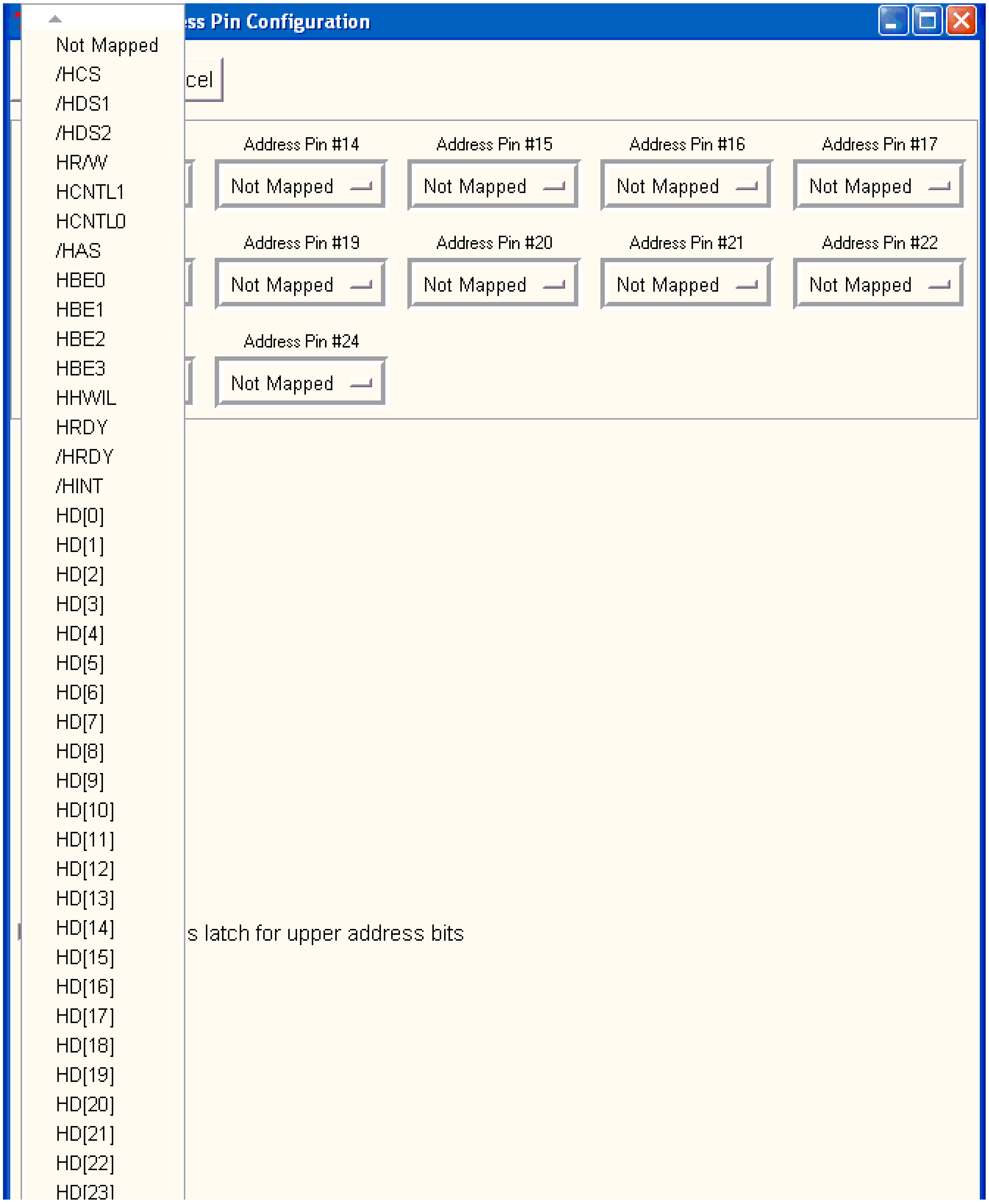
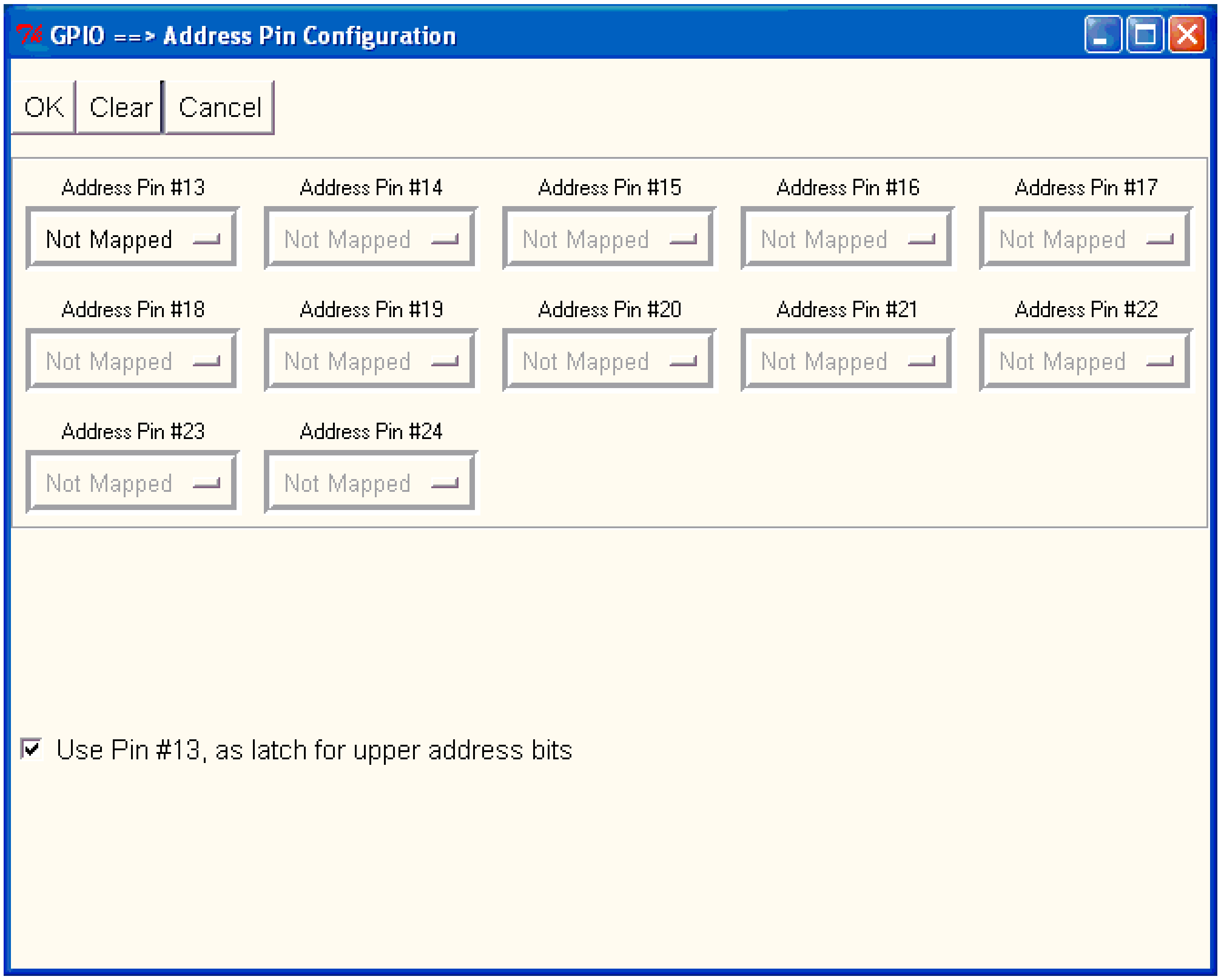


Figure 26. Mapping a GPIO Pin Using GPIO Pull Down Menu

All available GPIO pins are listed. Once a GPIO pin is selected to map to any of the address pins, the utility will not allow it to be mapped again. Any attempt to select the same pin twice, will result in the second attempt always showing the address pin as “Not Mapped”.

An alternate approach to defining a GPIO pin for each address pin in an extended range, is to use a single GPIO pin as a latch. The genBootCfg utility and the example secondary bootloader support this option as well. To select a single GPIO to use as an address latch, simply click in the check box labeled “Use Pin #13, as latch for upper address”. (For TQFP package this will read use Pin #12, instead of Pin #13). When this option is used, the utility will disable all pin mappings except one. Choose the GPIO pin needed, by highlighting and clicking it in the pull down menu.



Address Pin #13	Address Pin #14	Address Pin #15	Address Pin #16	Address Pin #17
Not Mapped	Not Mapped	Not Mapped	Not Mapped	Not Mapped
Address Pin #18	**Address Pin #19**	**Address Pin #20**	**Address Pin #21**	**Address Pin #22**
Not Mapped	Not Mapped	Not Mapped	Not Mapped	Not Mapped
Address Pin #23	**Address Pin #24**			
Not Mapped	Not Mapped			

[✓] Use Pin #13, as latch for upper address bits

Figure 27. Selecting GPIO Pin as Latch

To keep the pin configuration, click **OK**.

NOTE: The numbering of the pins as given, are in counted in relation to the DSP's EMIF and not from the perspective of any attached FLASH device. Each application must determine for the FLASH device used, which address pins must be connected from the FLASH to the given GPIO of the DSP to create the effective address.

6.2.8 Output Files

Once all configurations are complete, to save the settings go to the FILE pulldown menu and select SAVE. The utility will then create two files: “filename.cfg and filenamecfg.c”. The “.cfg” file contains the raw configuration information.

Example 1. “.cfg “file

```
##==================================================================## ## Boot Configuration File :
## ## Date: Tuesday June 21,2005 11:12:43 ##
##==================================================================##
#=================================================================== # PLL Configuration
#=================================================================== -pllCfg 0x1 -pllcfgosc
25.000000 -pllcfgcpu 0x0000012C -pllcfgemif 0x00000064 -pllcfgintosc 0x00000000 -pllcfgweight
0x00000003 -pllcfgpllm 0x00000018 -pllcfgdiv0 0x00000000 -pllcfgdiv1 0x00000001 -pllcfgdiv2
0x00000003 -pllcfgdiv3 0x00000005
#=================================================================== # SDRAM Configuration
#=================================================================== -sdramCfg 0
#=================================================================== # ASYNC Ram Configuration
#=================================================================== -asyncRamCfg 0x1 -asyncRamA1CR
0xBFFFFFFD -asyncRamAWCCR 0x10000080
```

6.2.8.2 TIBOOT Section and TIBootSetup Symbol

The “*cfg.c” file created by save, places all generated code within a named section called, “.TIBOOT”. The “TIBootSetup” function defined in the “*cfg.c” file is contained in this section, and is the entry point for all boot configuration for PLL, EMIF, etc. The genAIS tool looks for this section when –cfgtype “c” option is chosen. genAIS automatically places code/data for “.TIBOOT” as the very first section to load within the AIS data stream. Immediately following load of this section, genAIS places an AIS JUMP command with the address of the “TIBootSetup” function as target. This forces execution of the configuration code. Normal AIS processing will continue after “TIBootSetup” function has completed and control returns to the on-chip bootloader.

6.3 genAIS

genAIS is a Perl script that takes an application “.out” file as input and produces an AIS data stream as output. The utility supports creation of AIS data streams for I2C master/slave, SPI master/slave, and can also produce “raw” AIS output for application specific use, or a stream that can be used with a sample secondary boot loader that is included with this applications note. genAIS is a command line script. It can be invoked within a makefile or other batch mode utility. The script takes an input application “.out” file and produces an AIS data stream in either ASCII text format, raw binary, or C672x assembly using the assembler’s “.word” directive. A list of options is given in Table 13.

Table 13. genAIS options

Option	Description	Values
-ping (optional)	Specifies number of words to transmit when PINGing device in I2C or SPI slave modes	User specified decimal number

-cfgtype (optional)	Specified whether tool should generate AIS SET commands to initialize PLL, EMIF, etc for boot, or should assume cfg was built with application file, and use AIS JUMP command to execute the initialization code.	ais or c
		ais => SET command generated
		c => JUMP to initialization code from cfg.c is generated
		default => c

-i (required)	Input filename (*.out)	No default value, this is a required option
-o (optional)	Output filename	Default value is: inputfilename.asm if output type is ASM or, inputfilename.ais if output type is ASCII
-cfg (optional for SPI and I2C bootmodes, required for tisecboot bootmode)	Configuration Filename	name of configuration file created using genBootCfg utility (see Section 7 for description)
-crc (optional)	Request CRC be generated for section load verification	0 or 1 0 selects no crc 1 selects crc generation Default = 1
-otype (optional)	Selects output file type	asm or ascii asm **->** selects TMS320C672x assembly as output ascii **->** selects ASCII text output
-bootmode (required)	Selects boot mode	spimaster **->** selects SPI master 8/16 bit mode spislave **->** selects SPI slave 8/16 bit mode i2cmaster **->** selects I2C master 8/16 bit mode i2cslave **->** selects I2C slave 8/16 bit mode tisecboot **->** selects use of TI sample secondary bootloader (see Section 7 for description) Default **->** spislave raw -> Selects raw AIS data stream
-pf (required for tisecboot mode if memory type is 16 bit)	Specifies data word size in bits of Parallel Flash	8 **->** 8 bit Parallel Flash 16 **->** 16 bit Parallel Flash Default **->** 8 **NOTE:** This option is only valid when specifying tisecboot as bootmode.
-pkg (optional if default is acceptable or if -cfg option is being used)	Specifies device package type	bga **->** Ball Grid Array [GDH suffix] tqfp **->** Thin Quad Flatpack [RFP suffix] Default -> bga **NOTE:** Specifying this option is unnecessary when importing a configuration file with -cfg option.

A sample invocation of genAIS for SPI slave mode, generating ASCII output, is:

```
perl genAIS.pl -I myApplication.out -o myApplication.ais -bootmode spislave
```

6.3.1 -Bootmode option

The –bootmode option is used to select the correct AIS data stream for the physical boot mode selected for TMS320C672xx devices. A simple assembly input file, shown in the code below, will be used as the application source for displaying the AIS generation results for each optional bootmode. An example for “tisecboot” mode is included in a separate section.

```
;======================================= ; Sample Assembly Source File ; a = 6; ; while(1) { ; b
= a + 1; ; c = b + 2; ; } ; ;======================================= .global _a,_b,_c .sect
"myData" _a .word 0xA _b .word 0xB _c .word 0xC .text .global Start Start: MVKL .S1 _a,A3 MVKL
.S1 _c,A5 MVKL .S1 _b,A4 MVKH .S1 _a,A3 || MVK .S2 6,B4 STW .D1T2 B4,*A3 || MVKH .S1 _c,A5 MV
.L2X A3,B5 || MVKH .S1 _b,A4 loop: LDW .D2T2 *B5,B4 NOP 4 ADD .L2 1,B4,B4 STW .D1T2 B4,*A4 NOP 2
LDW .D1T1 *A4,A3 NOP 4 ADD .L1 2,A3,A3 STW .D1T1 A3,*A5 NOP 2 B .S1 loop NOP 5
```

This example was linked with the following linker MEMORY and SECTIONS directives

```
/****************************************************************************/ /* Specify the
Memory Configuration */
/****************************************************************************/ MEMORY
 { ROM : origin = 0x00001000 length = 0x000BF000
 VEC : origin = 0x10000000 length = 0x00000A00
 RAM : origin = 0x10001C00 length = 0x0003E400
 SDRAM : origin = 0x80000000 length = 0x08000000
 ASYNC2 : origin = 0x90000000 length = 0x00008000
 }
/****************************************************************************/
 /* Specify the Output Sections */
 /****************************************************************************/
SECTIONS
 .TIBoot: load = RAM
 .text: load = RAM
 myData: load = RAM
 .stack load = RAM
 .cinit load = RAM
 .cio load = RAM
 .const load = RAM
 .data load = RAM
 .switch load = RAM
 .far load = RAM
 .bss load = RAM
 .sysmem load = RAM
 .pinit load = RAM }
```

The next example shows the memory addresses for global symbols and sections defined in the example source.

```
GLOBAL SYMBOLS: SORTED BY Symbol Address address name -------- ---- 10001c00 ___end__ 10001c00
___edata__ 10001c00 ___data__ 10001c00 end 10001c00 edata 10001c00 ___text__ 10001c00 ___bss__
10001c00 .text 10001c00 .bss 10001c00 .data 10001c00 $bss 10001c00 Start 10001c60 etext 10001c60
_a 10001c60 ___etext__ 10001c64 _b 10001c68 _c ffffffff ___c_args__ ffffffff ___binit__ ffffffff
binit
```

The same code and linkage are used to illustrate output from each –bootmode option.

6.3.1.1 -bootmode i2cslave/spislave

The AIS data stream is exactly the same for “i2cslave” and “spislave” bootmodes. The data stream for I2C/SPI slave modes requires the host to send a transmit start word, followed by an AIS, PING_DEVICE command. The AIS data stream created when using –bootmode spislave or –bootmode i2cslave contains the transmit start word, and PING_DEVICE command as part of the stream. Please refer to Section 3 for details of PING_DEVICE command. The number of words transmitted as part of the PING_DEVICE command is configurable using the genAIS –ping option. The default value is 10.

Using AIS data stream was generated by the following invocation of genAIS tool:

```
genAis -I docExample.out -o docExample_spiSlave.ascii -bootmode spislave -otype ascii
```

Table 14. AIS Data Output

AIS Command	Data Value in File docExample_spiSlave.ascii
AIS Magic Number	0x41504954
XMT_START Word	0x00005853
PING_DEVICE Command	0x5853590B
Number of data words transmitted for ping	0x0000000A
First data word of ping command	0x00000001

Table 14. AIS Data Output (continued)

AIS Command	Data Value in File docExample_spiSlave.ascii
Second data word of ping command	0x00000002
	0x00000003
	0x00000004
	0x00000005
	0x00000006
	0x00000007
	0x00000008
	0x00000009
Last data word of ping command	0x0000000A
ENABLE_CRC Command	0x58535903
SECTION_LOAD Command (.text section)	0x58535901
Section Load Address	0x10001C00
Section size in 8-bit bytes	0x00000060
First 32 bit word of section data	0x018E3028
Second 32 bit word of section data	0x028E3428
	0x020E3228
	0x01880069
	0x0200032A
	0x020C0277
	0x02880068
	0x028C105B
	0x02080068
	0x021402E6
	0x00006000
	0x0210205A
	0x02100276
	0x00002000
	0x01900264
	0x00006000
	0x018C4058
	0x01940274
	0x00002000
	0x0FFFFC90
	0x00008000
	0x00000000
	0x00000000
Last word of section data	0x00000000
REQUEST_CRC Command	0x58535902
Expected CRC value for this section	0xB0EC107D
Offset to last valid AIS command in stream **NOTE:** This offset is used to reposition stream to last command if re-transmission is attempted due to CRC error.	0xFFFFFF88
SECTION_LOAD Command (myData section0	0x58535901
Section load address	0x10001C60

Table 14. AIS Data Output (continued)

AIS Command	Data Value in File docExample_spiSlave.ascii
Section size in 8bit bytes	0x0000000C
First 32bit word of section data	0x0000000A
	0x0000000B
Last 32bit word of section data	0x0000000C
REQUEST_CRC Command	0x58535902
Expected CRC value for this section	0xBBE311D7
Offset to last valid AIS command in stream **NOTE:** This offset is used to reposition stream to last command if re-transmission is attempted due to CRC error)	0xFFFFFFDC
JUMP_CLOSE Command	0x58535906
Start address of application code	0x10001C00

NOTE: Please note that although the stream contains the AIS magic number as the first word of the file, this data is not transmitted by the MASTER. The first data word transmitted by the host should be the XMT_START instruction, to begin MASTER-to-DSP handshake.

After transmitting the XMT_START word, the host should wait to receive acknowledgment from the DSP. If the XMT_START is received correctly, the DSP will respond with appropriate RECV_START word. For I2C/SPI slave modes, this values is 0x5253. After the host receives the RECV_START, then continue transmission with the PING_DEVICE command and its associated data.

When the DSP receives the PING_DEVICE command, it will read the first data word of the command. This word tells the DSP how many data words to expect as part of the ping command sequence. The DSP sends this same data word back to the MASTER in acknowledgment that it has received the PING_DEVICE command. The MASTER then proceeds to send the remaining data words to complete PING.

The MASTER device then transmits any valid sequence of AIS commands and data until the JUMP_CLOSE command is sent.

When the DSP encounters a REQUEST_CRC in the AIS stream, it will send to the MASTER device the current CRC value it has calculated for the loaded code/data. If this matches the CRC value the MASTER sends as part of the REQUEST_CRC command, no further action is required, and the DSP wait to process the next AIS command sent from the MASTER device. However, if the CRC value is in error, the HOST has the option of terminating the boot process or attempting re-transmission of last command/data. The second word of the REQUEST_CRC command contains a signed offset that points to the last valid AIS command previously encountered in the AIS data stream. To retry transmission, the MASTER adjusts the current data stream pointer by the amount specified in the REQUEST_CRC command. The repositions the data stream and the MASTER begins transmission of AIS commands/data from this point.

For example of software that implements the MASTER side of I2C/SPI slave mode using AIS, please see the example: TMS320C672xxBootUtils\Examples\generic\spiSlave that is included with the .zip file attached to this application note. The example code was run using 6713DSK as the MASTER device and the TMS320C672xx present on the PADK as SPI slave. The connections used for SPI communication were as follows:

Using McBSP1 of TMS320C6713 connected to SPI0 of TMS320C6727, connect following pins:

TMS320C6713 McBSP1		TMS320C6727 SPI0
DR1	→	SPI0_SOMI
DX1	→	SPI0_SIMO
CLKX1	→	SPIO_CLK
FSR1	→	SIO0_SCSn

6.3.1.2 -bootmode i2cmaster/spimaster

In this bootmode the TMS320672xx acts as the MASTER SPI device. The AIS data stream is exactly the same for both i2cmaster and spimaster modes. Table 15 presents the AIS data stream for example.asm, when i2cmaster bootmode is selected:

```
genAis -I docExample.out -o docExample_i2cMaster.ascii -bootmode i2cmaster -otype ascii
```

Table 15. AIS Data Output in file docExample_i2cMaster.ascii

AIS Command	Data From file docExample_i2cMaster.ascii
AIS Magic Word	0x41504954
REQUEST_CRC Command	0x58535903
SECTION_LOAD Command (.text section)	0x58535901
Section load address	0x10001C00
Section size in 8bit bytes	0x00000060
First 32bit word of section data	0x018E3028
Second 32 bit word of section data	0x028E3428
	0x020E3228
	0x01880069
	0x0200032A
	0x020C0277
	0x02880068
	0x028C105B
	0x02080068
	0x021402E6
	0x00006000
	0x0210205A
	0x02100276
	0x00002000
	0x01900264
	0x00006000
	0x018C4058
	0x01940274
	0x00002000
	0x0FFFFC90
	0x00008000
	0x00000000
	0x00000000
Last 32 bit word of section data	0x00000000

Table 15. AIS Data Output in file docExample_i2cMaster.ascii (continued)

AIS Command	Data From file docExample_i2cMaster.ascii
REQUEST_CRC Command	0x58535902
Expected CRC value	0xB0EC107D
Offset to last valid AIS command in stream **NOTE:** This offset is used to reposition stream to last command if re-transmission is attempted due to CRC error)	0xFFFFFF88
SECTION_LOAD Command	0x58535901
Section load address	0x10001C60
Section size in 8bit bytes	0x0000000C
First 32bit word of section data	0x0000000A
	0x0000000B
Last 32bit word of section data	0x0000000C
REQUEST_CRC Command	0x58535902
Expected CRC value	0xBBE311D7
Offset to last valid AIS command in stream **NOTE:** This offset is used to reposition stream to last command if re-transmission is attempted due to CRC error	0xFFFFFFDC
JUMP_CLOSE command	0x58535906
Start address of application code	0x10001C00

When the DSP is the MASTER I2C/SPI device, it reads the first word of the AIS stream and expects to find the AIS magic word. Therefore, this MUST be the first word burned into the EEPROM when DSP is Master for either I2C or SPI boot modes. If the DSP reads the AIS magic word, it will then begin to process all subsequent AIS commands/data in the stream, until JUMP_CLOSE command is encountered. Once JUMP_CLOSE command is processed the DSP branches to the address given in the instruction to begin execution of application.

If a CRC error is encountered when processing the REQUEST_CRC command, the DSP will adjust the address pointer to point to last valid AIS command read from EEPROM device to re-try fetch of data. The DSP currently will try to process a section load twice before aborting the boot process.

6.3.1.3 -bootmode raw

The “raw” bootmode option produces a simple AIS data stream that could be used when implementing a customized secondary boot loader for EMIFA/FLASH boot or as input to HOST software that implements UHPI boot. The data stream produced by genAIS tool is same as for I2C/SPI master boot mode.

6.3.2 -cfgtype option

The –cfgtype option is used to select configuration of PLL, EMIF ,etc via AIS SET Commands or to create a JUMP to compiled code that is linked with the application for that purpose. When using the –cfgtype “ais” option, the genAIS tool creates entries in the AIS data stream for SET commands that will effect peripheral configuration. The –cfg option MUST be used in conjunction with option –cfgtype “ais”.

6.3.2.1 -cfgtype "ais"

This option depends on the input from a “*.cfg” file created by prior invocation of the genBootCfg utility. Use genBootCfg to create the configuration data necessary to support your board configuration for boot. Once this has been done, then invoke genAIS utility using –cfg and –cfgtype options to produce the AIS data stream.

The code below shows a sample “*.cfg” file that has EMIF configuration for ASYNC RAM.

```
##=====================================================================## ## Boot Configuration
File : ## ## C:/Lyrtech/PADK/dsp/demos/TMS320C672xxBootUtils/Examples/gener ## ## docExample.cfg
## ## Date: Friday October 21,2005 10: 6: 9 ##
##=====================================================================##
#===================================================================== # PLL Configuration
#===================================================================== -pllCfg 0x0
#===================================================================== # I2C Clock Configuration
#===================================================================== -i2cClkCfg 0
#===================================================================== # SDRAM Configuration
#===================================================================== -sdramCfg 0
#===================================================================== # ASYNC Ram Configuration
#===================================================================== -asyncRamCfg 0x1 -
asyncRamA1CR 0x1DF6EFFD -asyncRamAWCCR 0x10000080
```

An AIS stream containing the SET commands may be generated with the following example invocation of genAIS:

```
genAis -I docExample.out -o docExample_cfgTypeAis.ascii -bootmode spimaster -otype ascii -cfgtype
ais -cfg docExample.cfg
```

Table 16. AIS Data Stream From File, docExample_cfgTypeAis.ascii

AIS Magic Word	0x41504954
SET Command	0x58535907
Byte address of memory location to modify	0xF0000004
Data value to be written at specified address	0x10000080
Data type - (0x2 → 32bit data)	0x00000002
Number of cycles to wait after data is written	0x00000000
SET Command	0x58535907
Byte address of memory location to modify	0xF0000010
Data value to be written at specified address	0x1DF6EFFD
Data type - (0x2 → 32bit data)	0x00000002
Number of cycles to wait after data is written	0x00000000
ENABLE_CRC Command	0x58535903
SECTION_LOAD Command	0x58535901
Section load address	0x10001C00
Section size in 8bit bytes	0x00000060
First 32bit word of section data	0x018E3028
Second 32bit word of section data	0x028E3428
	0x020E3228
	0x01880069
	0x0200032A
	0x020C0277
	0x02880068
	0x028C105B
	0x02080068
	0x021402E6
	0x00006000
	0x0210205A
	0x02100276
	0x00002000

Table 16. AIS Data Stream From File, docExample_cfgTypeAis.ascii (continued)

	0x01900264
	0x00006000
	0x018C4058
	0x01940274
	0x00002000
	0x0FFFFC90
	0x00008000
	0x00000000
	0x00000000
Last 32bit word of section data	0x00000000
REQUEST_CRC Command	0x58535902
Expected CRC value for this section	0xB0EC107D
Offset to last valid AIS command in stream **NOTE:** This offset is used to reposition stream to last command if re-transmission is attempted due to CRC error	0xFFFFFF88
SECTION_LOAD Command	0x58535901
Section load address	0x10001C60
Section size in 8bit bytes	0x0000000C
First 32bit word of section data	0x0000000A
	0x0000000B
Last 32bit word of section data	0x0000000C
REQUEST_CRC Command	0x58535902
Expected CRC value for this section	0xBBE311D7
Offset to last valid AIS command in stream **NOTE:** This offset is used to reposition stream to last command if re-transmission is attempted due to CRC error.	0xFFFFFFDC
JUMP_CLOSE Command	0x58535906
Start address of application code	0x10001C00

7 Boot Examples

This applications note contains attached code for several boot examples. The examples cover creation of secondary bootloader for FLASH/EMIFA boot, generic examples for producing AIS data in a format compatible with EEPROM programmers, and several examples that were created for use with TMS320C672xx Performance Audio Development Kit (PADK) available from Lyrtech, Inc. All source code and data for the examples maybe found in the attachment to this document available at this link:

http://www-s.ti.com/sc/techlit/sprc203.zip.

The secondary bootloader is easy to use and simply links with your application code. The resulting executable file is processed by the genAIS utility. The output of genAIS is then burned into the FLASH memory. When the C672xx is released from reset, the on-chip bootloader will copy the first 1024 bytes of the secondary bootloader to memory. The secondary bootloader will then load the rest of it's own code, plus the remaining application code. It will then branch to the start of application code for execution. The secondary bootloader works by processing the application code as an AIS stream. The genAIS utility converts the combined application + secondary bootloader output file into a specialized AIS stream. The secondary bootloader code is not converted to AIS. It is burned to flash as raw data. In Parallel FLASH mode the on-chip bootloader loads the first 1024 bytes of the secondary bootloader which contains the boot strap code necessary to complete load of secondary boot. The secondary boot loader than processes the rest of the data in the Parallel FLASH as a modified AIS data stream. This modified AIS data stream contains the application code and configuration data ,such as PLL and EMIF configurations required during boot. Three files are provided with this application's note to implement the secondary bootloader.

- TISecondaryBoot.h – header file containing C type and constant definitions to bootloader code
- TISecondaryBoot.c/TISecondaryBoot.obj – main source for secondary bootloader.
- TIsecondaryBootLnk.cmd – linker command file to use when linking code for secondary boot.

NOTE: The secondary bootloader assumes GPIO's are used as either address pins or as a single GPIO for latch.

AIS was extended to include two new commands to facilitate paging of extended addresses for EMIF boot:

- `PAGE_SWAP_COMMAND` – takes a list of GPIO pin configurations needed to access the next page of memory in the FLASH. This is an atomic command and is the last command on issued on the page to ensure that the address pins are set correctly to access the first word/byte of the next memory page. Once this command is complete, the very next fetch from the AIS stream will be to the next page of FLASH memory.
- `LATCH_ADDRESS_COMMAND` – takes a GPIO pin configuration needed to effectively latch the address to enable switch of parallel FLASH memory page. This is also an atomic command which is issued as the last command in the current memory page. Once execute, AIS commands are fetched beginning with the first word/byte of the next memory page in FLASH.

When a GPIO is used as a latch, the last word of the LATCH_ADDRESS_COMMAND contains a memory address to present on the EMIF bus that will latch the specified page of FLASH memory. Currently the page is specified in the lower 16-bits of the address. Table 17 lists the expected address latch value for the first six pages of a 16-bit Parallel FLASH.

Table 17. Address Range to Page Latch Address Mapping for 16-bit FLASH

EMIF Address Range	Corresponding Latch Address
0x90000000-0x90007FFF	0x900000000
0x90008000-0x9000FFFF	0x900000004
0x90010000-0x90017FFF	0x900000008
0x90018000-0x9001FFFF	0x90000000C
0x90020000-0x90027FFF	0x900000010
0x90028000-0x9002FFFF	0x900000014
0x90030000-0x90037FFF	0x900000018

7.1 Building an Application Using an Example Secondary Bootloader

The example secondary bootloader included with this application's note, is intended for use with Parallel Flash bootload only. It extends the address range for Parallel FLASH boot, by using GPIO pins as address pins. The GPIO **->** address pin mapping is configurable using the genBootCfg bootloader utility described earlier in this chapter. GPIO pins from any of UHPI (BGA only), or MCASP0/1, can be mapped as address pins. Up to 11 pins may be defined. Alternatively, the address range could also be expanded by defining a single GPIO to act as a latch for the upper address. In this current implementation, the page selection for latched addresses uses the lower bits of the address presented on the EMIF address bus to specify memory page. This sample secondary bootloader requires the use of both genBootCfg and genAIS utilities. Three files are provided with this application's note to implement the secondary bootloader.

To build and use the example secondary bootloader follow these steps:

1. Invoke genBootCfg utility and configure PLL, and EMIF ASYNC RAM interface as needed for boot. If extending the address range for Parallel Flash, GPIO pins used for address pin extension must also be configured.
2. Link the application code, the “cfg.c” file, and TISecondaryBoot.obj using the TISecondaryBootLnk.cmd file.
3. Invoke genAIS with *.out file from the link in step 2, and setting –bootmode tisecboot., setting Parallel Flash size to 8 or 16 as appropriate, and including the “.cfg” file from genBootCfg. (i.e. genAIS –I my.out –cfg my.cfg –o my.ascii –otype ascii –pf 16 –bootmode tisecboot)
4. Burn the resulting data output from genAIS to Flash.

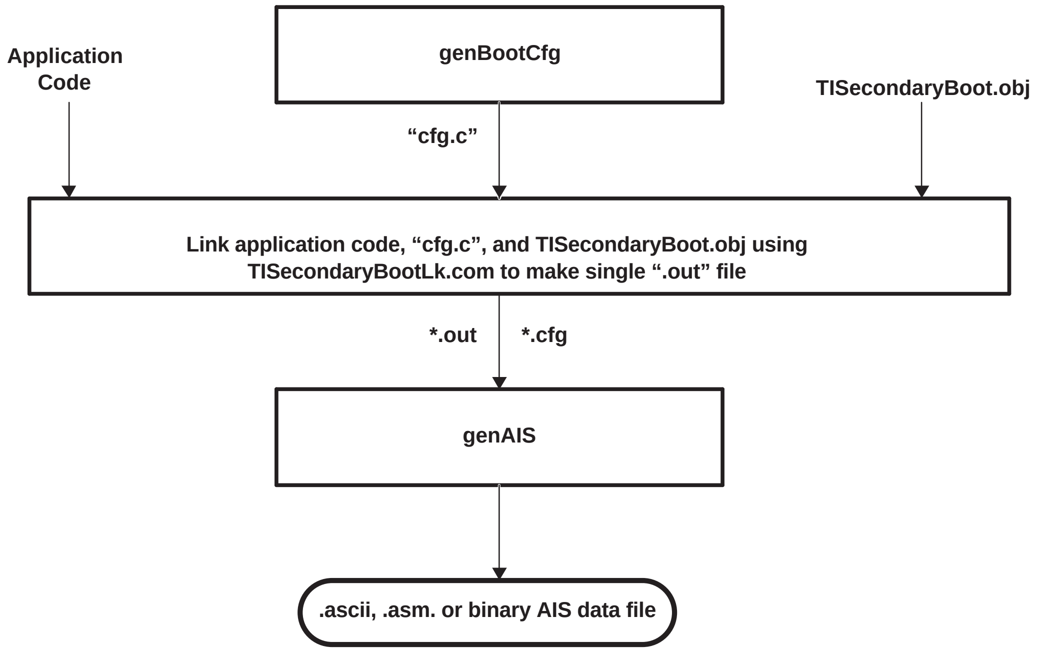

Application Code
genBootCfg
TISecondaryBoot.obj
“cfg.c”
Link application code, “cfg.c”, and TISecondaryBoot.obj using TISecondaryBootLk.com to make single “.out” file
*.out
*.cfg
genAIS
.ascii, .asm. or binary AIS data file

Figure 28. Build Flow

Figure 28 shows the build flow for the example secondary bootloader.

The output from genAIS is stored in the file as either ascii, assembly, or binary data. The contents of the output file is a specialized AIS data stream formulated specifically for the needs of the secondary bootloader. It is NOT a generalized format that can be used for any other boot method. If booting for SPI/I2C boot, choose the appropriate boot mode. The secondary boot loader extends the AIS languages to include two new AIS commands. These commands enable page swapping when using GPIO pins as address pins, or when using a single GPIO pin as an address latch.

The contents of the file should be burned to FLASH beginning at FLASH address 0x00000000.

7.2 Sample Projects using Secondary Bootloader

Two generic examples are included with the software linked with this document, gpioAsAddr and gpioAsLatch. This example code and corresponding linker command files are in some respects atypical of a normal application. These are designed specifically to illustrate the capabilities for page swap, so the code was artificially packed with fill space to force crossing of address page boundaries. The linker command file has been annotated to indicate which sections are needed for application code, and which are there simply for the purpose of forcing paging for the example. The example code includes the following files:

Example code for illustrating use of GPIO pins mapped as address pins to extend the EMIF A address range can be found in the folder TMS320C672xxBootUtils\generic\gpio.

The example includes the following files:

```
genBootCfgFile.bat - batch file used to invoke genBootCfg
 genAISFile,bat - batch file used to invoke genAIS tool
 appMain.c - application main .c file
 fill1space.asm - C6000 assembly file, defines first fill section
 fill2space.asm - C6000 assembly file, defines second fill section
 fill3space.asm - C6000 assembly file, defines third fill section
 gpioAsAddr - File folder containing project files
 TISecondaryBootLnk.cmd - linker command file
 gpioAsAddr.ais - AIS data stream generated by genAIS utility from gpioAsAddr.out file
 gpioAsAddrcfg.c - C configuration file generated by genBootCfg utility
 gpioAsAddr.cfg - configuration data file generated by genBootCfg utility
```

The same source for application code is used to illustrate using a GPIO pin as an address latch, to latch upper bits of the address to extend EMIFA addressing range. The source file for this example can be found in folder TMS320C672xxBootUtils\generic\latch.

The example includes the following files:

```
genBootCfgFile.bat - batch file used to invoke genBootCfg
 genAISFile,bat - batch file used to invoke genAIS tool
 appMain.c - application main .c file
 fill1space.asm - C6000 assembly file, defines first fill section
 fill2space.asm - C6000 assembly file, defines second fill section
 fill3space.asm - C6000 assembly file, defines third fill section
 gpioAsLatch - File folder containing project files
 TISecondaryBootLnk.cmd - linker command file
 gpioAsLatch.ais - AIS data stream generated by genAIS utility from gpioAsAddr.out file
 gpioAsLatchcfg.c - C configuration file generated by genBootCfg utility
 gpioAsLatch.cfg - configuration data file generated by genBootCfg utility
```

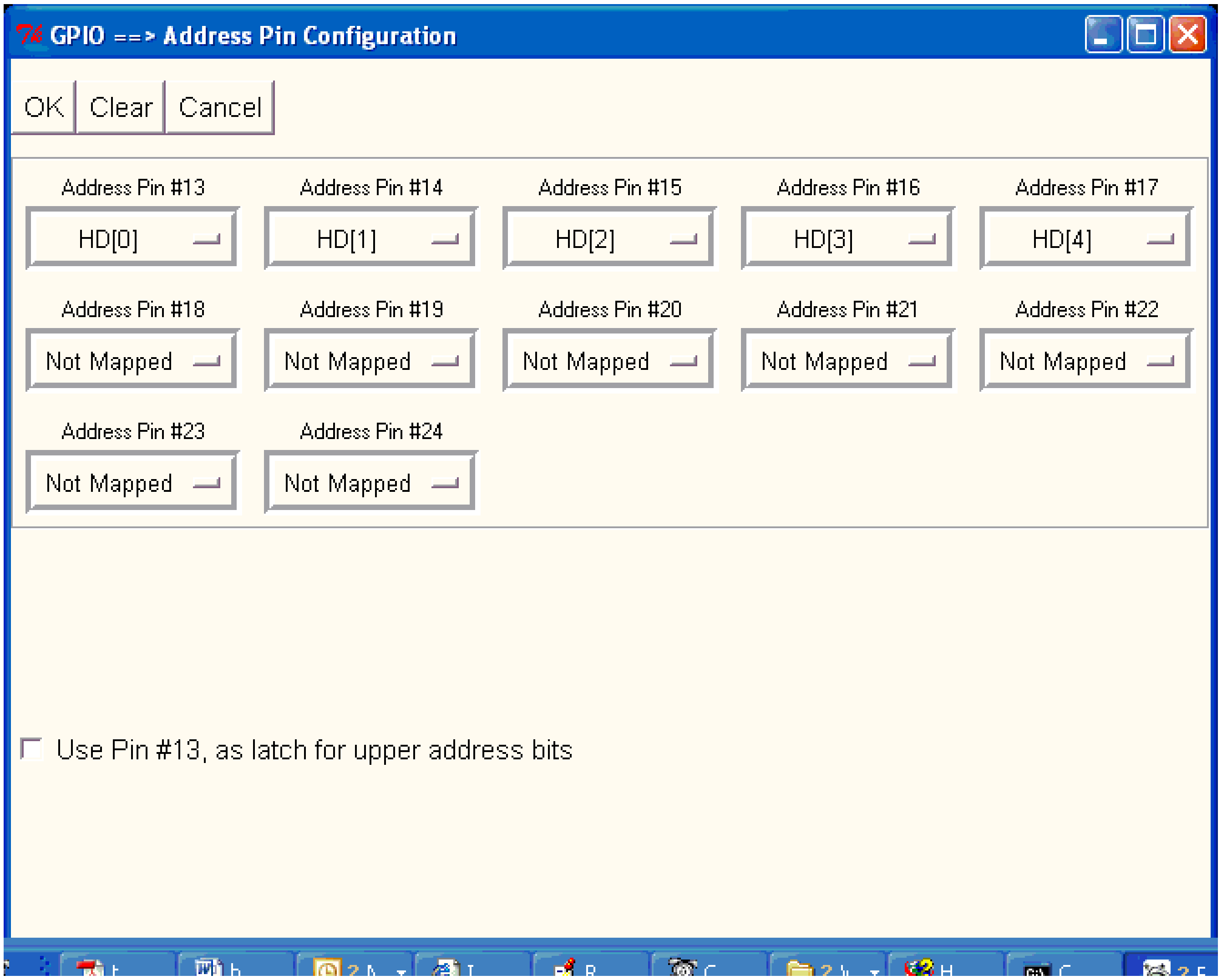
WARNING

Please note that when building/rebuilding the sources for this project, that -ml3 option is required for the compile. The code will not rebuild without this option, since the distance between called functions exceeds the limit for relative branches.

7.2.1 Example 1 - GPIO Pins Mapped as Address Pins (gpioAsAddr)

To map GPIO pins as address pins, first invoke the genBootCfg utility to setup the GPIO pin-to-address pin map. In this example, the UHPI data pins, HD[0-4], have been mapped as address pins and ASYNC RAM has been configured for 16 bit memory.

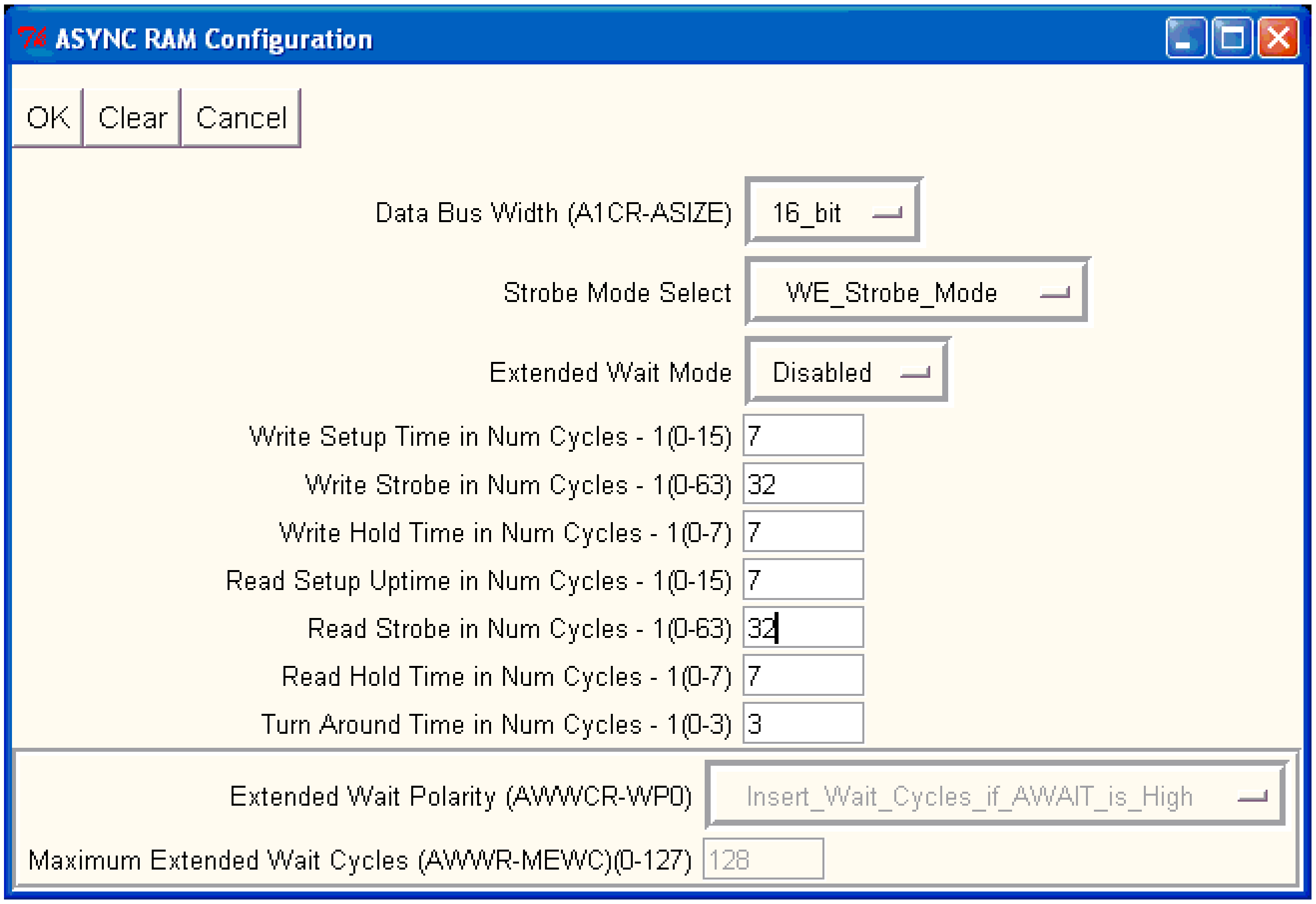
Figure 29 shows pin configuration page of genBootCfg utility. Figure 30 shows the ASYNC RAM setup page in genBootCfg. The resulting configuration file, gpioAsAddr.cfg is displayed in Example 2, and the accompanying cfg.c file in Example 3.



Address Pin #13	Address Pin #14	Address Pin #15	Address Pin #16	Address Pin #17
HD[0]	HD[1]	HD[2]	HD[3]	HD[4]
Address Pin #18	Address Pin #19	Address Pin #20	Address Pin #21	Address Pin #22
Not Mapped	Not Mapped	Not Mapped	Not Mapped	Not Mapped
Address Pin #23	Address Pin #24			
Not Mapped	Not Mapped			

☐ Use Pin #13, as latch for upper address bits

Figure 29. GPIO Pins



Data Bus Width (A1CR-ASIZE) 16_bit

Strobe Mode Select WE_Strobe_Mode

Extended Wait Mode Disabled

Write Setup Time in Num Cycles - 1(0-15) 7

Write Strobe in Num Cycles - 1(0-63) 32

Write Hold Time in Num Cycles - 1(0-7) 7

Read Setup Uptime in Num Cycles - 1(0-15) 7

Read Strobe in Num Cycles - 1(0-63) 32

Read Hold Time in Num Cycles - 1(0-7) 7

Turn Around Time in Num Cycles - 1(0-3) 3

Extended Wait Polarity (AWWCR-WP0) Insert_Wait_Cycles_if_AWAIT_is_High

Maximum Extended Wait Cycles (AWWR-MEWC)(0-127) 128

Figure 30. ASYNC RAM Setup

Example 2. gpioAsAddr.cfg file

```
##====================================================================## ## Boot Configuration File :
## ## C:/Lyrtech/PADK/dsp/demos/TMS320C672xxBootUtils/Examples/gener ## ## gpioAsAddr.cfg ## ## Date:
Wednesday September 7 ,2005 13:22:46 ##
##====================================================================##
#==================================================================== # PLL Configuration
#==================================================================== -pllCfg 0x1 -pllcfgosc
25.000000 -pllcfgcpu 0x0000012C -pllcfgemif 0x00000064 -pllcfgintosc 0x00000001 -pllcfgweight
0x00000003 -pllcfgpllm 0x00000018 -pllcfgdiv0 0x00000000 -pllcfgdiv1 0x00000001 -pllcfgdiv2
0x00000003 -pllcfgdiv3 0x00000005
#==================================================================== # I2C Clock Configuration
#==================================================================== -i2cClkCfg 0
#==================================================================== # SDRAM Configuration
#==================================================================== -sdramCfg 0
#==================================================================== # ASYNC Ram Configuration
#==================================================================== -asyncRamCfg 0x1 -asyncRamA1CR
0x9E0EE1FD -asyncRamAWCCR 0x10000080
#==================================================================== # Address Pin Configuration
#==================================================================== -pinCfg 0x1 -useAddressLatch
0x00000000 -pin0 HD[0] -pin1 HD[1] -pin2 HD[2] -pin3 HD[3] -pin4 HD[4] -pincount 5 -pinEnableReg0
0x4300000C -pinEnableMask0 0x00000080 -pinEnableMaskMode0 0x00000001 -pinDirectionReg0 0x43000010 -
pinDirectionMask0 0x00000001 -pinDirectionMaskMode0 0x00000001 -pinSetReg0 0x43000014 -pinSetMask0
0x00000001 -pinSetMode0 0x00000001 -pinClearReg0 0x43000014 -pinClearMask0 0xFFFFFFFE -pinClearMode0
0x00000002 -pinDisableReg0 0x4300000C -pinDisableMask0 0x00000000 -pinDisableMaskMode0 0x00000002 -
pinGlobalSetupFlag0 0x00000000 -pinEnableReg1 0x4300000C -pinEnableMask1 0x00000080 -
pinEnableMaskMode1 0x00000001 -pinDirectionReg1 0x43000010 -pinDirectionMask1 0x00000002 -
pinDirectionMaskMode1 0x00000001 -pinSetReg1 0x43000014 -pinSetMask1 0x00000002 -pinSetMode1
0x00000001 -pinClearReg1 0x43000014 -pinClearMask1 0xFFFFFFFD -pinClearMode1 0x00000002 -
```

Example 2. (continued)

```
pinDisableReg1 0x4300000C -pinDisableMask1 0x00000000 -pinDisableMaskMode1 0x00000002 -
pinGlobalSetupFlag1 0x00000000 -pinEnableReg2 0x4300000C -pinEnableMask2 0x00000080 -
pinEnableMaskMode2 0x00000001 -pinDirectionReg2 0x43000010 -pinDirectionMask2 0x00000004 -
pinDirectionMaskMode2 0x00000001 -pinSetReg2 0x43000014 -pinSetMask2 0x00000004 -pinSetMode2
0x00000001 -pinClearReg2 0x43000014 -pinClearMask2 0xFFFFFFFB -pinClearMode2 0x00000002 -
pinDisableReg2 0x4300000C -pinDisableMask2 0x00000000 -pinDisableMaskMode2 0x00000002 -
pinGlobalSetupFlag2 0x00000000 -pinEnableReg3 0x4300000C -pinEnableMask3 0x00000080 -
pinEnableMaskMode3 0x00000001 -pinDirectionReg3 0x43000010 -pinDirectionMask3 0x00000008 -
pinDirectionMaskMode3 0x00000001 -pinSetReg3 0x43000014 -pinSetMask3 0x00000008 -pinSetMode3
0x00000001 -pinClearReg3 0x43000014 -pinClearMask3 0xFFFFFFF7 -pinClearMode3 0x00000002 -
pinDisableReg3 0x4300000C -pinDisableMask3 0x00000000 -pinDisableMaskMode3 0x00000002 -
pinGlobalSetupFlag3 0x00000000 -pinEnableReg4 0x4300000C -pinEnableMask4 0x00000080 -
pinEnableMaskMode4 0x00000001 -pinDirectionReg4 0x43000010 -pinDirectionMask4 0x00000010 -
pinDirectionMaskMode4 0x00000001 -pinSetReg4 0x43000014 -pinSetMask4 0x00000010 -pinSetMode4
0x00000001 -pinClearReg4 0x43000014 -pinClearMask4 0xFFFFFFEF -pinClearMode4 0x00000002 -
pinDisableReg4 0x4300000C -pinDisableMask4 0x00000000 -pinDisableMaskMode4 0x00000002 -
pinGlobalSetupFlag4 0x00000000 #=================================================================
# Address Pin Configuration #============================================================ -
pkgType BGA
```

Example 3. cfg.c file

```
//===================================================================// // Boot Configuration File :
// // C:/Lyrtech/PADK/dsp/demos/TMS320C672xxBootUtils/Examples/gener // // gpioAsAddrcfg.c // //
Date: Wednesday September 7 ,2005 13:22:46 //
//===================================================================// #include <TISecondaryBoot.h>
//===================================================================// // Function Prototypes //
//===================================================================// far void TIBootPllCfg(void);
far void TIBootAsyncRamCfg(void);
//===================================================================// // Boot Configuration Setup
Function // // This code along with PLL Configuration, SDRAM Configuration // // ASYC RAM
Configuration, I2C Clock Configuration, will be // // loaded first. A branch will then be executed
after load of // // all boot configuration code to the bootSetup function. // // After the boot setup
has been performed, normal AIS // // processing will continue at that point. //
//===================================================================// #pragma
CODE_SECTION(TIBootSetup,".TIBoot") void TIBootSetup(void) { TIBootPllCfg(); TIBootAsyncRamCfg(); }
//===================================================================// // PLL Configuration // //
Input Oscillator Frequency: 25.00 // // Cpu Clock Frequency : 300.00 // // Emif Clock Frequency :
100.00 // // PLLM : 24 // // DIV0 : 0 // // DIV1 : 1 // // DIV2 : 3 // // DIV3 : 5 //
//===================================================================// #pragma
CODE_SECTION(TIBootPllCfg, ".TIBoot") far void TIBootPllCfg(void) { int I; // configure the PLL // //
Make sure SDRAM is in Self-Refresh mode before setting PLL // // By setting SR bit in EMIF SDCR
register to 1 // *(unsigned char *)TIBOOT_EMIF_SDCR = 0x8; // 1. In PLLCSR, write PLLEN = 0 (bypass
mode) // *(volatile unsigned int *)TIBOOT_PLL_PLLCSR = TIBOOT_PLLDISABLE; // 2. Wait 4 cycles of the
slowest of PLLOUT or reference// clock source (CLKIN or OSCIN) asm(" nop 4"); // 3. In PLLCSR, write
PLLRST = 1 (PLL is reset) *(volatile unsigned int *)TIBOOT_PLL_PLLCSR = TIBOOT_PLLDISABLE |
TIBOOT_PLLRESET; // 4. If necessary, program PLLDIV0 and PLLM // DIV0 - Before PLL(set to/1)
*(volatile unsigned int *)TIBOOT_PLL_PLLDIV0 = TIBOOT_DIVENABLED | 0x00000000; *(volatile unsigned
int *)TIBOOT_PLL_PLLM = 0x00000018; // 5. If necessary, program PLLDIV1-n. Note that you must apply
the GO operation to // change these dividers to new ratios. // DIV1 - After PLL- SYSCLK1 DSP Core //
DIV2 - After PLL- SYSCLK2 PERIPHS (Always twice DIV3) // DIV3 - After PLL- SYSCLK3 EMIF CLOCK
*(volatile unsigned int *)TIBOOT_PLL_PLLDIV1 = TIBOOT_DIVENABLED | 0x00000001; asm(" nop 4"); asm("
nop 4"); *(volatile unsigned int *)TIBOOT_PLL_PLLDIV2 = TIBOOT_DIVENABLED | 0x00000003; asm(" nop
4"); asm(" nop 4"); *(volatile unsigned int *)TIBOOT_PLL_PLLDIV3 = TIBOOT_DIVENABLED | 0x00000005; //
Enable PLL Align control. *(volatile unsigned int *)TIBOOT_PLL_PLLALNCTL = TIBOOT_PLLALN1 |
TIBOOT_PLLALN2 | TIBOOT_PLLALN3; *(volatile unsigned int *)TIBOOT_PLL_PLLCMD = TIBOOT_PLLGOSET; while
(*(volatile unsigned int *)TIBOOT_PLL_PLLSTAT == TIBOOT_PLLGOWAIT){ *(volatile unsigned int
*)TIBOOT_PLL_PLLCMD = TIBOOT_PLLGOCLR; } // 6. Wait for PLL to properly reset // Reset wait time is
125 ns for(I=0; I< 8;++I) {}; // 7. In PLLCSR, write PLLRST = 0 to bring PLL out of reset *(volatile
unsigned int *)TIBOOT_PLL_PLLCSR = TIBOOT_PLLDISABLE | TIBOOT_PLLRESETRELEASE; // 8. Wait for PLL to
lock for(I=0; I< 4787;++I) {}; // 9. In PLLCSR, write PLLEN = 1 to enable PLL mode *(volatile
unsigned int *)TIBOOT_PLL_PLLCSR = TIBOOT_PLLENABLE | TIBOOT_PLLRESETRELEASE; for(I=0; I < 4787; ++I)
{}; // 10. Wait for Lock bit to become 1 while ( ((*(volatile unsigned int *)TIBOOT_PLL_PLLCSR) &
TIBOOT_PLLLOCKED) == 0 ) { } // ---- done PLL Programation ---- // take CFG bridge out of reset
*(volatile unsigned int *)TIBOOT_CFGBRIDGE_REGISTER |= 1; asm(" nop 9"); *(volatile unsigned int
*)TIBOOT_CFGBRIDGE_REGISTER &= 0xFFFFFFFE; // Make sure SDRAM exits Self-Refresh Mode // By setting
```

(continued)

```
SR bit in EMIF SDCR register to 0 *(volatile unsigned char *)TIBOOT_EMIF_SDCR = 0x0; }
//=================================================================// // ASYNC Ram Configuration //
// Asynchronous 1 Configuration Register: // // register mask: 0x9E0EE1FD // // SS: 1 // // EW: 0 //
// W_SETUP: 7 // // W_STROBE: 32 // // W_HOLD: 7 // // R_SETUP: 7 // // R_STROBE: 0 // // R_HOLD: 7
// // TA: 3 // // ASIZE: 1 // // // // Asynchronous Wait Cycle Configuration Register // // register
mask: 0x10000080 // // WP0: 1 // // MEWC: 128 //
//========================================================
==========// #pragma CODE_SECTION(TIBootAsyncRamCfg,".TIBoot") void TIBootAsyncRamCfg() { *(volatile
unsigned int *)TIBOOT_EMIF_AWCCR = 0x10000080u; *(volatile unsigned int *)TIBOOT_EMIF_A1CR =
0x9E0EE1FDu; }
```

Once the configuration files have been created, compile and link the configuration files, the secondary boot loader and your application code together into a single output file (*.out). as is illustrated in the project below.

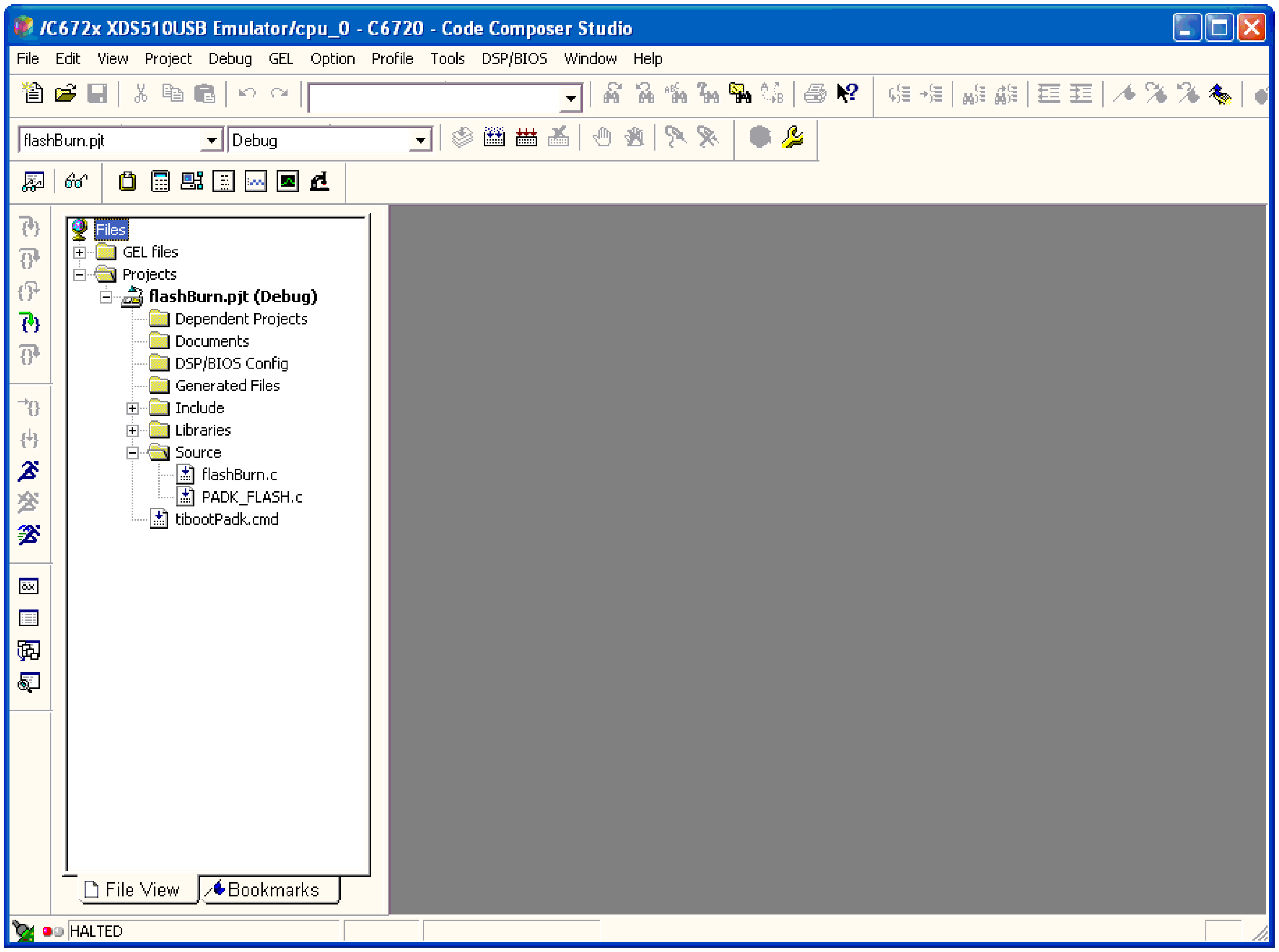


Figure 31. Project File

After the project has been built, the resulting ".out" file from the link and the ".cfg" file output from genBootCfg should be passed into the genAIS utility to transform into a specialized AIS stream. i.e. `genAis -I gpioAsAddr.out -o gpioAsAddr.ais -bootmode tisecboot -pf 16 -otype ascii -cfg gpioAsAddr.cfg.`

The example code included as attachment to this applications note contains a sample batch file which can be used to create the AIS stream for this project.

The genAIS utility partitions the initialized code and data sections so that they will fall within page boundaries. The last Section of each page contains the GPIO pin configuration necessary to properly address the next page of memory. Figure 32 shows a portion of the AIS stream where the SWAP_PAGE_COMMAND is used to effect the configuration for transition from page 0 (0x90000000-0x90007ffff) to page 1 (0x90008000-0x0x9000FFFF) of the memory.

0x58535902	;Flash Address 0x90007FC8	- Check sum Request (last section on this page that was loaded)
0x4DA63199	;Flash Address 0x90007FCC	- Expected check sum value
0xFFFF9F44	;Flash Address 0x90007FD0	- Seek offset to retry if current CRC value does not match expected CRC
0x585359F1	;Flash Address 0x90007FD4	- Page Swap Command
0x00000003	;Flash Address 0x90007FD8	- Number of pin configurations
0x4300000C	;Flash Address 0x90007FDC	- Configuration for pin enable
0x00000080	;Flash Address 0x90007FE0	
0x00000001	;Flash Address 0x90007FE4	
0x43000010	;Flash Address 0x90007FE8	- Configuration for pin direction
0x00000001	;Flash Address 0x90007FEC	
0x00000001	;Flash Address 0x9007FF0	
0x43000014	;Flash Address 0x90007FF4	- Configuration for pin set
0x00000001	;Flash Address 0x90007FFC	
0x58535901	;Flash Address 0x90008000	- Section Load Command (on next page
0x10009564	;Flash Address 0x90008004	
0x00005F7C	;Flash Address 0x90008008	
0x00000000	;Flash Address 0x9000800C	
0x00000000	;Flash Address 0x90008010	

The utility automatically splits section data to fit within a page and will partition sections appropriately to complete loading on consecutive pages if required. The complete AIS data stream for this example is included with the code attached to this application note.

The application main appMain.c contains a simple while loop that invokes three functions. The link of the code was designed such that the main application code is loaded on the last page of the FLASH that is accessed. If application runs successfully, this ensures that all pages were accessed and all code loaded correctly.

```
#pragma CODE_SECTION(main,"apptext")
far unsigned int page0(void);
far unsigned int page1(void);
far unsigned int page2(void);
volatile unsigned int a,b,c;
void main(void){
while(1){
a = page0();
b = page1();
c = page2();
}
}
#pragma CODE_SECTION(page0,"appPage0")
far unsigned int page0(void) {
return 0x11111111;
}
#pragma CODE_SECTION (page1,"appPage1")
far unsigned int page1(void) {
return 0x22222222;
}
#pragma CODE_SECTION(page2,"appPage2")
far unsigned int page2(void) {
return 0x33333333;
}
```

If all code is loaded properly the DSP will be executing a loop, and variables a,b,c will be 0x11111111, 0x22222222, and 0x33333333 respectively.

7.2.2 Example 2 -GPIO Pin Used as Address Latch (gpioAsLatch)

This example uses the same code base as the gpioAsAddr example shown in the last section. The only difference is in the configuration of the GPIO pins in genBootCfg utility. In this example, a single GPIO pin has been mapped to be used as an address latch. For the purposes of this example MCASP pin AXR[0] is configured as an address latch.

The resulting configuration file, contains a flag that tells the genAIS utility that an address latch is being used.

```
#====================================================================
 # Address Pin Configuration
 #====================================================================
 -pinCfg 0x1
 -useAddressLatch 0x00000001
 -pin0 AXR0[0]
 -pincount 1
 -pinEnableReg0 0x44000010
 -pinEnableMask0 0x00000001
 -pinEnableMaskMode0 0x00000001
 -pinDirectionReg0 0x44000014
 -pinDirectionMask0 0x00000001
 -pinDirectionMaskMode0 0x00000001
 -pinSetReg0 0x4400001C
 -pinSetMask0 0x00000001
 -pinSetMode0 0x00000004
 -pinClearReg0 0x44000020
 -pinClearMask0 0x00000001
 -pinClearMode0 0x00000004
 -pinDisableReg0 0x44000010
 -pinDisableMask0 0x00000000
 -pinDisableMaskMode0 0x00000002
 -pinGlobalSetupFlag0 0x00000000
```

Once configuration is complete, compile and link the application code, the *cfg.c” file, and TISecondaryBoot.c files to form a single “.out” file as shown below.

After the “.out” file had been built, the “.out” and configuration file “*.cfg” is input to the genAIS utility to create the AIS stream. This AIS command for transitioning between memory pages is slightly different than in the previous example. In this instance the USE_ADDRESS_LATCH command is given to cause transition to the next page. As before, the genAIS utility automatically partitions code/data to fit within page boundaries, and issues the `USE_ADDRESS_LATCH` command as the last command on a page to effect smooth transition to the next page of memory.

0x58535902	;Flash Address 0x90007FBC	- Request CRC from last section
0xAD19BB1C	;Flash Address 0x90007FC0	- Expected CRC value
0xFFFF9F50	;Flash Address 0x90007FC4	- Offset to last command in case current CRC does not match expected CRC
0x585359F2	;Flash Address 0x90007FC8	- USE_LATCH_COMMAND
0x44000010	;Flash Address 0x90007FCC	- Start of configuration for GPIO pin
0x00000001	;Flash Address 0x90007FD0	
0x00000001	;Flash Address 0x90007FD4	
0x44000014	;Flash Address 0x90007FD8	
0x00000001	;Flash Address 0x90007FDC	
0x00000001	;Flash Address 0x90007FE0	
0x4400001C	;Flash Address 0x90007FE4	
0x00000001	;Flash Address 0x90007FE8	
0x00000004	;Flash Address 0x90007FEC	
0x44000020	;Flash Address 0x90007FF0	
0x00000001	;Flash Address 0x90007FF4	
0x00000004	;Flash Address 0x90007FF8	
0x90000004	;Flash Address 0x90007FFc	- Address to force on address bus to latch

The address latch as implemented in the secondary boot loader, asserts the GPIO pin (GPIO goes high), forces read of latch address, in this case 0x90000004, and then de-asserts the GPIO (GPIO goes low). It assumes that the lower 16bits of the address are captured to an external address register which will manage the upper address bits. Complete code for this example, may be found in the files attached to this applications note.

7.3 Generating AIS Stream for Use with EEPROM Programmer

If several EEPROM devices are being programmed via an EEPROM programmer that requires one of the standard hexadecimal formats such as Intel MCS-86 or Motorola Exorciser, this may be accomplished by following steps:

1. Invoke genBootCfg to effect any configuration requirements for boot (i.e. ASYNC/SDRAM configuration, PLL configuration, etc.)
2. Compile/link application code and “cfg.c” file together into single “.out” (compile/link TISecondaryBoot.c if using example secondary boot loader)
3. Invoke genAIS tool, using –otype asm as the output type.
4. Assemble and link the output assembly file from genAIS utility.
5. Invoke hex utility “hex6x.exe” with “.out” as input and choose appropriate options to generate the hexadecimal format required.

A project and set of batch command files to illustrate generation of hexadecimal files can be found in the folder:

```
TMS320C672xxBootUtils\Examples\generic\eeprom
```

In the files attached to the applications note. The project files are the same as for the gpioAsLatch project discussed in a previous section. There are additional files to assemble and link the .asm file created by call to genAIS, and then invoke the hex conversion utility to create a formatted hexadecimal file. The command line shown below invokes genAIS with –otype asm to create the assembly file.

```
genAis -I gpioAsLatch.out -o gpioAsLatchAis.asm -bootmode tisecboot -pf 16 -otype asm -cfg
gpioAsLatch.cfg
```

After the AIS output file, gpioAsLatchAis.asm has been created, assemble and link the .asm file. This can be accomplished with the following command line: cl6x -mv6700 gpioAsLatchAis.asm -z gpioAsLatchAis.obj -o gpioAsLatchAis.out -m gpioAsLatchAis.map -l TIEmifBootLnk.cmd The linker command file TIEmifBootLnk.cmd contains a single MEMORY and SECTIONS directive as shown in Figure 38.

```
/****************************************************************************/
 /* lnk.cmd v#####
*/
 /* Copyright  ) 1996@%%%% Texas Instruments Incorporated */
 /* Usage: lnk6x <obj files...> -o <out file> -m <map file> lnk.cmd */
 /* cl6x <src files...> -z -o <out file> -m <map file> lnk.cmd */
 /* */
 /* Description: This file is a sample linker command file that can be */
 /* used for linking the assembled output of the genAIS tool */
 /* -otype asm command. It places the output AIS stream */
 /* in EMIFA address space for EMIF boot. */
 /* */
 /****************************************************************************/
/* SPECIFY THE SYSTEM MEMORY MAP */
MEMORY {
 BOOTSRAM: o = 90000000h, l = 10000000h
}
SECTIONS {
 .text > BOOTSRAM }
```

The last step is to invoke the hex conversion utility. In this example, the utility will generate a single output file in Intel MCS-86 format. Please refer to the *TMS320C6000 Assembly Language Tools User’s Guide* (spru186) for details about available hexadecimal formats and full set of options.

```
 hex6x gpioAsLatchAis.out -o gpioAsLatchAis.hex hexCmd.cmd
```

The "hexCmd" file contains options to select the format and define The memory map for the FLASH/EEPROM.

```
 -I /* Specify Intel Format */
 -romwidth 16 /* Specify FLASH virtual memory width */
 -memwidth 16 /* Specify FLASH physical memory width */
 ROMS {
 /* Specify starting address and length in bytes for ROM */
 ROM0: o = 0x00000000 , l = 0x10000000
 }
SECTIONS {
 .text : paddr = 0x0000000
}
```

The first few lines of the resulting hex conversion are shown here. The complete file is included with the example code attached to this application note.

```
:20000000000100000000000000000000000000000000000000000000000000000000000DF
:200020000000000054F601BC20000000BE2A07BB00EA078807A207BF002A073A00EA0708A3
:2000400042A02BA002A020400EA0288006A024802F60214200000000028019210280203B
:20006000907902140068018802740190200000000542A0208905B0290006A020002F60214CA
:20008000200000000C28023C00E9020800F8018002740190200000000105801940265018C4A
:2000A000105A021002E602104000000049A0018C9BFA000C1290300080000000AE2A020047
:2000C000006A020803620010682A0180006A018840000000142A023A00EA020802E6029009
:2000E00060000000805A031402F6031002F40214200000000C2801BC00E801880264020CA2
:2001000060000000205802100274020C200000000C2B023C905801900265018C00EA02087B
:200120002E602104000000049A0018C9BFA000CF3102FFF80000000002A0212006A02080B
:20014000003620010A62A0180006A01884000000052E601BC600000000362000C8000000060
:20016000005A07BF042801BA00E801880264018C600000000264018C6000000022F401BC8E
:2001800020000000042A02BA00EA028802E602146000000000805A021002F602142000000069
:2001A00022E4023C60000000005A07BD0362000C80000000000000000000000000000000008C
:2001C000000000000000000000000000000000000000000000000000000000000000001F
```

7.4 Boot Examples for Professional Audio Development Kit (PADK)

The PADK is a professional audio development platform created and distributed by LyrTech, Inc. The next two examples illustrate boot from FLASH/EMIFA and I2C master modes on this platform. Please unzip the folder “TMS320C672xxBootUtils” found in the .zip file attached to this document into the LyrTech installation directory under “installdir\dsp\demos”, where installdir is whatever directory chosen to install the LyrTech PADK support files. All projects provided with these examples use relative pathnames from this directory. If installed in the “installdir\dsp\demos” directory they should build and run without modification.

7.4.1 FLASH/EMIFA Boot for PADK

The example demonstrating FLASH/EMIFA boot for the PADK also demonstrates how to customize the secondary bootloader for board requirements. The files for this example are listed below:

flashburn.c	- 'C' source file containing code to burn AIS stream to FLASH
main.c	- Switches and Led example code provided by LyrTech, Inc. distributed with the PADK.
genBootCfgFile.bat	- batch file that can be used to invoke genBootCfg
genAisFile.bat	- batch file for creation of AIS boot stream
TISecondaryBootPADK.cmd	- linker command file
TISecondaryBootPADK.c	- modified version of secondary bootloader source code.
SwitchesAndLedscfg.c	- configuration 'C' source FILE created by call to genBootCfg
SwitchesAndLeds.cfg	- configuration file crated by call to genBootCfg
C672xRomPatchV1_00_00.lib	- library containing patch for C672x ROM code
applyPatch.obj	- object file to apply the patch to ROM code from library source
SwitchesAndLeds.pjt	- Code Composer Studio™ project file to build SwitchesAndLeds.out file

The PADK uses a page register configured within an FPGA to effect paging of the FLASH memory. Since, the secondary boot loader assumes use of GPIO's for this purpose, the secondary bootloader had to be modified to for use with the PADK. The secondary bootloader uses a function, 'TISecondaryBoot_fetchAis()", to fetch the next AIS command/data from the AIS stream in FLASH. Adapting the secondary bootloader simply required replacing this function with a custom version that properly addressed paging for PADK. In addition the portions of the code that implemented the USE_ADDRESS_LATCH, and PAGE_SWAP_COMMAND were removed. The following code is the modified secondary bootloader source.

```
case JUMP_COMMAND:
 {
 jump_address = (void (*)())TISecondaryBoot_fetchAis();
 (*jump_address)();
 TISecondaryBootStatus.lastAisCmd = JUMP_COMMAND;
 break;
 }
 case JUMP_CLOSE_COMMAND:
 {
 TISecondaryBootStatus.lastAisCmd = JUMP_CLOSE_COMMAND;
 jump_address = (void (*)())TISecondaryBoot_fetchAis();
 (*jump_address)();
 break;
 }
default :
 {
 TISecondaryBoot_abort(TIBOOT_ERR_INVALID_AIS_CMD);
 break;
 }
 }
```

Additionally, the function TISecondaryBoot_fetchAis is transformed to set the page address register. The original source simply incremented the pointer to the address in EMIFA memory space, with the assumption that the PAGE_SWAP_COMMAND or USE_ADDDRESS_LATCH command guaranteed that the pointer was always within a page boundary. To adjust the algorithm for paging on the PADK, the address is shifted to get upper bits of the memory address for page register.

```
#pragma CODE_SECTION(TISecondaryBoot_fetchAis,".TIBootStrap")
unsigned TISecondaryBoot_fetchAis(void) {
// Set Page pointer
unsigned int val = *TISecondaryBootStatus.aisStreamPtr;
++TISecondaryBootStatus.aisStreamPtr;
// Read the data
return val;
}

#pragma CODE_SECTION(TISecondaryBoot_fetchAis,".TIBootStrap")
unsigned TISecondaryBoot_fetchAis(void) {
```

```
unsigned int offset;
// Set Page pointer
offset = (unsigned int)(TISecondaryBootStatus.aisStreamPtr) - ASYNC_CE;
FPGA_REG(5) = offset >> 13;
TISecondaryBootStatus.aisStreamPtr++;
// Read the data
 return ((unsigned *)FPGA_FLASH)[(offset & 0x1FFF)/4]; }
```

The symbols, `ASYNC_CE, FLASH_FPGA, and FPGA_REG(5)` are define in the PADK.h file found in 'api' folder with the software distributed by LyrTech for the PADK. The example uses the code from the SwitchesAndLeds demo found in the LyrTech installation folder:`installdir\PADK\dsp\demos\SwiteshAndLeds`. The main code is unmodified in this example. The project has been expanded to include the configuration file `SwitchesAndLedscfg.c`, created by call to genBootCfg and the source for the modified secondary boot loader code TISecondaryBootPadk.c. Figure 44 shows the required source files to build the `SwitchesAndLeds.out` file.

The essential steps to building the FLASH example are the same as mentioned in previous sections:

1. Use genBootCfg utility to configure PLL, EMIF, etc for boot. For this example, minimum configuration requires setting EMIF for 16 bit ASYNC RAM interface.
2. Compile application code + configuration 'C' source + secondary boot loader source (in this instance using the PADK specific secondary boot loader).
3. Use genAIS tool to create an AIS stream with the ".out" file created in step 2 as the input file.
4. Program FLASH memory with the code/data in the AIS output from step 3.

The PADK specific secondary boot loader requires a raw AIS data stream without use of PAGE_SWAP_COMMAND or USE_ADDRESS_LATCH command. To generate a raw AIS stream, invoke genAIS utility with "-bootmode raw" as appears below:

```
genAis -I SwitchesAndLeds.out -o SwitchesandLeds.ais -bootmode raw -pf 16 -otype ascii
```

Once the AIS stream has been created, the data stream should be programmed to FLASH memory. A small project to program the data to FLASH is included with the files attached to this applications note. Enclosed in the folder `TMS320C672xxBootUtils\Examples\Padk_examples\flashBoot\flashBurn` please find flashburn.pjt. Open this project in Code Composer Studio, it can then be built and run to program the FLASH.

When compiled and run the FLASH programming code will generate a message to standard out indicating the FLASH programming was successful.

After FLASH program is complete, check the boot mode pins on the PADK to make sure that they are configured for parallel FLASH boot mode. Please refer to the Professional Audio Development Kit Technical Reference Guide for details. Disconnect the device from Code Composer Studio, or close Code Composer Studio. Toggle the power switch on the PADK.

At this point, the SwitchesAndLeds example should be loaded and running. Toggle any of the user switches to see corresponding LEDS light up.

7.4.2 I2C Master Boot for PADK

This example creates an AIS stream for use with I2C Master boot. Even though this example runs on the PADK, the stream produced is independent of the platform and the same methods used here to create the AIS stream can be used to generate any stream for I2C Master boot mode for the TMS320C672x devices. The I2C master boot mode is implemented within the ON-CHIP bootloader in TMS320C672x ROM, so no other source is required other than the application code and configuration code for PLL, EMIF for boot if required.

This example uses a modified version of the SwitchesAndLeds example code to simply blink a single LED. The files included with this example are:

blinkLed1.c	- C source file for application code
i2cMastercfg.c	- configuration 'C' source generated by call to genBootCfg
genBootCfgFile.bat	- sample batch file to invoke genBootCfg utility
genAISFile.bat	- sample batch file to invoke genAIS utility to create AIS stream for I2C master boot
padk.cmd	- sample linker command file

Although not required, a boot configuration file was created to adjust the I2C clock to enable speed up of boot process. This was done via invocation of genBootCfg utility and selecting I2C clock configuration. The resulting configuration was saved to i2cMasterCfg.c. This step is useful for adjusting the clock to meet the operating requirements for high hold and low hold times of the I2C EEPROM. The linker command file “padk.cmd” was modified to include load of the “.TIBoot” section. This section is created by the genBootCfg tool and this is where all boot configuration PLL and EMIF settings required for boot are placed. The genAIS utility looks for the code/data for this section and places it first in the AIS data stream. Once this code is loaded, an AIS **jump** command is issued to branch to this code for execution. The code runs, changing PLL , EMIF etc to required values for boot. The boot process then continues with parsing of remaining AIS commands/data.

The project file for this example are shown in Figure 47.

After building the project, the genAIS utility is invoked to produce the AIS stream for I2C Master boot as shown here:

```
genAis -I .blinkLed1.out -o blinkLed1.ais -bootmode i2cmaster -otype ascii
```

The resulting AIS file can then be programmed to the I2C EEPROM. A sample project to write the AIS stream to the I2C EEPROM deice on the PADK has been provided with this example. The code may be found in the folder `TMS320C672xxBootUtils\Examples\Padk_examples\i2cMaster\progI2c`. Open project file progI2c.pjt in Code Composer Studio.

When this project is compiled and run it will generate a message in the standard output window indicating that I2C EEPROM was programmed as shown in Figure 49.

Once the I2C EEPROM has been programmed, disconnect the device or close Code Composer Studio. Check for boot mode pin configuration for I2C master boot. Please refer to the Professional Audio Development Kit Technical Reference Guide for details. Toggle the power switch on the PADK. The blinkLed1 code should be loaded and starting to run. It may take a second or so before LED 1 starts blinking.

8 Troubleshooting On-chip BootLoad

In the event that a problem is encountered during boot, there are some steps that can be taken to help debug the process. The on-chip bootloader, that runs from the TMS320C672x ROM at device reset, traps to a specific segment of code and executes an infinite loop when the boot process fails. It writes an error code to memory location in internal RAM, 0x10000708, that can be used to help determine the cause of boot failure. Table 18 below lists the possible error codes and brief explanation.

Table 18. On-Chip BootLoader Error Codes

Error Code	Explanation
1	Incorrect keyword
2	Transmit sync error
3	CRC error
5	Unsupported bootmode error

The following sections discuss briefly each error code and possible causes.

8.1 Incorrect Key Word

The “incorrect key word” error code is generated during SPI or I2C master boot modes. When in SPI or I2C master boot mode, the first word read from the serial device MUST be the AIS magic word "0x41504954". If any other data value is read from the device at this location, the boot loader will abort the boot process and write the error code into location 0x10000708.

- Check the AIS data stream to make sure the a valid key word is the first data written to the device.
- Check endianness of data. The on-chip bootloader expects data to be received MSB first.
- Check device/board configuration to make sure proper communication exists between SPI/I2C device and TSM320C672x.

8.2 Transmit Sync Error

A transmit sync error whenever the on-chip bootloader attempts to process the next valid AIS command in the stream and encounters an invalid command. It expects a valid AIS command starting with 0x585359xx to be transmitted under following circumstances:

- Immediately after receiving the valid key word for SPI/I2C master modes.
- Immediately after receiving/processing PING device command for SPI/I2C slave modes.
- Immediately after it has completed processing any other valid command that was received in the data stream and is fetching the next command from the stream.

If any other data is encountered at that point, the bootloader aborts the boot process and writes “Transmit Sync Error” code to the boot error “register”, 0x10000708. When this error occurs, check the AIS data stream to make sure that a valid AIS command (see Section 3 for a list of valid AIS commands) is being transmitted at this point in the data stream.

8.3 CRC Error

A CRC error will occur whenever the expected CRC value received by the on-chip bootloader while processing a REQUEST_CRC command, does not match the current CRC value calculated by the bootloader. Please refer to Section 9 and Appendix A for details on generating CRC values compatible with on-chip bootloader software. When this error is encountered:

- Check the CRC value calculated and transmitted with the Request CRC command to make sure that is calculated using same method employed by the on-chip bootloader.
- Check communication between SPI/I2C device and TMS320C672x.

8.4 Unsupported BootMode Error

If this error occurs, please check the boot mode pin configuration. Please refer to Table 4 for a list valid boot pin configurations. One way to determine which boot mode was set, is to open Code Compose Studio after device reset and check the CFGPIN0 and CFGPIN1 registers to make sure the required mode was set.

9 Calculating CRC

The on-chip bootloader uses a 32bit CRC. Code for calculating the CRC is given in the Appendix A. The CRC as calculated for the on-chip bootloader requires 3 calls to the BL_updateCrc function. The first call is made sending the section load address as the data word. The second call uses the section size in bytes as the data word. The third call sends the actual section data, calculating a CRC across all the data elements in the section. So the final CRC is a combination of the CRC's calculated for section address ,section size and section data. A sample set of calls to the function to create the expected CRC value is shown below:

```
unsigned int crc;
unsigned int sectionAddr;
unsigned int sectionSize;
unsigned int *sectionData;
crc = BL_updateCRC(&sectionAddr, 4, 0);
crc = BL_updateCRC(&sectionSize, 4, crc);
crc = BL_updateCRC(sectionData, sectionSize, crc);
```

The last 'crc' value calculated, is the value that should be written as the expected CRC for the REQUEST_CRC command. If calculating a single CRC for the entire application load, simply pass each successive "crc" value into the subsequent calls to BL_updateCRC.

```
typedef struct {
unsigned int sectionAddr;
unsigned int sectionSize;
unsigned int *sectionData;
} SectionDatObj;
SectionDataObj mySections[10];
unsigned int crc;
crc = 0;
for(i=0;i<10;i++) {
crc = BL_updateCRC(&(mySections[i].sectionAddr), 4, crc);
crc = BL_updateCRC(&(mySections[i].sectionSize), 4, crc);
crc = BL_updateCRC(mySections[i].sectionData, mySections[i].sectionSize, crc);
}
```

10 Memory Allocation

The on-chip bootloader requires a small amount of on-chip RAM used as local stack/data space during boot. Internal memory locations 0x10000400 – 0x10000FFF are reserved for use by the on-chip bootloader during boot process. DO NOT allocate any initialized sections within this memory range when linking application code. Doing so may cause the boot process to fail. Examples of initialized sections are compiler generated sections such as .text, .switch, .cinit, which contain code/data that will be loaded directly into the memory space when application is booted. Uninitialized sections, such as .bss, or .far may be allocated in this memory space, since these will not be populated until after boot of application code is complete.

When using any of the ROM'ed applications present in TMS320C672xx Internal ROM, such as DSPBIOS or DSPLIB, please refer to the *TMS320C672xx ROM Data Sheet* (SPRS277) for any further memory allocation constraints that may apply.

11 Determining On-chip BootLoader/ROM Version

The ROM version may be read from on-chip ROM memory location 0x0000000C. The current ROM/Bootloader revision is 0xC9230C10.

12 Cache Considerations

The ROMed bootloader software disables the cache at start of boot process. It does not invalidate nor enable cache prior to branching to the loaded application code. Therefore, the application must take responsibility for properly invalidating, and enabling the cache, if cache is used. Figure 32 illustrates how to invalidate, enable, and bypass cache operation.

```
#define L1PSAR              *(unsigned int *)(0x20000000u)
#define L1PICR              *(unsigned int *)(0x20000004u)
#define L1P_INVALIDATE                       (0x80000000u)
#define CACHE_ENABLE                         (0x00000040u)
#define CACHE_FREEZE                         (0x00000060u)
#define CACHE_BYPASS                         (0x00000080u)
#define CACHE_CONTROL_MASK                   (0x000000E0u)

// Declare CSR register
extern cregister volatile unsigned int CSR;


// This function Invalidates and Enables Cache
void cache_enable() {

   // Invalidate all lines of cache by setting L1P bit
     L1PICR = L1PICR | L1P_INVALIDATE;

   // Make Sure Cache Invalidate is Complete
     while (L1PICR != 0x00000000u){
        }

   // Enable the Cache
     CSR = (CSR & (~CACHE_CONTROL_MASK)) | CACHE_ENABLE;
}


// This function Invalidates and by passes Cache
// forcing all fetches from memory contents.
void cache_bypass() {

   // Invalidate all lines of cache by setting L1P bit
     L1PICR = L1PICR | L1P_INVALIDATE;

   // Make Sure Cache Invalidate is Complete
     while (L1PICR != 0x00000000u){
        }

   // Bypass the Cache
     CSR = (CSR & (~CACHE_CONTROL_MASK)) | CACHE_BYPASS;
}
```

Figure 32. Bypass Cache Operation

Appendix A Calculating the CRC

The CRC calculated to process the REQUEST_CRC command is based on the following algorithm, where “data_ptr” points to the first data element in the current section, “section_size” is the size of the section expressed in 8bit bytes, and “crc” is current crc value.

```
unsigned int BL_updateCRC(unsigned int *data_ptr, unsigned int section_size, unsigned int crc) {
unsigned int n, crc_poly = 0x04C11DB7; /* CRC - 32 */ unsigned int msb_bit; unsigned int
residue_value; int bits; for( n = 0; n < (section_size>>2); n++ ) { bits = 32; while( --bits >= 0
) { msb_bit = crc & 0x80000000; crc = (crc << 1) ^ ( (*data_ptr >> bits) & 1 ); if ( msb_bit )
crc = crc ^ crc_poly; } data_ptr ++; } switch(section_size & 3) { case 0: break; case 1:
residue_value = (*data_ptr & 0xFF) ; bits = 8; break; case 2: residue_value = (*data_ptr &
0xFFFF) ; bits = 16; break; case 3: residue_value = (*data_ptr & 0xFFFFFF) ; bits = 24; break; }
if(section_size & 3) { while( --bits >= 0 ) { msb_bit = crc & 0x80000000; crc = (crc << 1) ^ (
(residue_value >> bits) & 1 ); if ( msb_bit ) crc = crc ^ crc_poly; } } return( crc ); }
```