TEXAS
INSTRUMENTS

Booting a TMS320C32 Target System in a C Environment

Application
Report

1996 **Digital Signal Processing Solutions**

Printed in U.S.A., October 1996

Booting a TMS320C32 Target System in a C Environment

Peter Galicki
DSP Products – SC Group

Literature Number: SPRA067
October 1996

PRINTED WITH SOY INK™

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

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

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

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

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

Copyright © 1996, Texas Instruments Incorporated

Contents

Title *Page*

ABSTRACT **1**

INTRODUCTION **1**

GENERATING A COFF FILE **2**
- Complier 2
- Assembler 2
- Linker 2
- The .OUT (COFF) File 2

LOADING THE COFF FILE TO THE TARGET SYSTEM **5**

DEBUGGER BOOT **6**
- RAM Model (Linker –cr Option) 6
- ROM Model (Linker –c Option) 6

EPROM BOOT **9**
- Microprocessor Mode (Linker –c Option) 9
- Microcomputer/Boot Loader Mode (Linker –cr Option) 9

BOOT TABLE MEMORY CONSIDERATIONS **13**

HOST LOAD **16**
- Boot From Serial Port 16
- Boot From a Latch 16
- Asynchronous Boot From a Communications Port 16

Appendices

Title *Page*

APPENDIX A – THE ’C32 BOOT TABLE EXAMPLES **21**

APPENDIX B – THE ’C32 BOOT LOADER PROGRAM **27**
- ’C32 Boot Loader Opcodes 27

APPENDIX C – MEMORY ACCESS FOR C PROGRAMS **37**

APPENDIX D – MEMORY INTERFACE AND ADDRESS TRANSLATION **43**

List of Illustrations

Figure *Title* *Page*

1 Compile, Assemble, and Link Flow 4

2 Debugger Load (Linker –cr Option) 7

3 Debugger Load (Linker –cr Option) 8

4 32-Bit EPROM Boot in the Microprocessor Mode (Linker –c Option) 11

5 8-Bit EPROM Boot Using the On-Chip Boot Loader (Linker –cr Option) 12

6 Memory Configuration for Normal Program Execution 14

7 Boot Table Memory Configuration 15

8 Boot From Host Using Serial Port (Linker –cr Option) 17

9 Boot From Host Using an 8-Bit Latch (Linker –cr Option) 18

10 Boot From Host Using Asynchronous Communications Port (Linker –cr Option) 19

11 Boot From a 32-Bit Wide ROM to 8-, 16-, and 32-Bit Wide RAM 22

12 Boot From a 16-Bit Wide ROM to 8-, 16-, and 32-Bit Wide RAM 23

13 Boot From a Byte Wide ROM to 8-, 16-, and 32-Bit Wide RAM 24

14 Boot From Serial Port to 8-, 16-, and 32-Bit Wide RAM 25

15 TMS320C32 Boot Loader Program 28

16 Memory Allocation in C Programs 38

17 Dynamic Memory Allocation for TMS320C32 (One Block of 32-Bit Memory) 39

18 Dynamic Memory Allocation for TMS320C32 (One Block of 16-Bit Memory) 41

19 Dynamic Memory Allocation for TMS320C32
(One Block Each of 32-, 16-, and 8-Bit Memory) 42

20 Data and Program Packing (Program and One Data Size) 44

21 Data and Program Packing (Program and Two Data Sizes) 45

22 Address Translation for 32-Bit Data Stored in 32-Bit Wide Memory 47

23 Address Translation for 16-Bit Data Stored in 32-Bit Wide Memory 48

24 Address Translation for 8-Bit Data Stored in 32-Bit Wide Memory 49

25 Address Translation for 32-Bit Data Stored in 16-Bit Wide Memory 50

26 Address Translation for 16-Bit Data Stored in 16-Bit Wide Memory 51

27 Address Translation for 8-Bit Data Stored in 16-Bit Wide Memory 52

28 Address Translation for 32-Bit Data Stored in 8-Bit Wide Memory 53

29 Address Translation for 16-Bit Data Stored in 8-Bit Wide Memory 54

30 Address Translation for 8-Bit Data Stored in 8-Bit Wide Memory 55

ABSTRACT

This application report describes methods to boot a TMS320C32 target system in a C environment.

INTRODUCTION

A DSP system uses a boot procedure following power-up or reset to initialize the system volatile memory (such as SRAM) with the application program/data and to start execution of the application code. The SRAM loads from a nonvolatile medium (EPROM) or from a PC development platform using a debugger/loader program. The loader uses an emulator cable to move the load file from the PC hard disk to the SRAM on the DSP target board. An EPROM boot causes the DSP to start program execution directly from 16- or 32-bit EPROM (microprocessor mode). A hard-wired on-chip boot loader program copies the boot table from the 8-bit EPROM to internal or external SRAM and then starts execution from the SRAM (microcomputer/boot loader mode).

TI supports four ways to boot a DSP system following power-up/reset. Each boot procedure uses a different combination of ’C32 silicon features, software, and hardware tools. Each combination forms an integrated development environment that includes features to support most system boot requirements.

A boot development flow includes two major tasks:

- Use C source debugger and assembly level tools to compile, assemble and link the boot code/data to create a binary common object file format (COFF) executable object.
- Load the COFF file into the DSP target system.

Generating the COFF file (linker output .OUT file) uses the same flow for all boot methods.

GENERATING A COFF FILE

Generating a COFF file requires compiling the source code with the C compiler, then assembling and linking the resulting assembly files with the assembly level tools. A text editor creates additional assembly files or the files are extracted from the RTS30 library. The linking process resolves all external references between program files and generates the .OUT COFF file subject to specified options (such as –c or –cr boot options).

Compiler

Figure 1 shows how one or more C files are compiled into multiple assembly files. Each assembly file is constructed from former C functions that were individually decomposed into standard logical sections. The program code is assigned to .TEXT, stack to .STACK, dynamically allocated memory to .SYSMEM, switch tables to .CONST, uninitialized variables to .BSS, and initialized variables to .CINIT. If, following system reset, the program executes directly out of EPROM (microprocessor mode), a separate assembly file holds the reset vector (and possibly other interrupt vectors). The reset vector points to the address contained in the c_int00 symbol that the linker resolves with the beginning of the BOOT.ASM routine (from the RTS30 library).

Assembler

The assembler assembles all .ASM files into their respective .OBJ files. Since each .ASM file may have a .TEXT section fragment for each function in the file, its .OBJ counterpart groups all the fragments into a single contiguous .TEXT section. This applies to all sections in that file. The results of the assembler process are multiple .OBJ files composed of single instances of all standard C sections. In addition to the object files generated by the user, the subsequent boot procedures require another .OBJ file. The BOOT.ASM file can be extracted from the RTS30 library and assembled separately into the BOOT.OBJ. The BOOT.OBJ is the first routine executed following reset. It initializes the C environment by setting up the system stack, processing initialized variables, setting up the page pointer, and calling the main function. While the BOOT.ASM is required for a C program, other optional files may be extracted from the library, such as MALLOC.ASM, which is used to allocate additional memory at run time.

Linker

The linker assigns physical addresses to logical program sections from .OBJ files. A linker command file defines the available physical memory segments (MEMORY directive), assigns one or more sections to individual memory segments (SECTIONS directive), and lists all the object files containing the sections to be processed. The order in which object files are listed is important and reflects the order in which individual sections are stacked in physical memory. For that reason the BOOT.OBJ file must always be the first one listed, since it represents the execution entry point for every C program. The BOOT.OBJ global symbol c_int00 provides the entry address that can be resolved to other files that are linked with the BOOT.OBJ (for example, the vector file that needs an address for the reset vector). Depending on the method, the linker can be invoked with the –c or –cr option. These two options control how a C program's initialized variables are handled during the later stages of the boot process.

The .OUT (COFF) File

After resolving the external references among all program sections, the linker builds the .OUT file. The .OUT file is constructed in the binary COFF format and it contains all the sections listed in the linker SECTIONS directive. It contains all the information about the program, how to load it into the target DSP system, and even symbol information for the debugger that is later used to verify the code. All C and assembly symbols, such as subroutine labels, etc., can be made visible in the debugger window (by embedding them in the COFF file), provided that they are declared as global symbols and the appropriate options are used with the code generation tools.

Some .OUT sections contain only the starting addresses and no code or data. They include the .STACK section for the system stack, the .SYSMEM section for dynamically allocated memory, and the .BSS section for uninitialized data. The boot process also uses the .BSS section as a destination for the initialized variables that are originally stored in the .CINIT section of the .OUT file. Although they contain no data, the .STACK and .SYSMEM sections are included in .OUT to allow the debugger tools to verify that the physical memory for those sections exists on the target board. Other sections in the .COFF file, such as .VECTORS, .CONST, and .TEXT, contain the starting addresses and the contents of the sections. When the debugger loads the .TEXT section into the target system, for example, the opcodes for all assembly instructions for the entire program are copied beginning at the section starting address.

The .CINIT section is special because it contains initialized variables. Once the .OUT file is generated it can be burned into a 16- or 32-bit wide EPROM and the program can start executing directly from that EPROM following reset (in the microprocessor mode). But if the initialized variables also reside in the same EPROM, they are not really variables since one cannot write to an EPROM device and actually change the values of those variables. For that reason, before user program execution commences, the BOOT.ASM library routine copies the initialized variables from the EPROM .CINIT section to the SRAM .BSS section, one array of data at a time. Figure 1 shows that the .CINIT section is divided into individual array records in which each array has a length, data content, and destination address in the SRAM .BSS section. The .BSS section is the final destination for initialized variables, while the .CINIT EPROM section is a temporary holding place before power-up/reset. The .CINIT section also stores the –c/-cr linker option selection for use in the later stages of the boot process.

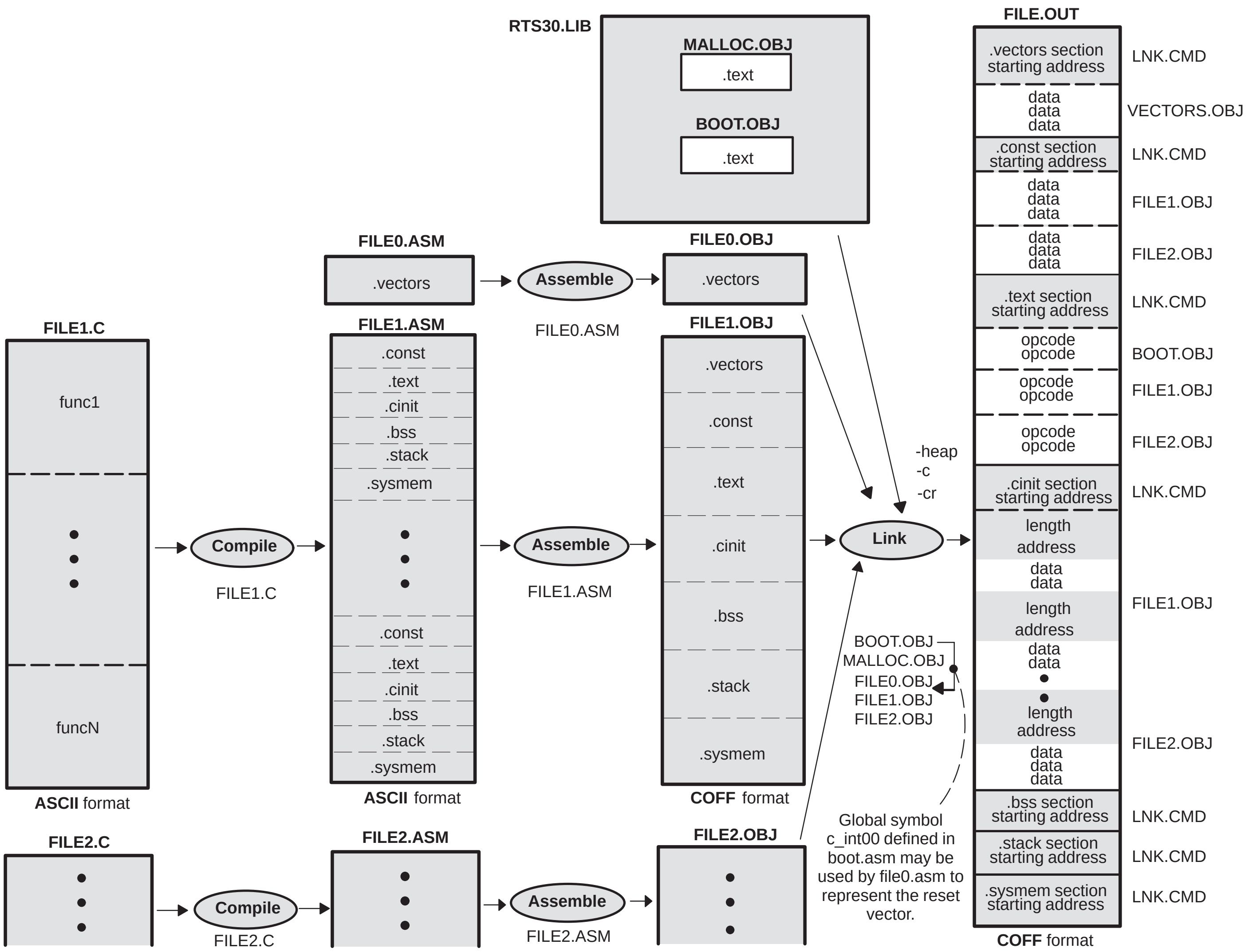

RTS30.LIB
MALLOC.OBJ
.text
BOOT.OBJ
.text
FILE0.ASM
.vectors
Assemble
FILE0.ASM
FILE0.OBJ
.vectors
FILE1.C
func1
funcN
ASCII format
Compile
FILE1.C
FILE1.ASM
.const
.text
.cinit
.bss
.stack
.sysmem
.const
.text
.cinit
.bss
.stack
.sysmem
ASCII format
Assemble
FILE1.ASM
FILE1.OBJ
.vectors
.const
.text
.cinit
.bss
.stack
.sysmem
COFF format
-heap
-c
-cr
Link
BOOT.OBJ
MALLOC.OBJ
FILE0.OBJ
FILE1.OBJ
FILE2.OBJ
Global symbol c_int00 defined in boot.asm may be used by file0.asm to represent the reset vector.
FILE2.C
Compile
FILE2.C
FILE2.ASM
Assemble
FILE2.ASM
FILE2.OBJ
FILE.OUT
.vectors section starting address
LNK.CMD
data data data
VECTORS.OBJ
.const section starting address
LNK.CMD
data data data
FILE1.OBJ
data data data
FILE2.OBJ
.text section starting address
LNK.CMD
opcode opcode
BOOT.OBJ
opcode opcode
FILE1.OBJ
opcode opcode
FILE2.OBJ
.cinit section starting address
LNK.CMD
length address data data
length address data data
FILE1.OBJ
length address data data data
FILE2.OBJ
.bss section starting address
LNK.CMD
.stack section starting address
LNK.CMD
.sysmem section starting address
LNK.CMD
COFF format

Figure 1. Compile, Assemble, and Link Flow

LOADING THE COFF FILE TO THE TARGET SYSTEM

Loading the COFF file to the DSP target system extracts program and data content as well as control information and then applies the control to place the program/data in the target memory. Some control information embedded in the COFF file may not apply directly to the program/data content. For example, the COFF file may include a symbol table for the debugger or a memory width control word for the on-chip boot loader.

Using the debugger to load the COFF file to target memory requires connecting the target board to the PC (on which the debugger is running) with an emulator cable and pod, and then transferring the COFF file with the LOAD command. The linker –c/-cr options control processing of the .CINIT section during the load operation.

The COFF file can also be loaded to a target system from an EPROM. The Hex30 utility converts the COFF file to an EPROM-programmer-compatible file that can be programmed to the EPROM. In the microprocessor mode, the program executes directly from the EPROM. In the microcontroller/boot loader mode, the on-chip boot loader first expands the EPROM contents into target SRAM and the program executes from there. In either case, the C program begins execution at the start of the BOOT.ASM library routine to initialize the C environment before the rest of the C program runs.

DEBUGGER BOOT

Figures 2 and 3 show how to load the COFF file into the target system using the debugger load command.

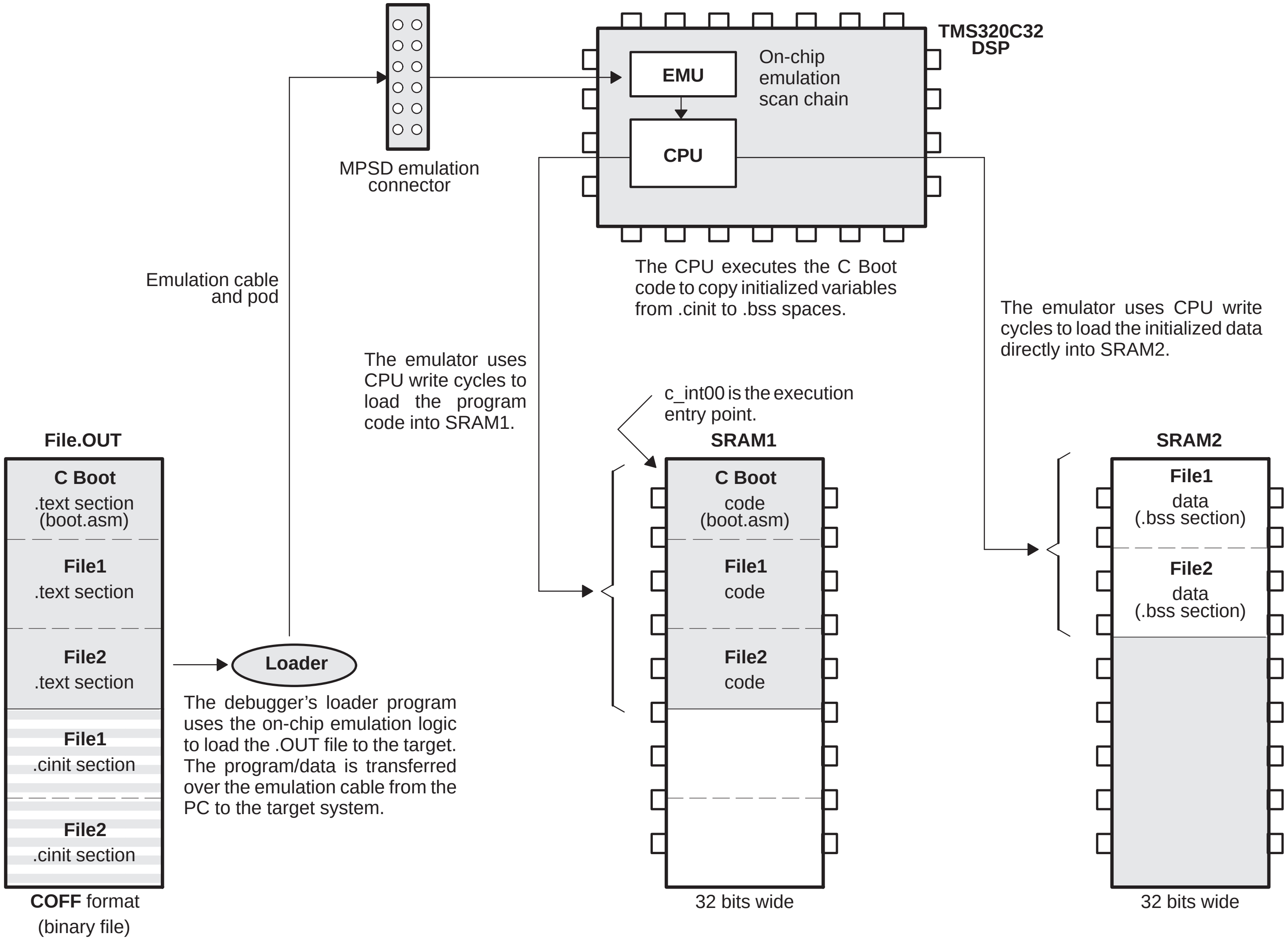
The debugger is a standard TI software development tool that runs on a PC platform. The debugger accesses the target board through the PC emulator card and cable. The cable connects to the target board through a 12-pin connector that routes the signals to the DSP’s emulation pins. The emulation pins control the operation of the MPSD scan chain in the processor. Depending on the command issued by the debugger, the emulation circuitry in the scan chain stops or resumes processor operation, examines/loads registers or memory, sets breakpoints, or executes code one instruction at a time (called single-step execution). The debugger LOAD command reads the COFF file from the PC hard drive, extracts program/data content and transfers it through the emulator cable to the target board’s memory.

RAM Model (Linker –cr Option)

When the COFF file is loaded into the target board’s memory, most sections in the file are processed by copying the program/data to the address defined at the beginning of each section; however, the initialized variables in the .CINIT section are processed differently. If the COFF file was generated by the linker using a –cr option, the .CINIT section of the file is loaded using the RAM model (see Figure 2). The RAM model assumes that the target memory is composed exclusively of SRAM devices. Thus, the initialized variables can be directly copied to the SRAM .BSS section, one array at a time, without first placing them in a temporary EPROM .CINIT section. Once the initialized variables have been loaded into SRAM, they can be read or written to by the CPU without further initialization steps by the BOOT.ASM at the beginning of C program execution.

ROM Model (Linker –c Option)

If the COFF file was created with the linker –c option, the loader places the .CINIT section in the target memory according to the ROM model. The ROM model copies the .CINIT section as one block to the address specified at the beginning of the same .CINIT section. Following the load operation, the ROM model expects the BOOT.ASM routine (at the beginning of the C program) to further process the .CINIT section by copying its contents to the SRAM .BSS section, one array at a time. After the COFF load operation, the memory content is the same as that created by the RAM model with one exception: the target SRAM still contains the temporary .CINIT section that serves no purpose after being processed by the BOOT.ASM. The ROM model can still be useful, for example, to simulate the microprocessor mode EPROM boot (see Figure 4). During the development cycle, instead of burning a new EPROM each time the code is modified, the EPROM can be removed and replaced with an equivalent SRAM device (by reconfiguring a few jumpers, perhaps). The ROM model allows using the loader to quickly load and debug the modified code, while preserving the bus activity at power up to simulate an EPROM boot.


MPSD emulation connector
TMS320C32 DSP
EMU
On-chip emulation scan chain
CPU
Emulation cable and pod
File.OUT
C Boot
.text section
(boot.asm)
File1
.text section
File2
.text section
File1
.cinit section
File2
.cinit section
COFF format
(binary file)
Loader
SRAM1
C Boot
code
(boot.asm)
File1
code
File2
code
32 bits wide
SRAM2
File1
data
(.bss section)
File2
data
(.bss section)
32 bits wide

The CPU executes the C Boot code to copy initialized variables from .cinit to .bss spaces.

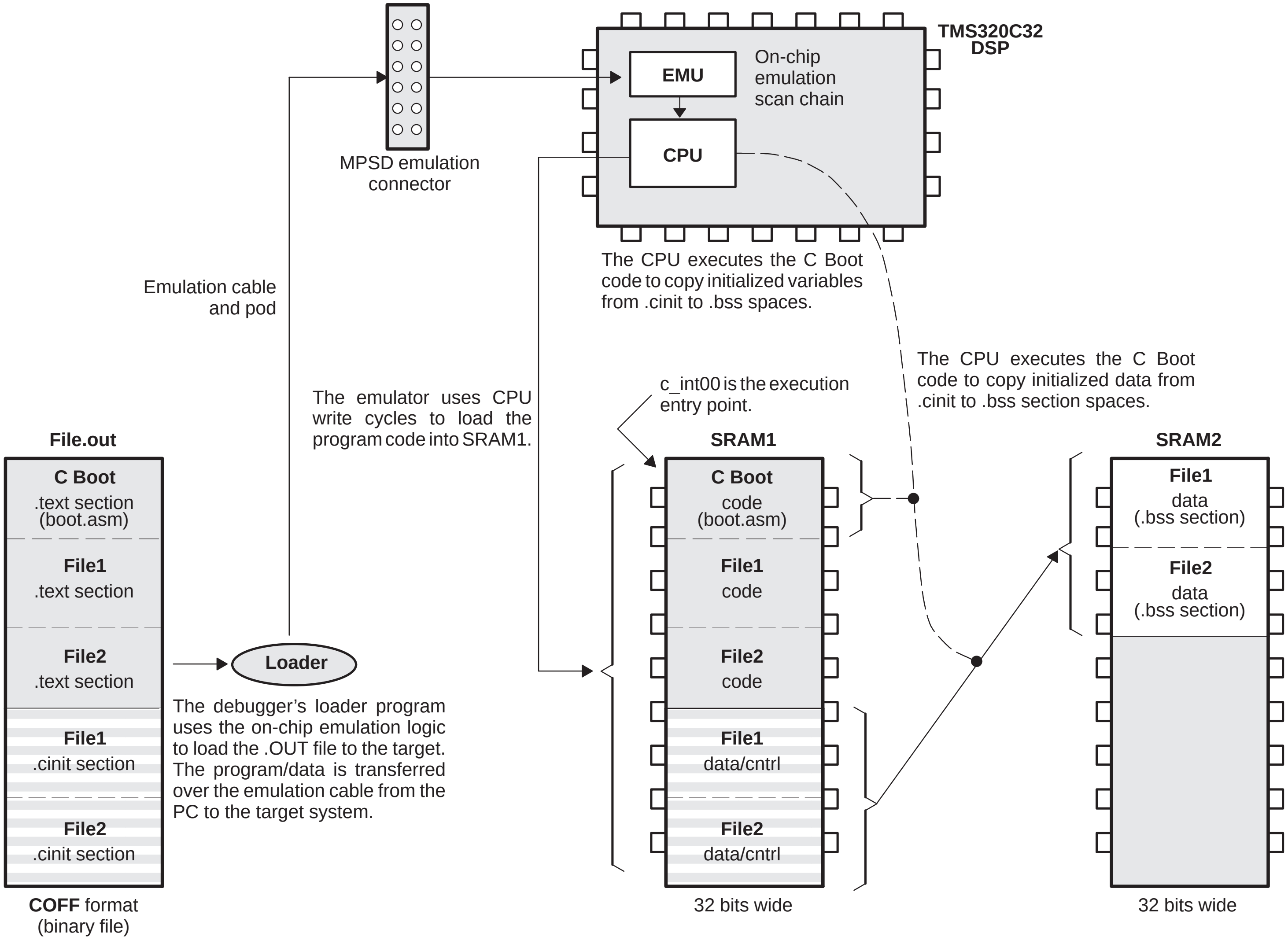
The emulator uses CPU write cycles to load the initialized data directly into SRAM2.

The emulator uses CPU write cycles to load the program code into SRAM1.

c_int00 is the execution entry point.

The debugger's loader program uses the on-chip emulation logic to load the .OUT file to the target. The program/data is transferred over the emulation cable from the PC to the target system.

Figure 2. Debugger Load (Linker –cr Option)



MPSD emulation connector
TMS320C32 DSP
EMU
On-chip emulation scan chain
CPU
Emulation cable and pod
File.out
C Boot
.text section
(boot.asm)
File1
.text section
File2
.text section
File1
.cinit section
File2
.cinit section
COFF format (binary file)
Loader
SRAM1
C Boot
code
(boot.asm)
File1
code
File2
code
File1
data/cntrl
File2
data/cntrl
32 bits wide
SRAM2
File1
data
(.bss section)
File2
data
(.bss section)
32 bits wide

The CPU executes the C Boot code to copy initialized variables from .cinit to .bss spaces.

The emulator uses CPU write cycles to load the program code into SRAM1.

c_int00 is the execution entry point.

The CPU executes the C Boot code to copy initialized data from .cinit to .bss section spaces.

The debugger's loader program uses the on-chip emulation logic to load the .OUT file to the target. The program/data is transferred over the emulation cable from the PC to the target system.

Figure 3. Debugger Load (Linker –c Option)

EPROM BOOT

Booting a DSP target board from C code stored in nonvolatile memory and accessible to the DSP can be done in two ways. If the DSP is powered up in the microprocessor mode, the reset causes the program to start execution from 32- or 16-bit EPROM by fetching the reset vector from memory address 000000h and branching to the reset interrupt service routine (ISR) pointed to by that vector.

On the other hand, if the DSP is powered up in the microcomputer/boot loader mode, program execution starts with the on-chip boot loader program. The boot loader reads the COFF file from an 8-bit EPROM and expands it to the system SRAM from which it can be executed (16 or 32 bits wide). In either case, program entry occurs at the beginning of the BOOT.ASM library routine to initialize the C environment prior to execution of the C code.

Microprocessor Mode (Linker –c Option)

Before the binary COFF file can be burned into an EPROM, it must be converted to an ASCII format that an EPROM programmer can recognize (see Figure 4). The HEX conversion utility converts COFF files to a programmer object file format such as Intel Hex. The EPROM programmer uses the converted files to program one or more EPROMs that can be inserted into the DSP target board.

If the linker –c option is used to create the COFF file (ROM model), the HEX utility copies the .CINIT section directly into the programmer object file without processing its content. In other words, the .CINIT section in the programmed EPROM contains the initialized data as well as destination addresses and lengths in .BSS for individual .CINIT data arrays. To start program execution from EPROM at power up, the DSP must be configured in the microprocessor mode by pulling the MCBL/$\overline{\text{MP}}$ pin low. Triggered by the low-to-high transition of the $\overline{\text{RESET}}$ pin, the DSP executes the reset vector fetch read cycle. The reset vector points to the BOOT.ASM routine which is executed next. The linker –c option sets a control bit in the .CINIT section of the COFF file.

When the BOOT.ASM program executes the .CINIT section, it checks the –c/–cr control bit. The –c option (ROM model) causes the BOOT.ASM to copy the contents of each array within the .CINIT section to its destination in the .BSS section mapped to SRAM. The initialized variables must be copied from EPROM to SRAM at the beginning of program execution because they cannot be modified in EPROM (variable data must be changeable during program execution).

Microcomputer/Boot Loader Mode (Linker –cr Option)

The TMS320C32 features an on-chip hardwired boot loader program in the internal programmable logic array (PLA). The boot loader reduces the DSP target board cost by replacing multiple fast EPROMs with a single 8-bit slow (inexpensive) EPROM. Because the ’C32 cannot execute code from memory that is only eight bits wide, the on-chip boot loader program reads the boot table from the byte-wide EPROM and reconstructs all sections of the original COFF file one byte at a time before placing the program/data in SRAM (see Figure 5).

To power up the DSP in the boot loader mode, the MCBL/$\overline{\text{MP}}$ pin must be held high when the $\overline{\text{RESET}}$ signal is deasserted. At that stage, the DSP starts executing the boot loader code from internal address 000045h. Immediately after it starts execution, the boot loader checks the interrupt flag (IF) register. At this point, all interrupts are disabled and remain disabled until the application program enables them. Depending on which external interrupt is asserted, the boot loader looks for the boot table at one of three external memory locations or at the serial port. At this point, the interrupt pins carry a message to the boot loader telling it where to get the boot table after reset.

The boot table structure resembles the COFF file from which it was derived by the HEX conversion utility. The main feature that distinguishes the boot table from a regular HEX utility output (such as the microprocessor mode boot example) is that in addition to the contents of the COFF sections, the boot table includes special control words for the on-chip boot loader program to instruct it how to assemble and load those sections. Each section is built into a block preceded by three control words: block size, destination address, and destination memory width/data size. Multiple blocks can be transferred to selected parts of the DSP memory map. To format the COFF file into the boot table, the program section to be booted must be identified to the HEX conversion utility with the SECTIONS directive. The boot table is constructed of the COFF sections identified in the SECTIONS directive and marked with the BOOT option (see Figure 5).

If the linker uses the –cr option to create the COFF file, the HEX utility processes the COFF .CINIT section and assigns the addresses in the .BSS section to the corresponding .CINIT arrays in the boot table. Every C program starts execution with the BOOT.ASM routine, but because one of the BOOT.ASM control flags indicates that the COFF file was created with the linker –cr option, the code skips transfer of .CINIT contents to .BSS. The HEX utility does that task by placing all the initialized variables in .BSS while creating the boot table without relying on BOOT.ASM to make the transfer at run time (see Figure 5).

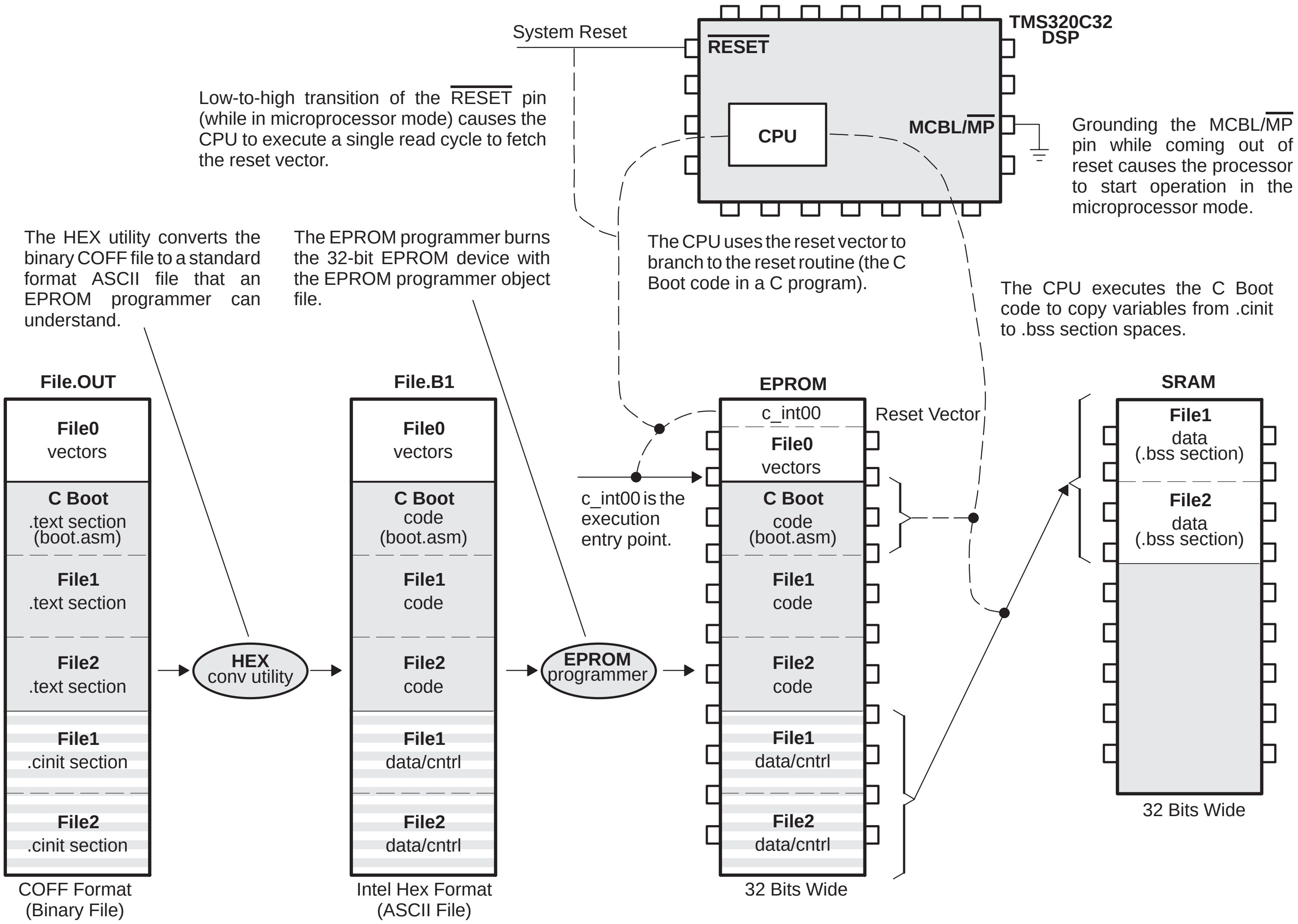
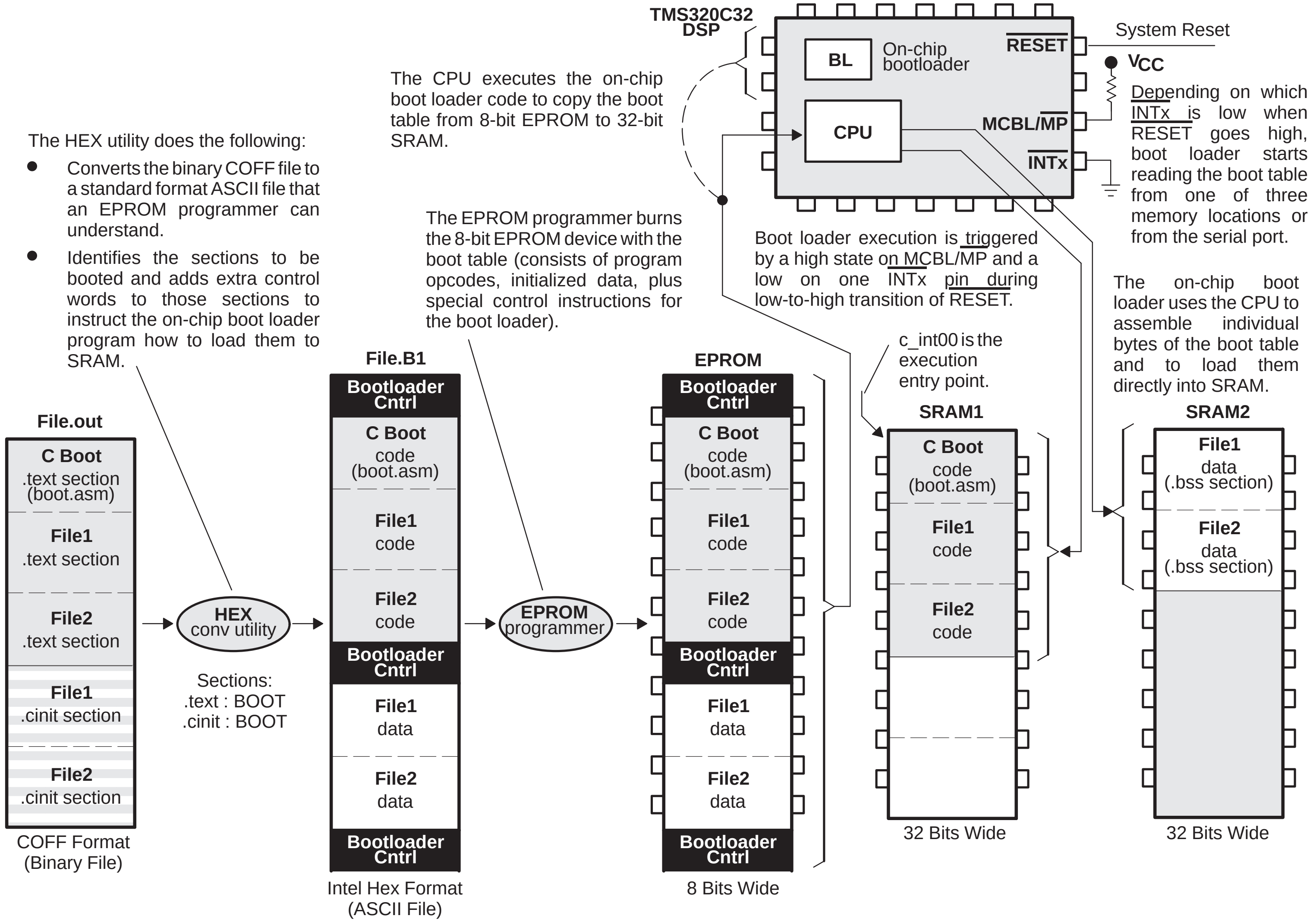

System Reset
RESET
TMS320C32 DSP
CPU
MCBL/MP
Low-to-high transition of the RESET pin (while in microprocessor mode) causes the CPU to execute a single read cycle to fetch the reset vector.
Grounding the MCBL/MP pin while coming out of reset causes the processor to start operation in the microprocessor mode.
The HEX utility converts the binary COFF file to a standard format ASCII file that an EPROM programmer can understand.
The EPROM programmer burns the 32-bit EPROM device with the EPROM programmer object file.
The CPU uses the reset vector to branch to the reset routine (the C Boot code in a C program).
The CPU executes the C Boot code to copy variables from .cinit to .bss section spaces.
File.OUT
File0 vectors
C Boot .text section (boot.asm)
File1 .text section
File2 .text section
File1 .cinit section
File2 .cinit section
COFF Format (Binary File)
HEX conv utility
File.B1
File0 vectors
C Boot code (boot.asm)
File1 code
File2 code
File1 data/cntrl
File2 data/cntrl
Intel Hex Format (ASCII File)
EPROM programmer
c_int00 is the execution entry point.
EPROM
c_int00
Reset Vector
File0 vectors
C Boot code (boot.asm)
File1 code
File2 code
File1 data/cntrl
File2 data/cntrl
32 Bits Wide
SRAM
File1 data (.bss section)
File2 data (.bss section)
32 Bits Wide

Figure 4. 32-Bit EPROM Boot in the Microprocessor Mode (Linker –c Option)



The HEX utility does the following:

- Converts the binary COFF file to a standard format ASCII file that an EPROM programmer can understand.
- Identifies the sections to be booted and adds extra control words to those sections to instruct the on-chip boot loader program how to load them to SRAM.

The CPU executes the on-chip boot loader code to copy the boot table from 8-bit EPROM to 32-bit SRAM.

The EPROM programmer burns the 8-bit EPROM device with the boot table (consists of program opcodes, initialized data, plus special control instructions for the boot loader).

Boot loader execution is triggered by a high state on MCBL/MP and a low on one INTx pin during low-to-high transition of RESET.

Depending on which INTx is low when RESET goes high, boot loader starts reading the boot table from one of three memory locations or from the serial port.

The on-chip boot loader uses the CPU to assemble individual bytes of the boot table and to load them directly into SRAM.

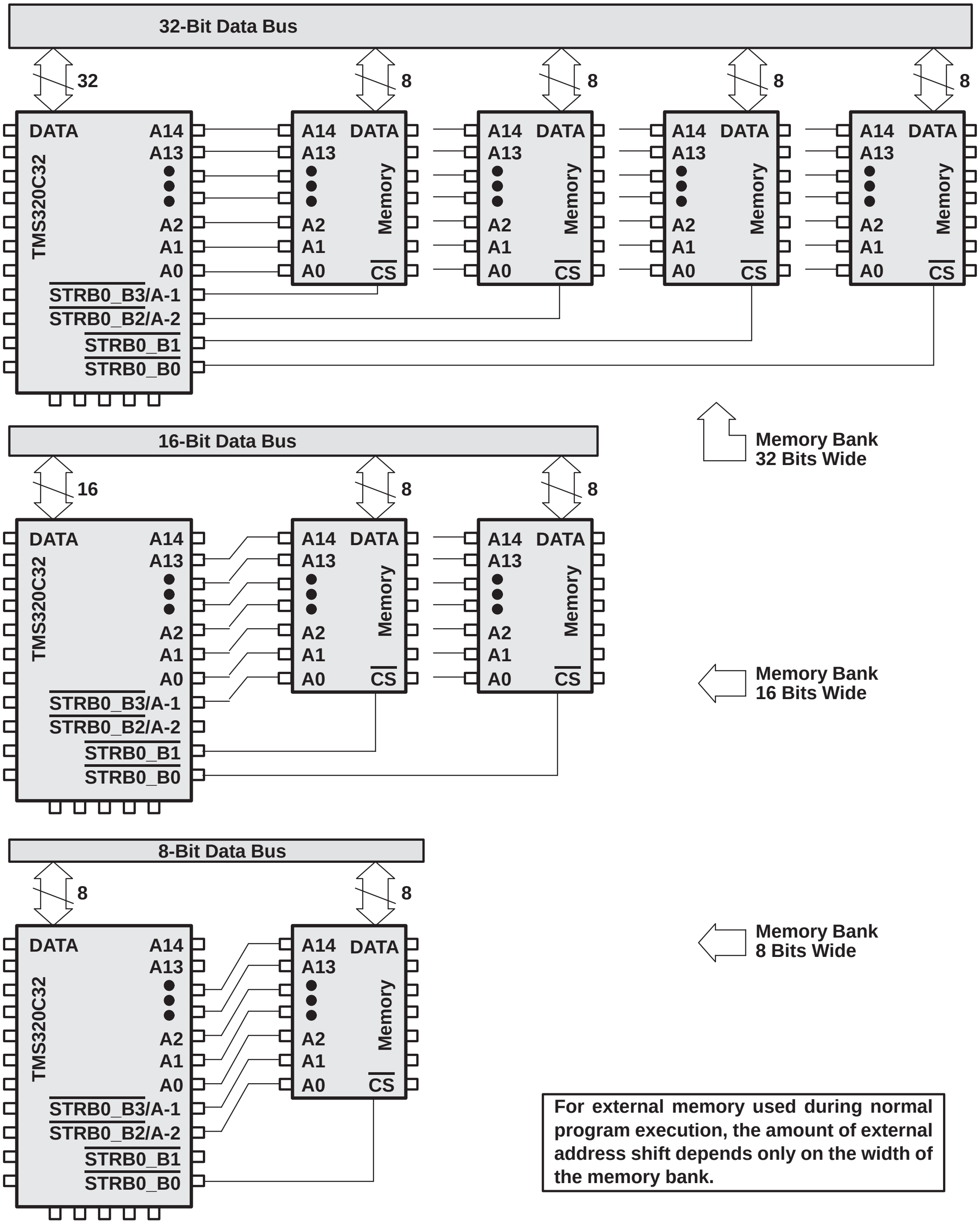
c_int00 is the execution entry point.

TMS320C32 DSP
BL
On-chip bootloader
CPU
RESET
MCBL/MP
INTx
System Reset
VCC
File.out
C Boot
.text section (boot.asm)
File1
.text section
File2
.text section
File1
.cinit section
File2
.cinit section
COFF Format (Binary File)
HEX conv utility
Sections:
.text : BOOT
.cinit : BOOT
File.B1
Bootloader Cntrl
C Boot
code (boot.asm)
File1
code
File2
code
Bootloader Cntrl
File1
data
File2
data
Bootloader Cntrl
Intel Hex Format (ASCII File)
EPROM programmer
EPROM
Bootloader Cntrl
C Boot
code (boot.asm)
File1
code
File2
code
Bootloader Cntrl
File1
data
File2
data
Bootloader Cntrl
8 Bits Wide
SRAM1
C Boot
code (boot.asm)
File1
code
File2
code
32 Bits Wide
SRAM2
File1
data (.bss section)
File2
data (.bss section)
32 Bits Wide

Figure 5. 8-Bit EPROM Boot Using the On-Chip Boot Loader (Linker –cr Option)

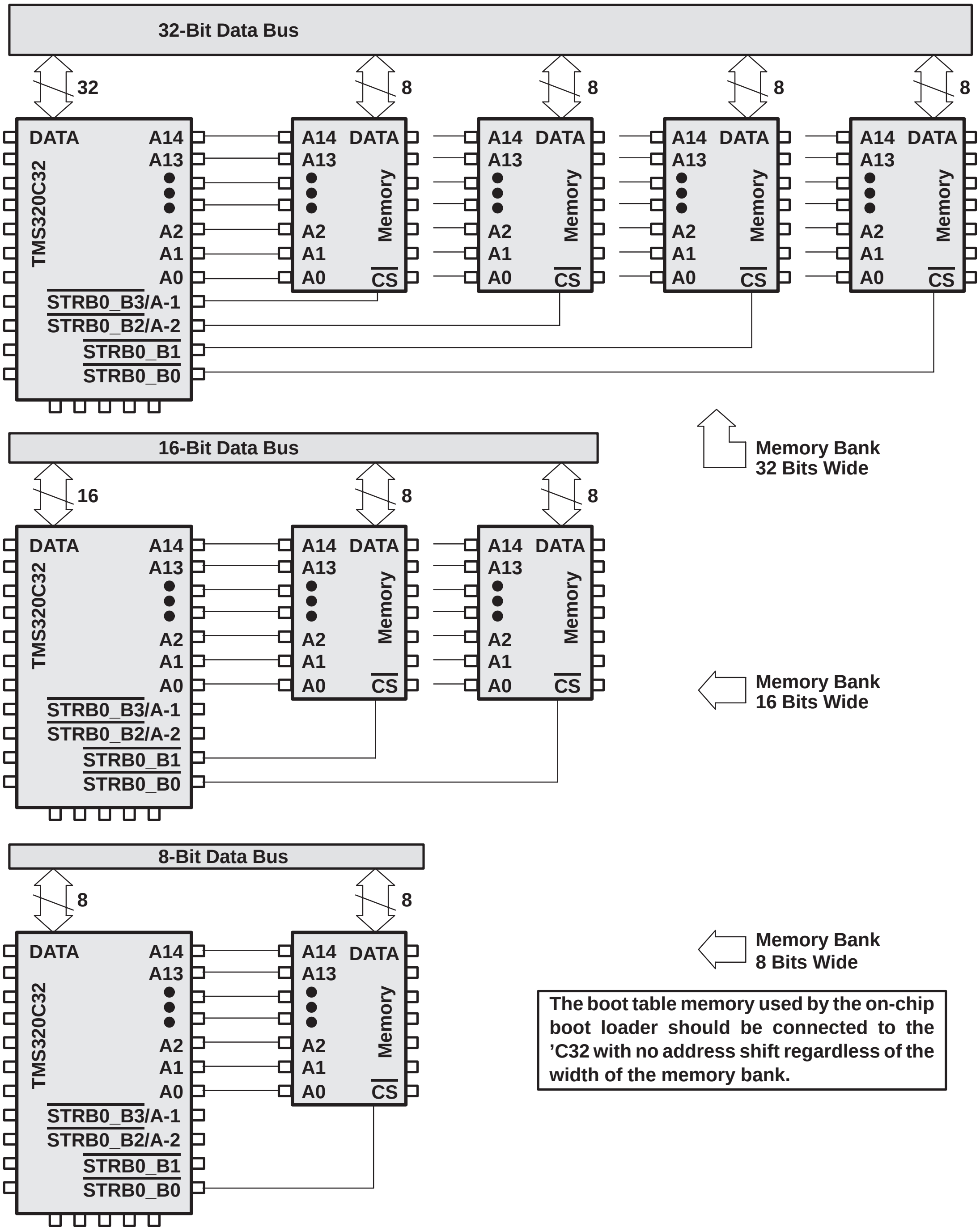
BOOT TABLE MEMORY CONSIDERATIONS

There is a significant difference in the methods of interfacing the external memory holding the boot table and the program/data memory used during normal code execution. The address presented on the ’C32’s pins may be shifted by one or two bits depending on the size of the memory bank (see Figure 6), but the external memory holding the boot table must have *no* address shift relative to the ’C32 address pins regardless of the width of the boot memory (see Figure 7). The boot loader program reads the boot table memory width from the first word of the boot table. It reads the boot table contents as 32-bit data and, depending on the memory width, it reconstructs the program and data before sending it to the memory map. Due to this difference in the address shift, the byte-wide EPROM containing the boot table is not best suited to store normal data unless special hardware is added to handle the address shift.


32-Bit Data Bus
32
8
DATA
A14
A13
A2
A1
A0
TMS320C32
STRB0_B3/A-1
STRB0_B2/A-2
STRB0_B1
STRB0_B0
Memory
CS
Memory Bank
32 Bits Wide
16-Bit Data Bus
16
Memory Bank
16 Bits Wide
8-Bit Data Bus
Memory Bank
8 Bits Wide

For external memory used during normal program execution, the amount of external address shift depends only on the width of the memory bank.

Figure 6. Memory Configuration for Normal Program Execution


32-Bit Data Bus
32
8
8
8
8
DATA
TMS320C32
A14
A13
A2
A1
A0
STRB0_B3/A-1
STRB0_B2/A-2
STRB0_B1
STRB0_B0
A14
A13
A2
A1
A0
DATA
Memory
CS
Memory Bank 32 Bits Wide
16-Bit Data Bus
16
8
8
Memory Bank 16 Bits Wide
8-Bit Data Bus
8
8
Memory Bank 8 Bits Wide

The boot table memory used by the on-chip boot loader should be connected to the ’C32 with no address shift regardless of the width of the memory bank.

Figure 7. Boot Table Memory Configuration

HOST LOAD

While some DSP systems stand alone, others may be embedded DSPs controlled by a host such as a microcontroller or another DSP. During system power up, the DSP boot table may be transferred from the host to the DSP through a serial port or through a byte-wide latch. This eliminates the need for a dedicated boot EPROM on the DSP side of the system. On the host side, the DSP boot table may be temporarily stored in an EPROM prior to the DSP boot. Following reset, the host transfers the boot table to the DSP to initialize it and start program execution.

Boot From Serial Port

If the DSP powers up in the microcomputer/boot loader mode (MCBL/$\overline{\text{MP}}$ high), the low on the $\overline{\text{INT3}}$ pin and high on all other $\overline{\text{INTx}}$ pins causes the on-chip boot loader program to read the boot table from the serial port. Most microcontrollers also feature a serial port, and in many cases the two ports can be connected directly without additional glue logic for an economical host/DSP interface. Following the boot, the serial channel can also be used by the host to send/receive data and to control the operation of the DSP (see Figure 8). Generating the boot table for this example requires linking the object files with the –cr option (RAM model), and then appending the HEX utility's SECTION directive with the :BOOT keyword to identify the COFF sections to be included in the boot table.

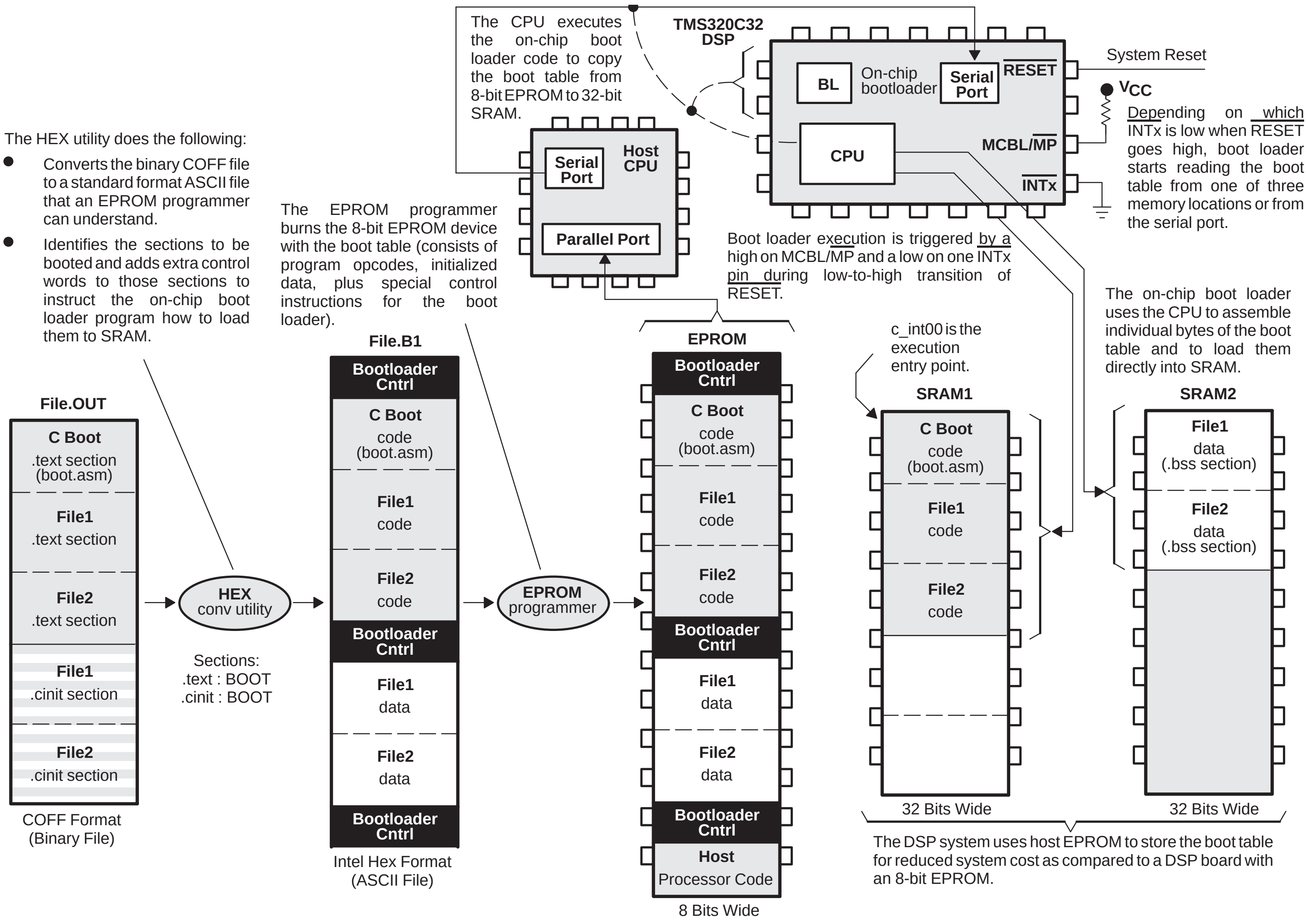
Boot From a Latch

If the host processor does not have a serial port, the DSP can be booted from the host using an 8-bit latch. During the boot operation, the host feeds the boot table bytes to the latch on one side, while the DSP reads the data from the other. Following reset, interrupts 0, 1, and 2 direct the DSP boot loader to the latch address. The same interrupts cause the boot loader to think that it is reading from the parallel port, so some control/decode logic is required to make the DSP think that it is reading from memory instead of from a latch. The same glue logic must also be connected to the host side of the latch to ensure proper data transfer synchronization between two asynchronous systems (see Figure 9). At power up, the DSP boot table most likely resides in the host's EPROM, and the host outputs the boot table to the latch one byte at a time following reset. Creating the boot table for this operation uses the same linker/COFF options as for the host/serial boot and the direct EPROM boot.

Asynchronous Boot From a Communications Port

If the host processor has an asynchronous communications capability, then the 'C32 can make a glueless connection to the host's communication port (see Figure 10). In addition to the data bus, three 'C32 pins are involved in the asynchronous boot: XF0, XF1, and $\overline{\text{IACK}}$. The XF1 pin serves as the data ready input to the 'C32 and XF0 is the data acknowledge.

The $\overline{\text{IACK}}$ pin pulses when there is no valid data present on the data lines (needed for TMS320C4x comm-port interface). For this boot load mode, it is assumed that the host (such as a TMS320C4x) connects directly to the data ready and data acknowledge control lines. The host drives the data ready signal low to indicate to the DSP that the next byte of the boot table has been placed on the data lines. The DSP responds by pulling the data acknowledge signal low after reading the data. When the host sees the data acknowledge signal, it stops driving the data bus and brings the data ready line high. To complete the handshaking transaction, the DSP brings the data acknowledge signal high to request the next byte from the host. The boot table for this type of boot operation is created with the linker –cr option (RAM model) and HEX conversion utility SECTIONS directive :BOOT keyword — the same options used for other boot load procedures involving the on-chip boot loader program.



File.OUT
C Boot
.text section
(boot.asm)
File1
.text section
File2
.text section
File1
.cinit section
File2
.cinit section
COFF Format
(Binary File)
HEX
conv utility
File.B1
Bootloader Cntrl
C Boot
code
(boot.asm)
File1
code
File2
code
Bootloader Cntrl
File1
data
File2
data
Bootloader Cntrl
Intel Hex Format
(ASCII File)
EPROM
programmer
EPROM
Bootloader Cntrl
C Boot
code
(boot.asm)
File1
code
File2
code
Bootloader Cntrl
File1
data
File2
data
Bootloader Cntrl
Host
Processor Code
8 Bits Wide
Serial Port
Host CPU
Parallel Port
TMS320C32 DSP
BL
On-chip bootloader
Serial Port
RESET
CPU
MCBL/MP
INTx
System Reset
VCC
SRAM1
C Boot
code
(boot.asm)
File1
code
File2
code
32 Bits Wide
SRAM2
File1
data
(.bss section)
File2
data
(.bss section)
32 Bits Wide

The HEX utility does the following:

- Converts the binary COFF file to a standard format ASCII file that an EPROM programmer can understand.
- Identifies the sections to be booted and adds extra control words to those sections to instruct the on-chip boot loader program how to load them to SRAM.

Sections:
.text : BOOT
.cinit : BOOT

The EPROM programmer burns the 8-bit EPROM device with the boot table (consists of program opcodes, initialized data, plus special control instructions for the boot loader).

The CPU executes the on-chip boot loader code to copy the boot table from 8-bit EPROM to 32-bit SRAM.

Boot loader execution is triggered by a high on MCBL/$\overline{\text{MP}}$ and a low on one $\overline{\text{INTx}}$ pin during low-to-high transition of $\overline{\text{RESET}}$.

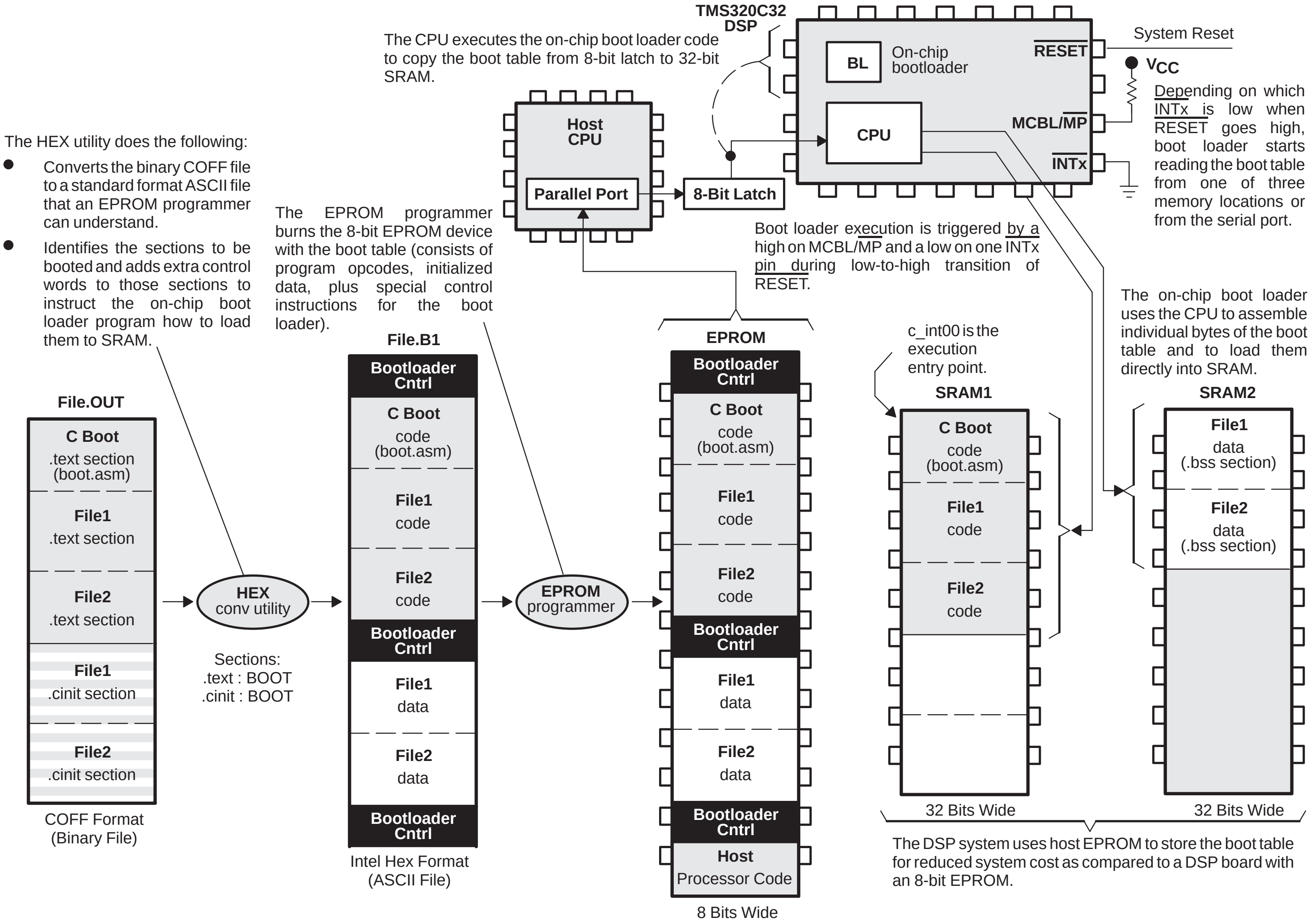
Depending on which $\overline{\text{INTx}}$ is low when $\overline{\text{RESET}}$ goes high, boot loader starts reading the boot table from one of three memory locations or from the serial port.

c_int00 is the execution entry point.

The on-chip boot loader uses the CPU to assemble individual bytes of the boot table and to load them directly into SRAM.

The DSP system uses host EPROM to store the boot table for reduced system cost as compared to a DSP board with an 8-bit EPROM.

Figure 8. Boot From Host Using Serial Port (Linker –cr Option)



The HEX utility does the following:

- Converts the binary COFF file to a standard format ASCII file that an EPROM programmer can understand.
- Identifies the sections to be booted and adds extra control words to those sections to instruct the on-chip boot loader program how to load them to SRAM.

The EPROM programmer burns the 8-bit EPROM device with the boot table (consists of program opcodes, initialized data, plus special control instructions for the boot loader).

The CPU executes the on-chip boot loader code to copy the boot table from 8-bit latch to 32-bit SRAM.

Boot loader execution is triggered by a high on MCBL/MP and a low on one INTx pin during low-to-high transition of RESET.

Depending on which INTx is low when RESET goes high, boot loader starts reading the boot table from one of three memory locations or from the serial port.

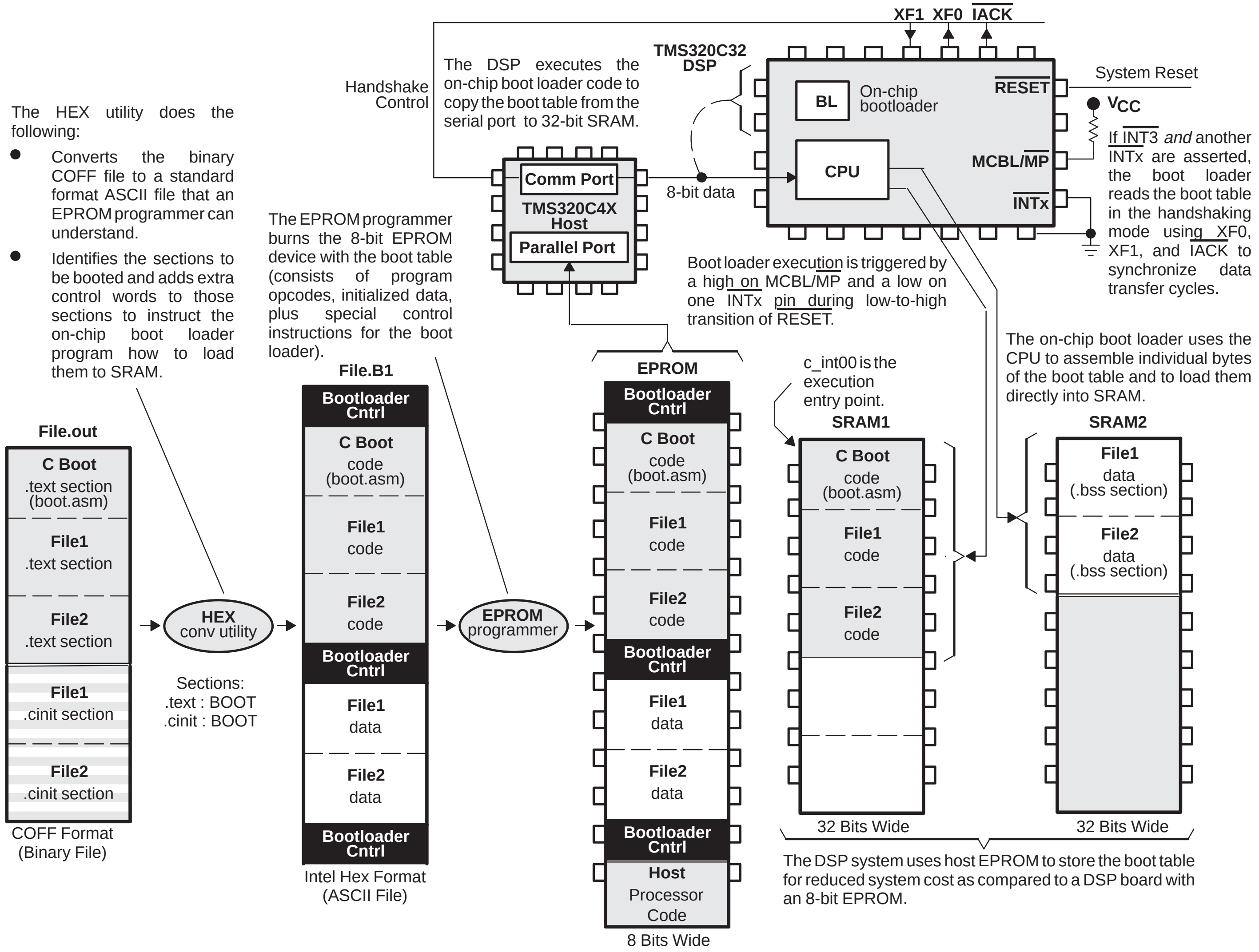
The on-chip boot loader uses the CPU to assemble individual bytes of the boot table and to load them directly into SRAM.

c_int00 is the execution entry point.

The DSP system uses host EPROM to store the boot table for reduced system cost as compared to a DSP board with an 8-bit EPROM.

TMS320C32 DSP
BL
On-chip bootloader
CPU
RESET
MCBL/MP
INTx
System Reset
V_{CC}
Host CPU
Parallel Port
8-Bit Latch
File.OUT
C Boot
.text section
(boot.asm)
File1
.text section
File2
.text section
File1
.cinit section
File2
.cinit section
COFF Format
(Binary File)
HEX
conv utility
Sections:
.text : BOOT
.cinit : BOOT
File.B1
Bootloader Cntrl
C Boot
code
(boot.asm)
File1
code
File2
code
Bootloader Cntrl
File1
data
File2
data
Bootloader Cntrl
Intel Hex Format
(ASCII File)
EPROM
programmer
EPROM
Bootloader Cntrl
C Boot
code
(boot.asm)
File1
code
File2
code
Bootloader Cntrl
File1
data
File2
data
Bootloader Cntrl
Host
Processor Code
8 Bits Wide
SRAM1
C Boot
code
(boot.asm)
File1
code
File2
code
32 Bits Wide
SRAM2
File1
data
(.bss section)
File2
data
(.bss section)
32 Bits Wide

Figure 9. Boot From Host Using an 8-Bit Latch (Linker –cr Option)



XF1
XF0
IACK
TMS320C32 DSP
BL
On-chip bootloader
CPU
RESET
System Reset
V_{CC}
MCBL/MP
INTx
Handshake Control
Comm Port
TMS320C4X Host
Parallel Port
8-bit data
File.out
C Boot
.text section (boot.asm)
File1
.text section
File2
.text section
File1
.cinit section
File2
.cinit section
COFF Format (Binary File)
HEX conv utility
Sections:
.text : BOOT
.cinit : BOOT
File.B1
Bootloader Cntrl
C Boot
code (boot.asm)
File1
code
File2
code
Bootloader Cntrl
File1
data
File2
data
Bootloader Cntrl
Intel Hex Format (ASCII File)
EPROM programmer
EPROM
Bootloader Cntrl
C Boot
code (boot.asm)
File1
code
File2
code
Bootloader Cntrl
File1
data
File2
data
Bootloader Cntrl
Host
Processor Code
8 Bits Wide
SRAM1
C Boot
code (boot.asm)
File1
code
File2
code
32 Bits Wide
SRAM2
File1
data (.bss section)
File2
data (.bss section)
32 Bits Wide

The HEX utility does the following:

- Converts the binary COFF file to a standard format ASCII file that an EPROM programmer can understand.
- Identifies the sections to be booted and adds extra control words to those sections to instruct the on-chip boot loader program how to load them to SRAM.

The EPROM programmer burns the 8-bit EPROM device with the boot table (consists of program opcodes, initialized data, plus special control instructions for the boot loader).

The DSP executes the on-chip boot loader code to copy the boot table from the serial port to 32-bit SRAM.

If INT3 *and* another INTx are asserted, the boot loader reads the boot table in the handshaking mode using XF0, XF1, and IACK to synchronize data transfer cycles.

Boot loader execution is triggered by a high on MCBL/MP and a low on one INTx pin during low-to-high transition of RESET.

The on-chip boot loader uses the CPU to assemble individual bytes of the boot table and to load them directly into SRAM.

c_int00 is the execution entry point.

The DSP system uses host EPROM to store the boot table for reduced system cost as compared to a DSP board with an 8-bit EPROM.

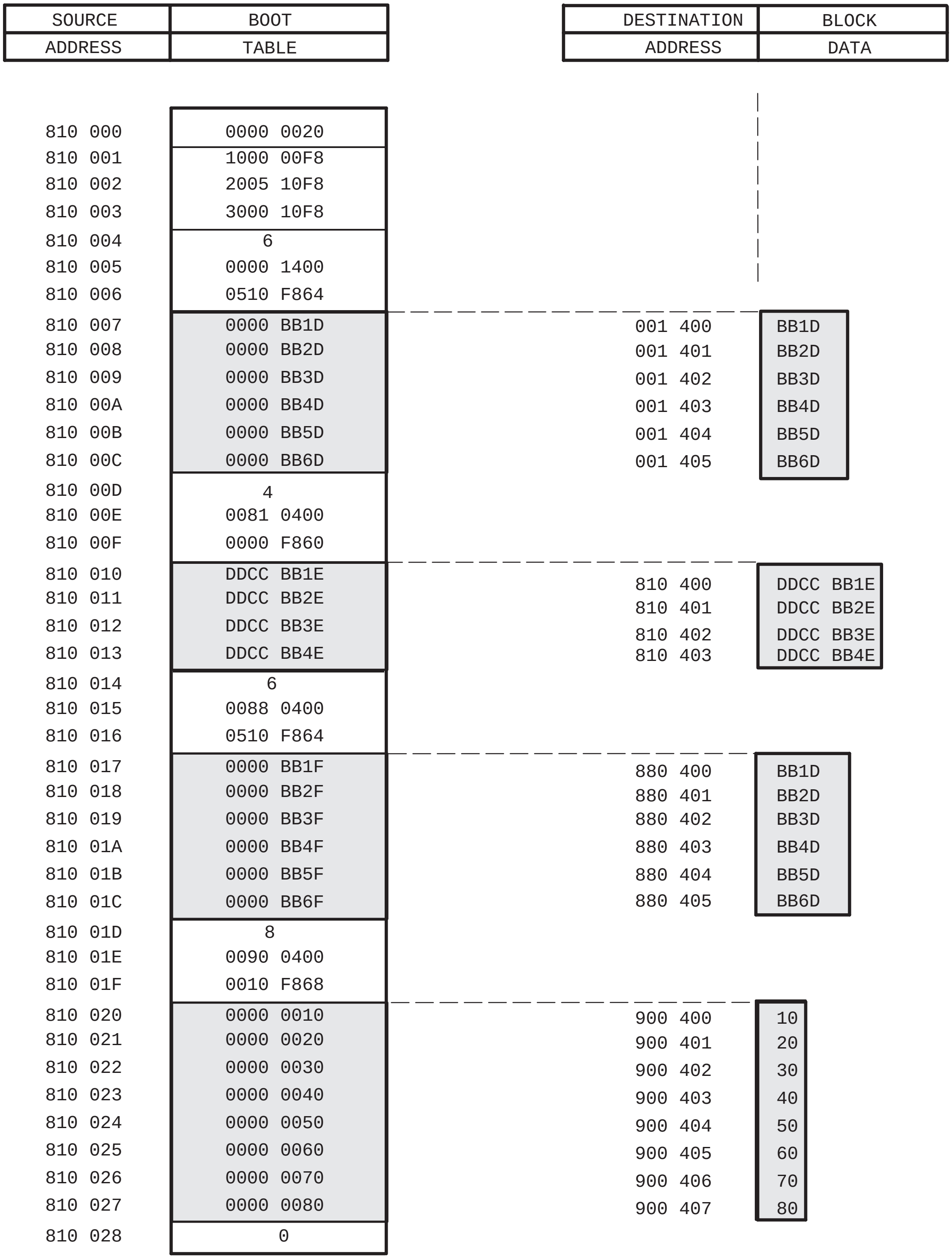
Figure 10. Boot From Host Using Asynchronous Communications Port (Linker –cr Option)

APPENDIX A—THE ’C32 BOOT TABLE EXAMPLES

Figures 11 through 14 show four instances of the boot table, each containing four blocks. The destination for the first and third block is 16-bit $\overline{\text{STRB0}}$ memory. The second block is booted to the 32-bit $\overline{\text{IOSTRB}}$ memory. Block 4 is destined for the 8-bit memory in the $\overline{\text{STRB1}}$ portion of the memory map.

Each figure represents a boot from a different source medium. In Figure 11, the boot table resides in the 32-bit $\overline{\text{IOSTRB}}$ memory (pointed to by $\overline{\text{INT1}}$ pin low after reset in the microcontroller/boot loader mode). The boot table in Figure 12 is stored in the 16-bit $\overline{\text{STRB0}}$ memory (pointed to by $\overline{\text{INT0}}$). The boot table in Figure 13 resides in the 8-bit $\overline{\text{STRB1}}$ memory (pointed to by $\overline{\text{INT2}}$). The final example, shown in Figure 14, represents the boot table stored in the host memory before being sent to the ’C32 over the serial port. Unlike the boot from memory, the serial port boot table omits the memory width control word from the beginning of the table.

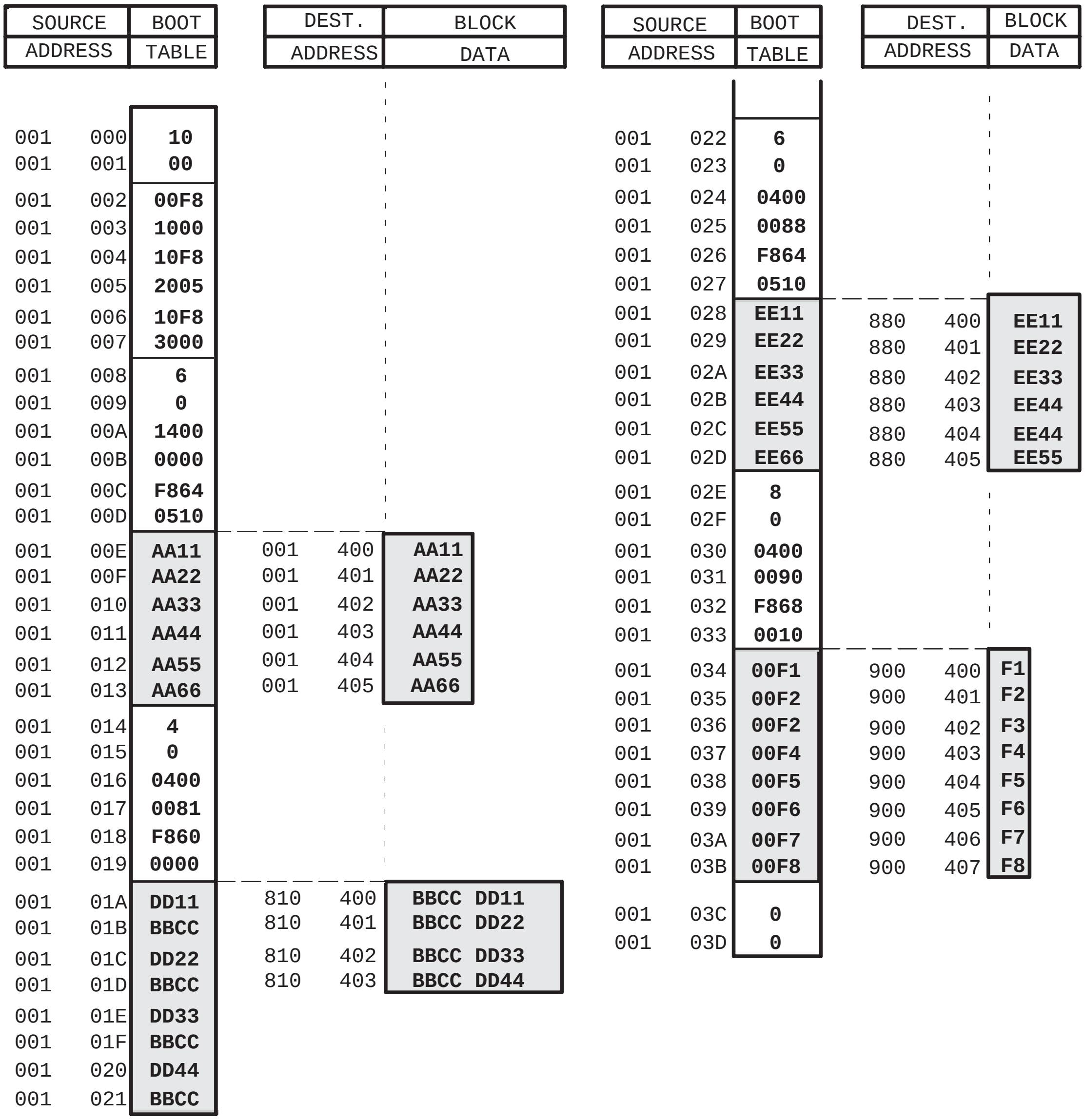
The shaded areas of the boot table examples represent the contents of the individual blocks of code or data. The unshaded portions are the control words that instruct the boot loader program to transfer the blocks to the memory map.



SOURCE ADDRESS	BOOT TABLE
810 000	0000 0020
810 001	1000 00F8
810 002	2005 10F8
810 003	3000 10F8
810 004	6
810 005	0000 1400
810 006	0510 F864
810 007	0000 BB1D
810 008	0000 BB2D
810 009	0000 BB3D
810 00A	0000 BB4D
810 00B	0000 BB5D
810 00C	0000 BB6D
810 00D	4
810 00E	0081 0400
810 00F	0000 F860
810 010	DDCC BB1E
810 011	DDCC BB2E
810 012	DDCC BB3E
810 013	DDCC BB4E
810 014	6
810 015	0088 0400
810 016	0510 F864
810 017	0000 BB1F
810 018	0000 BB2F
810 019	0000 BB3F
810 01A	0000 BB4F
810 01B	0000 BB5F
810 01C	0000 BB6F
810 01D	8
810 01E	0090 0400
810 01F	0010 F868
810 020	0000 0010
810 021	0000 0020
810 022	0000 0030
810 023	0000 0040
810 024	0000 0050
810 025	0000 0060
810 026	0000 0070
810 027	0000 0080
810 028	0

DESTINATION ADDRESS	BLOCK DATA
001 400	BB1D
001 401	BB2D
001 402	BB3D
001 403	BB4D
001 404	BB5D
001 405	BB6D
810 400	DDCC BB1E
810 401	DDCC BB2E
810 402	DDCC BB3E
810 403	DDCC BB4E
880 400	BB1D
880 401	BB2D
880 402	BB3D
880 403	BB4D
880 404	BB5D
880 405	BB6D
900 400	10
900 401	20
900 402	30
900 403	40
900 404	50
900 405	60
900 406	70
900 407	80

Figure 11. Boot From a 32-Bit Wide ROM to 8-, 16-, and 32-Bit Wide RAM



SOURCE ADDRESS	BOOT TABLE
001 000	10
001 001	00
001 002	00F8
001 003	1000
001 004	10F8
001 005	2005
001 006	10F8
001 007	3000
001 008	6
001 009	0
001 00A	1400
001 00B	0000
001 00C	F864
001 00D	0510
001 00E	AA11
001 00F	AA22
001 010	AA33
001 011	AA44
001 012	AA55
001 013	AA66
001 014	4
001 015	0
001 016	0400
001 017	0081
001 018	F860
001 019	0000
001 01A	DD11
001 01B	BBCC
001 01C	DD22
001 01D	BBCC
001 01E	DD33
001 01F	BBCC
001 020	DD44
001 021	BBCC

DEST. ADDRESS	BLOCK DATA
001 400	AA11
001 401	AA22
001 402	AA33
001 403	AA44
001 404	AA55
001 405	AA66
810 400	BBCC DD11
810 401	BBCC DD22
810 402	BBCC DD33
810 403	BBCC DD44

SOURCE ADDRESS	BOOT TABLE
001 022	6
001 023	0
001 024	0400
001 025	0088
001 026	F864
001 027	0510
001 028	EE11
001 029	EE22
001 02A	EE33
001 02B	EE44
001 02C	EE55
001 02D	EE66
001 02E	8
001 02F	0
001 030	0400
001 031	0090
001 032	F868
001 033	0010
001 034	00F1
001 035	00F2
001 036	00F2
001 037	00F4
001 038	00F5
001 039	00F6
001 03A	00F7
001 03B	00F8
001 03C	0
001 03D	0

DEST. ADDRESS	BLOCK DATA
880 400	EE11
880 401	EE22
880 402	EE33
880 403	EE44
880 404	EE44
880 405	EE55
900 400	F1
900 401	F2
900 402	F3
900 403	F4
900 404	F5
900 405	F6
900 406	F7
900 407	F8

Figure 12. Boot From a 16-Bit Wide ROM to 8-, 16-, and 32-Bit Wide RAM

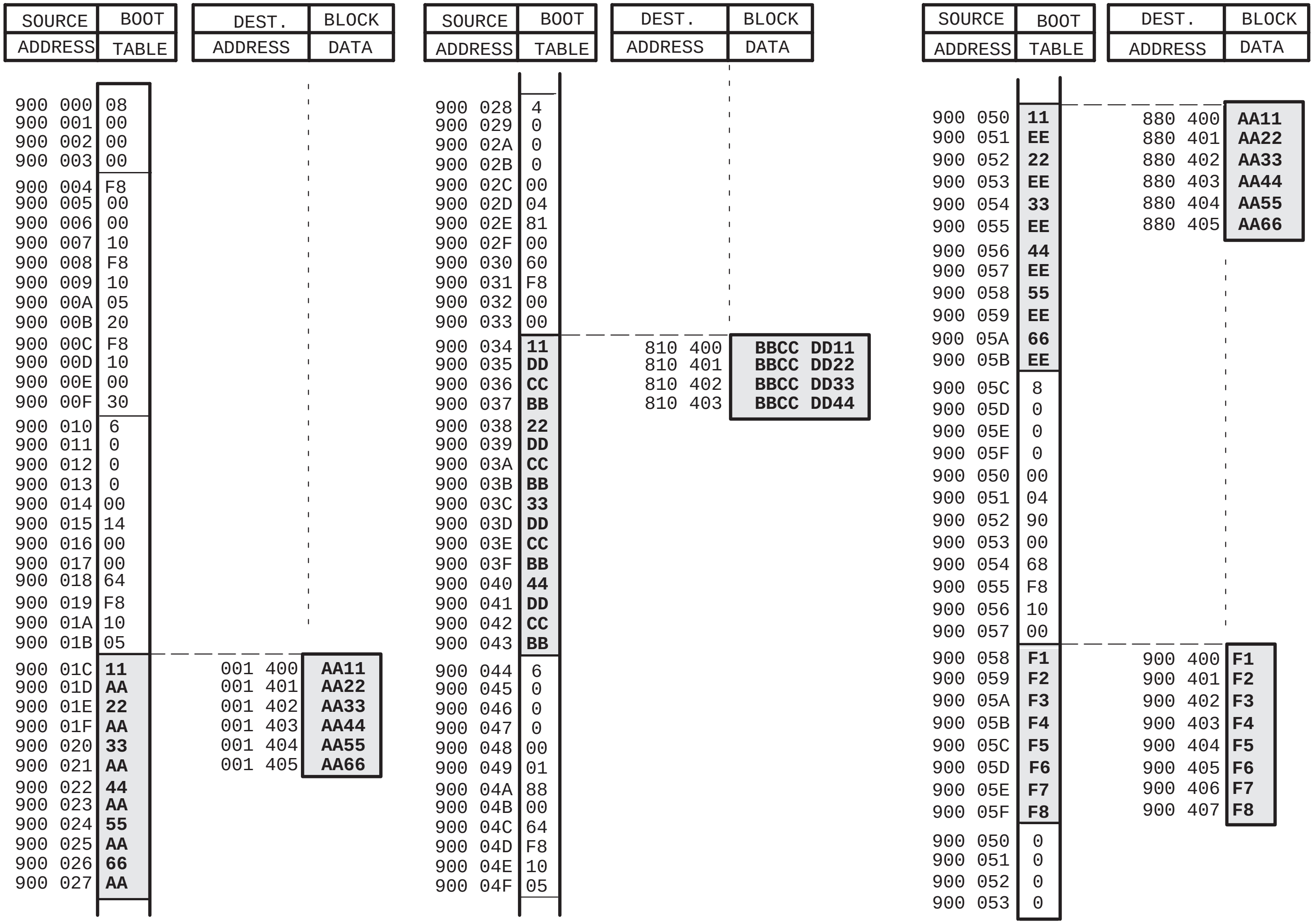
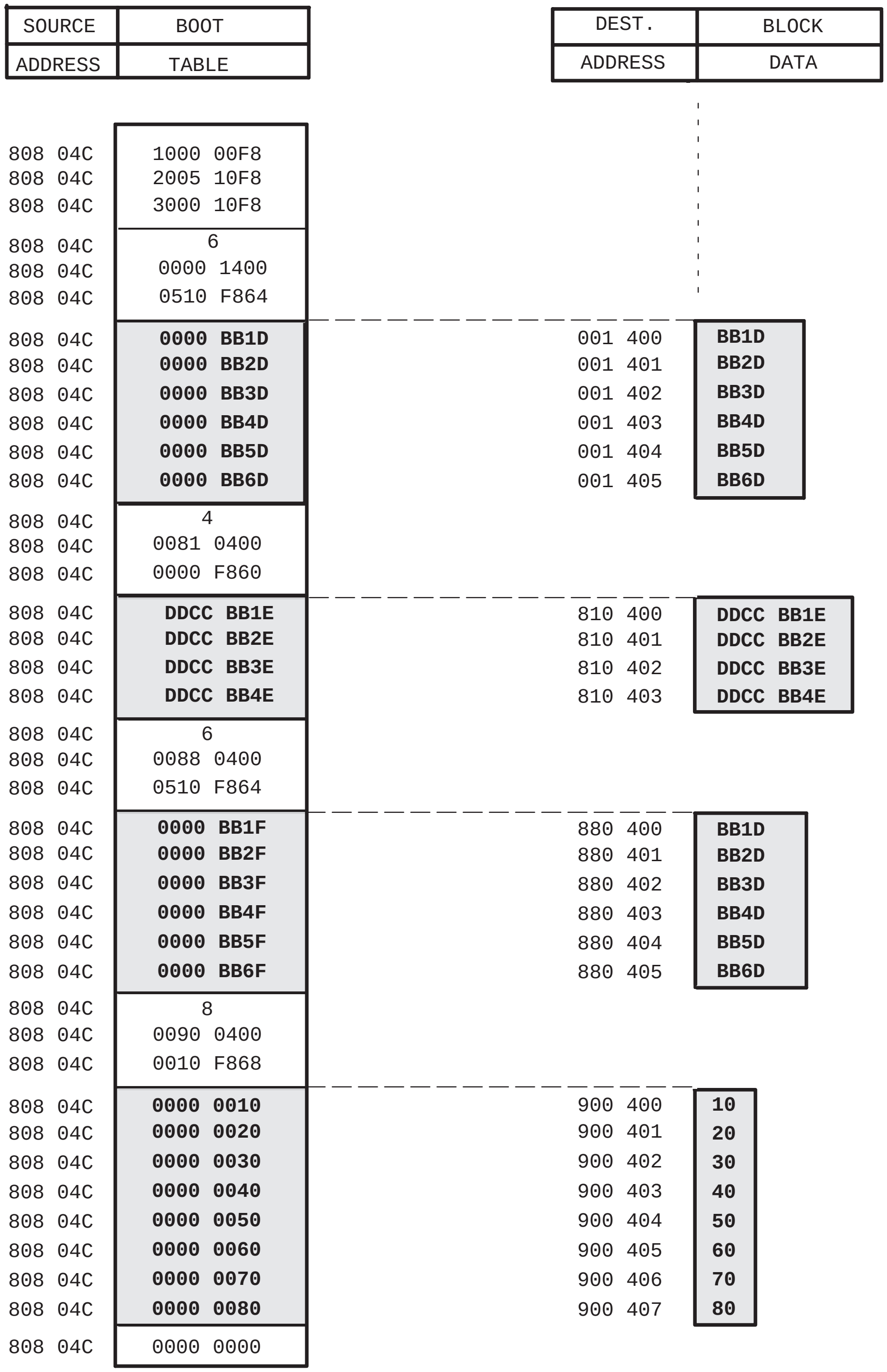

SOURCE ADDRESS
BOOT TABLE
DEST. ADDRESS
BLOCK DATA
900 000 08
900 001 00
900 002 00
900 003 00
900 004 F8
900 005 00
900 006 00
900 007 10
900 008 F8
900 009 10
900 00A 05
900 00B 20
900 00C F8
900 00D 10
900 00E 00
900 00F 30
900 010 6
900 011 0
900 012 0
900 013 0
900 014 00
900 015 14
900 016 00
900 017 00
900 018 64
900 019 F8
900 01A 10
900 01B 05
900 01C 11
900 01D AA
900 01E 22
900 01F AA
900 020 33
900 021 AA
900 022 44
900 023 AA
900 024 55
900 025 AA
900 026 66
900 027 AA
001 400 AA11
001 401 AA22
001 402 AA33
001 403 AA44
001 404 AA55
001 405 AA66
SOURCE ADDRESS
BOOT TABLE
DEST. ADDRESS
BLOCK DATA
900 028 4
900 029 0
900 02A 0
900 02B 0
900 02C 00
900 02D 04
900 02E 81
900 02F 00
900 030 60
900 031 F8
900 032 00
900 033 00
900 034 11
900 035 DD
900 036 CC
900 037 BB
900 038 22
900 039 DD
900 03A CC
900 03B BB
900 03C 33
900 03D DD
900 03E CC
900 03F BB
900 040 44
900 041 DD
900 042 CC
900 043 BB
900 044 6
900 045 0
900 046 0
900 047 0
900 048 00
900 049 01
900 04A 88
900 04B 00
900 04C 64
900 04D F8
900 04E 10
900 04F 05
810 400 BBCC DD11
810 401 BBCC DD22
810 402 BBCC DD33
810 403 BBCC DD44
SOURCE ADDRESS
BOOT TABLE
DEST. ADDRESS
BLOCK DATA
900 050 11
900 051 EE
900 052 22
900 053 EE
900 054 33
900 055 EE
900 056 44
900 057 EE
900 058 55
900 059 EE
900 05A 66
900 05B EE
900 05C 8
900 05D 0
900 05E 0
900 05F 0
900 050 00
900 051 04
900 052 90
900 053 00
900 054 68
900 055 F8
900 056 10
900 057 00
900 058 F1
900 059 F2
900 05A F3
900 05B F4
900 05C F5
900 05D F6
900 05E F7
900 05F F8
900 050 0
900 051 0
900 052 0
900 053 0
880 400 AA11
880 401 AA22
880 402 AA33
880 403 AA44
880 404 AA55
880 405 AA66
900 400 F1
900 401 F2
900 402 F3
900 403 F4
900 404 F5
900 405 F6
900 406 F7
900 407 F8

Figure 13. Boot From a Byte Wide ROM to 8-, 16-, and 32-Bit Wide RAM



SOURCE ADDRESS	BOOT TABLE
808 04C	1000 00F8
808 04C	2005 10F8
808 04C	3000 10F8
808 04C	6
808 04C	0000 1400
808 04C	0510 F864
808 04C	**0000 BB1D**
808 04C	**0000 BB2D**
808 04C	**0000 BB3D**
808 04C	**0000 BB4D**
808 04C	**0000 BB5D**
808 04C	**0000 BB6D**
808 04C	4
808 04C	0081 0400
808 04C	0000 F860
808 04C	**DDCC BB1E**
808 04C	**DDCC BB2E**
808 04C	**DDCC BB3E**
808 04C	**DDCC BB4E**
808 04C	6
808 04C	0088 0400
808 04C	0510 F864
808 04C	**0000 BB1F**
808 04C	**0000 BB2F**
808 04C	**0000 BB3F**
808 04C	**0000 BB4F**
808 04C	**0000 BB5F**
808 04C	**0000 BB6F**
808 04C	8
808 04C	0090 0400
808 04C	0010 F868
808 04C	**0000 0010**
808 04C	**0000 0020**
808 04C	**0000 0030**
808 04C	**0000 0040**
808 04C	**0000 0050**
808 04C	**0000 0060**
808 04C	**0000 0070**
808 04C	**0000 0080**
808 04C	0000 0000

DEST. ADDRESS	BLOCK DATA
001 400	**BB1D**
001 401	**BB2D**
001 402	**BB3D**
001 403	**BB4D**
001 404	**BB5D**
001 405	**BB6D**
810 400	**DDCC BB1E**
810 401	**DDCC BB2E**
810 402	**DDCC BB3E**
810 403	**DDCC BB4E**
880 400	**BB1D**
880 401	**BB2D**
880 402	**BB3D**
880 403	**BB4D**
880 404	**BB5D**
880 405	**BB6D**
900 400	**10**
900 401	**20**
900 402	**30**
900 403	**40**
900 404	**50**
900 405	**60**
900 406	**70**
900 407	**80**

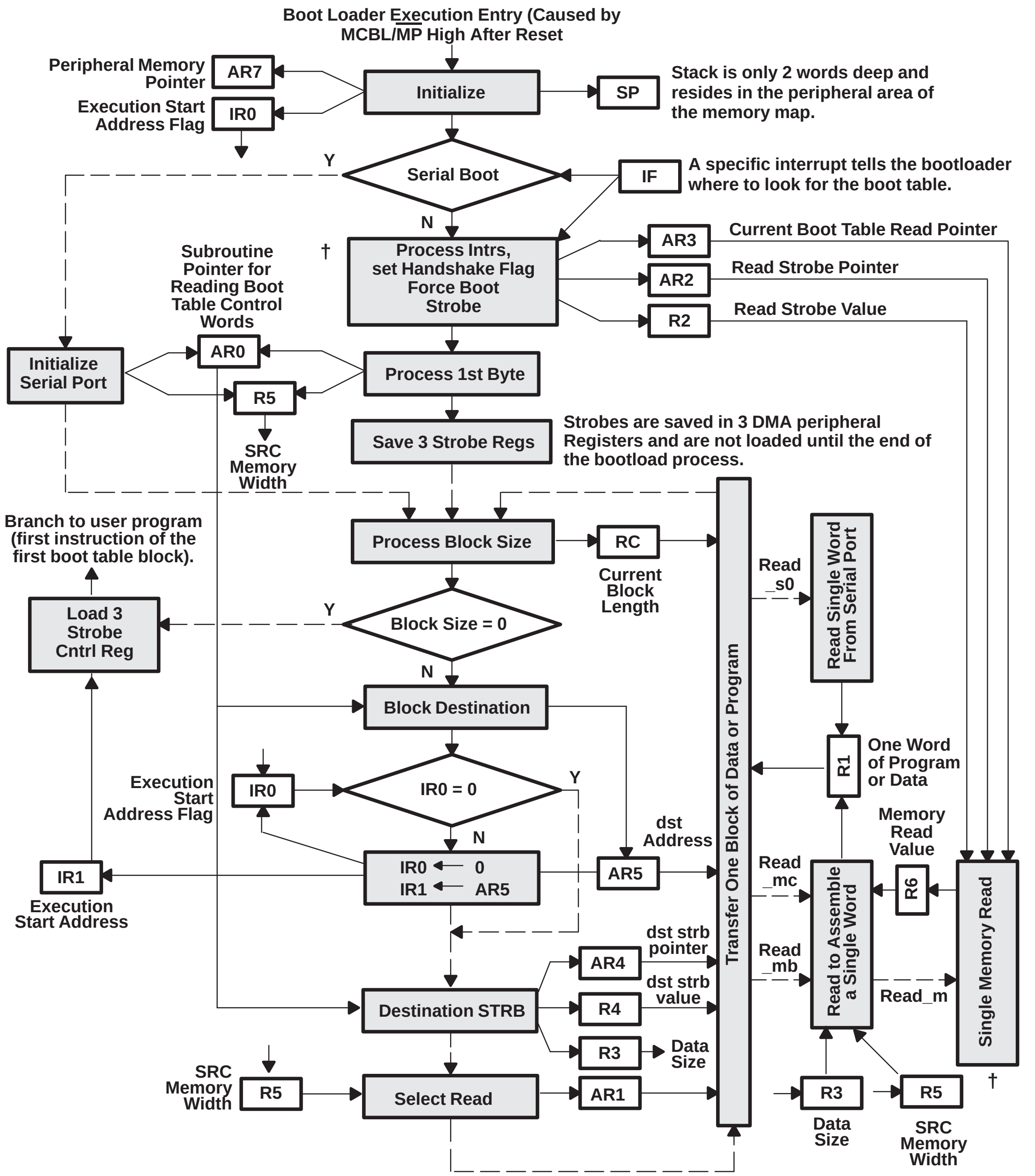
Figure 14. Boot From Serial Port to 8-, 16-, and 32-Bit Wide RAM

APPENDIX B—THE ’C32 BOOT LOADER PROGRAM

This appendix describes the on-chip boot loader program that initializes the DSP system following power up or reset. Figure 15 shows the program flowchart with shaded areas representing portions of code and the square shapes depicting registers containing data.

’C32 Boot Loader Opcodes

The boot loader reads the boot table from one of three memory locations (1000h, 810000h, 900000h) or from the serial port. The boot loader processes each block of the boot table separately. First, the words of program or data are assembled from bytes (or half-words). The assembled words are then written to their destinations one at a time. Each block can be transferred to any memory address range within the memory map. The blocks in the boot table are preceded by three control words: block size, destination address, and strobe control register value. The boot loader ends execution when it finds a zero for the size of the next block. At that point, it initializes the three strobe control registers and branches to the first instruction of the first block. For that reason, the first boot table block always contains program and not data. For the description of the boot loader operation, refer to the program listing included in this appendix and the *TMS320C32 User’s Guide: Addendum to the TMS320C32 User’s Guide* (literature number SPRU132). Table 1 lists the ’C32 boot loader opcodes.



Boot Loader Execution Entry (Caused by MCBL/MP High After Reset
Peripheral Memory Pointer
AR7
Execution Start Address Flag
IR0
Initialize
SP
Stack is only 2 words deep and resides in the peripheral area of the memory map.
Y
Serial Boot
IF
A specific interrupt tells the bootloader where to look for the boot table.
N
Subroutine Pointer for Reading Boot Table Control Words
†
Process Intrs, set Handshake Flag Force Boot Strobe
AR3
Current Boot Table Read Pointer
AR2
Read Strobe Pointer
R2
Read Strobe Value
Initialize Serial Port
AR0
Process 1st Byte
R5
SRC Memory Width
Save 3 Strobe Regs
Strobes are saved in 3 DMA peripheral Registers and are not loaded until the end of the bootload process.
Branch to user program (first instruction of the first boot table block).
Process Block Size
RC
Current Block Length
Read _s0
Read Single Word From Serial Port
Load 3 Strobe Cntrl Reg
Y
Block Size = 0
N
Block Destination
R1
One Word of Program or Data
Execution Start Address Flag
IR0
IR0 = 0
Y
N
Memory Read Value
IR1
Execution Start Address
IR0 ← 0
IR1 ← AR5
dst Address
AR5
Read _mc
R6
Transfer One Block of Data or Program
Read to Assemble a Single Word
Single Memory Read
dst strb pointer
AR4
Read _mb
dst strb value
Destination STRB
R4
Read_m
R3
Data Size
SRC Memory Width
R5
Select Read
AR1
R3
R5
†
Data Size
SRC Memory Width

[†] Handshake mode is enabled by setting the IOXF0 bit of IOF register to 1 when $\overline{\text{INT3}}$ *and* any of $\overline{\text{INT2}}$, $\overline{\text{INT1}}$, or $\overline{\text{INT0}}$ are asserted following reset.

Figure 15. TMS320C32 Boot Loader Program

Table 1. TMS320C32 Boot Loader Opcodes

ADDRESS	OPCODE	ADDRESS	OPCODE	ADDRESS	OPCODE	ADDRESS	OPCODE
00000000	00000045	00000034	00000000	00000068	1a660001	0000009d	086800a7
00000001	00000000	00000035	00000000	00000069	6a060004	0000009e	08650000
00000002	00000000	00000036	00000000	0000006a	09e6ffff	0000009f	08620000
00000003	00000000	00000037	00000000	0000006b	09eeffff	000000a0	080a000f
00000004	00000000	00000038	00000000	0000006c	09e50001	000000a1	08600111
00000005	00000000	00000039	00000000	0000006d	6a00fffa	000000a2	15400743
00000006	00000000	0000003A	00000000	0000006e	186e0002	000000a3	08670a30
00000007	00000000	0000003B	00000000	0000006f	04ee0000	000000a4	09e70010
00000008	00000000	0000003C	00000000	00000070	6a070002	000000a5	15470740
00000009	00000000	0000003D	00000000	00000071	72000053	000000a6	6a00ffcc
0000000A	00000000	0000003E	00000000	00000072	6f80fffe	000000a7	1a770020
0000000B	00000000	0000003F	00000000	00000073	70000008	000000a8	6a05fffe
0000000C	00000000	00000040	00000000	00000074	15410704	000000a9	02f70fdf
0000000D	00000000	00000041	00000000	00000075	70000008	000000aa	0841074c
0000000E	00000000	00000042	00000000	00000076	15410706	000000ab	78800000
0000000F	00000000	00000043	00000000	00000077	70000008	000000ac	08630003
00000010	00000000	00000044	00000000	00000078	15410708	000000ad	08730001
00000011	00000000	00000045	086f4040	00000079	70000008	000000ae	09930005
00000012	00000000	00000046	09ef0009	0000007a	08010001	000000af	18730001
00000013	00000000	00000047	08740023	0000007b	6a060007	000000b0	080e0003
00000014	00000000	00000048	1014000f	0000007c	08400704	000000b1	026e0001
00000015	00000000	00000049	0871ffff	0000007d	15400760	000000b2	09ee0003
00000016	00000000	0000004a	08000017	0000007e	08400706	000000b3	08000005
00000017	00000000	0000004b	02e0000f	0000007f	15400764	000000b4	04e00001
00000018	00000000	0000004c	04e00008	00000080	08400708	000000b5	6a050003
00000019	00000000	0000004d	6a05004f	00000081	15400768	000000b6	09e0ffff
0000001A	00000000	0000004e	080a000f	00000082	68000012	000000b7	09eeffff
0000001B	00000000	0000004f	026a0060	00000083	081b0001	000000b8	6a00fffb
0000001C	00000000	00000050	1a600004	00000084	187b0001	000000b9	186e0001
0000001D	00000000	00000051	536b4080	00000085	70000008	000000ba	08600000
0000001E	00000000	00000052	6a060008	00000086	080d0001	000000bb	08610000
0000001F	00000000	00000053	026a0004	00000087	4f100000	000000bc	02740003
00000020	00000000	00000054	1a600001	00000088	5312000d	000000bd	72000007
00000021	00000000	00000055	536b0008	00000089	53710000	000000be	18740003
00000022	00000000	00000056	6a060004	0000008a	70000008	000000bf	21871306
00000023	00000000	00000057	026a0004	0000008b	08040001	000000c0	09870000
00000024	00000000	00000058	1a600004	0000008c	02e1006c	000000c1	10010007
00000025	00000000	00000059	536b4800	0000008d	258c010f	000000c2	02000005
00000026	00000000	0000005a	6a05ffef	0000008e	09e4fff8	000000c3	6f80fff8
00000027	00000000	0000005b	1a600008	0000008f	08030004	000000c4	78800000
00000028	00000000	0000005c	6a050002	00000090	09e3fff0	000000c5	1a780002
00000029	00000000	0000005d	1a780080	00000091	02e30003	000000c6	1542c200*
0000002A	00000000	0000005e	08780006	00000092	1a61000c	000000c7	6a060002*
0000002B	00000000	0000005f	0862000f	00000093	52e30003	000000c8	08462301*
0000002C	00000000	00000060	09e20010	00000094	04e50000	000000c9	78800000*
0000002D	00000000	00000061	1042c200	00000095	52e900a7	000000ca	1b40c700*
0000002E	00000000	00000062	1542c200	00000096	536900ad	000000cb	1a780080*
0000002F	00000000	00000063	09eb0009	00000097	6400009b	000000cc	6a06fffd*
00000030	00000000	00000064	086800ac	00000098	70000009	000000cd	08462301*
00000031	00000000	00000065	08650001	00000099	1544c400	000000ce	08780002*
00000032	00000000	00000066	086e0020	0000009a	0c800000	000000cf	1a780080*
00000033	00000000	00000067	7200005d	0000009b	15412501**	000000d0	6a05fffe*
				0000009c	6a00ffdc	000000d1	08780006*
						000000d2	78800000*

```
*******************************************************************
* C32BOOT – TMS320C32 BOOT LOADER PROGRAM    (143 words)     March–96
*      (C) COPYRIGHT TEXAS INSTRUMENTS INCORPORATED, 1994      v.27
*=================================================================*
*
* NOTE:
*
*  1. Following device reset, the program waits for an external
*     interrupt. The interrupt type determines the initial address
*     from which the boot loader will start loading the boot table
*     to the destination memory:
*
```

INTERRUPT PIN	BOOT TABLE START ADDRESS	BOOT SOURCE
INTR0	1000h (STRB0	P_PORT
INTR1	810000h (IOSTRB)	P_PORT
INTR2	900000h (STRB1)	P_PORT
INTR3	80804Ch (sport0 Rx)	SERIAL
INTR0 and INT3	1000h (STRB0) ASYNC	PPORT,XF0/XF1
INTR1 and INT3	810000h (IOSTRB) ASYNC	PPORT,XF0/XF1
INTR2 and INT3	900000h (STRB1) ASYNC	PPORT,XF0/XF1

```
*     If INT3 is asserted together with (INT2 or INT1 or INT0)
*     following reset, that indicates that the boot table is to
*     be read asynchronously from EPROM using pins XF0 and XF1 for
*     handshaking. The handshaking protocol assumes that the data
*     ready signal generated by the host arrives through pin XF1.
*     The data acknowledge signal is output from the C32 on pin XF0.
*     Both signals are active low. The C32 will continuously toggle
*     the IACK signal while waiting for the host to assert data ready
*     signal (pin XF1).
*
*  2. The boot operation involves transfer of one or more source
*     blocks from the boot media to the destination memory. The block
*     structure of the boot table serves the purpose of distributing
*     the source data/program among different memory spaces. Each
*     block is preceded by several 32-bit control words describing
*     the block contents to the boot loader program.
*
*  3. When loading from serial port, the boot loader reads the source
*     data/program and writes it to the destination memory. There is
*     only one way to read the serial port. When loading from EPROM,
*     however, there are 4 ways to read and assemble the
*     source contents, depending on the width of boot memory and the
*     size of the program/data being transferred. Because there is a
*     possibility that reads and writes can span the same STRB space,
*     the boot loader loads the appropriate STRB control registers
*     before each read and write.
*
```

```
*  4. If the boot source is EPROM whose physical width is less than
*     32 bits, the physical interface of the EPROM device(s) to the
*     processor should be the same as that of the 32-bit interface.
*     (This involves a specific connection to C32’s strobe and
*     address signals). The reason for such arrangement is that
*     to function properly, the boot loader program always
*     expects 32-bit data from 32-bit wide memory during the boot
*     load operation. Valid boot EPROM widths are : 1, 2, 4, 8, 16
*     and 32 bits.
*  5. A single source block cannot cross STRB boundaries. For
*     example, its destination cannot overlap STRB0 space and IOSTRB
*     space. Additionally, all of the destination addresses of a
*     single source block should reside in physical memory of the
*     same width. It is also not permitted to mix prg and data in the
*     same source block.
*
*  6. The boot loader stops boot operation when it finds 0 in the
*     block size control word. Therefore, each boot table should
*     always end with a 0, prompting the boot loader to branch to the
*     first address of the first block and start program execution
*     from that location.
*
*===================================================================*
* C32 boot loader program register assignments, and altered mem
* locations
*===================================================================*
*
* AR7 – peripheral memory map              IOF – XF0 (handshake O)
* AR0 – read cntrl data subr pointer       IOF – XF1 (handshake I)
* AR1 – read block data/prg subr pointer
*
*  R2 – read STRB value            R4 – write STRB value
* AR2 – read STRB pointer         AR4 – write STRB pointer
* AR3 – read data/prg pointer     AR5 – write data/prg pointer
*
*                       read ––> R1 ––> write
*
* IR0 – EXEC start flag          stack – 808024h – TIM0 cnt reg
* IR1 – EXEC start address               808028h – TIM0 per reg
*                               IOSTRB – 808004h – DMA0 dst reg
*  R3 – data SIZE                STRB0 – 808006h – DMA0 dst reg
*  R5 – mem WIDTH                STRB1 – 808008h – DMA0 cnt reg
*
*  R6 – memory read value       AR6,R7,R0,BK – scratch registers
*
*===================================================================*

reset      .word    start        ; reset vector
           .space   44h          ; program starts @45h

*===================================================================*
```

```
* Init registers : 808000h ––> AR7,  808023h ––> SP,  –1 ––> IR0
*===================================================================*

start      LDI     4040h,AR7     ; load peripheral memory map
           LSH     9,AR7         ; base address = 808000h
           LDI     23h,SP        ; initialize stack pointer to
           OR      AR7,SP        ; 808023h (timer counter – 1)
           LDI     –1,IR0        ; reset exec start addr flag

*===================================================================*
* Test for INT3 and, if set exclusively, proceed with serial
* boot load. Else, load AR3 with 1000h if INT0, 810000h if INT1,
* 900000h if INT2. Also load appropriate boot strobe pointer  ––> AR2
* and force the boot strobe value to reflect 32bit memory width.
* If (INT0 or INT1 or INT2) and INT3, turn on the handshake mode.
*===================================================================*
wait1     LDI      IF,R0
          AND      0Fh,R0           ; clean
          CMPI     8,R0             ; test for INT3
          BEQ      serial   ;*******; serial boot load mode
          LDI      AR7,AR2

          ADDI     60h,AR2          ; 808060h (IOSTRB)   ––>  AR2
          TSTB     2,R0             ; test for  INT1
          LDINZ    4080h,AR3        ; 810000h / 2**9
          BNZ      exit3    ;*******;

         ADDI      4,AR2            ; 808064h  (STRB0)   ––>  AR2
         TSTB      1,R0             ; test for  INT0
         LDINZ     8,AR3            ; 001000h / 2**9
         BNZ       exit3    ;*******;

         ADDI      4,AR2            ; 808068h  (STRB1)   ––>  AR2
         TSTB      4,R0             ; test for  INT2
         LDINZ     4800h,AR3        ; 900000h / 2**9
         BZ        wait1    ;*******;

exit3    TSTB      8,R0           ;*; test#1 – INT3 asserted
         BZ        exit2          ;*; test#2 – INXF1 low (not used)
         TSTB      80h,IOF        ;*; enable handshake mode if
         LDI       6,IOF          ;*; test#1 passed

exit2    LDI       0Fh,R2
         LSH       16,R2            ; force boot data size to 32
         OR        *AR2,R2          ; force boot mem width to 32
         STI       R2,*AR2
         LSH       9,AR3            ; boot mem start addr ––> AR3
*                                                   xx000001 –  1 bit
*================================================== xx000010 –  2 bit
* Process MEMORY WIDTH control word (32 bits long)  xx000100 –  4 bit
*================================================== xx001000 –  8 bit
*                                                   xx010000 – 16 bit
*                                                   xx100000 – 32 bit
```

```
         LDI       read_mc,AR0      ; use memory to read cntrl words
                                    ;             read_mc -->  AR0
         LDI       1,R5             ; mem width = 1        (init)
         LDI       32,AR6           ; mem reads = 32       (init)
        CALLU      read_m           ; read memory once     (1st read)

loop2    TSTB      1,R6
         BNZ       label4
         LSH       -1,R6            ; look at next bit
         LSH       -1,AR6           ; decr mem reads
         LSH       1,R5             ; incr mem width   -->  R5
         BU        loop2    ;*******;

label4   SUBI      2,AR6
         CMPI      0,AR6            ; set flags
         BN        strobes  ;*******; total # of mem reads = 32/R5
label5  CALLU      read_m           ; read memory once
        DBU        AR6,label5  ;****;

*===================================================================*
* Read and save IOSTRB, STRB0 & STRB1 (to be loaded at end of
* boot load)
*===================================================================*

strobes  CALLU      AR0
          STI       R1,*+AR7(4)      ; IOSTRB    -->     (DMA src)
         CALLU      AR0
          STI       R1,*+AR7(6)      ; STRB0     -->     (DMA dst)
         CALLU      AR0
          STI       R1,*+AR7(8)      ; STRB1     -->     (DMA cnt)

*===================================================================*
* Process block size (# of bytes, half-words, or words after STRB
* cntrl)
*===================================================================*

block   CALLU      AR0               ; read boot memory cntrl word
         LDI       R1,R1             ; is this the last block ?
         BNZ       label2    ;*******; no, go around

         LDI      *+AR7(4),R0        ;                   (DMA src)
         STI       R0,*+AR7(60h)     ; restore IOSTRB
         LDI      *+AR7(6),R0        ;                   (DMA dst)
         STI      R0,*+AR7(64h)      ; restore  STRB0
         LDI      *+AR7(8),R0        ;                   (DMA cnt)
         STI      R0,*+AR7(68h)      ; restore  STRB1
         BU       IR1        ;*******; branch to start of program

label2   LDI      R1,RC              ; setup transfer loop
         SUBI     1,RC               ; RC - 1 --> RC
```

```
*===================================================================*
* Process block destination address, save start address of first
* block
*===================================================================*

      CALLU      AR0                 ; read boot memory cntrl word
       LDI       R1,AR5              ; set dest addr           -->
AR5
       CMPI      0,IR0               ; look at EXEC start addr flag
       LDINZ     AR5,IR1             ; if –1, EXEC start addr  -->
IR1
       LDINZ     0,IR0               ; set EXEC start addr flag

*===================================================================*
* (For internal destination, this word should be 0 or 60h. The first
* case will result in 0 --> DMA cntrl reg, in second case 0 -->
* IOSTRB reg).
* Process block destination strobe control (sss...sss 0110 xx00)
*======================================== strb value ==== 00 – IOSTRB
*                                                         01 –  STRB0
     CALLU      AR0                  ;                    10 –  STRB1
      LDI       R1,R4
      AND       6Ch,R1               ; dest mem strb pntr --> AR4
      OR3       AR7,R1,AR4

      LSH       –8,R4                ; dest memory strobe --> R4

      LDI       R4,R3
      LSH       –16,R3
      AND       3,R3                 ; dest data size     --> R3
      TSTB      0Ch,R1               ; (IOSTRB case)
      LDIZ      3,R3

*===================================================================*
* Look at R5 and choose serial or memory read for block data/program
*===================================================================*

        CMPI      0,R5
        LDIEQ    read_s0,AR1                 ; read serial port0
        LDINE    read_mb,AR1                 ; read memory

*===================================================================*
* Transfer one block of data or program
*===================================================================*

          RPTB     loop4
         CALLU     AR1                     ; read data/prg
          STI      R4,*AR4                 ; set write strobe
          NOP                              ; pipeline
loop4     STI      R1,*AR5++               ; write data/prg!!!!!!!!!!
           BU      block           ;*******; process next block
```

```
*===================================================================*
* Load R5 with 0, load read_s0 to AR0 and initialize serial port_0
*===================================================================*

serial    LDI     read_s0,AR0        ; use serial to read cntrl words
          LDI     0,R5                  ; memory WIDTH = serial
          LDI     0,R                   ; dummy
          LDI     AR7,AR2               ; dummy

          LDI     111h,R0               ; 0000111h --> R0
          STI     R0,*+AR7(43h)         ; set CLKR,DR,FSR as serial
          LDI     0A30h,R7              ;               port pins
          LSH     16,R7                 ; A300000h --> R7
          STI     R7,*+AR7(40h)         ; set serial global cntrl reg
          BU      strobes            ;*******; process first block

*===================================================================*
* Read a single value from serial or boot memory. The number of
* memory reads depends on mem WIDTH and data SIZE. R1 returns the
* read value. (Serial sim: NOP --> BZ read_s0 & LDI @4000H,R1 --> LDI
*  *+AR7(4Ch),R1)
*===================================================================*

read_s0  TSTB     20h,IF             ; look at RINT0 flag
         BZ       read_s0            ; wait for receive buffer full
         AND      0FDFh,IF           ; reset interrupt flag
         LDI      *+AR7(4Ch),R1      ; read data            -->  R1
         RETSU
*--------------------------------------------------------------------
read_mc  LDI      3,R3                  ; data size = 32, 3 --> R3

read_mb  LDI      1,BK                  ; 00000001  (ex: mem width=8)
         LSH      R5,BK                 ; 00000100
         SUBI     1,BK                  ; 000000FF  =  mask --> BK

         LDI      R3,AR6                ;  0 -    1 000  EXPAND
         ADDI     1,AR6                 ;  1 -   10 000  DATA --> AR6
         LSH     3,AR6                  ; 11 -  100 000  SIZE
         LDI        R5,R0
loop3    CMPI       1,R0
         BEQ        exit1               ; DATA SIZE
         LSH        -1,R0               ; --------- - 1       --> AR6
         LSH        -1,AR6              ; MEM WIDTH
         BU         loop3       ;*******;
exit1    SUBI       1,AR6

         LDI        0,R0                ; init shift value
         LDI        0,R1                ; init accumulator
loop1    ADDI       3,SP                ; 808027h --> SP
        CALLU       read_m              ; read memory once    --> R6
         SUBI       3,SP                ; 808024h --> SP
         AND3       R6,BK,R7            ; apply mask
         LSH        R0,R7               ; shift
         OR         R7,R1              ; accumulate        -->  R1
         ADDI       R5,R0              ; increment shift value
         DBU         AR6,loop1   ;*****; decrement #of chunks --> AR6
         RETSU

*==================================================================*
* Perform a single memory read from the source boot table.
* Handshake enabled if IOXF0 bit of IOF reg is set, disabled when
* reset. IACK will pulse continuously if handshake enabled and data
* not ready (to achieve zero-glue interface when connecting to a C40
* comm-port)
*==================================================================*

read_m  TSTB       2,IOF              ; handshake mode enabled ?
        STI        R2,*AR2            ; set read strobe !!!!!!!!!!!!!
        BNZ        loop5              ; yes, jump over
        LDI        *AR3++,R6          ; no,  just read memory & return
        RETSU
*------------------------------------------------------------   (C40)
loop5   IACK      *AR7               ;*; intrnl dummy read pulses IACK
        TSTB       80h,IOF           ;*; wait for data ready
        BNZ        loop5             ;*; (XF1 low from host)

        LDI        *AR3++,R6         ;*; read memory once   -->  R6

        LDI        2,IOF             ;*; assert data acknowledge
                                     ;*; (XF0 low to host)

loop6   TSTB       80h,IOF           ;*; wait for data not ready
        BZ         loop6             ;*; (XF1 high from host)

        LDI        6,IOF             ;*; deassert data acknowledge
                                     ;*; (XF0 high to host)
        RETSU
*==================================================================*
```

APPENDIX C—MEMORY ACCESS FOR C PROGRAMS

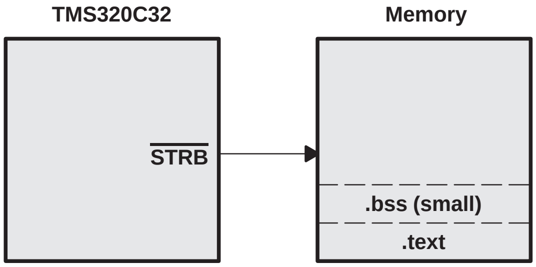
Two memory models can be used to access data when programming in C. In the small model (default), the external bus cycles use direct addressing to access data from memory. Direct addressing includes 16 bits of address in the instruction opcode. The address is combined with the 8-bit data page (defined beforehand) to access the data from memory. The 16-bit address limits the number of words that the small model can access to 64K words. However, this mode produces fast and compact code because each data access uses only a single instruction (see Figure 16).

The big model is not limited to 64K words because each data access in C explicitly sets the data page pointer (DP register). The 8-bit data page and 16-bit direct address are combined for a total address reach of 16M words but at a price of two instructions per data access (see Figure 16).

Dynamically allocated memory can be used if the application needs a large address reach, compact code size, and fast execution. The MALLOC function from runtime support library (RTS) can be called at run time to reserve a block of memory in the .SYSMEM section. Upon return, MALLOC returns a pointer to the newly allocated block. Any reference to that block of memory results in assembled code using indirect addressing, in which the opcode contains a pointer to the auxiliary register that holds the address of the operand (see Figure 16). Code referring to the dynamically allocated memory is fast and has a 16M-word address reach (24 bits). The price is a one-time call to MALLOC for each dynamically allocated array. For that reason, MALLOC is most efficient with large data arrays where the overhead associated with the call is insignificant when compared to a large number of data accesses that use the big arrays.

Figure 17 shows how to use MALLOC to allocate a block of 32-bit memory at run time. In this example, MALLOC is called three times to allocate memory from the heap.

- **Static memory – assigned at compile time**
- **Maximum size – 64K words**
- **Fast execution**

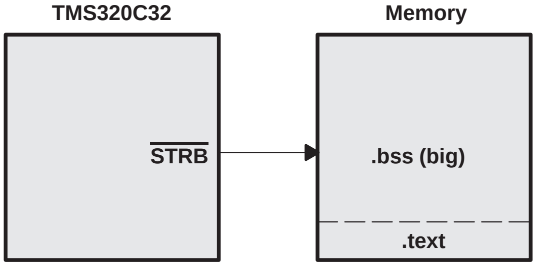

TMS320C32
Memory
STRB
.bss (small)
.text

C	ASSEMBLY

C = A + B	LDI @ 0FFFDh, R0
	LDI @0FFFEh, R1
	ADDI R0, R1
	STI R1, @ 0FFFh

(a) SMALL MODEL (DEFAULT)

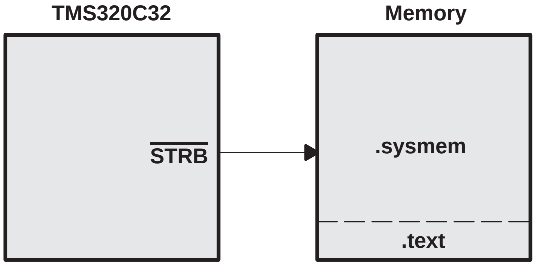
- **Static memory – assigned at compile time**
- **Maximum size – 64M words**
- **Slow execution**


TMS320C32
Memory
STRB
.bss (big)
.text

C	ASSEMBLY
C = A + B	LDP @ 880001h, DP LDI @ 880001h, R0 LDP @ 1002h, DP LDI @ 1002h, R1 LDP @ 8A0003, DP STI R1, @ 8A0003

(b) BIG MODEL (-mb OPTION)

- **Dynamic memory – assigned at execution time**
- **Maximum size – 64M words**
- **Fast execution**
- **Best for big arrays (one time overhead – malloc call)**


TMS320C32
Memory
STRB
.sysmem
.text

C	ASSEMBLY

C = A + B	LDI *AR0, R0
	LDI *AR1, R1
	ADDI R0, R1
	STI R1, *AR2

(c) RTS LIBRARY (MALLOC)

Figure 16. Memory Allocation in C Programs

C code

```
           •
           •
           •
int        *BUFFER_32                       /* declare a pointer to a pool of 32-bit memory */
           •
           •
           •
           •
           •
           •
           •
BUFFER_32 = MALLOC (2048 * sizeof (int))              /* allocate 2K words of memory */
dsp_func4 ( BUFFER_32)                                /* use the above memory */
           •
           •
           •
BUFFER_32 = MALLOC  (512 * sizeof (int))              /* allocate 0.5K words of memory */
dsp_func5 ( BUFFER_32)                                /* use the above memory */
           •
           •
           •
BUFFER_32  = MALLOC (1024 * sizeof (int))             /* allocate 1K words of memory */
dsp_func6 (BUFFER_32)                                 /* use the above memory */
           •
           •
           •
```

LINKER **command file**

```
           •
           •
           •
–heap   0x4000                         /* set the size of the dynamic 32-bit memory section */
           •
           •
           •
STRB_RAM        org = 0x1000,   len = 0x8000          /* define physical 32-bit memory */
           •
           •
           •
.sysmem    >   STRB_RAM                     /* assign logical section to physical memory */
           •
           •
           •
```

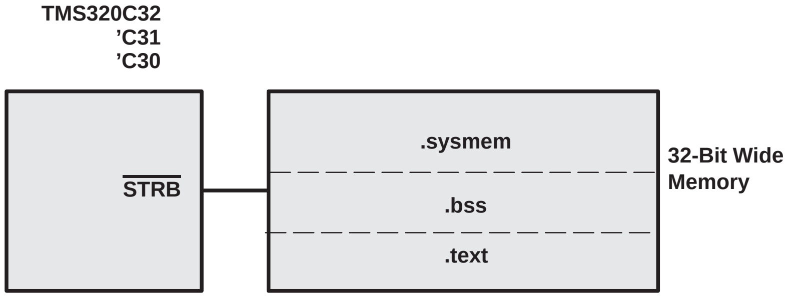


Figure 17. Dynamic Memory Allocation for TMS320C32 (One Block of 32- Bit Memory)

After each MALLOC call, the newly allocated block of memory can be used by other program functions by using the pointer BUFFER_32. The size of the heap (representing all of dynamically allocated memory) is defined in the linker command file by using the HEAP keyword followed by the size of the block. Any portion of the heap allocated with the MALLOC call is added to the .SYSMEM section. The SECTIONS directive can then be used to map the dynamically allocated sections to an address range in the physical memory.

Dynamically allocated memory provides the only method for a C program to access 8- or 16-bit wide memory. This means that physical memory that is less than 32 bits wide cannot be accessed using small or big model addressing. Instead, MALLOC8 and MALLOC16 RTS library functions can allocate blocks of 8- and 16-bit wide memory. These routines work like the 32-bit MALLOC by returning pointers to 8- or 16-bit memory blocks that can be used by code that follows the MALLOC call to access that memory (see Figures 18 and 19). The 8-bit data allocated by MALLOC8 is placed in the .SYSM8 section by the linker, while the 16-bit data is deposited in the .SYSM16 section. HEAP8 and HEAP16 linker keywords limit the total amount of 8- or 16-bit memory that the C compiler can allocate into those sections.

C code

```
          •
          •
          •
int       *BUFFER_16                          /* declare a pointer to a pool of 16-bit memory */
          •
          •
          •
*0x808064 = 0x5000                /* STRB0 Control Register : Data Size = 16, Memory Width = 16 */
          •
          •
          •
BUFFER_16 = MALLOC16(1024 * sizeof (int))               /* allocate 2K half-words of memory */
dsp_func4 ( BUFFER_16)                                   /* use the above memory */
          •
          •
          •
BUFFER_16 = MALLOC16  (512 * sizeof (int))              /* allocate 1K half-words of memory */
dsp_func5 ( BUFFER_16)                                  /* use the above memory */
          •
          •
          •
BUFFER_16  = MALLOC8 (2048 * sizeof (int))              /* allocate 4K half-words of memory */
dsp_func6 (BUFFER_16)                                   /* use the above memory */
          •
          •
          •
```

LINKER command file

```
          •
          •
          •
–heap 16  0x4000                  /* set the size of the dynamic 16-bit memory section */
          •
          •
          •
STRB0_RAM      org = 0x880000,   len = 0x8000           /* define physical 16-bit memory */
          •
          •
          •
.sysm16   >   STRB0_RAM                   /* assign logical section to physical memory */
          •
          •
          •
```

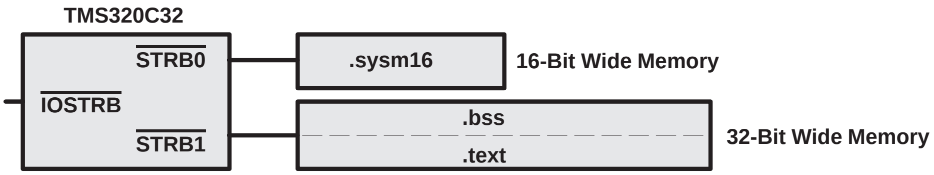

TMS320C32
STRB0
IOSTRB
STRB1
.sysm16
16-Bit Wide Memory
.bss
.text
32-Bit Wide Memory

Figure 18. Dynamic Memory Allocation for TMS320C32 (One Block of 16-bit Memory)

C code

```
            •
            •
int        *BUFFER_32                                        /* declare a pointer to a pool of 32-bit memory */
int        *BUFFER_16                                        /* declare a pointer to a pool of 16-bit memory */
int        *BUFFER_08                                        /* declare a pointer to a pool of 8-bit memory */
            •
            •
*0x808064 = 0x5000                   /* STRB0 Control Register : Data Size = 16, Memory Width = 16 */
*0x808068 = 0x0000                   /* STRB1 Control Register : Data Size =  8 , Memory Width =  8  */
            •
            •
BUFFER_32 = MALLOC    (1024 * sizeof (int))              /* allocate 1K words of memory */
BUFFER_16 = MALLOC16(1024 * sizeof (int))                /* allocate 2K halfwords of memory */
BUFFER_08 = MALLOC8  (1024 * sizeof (int))               /* allocate 4K bytes of memory */
dsp_func1 (BUFFER_32, BUFFER_16, BUFFER_08)              /* use the above memory */
            •
            •
BUFFER_32 = MALLOC    (2048 * sizeof (int))              /* allocate 2K words of memory */
BUFFER_16 = MALLOC16  (512 * sizeof (int))               /* allocate 1K half-words of memory */
dsp_func2 (BUFFER_32, BUFFER_16)                         /* use the above memory */
            •
            •
BUFFER_08  = MALLOC8 (4096 * sizeof (int))               /* allocate 16K bytes of memory */
dsp_func3 (BUFFER_08)                                    /* use the above memory */
            •
            •
```

LINKER command file

```
            •
            •
–heap       0x4000                         /* set the size of the dynamic 32-bit memory section */
–heap 16  0x4000                           /* set the size of the dynamic 16-bit memory section */
–heap   8  0x4000                          /* set the size of the  dynamic  8-bit memory section */
            •
            •
IOSTRB_RAM        org = 0x810000,   len = 0x8000             /* define physical 32-bit memory */
STRB0_RAM         org = 0x880000,   len = 0x8000             /* define physical 16-bit memory */
STRB1_RAM         org = 0x900000,   len = 0x8000             /* define physical  8-bit  memory */
            •
            •
.sysmem    >   IOSTRB_RAM                       /* assign logical section to physical memory */
.sysm16     >   STRB0_RAM                       /* assign logical section to physical memory */
.sysm8      >   STRB1_RAM                       /* assign logical section to physical memory */
            •
            •
```

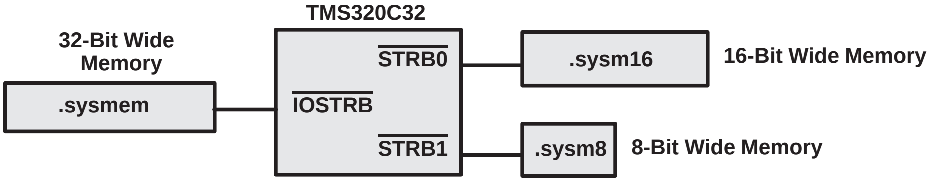


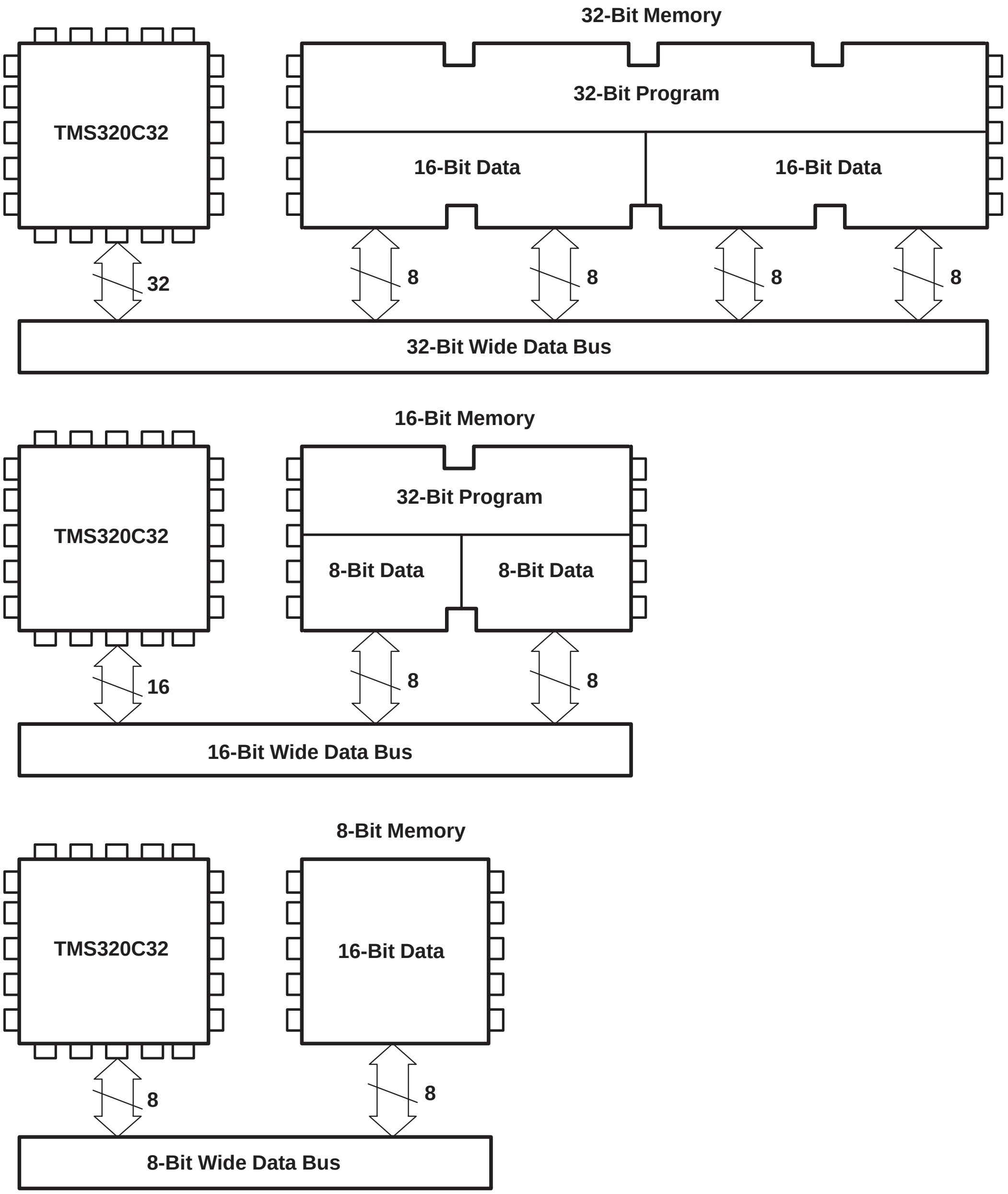
Figure 19. Dynamic Memory Allocation for TMS320C32
(One Block Each of 32-, 16-, and 8-Bit Memory)

APPENDIX D—MEMORY INTERFACE AND ADDRESS TRANSLATION

The ’C32 memory interface supports variable width memory and variable size data. The physical width of a memory bank connected to the ’C32 can be 8, 16, or 32 bits wide. When connecting 16-bit external memory, the A_{-1} address pin must be connected to the A0 pin of the memory device, causing a one-bit shift in the connection of the remaining address lines. For 8-bit memory, two extra address pins are used (A_{-1} and A_{-2}) effectively shifting the external address by two bits. No external address shift is needed for connecting 32-bit wide memory (or boot table memory regardless of its width).

The ’C32 can access data of any size regardless of the physical width of an external memory bank. For example, byte-wide data can be packed in 16-bit memory, or 32-bit data can be accessed from 8-bit wide memory. The latter takes four cycles. The variable data size feature is made possible by dividing the $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ controls into four signals each. The four control signals, in addition to being strobes, serve a byte-enable function.

Figure 20 shows examples of three ’C32 systems, each connected to a memory bank of a different width.

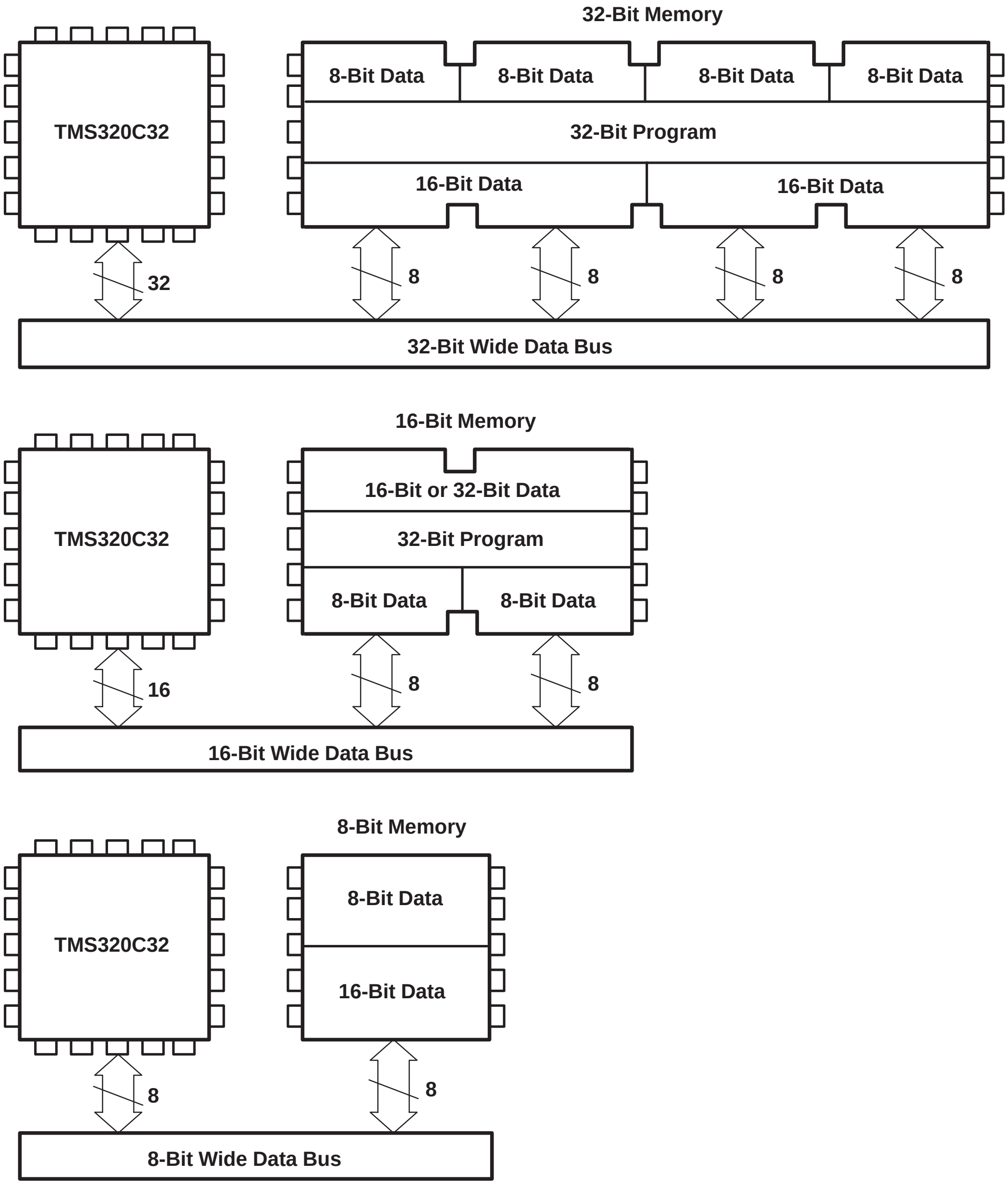

32-Bit Memory
TMS320C32
32-Bit Program
16-Bit Data
16-Bit Data
32
8
8
8
8
32-Bit Wide Data Bus
16-Bit Memory
TMS320C32
32-Bit Program
8-Bit Data
8-Bit Data
16
8
8
16-Bit Wide Data Bus
8-Bit Memory
TMS320C32
16-Bit Data
8
8
8-Bit Wide Data Bus

NOTE: 8-bit program is not supported.

Figure 20. Data and Program Packing (Program and One Data Size)

Regardless of memory width, the data inside each bank can be 8, 16. or 32 bits wide. Before data of a particular size can be accessed, the respective strobe control register must be programmed for that size. While the data size can vary, the program is always 32 bits wide. Even if they are different sizes, program and data can reside within the same physical bank of memory.

Up to two data sizes can reside simultaneously alongside the 32-bit program in a single bank (see Figure 21).


32-Bit Memory
TMS320C32
8-Bit Data
8-Bit Data
8-Bit Data
8-Bit Data
32-Bit Program
16-Bit Data
16-Bit Data
32
8
8
8
8
32-Bit Wide Data Bus
16-Bit Memory
TMS320C32
16-Bit or 32-Bit Data
32-Bit Program
8-Bit Data
8-Bit Data
16
8
8
16-Bit Wide Data Bus
8-Bit Memory
TMS320C32
8-Bit Data
16-Bit Data
8
8
8-Bit Wide Data Bus

NOTE: 8-bit program is not supported.

Figure 21. Data and Program Packing (Program and Two Data Sizes)

Since there are two strobes that support flexible memory ($\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$), they each can be programmed for a different data size (using the respective strobe control registers). By setting the strobe configuration bit in one control register, both $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ strobes can be mapped to $\overline{\text{STRB0}}$ control signals. In this overlay mode, data accesses to/from $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ portions of the memory map will drive $\overline{\text{STRB0}}$ signals to control a single memory bank. The glueless access to two data sizes and program from a single bank is a powerful ’C32 feature that minimizes system cost with no performance penalty.

The translation starts when an instruction requests a data read from a certain external address. Address locations referenced by program instructions are logical addresses. Before the logical address shows up on the external pins of the ’C32, it may undergo a 1- or 2-bit shift to the right that depends only on the size of the data being accessed. The address at the pins is a physical address. Before it is presented at the pins of the memory device, the physical address may again be shifted (this time to the left) if the memory is other than 32 bits wide. The physical-to-memory address shift is one bit for 16-bit wide memory and two bits for 32-bit memory. The following tables summarize the rules that apply to the variable data size and memory width for any ’C32 system.

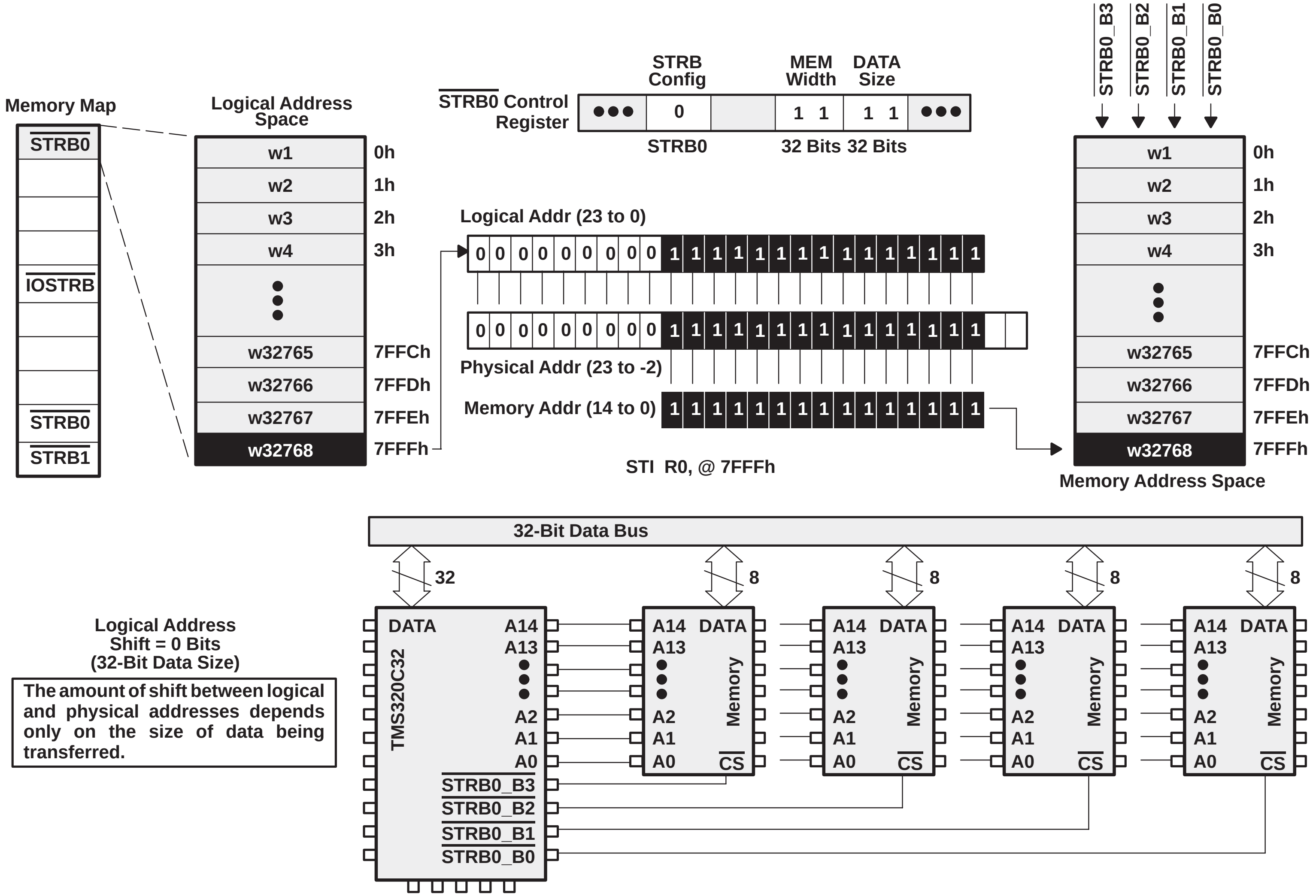
VARIABLE MEMORY WIDTH

MEMORY WIDTH	STROBES VALID	PHYSICAL ADDRESS LINES VALID	PHYSICAL ADDRESS TO MEMORY ADDRESS SHIFT (BITS)
32	$\overline{\mathrm{STRBx\_B3}}$ $\overline{\mathrm{STRBx\_B2}}$ $\overline{\mathrm{STRBx\_B1}}$ $\overline{\mathrm{STRBx\_B0}}$	A23–A0	0
16	$\overline{\mathrm{STRBx\_B1}}$ $\overline{\mathrm{STRBx\_B0}}$	A23–A0 A–1	1
8	$\overline{\mathrm{STRBx\_B0}}$	A23–A0 A–1 A–2	2

VARIABLE DATA SIZE

DATA SIZE	LOGICAL TO PHYSICAL ADDRESS SHIFT (BITS)
32	0
16	1
8	2

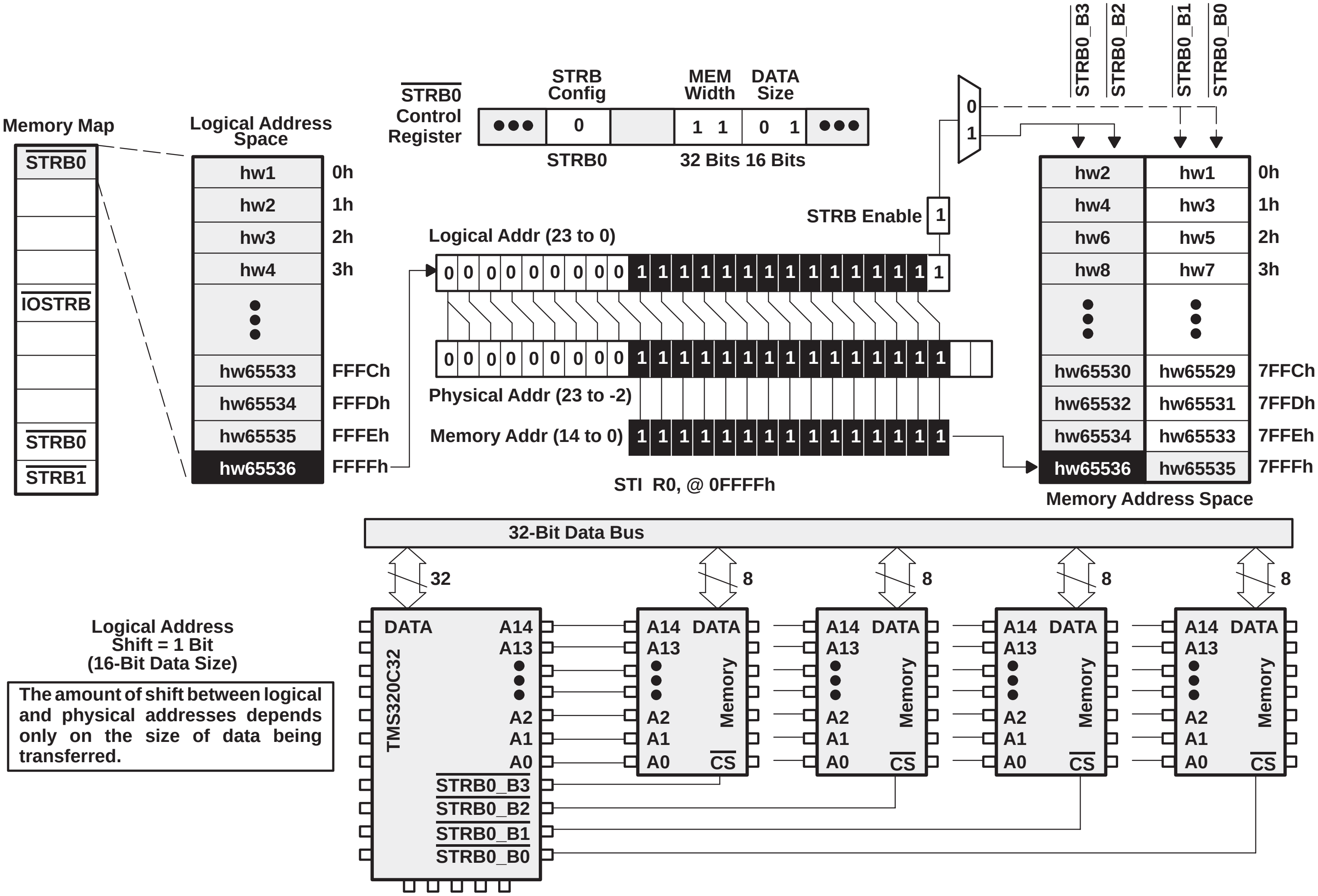
Figures 22 through 30 show how the address changes when accessing data of varying size from memory that is 32, 16, and 8 bits wide. The three data sizes and three memory widths comprise the nine cases that cover all possible combinations.


Memory Map
STRB0
IOSTRB
STRB0
STRB1
Logical Address Space
w1
w2
w3
w4
w32765
w32766
w32767
w32768
0h
1h
2h
3h
7FFCh
7FFDh
7FFEh
7FFFh
STRB0 Control Register
STRB Config
MEM Width
DATA Size
0
1 1
1 1
STRB0
32 Bits
32 Bits
Logical Addr (23 to 0)
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Physical Addr (23 to -2)
Memory Addr (14 to 0)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
STI R0, @ 7FFFh
STRB0_B3
STRB0_B2
STRB0_B1
STRB0_B0
Memory Address Space
32-Bit Data Bus
32
8
DATA
A14
A13
A2
A1
A0
TMS320C32
Memory
CS

Logical Address Shift = 0 Bits (32-Bit Data Size)

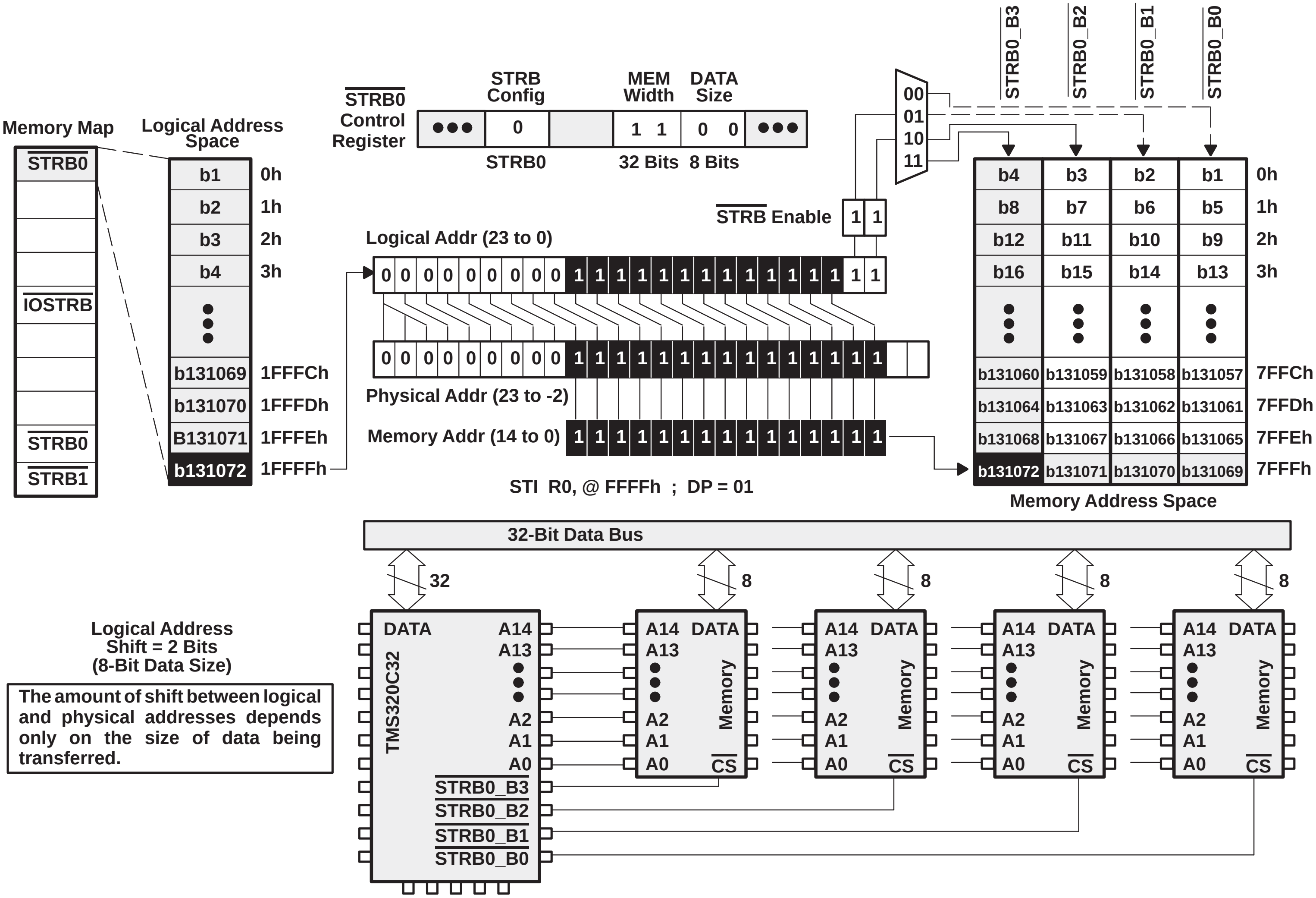
The amount of shift between logical and physical addresses depends only on the size of data being transferred.

Figure 22. Address Translation for 32-Bit Data Stored in 32-Bit Wide Memory


Memory Map
STRB0
IOSTRB
STRB0
STRB1
Logical Address Space
hw1
hw2
hw3
hw4
hw65533
hw65534
hw65535
hw65536
0h
1h
2h
3h
FFFCh
FFFDh
FFFEh
FFFFh
STRB0 Control Register
STRB Config
MEM Width
DATA Size
0
1 1
0 1
STRB0
32 Bits 16 Bits
STRB Enable
1
Logical Addr (23 to 0)
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Physical Addr (23 to -2)
Memory Addr (14 to 0)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
STI R0, @ 0FFFFh
STRB0_B3
STRB0_B2
STRB0_B1
STRB0_B0
0
1
hw2
hw1
hw4
hw3
hw6
hw5
hw8
hw7
hw65530
hw65529
hw65532
hw65531
hw65534
hw65533
hw65536
hw65535
0h
1h
2h
3h
7FFCh
7FFDh
7FFEh
7FFFh
Memory Address Space
32-Bit Data Bus
32
8
8
8
8
Logical Address
Shift = 1 Bit
(16-Bit Data Size)
DATA
TMS320C32
A14
A13
A2
A1
A0
STRB0_B3
STRB0_B2
STRB0_B1
STRB0_B0
A14 DATA
A13
A2
A1
A0
Memory
CS

The amount of shift between logical and physical addresses depends only on the size of data being transferred.

Figure 23. Address Translation for 16-Bit Data Stored in 32-Bit Wide Memory


Memory Map
STRB0
IOSTRB
STRB0
STRB1
Logical Address Space
b1 0h
b2 1h
b3 2h
b4 3h
b131069 1FFFCh
b131070 1FFFDh
B131071 1FFFEh
b131072 1FFFFh
STRB0 Control Register
STRB Config
0
STRB0
MEM Width
1 1
32 Bits
DATA Size
0 0
8 Bits
Logical Addr (23 to 0)
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Physical Addr (23 to -2)
0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Memory Addr (14 to 0)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
STRB Enable
1 1
00
01
10
11
STRB0_B3
STRB0_B2
STRB0_B1
STRB0_B0
b4 b3 b2 b1 0h
b8 b7 b6 b5 1h
b12 b11 b10 b9 2h
b16 b15 b14 b13 3h
b131060 b131059 b131058 b131057 7FFCh
b131064 b131063 b131062 b131061 7FFDh
b131068 b131067 b131066 b131065 7FFEh
b131072 b131071 b131070 b131069 7FFFh
Memory Address Space
STI R0, @ FFFFh ; DP = 01
32-Bit Data Bus
32
8
8
8
8
DATA
A14
A13
A2
A1
A0
TMS320C32
STRB0_B3
STRB0_B2
STRB0_B1
STRB0_B0
A14 DATA
A13
A2
A1
A0
Memory
CS

Logical Address Shift = 2 Bits (8-Bit Data Size)

The amount of shift between logical and physical addresses depends only on the size of data being transferred.

Figure 24. Address Translation for 8-Bit Data Stored in 32-Bit Wide Memory

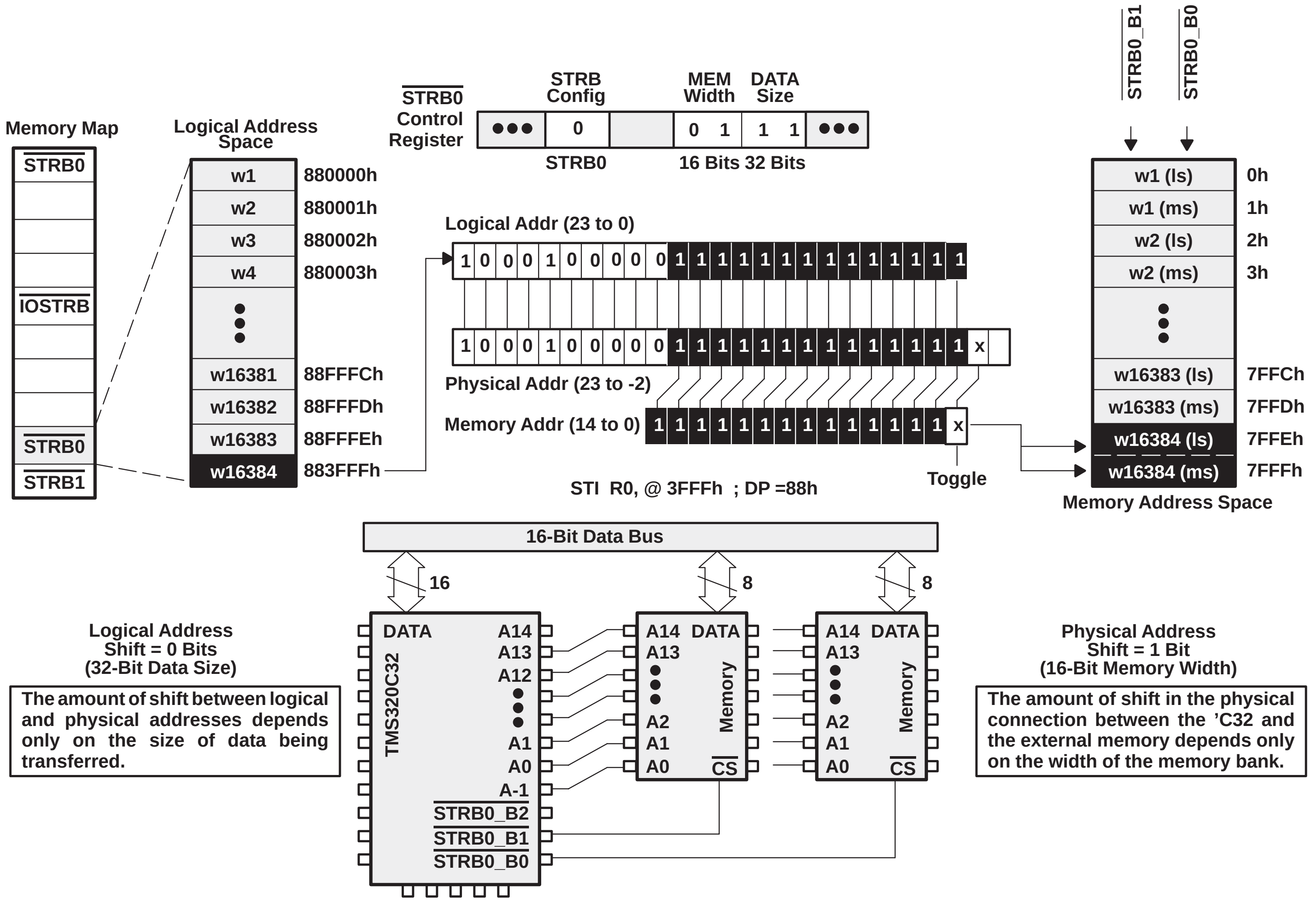
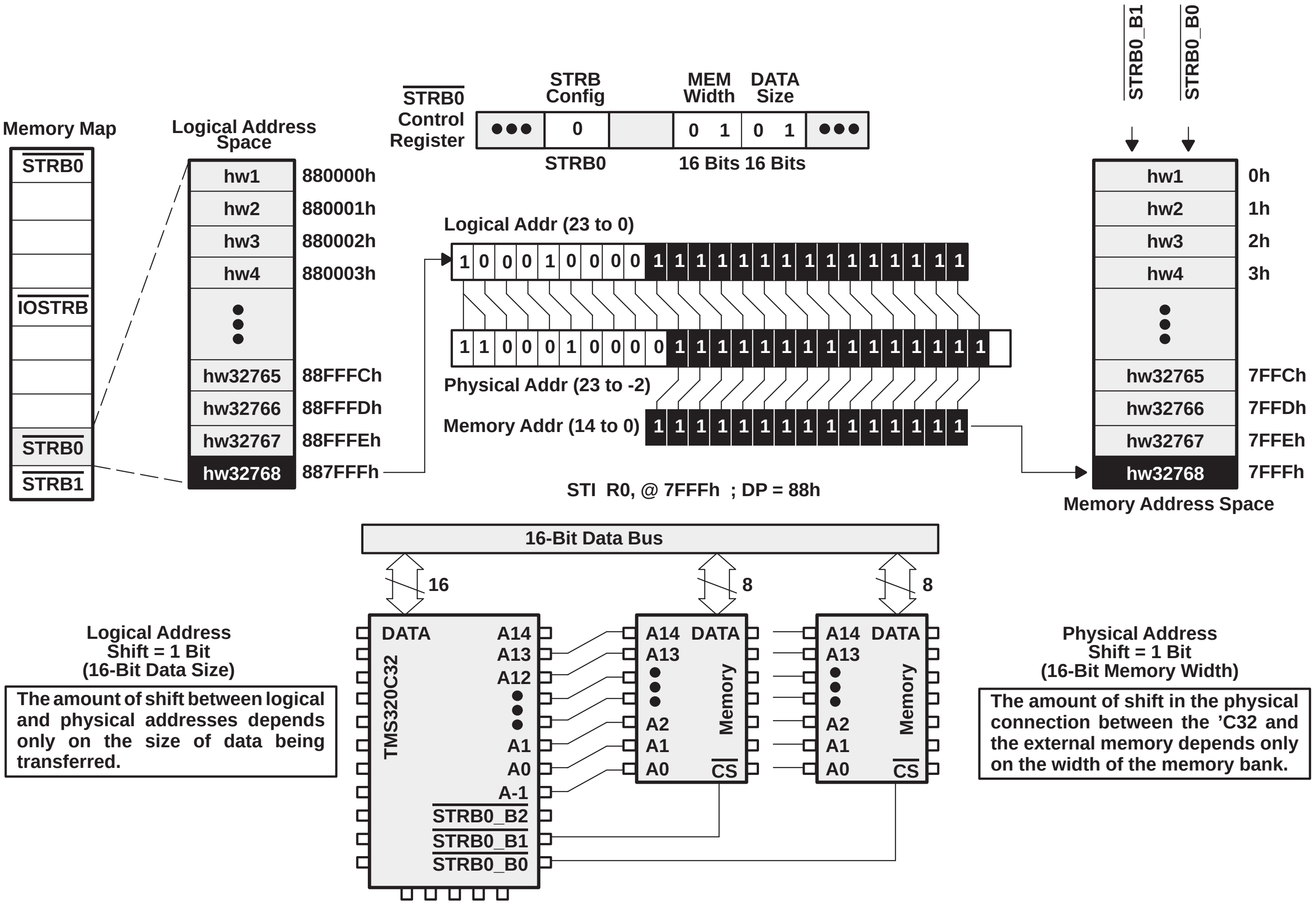

Memory Map
STRB0
IOSTRB
STRB0
STRB1
Logical Address Space
w1 880000h
w2 880001h
w3 880002h
w4 880003h
w16381 88FFFCh
w16382 88FFFDh
w16383 88FFFEh
w16384 883FFFh
STRB0 Control Register
STRB Config
MEM Width
DATA Size
0
0 1
1 1
STRB0
16 Bits 32 Bits
Logical Addr (23 to 0)
1 0 0 0 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 0 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 x
Physical Addr (23 to -2)
Memory Addr (14 to 0)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 x
Toggle
STI R0, @ 3FFFh ; DP =88h
STRB0_B1
STRB0_B0
w1 (ls) 0h
w1 (ms) 1h
w2 (ls) 2h
w2 (ms) 3h
w16383 (ls) 7FFCh
w16383 (ms) 7FFDh
w16384 (ls) 7FFEh
w16384 (ms) 7FFFh
Memory Address Space
16-Bit Data Bus
16
8
8
Logical Address
Shift = 0 Bits
(32-Bit Data Size)
The amount of shift between logical and physical addresses depends only on the size of data being transferred.
TMS320C32
DATA
A14
A13
A12
A1
A0
A-1
STRB0_B2
STRB0_B1
STRB0_B0
A14 DATA
A13
A2
A1
A0
Memory
CS
A14 DATA
A13
A2
A1
A0
Memory
CS
Physical Address
Shift = 1 Bit
(16-Bit Memory Width)
The amount of shift in the physical connection between the ’C32 and the external memory depends only on the width of the memory bank.

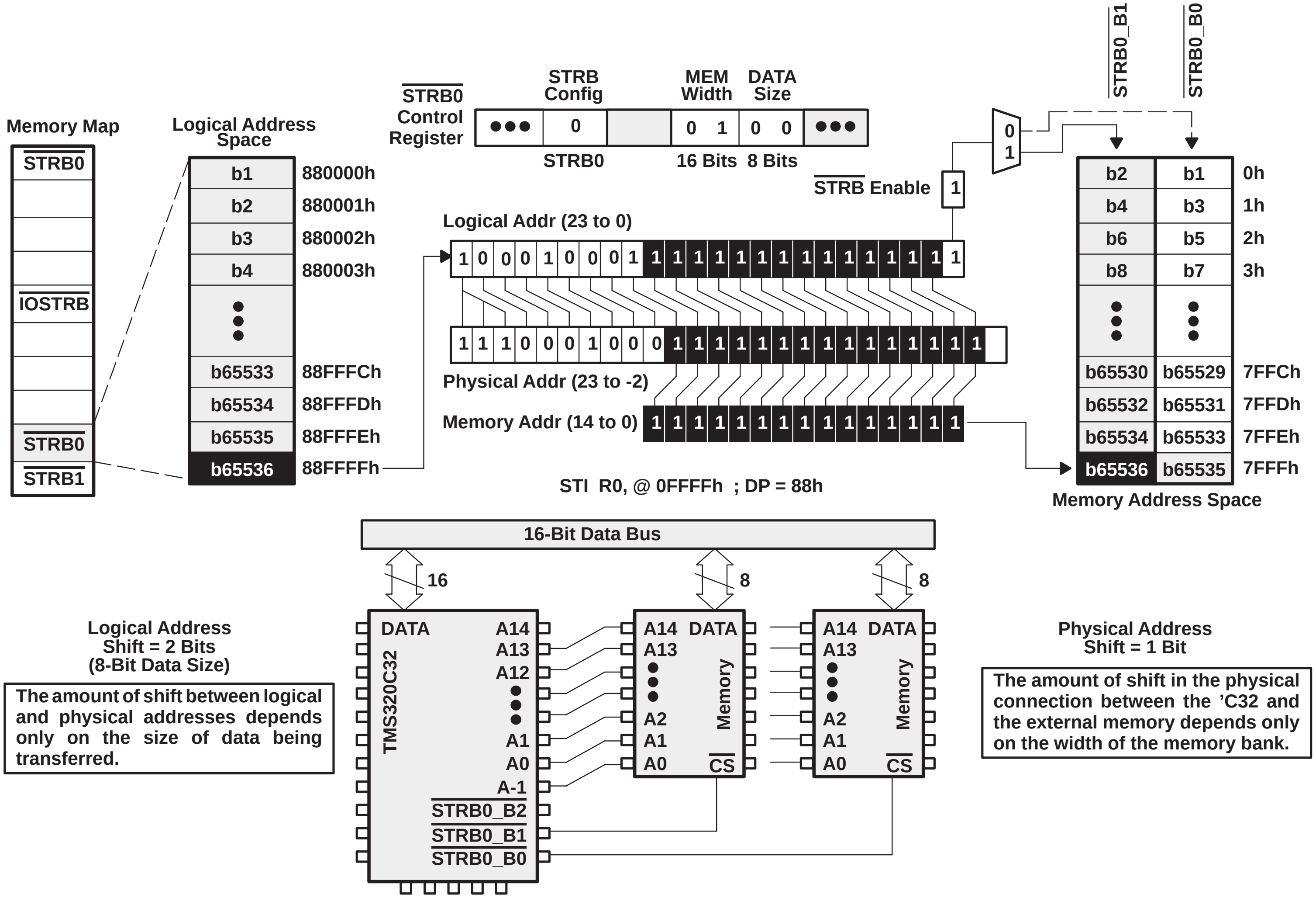
Figure 25. Address Translation for 32-Bit Data Stored in 16-Bit Wide Memory


Memory Map
STRB0
IOSTRB
STRB0
STRB1
Logical Address Space
hw1 880000h
hw2 880001h
hw3 880002h
hw4 880003h
hw32765 88FFFCh
hw32766 88FFFDh
hw32767 88FFFEh
hw32768 887FFFh
STRB0 Control Register
STRB Config
MEM Width
DATA Size
●●● 0 0 1 0 1 ●●●
STRB0
16 Bits 16 Bits
Logical Addr (23 to 0)
1 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Physical Addr (23 to -2)
Memory Addr (14 to 0)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
STI R0, @ 7FFFh ; DP = 88h
STRB0_B1
STRB0_B0
hw1 0h
hw2 1h
hw3 2h
hw4 3h
hw32765 7FFCh
hw32766 7FFDh
hw32767 7FFEh
hw32768 7FFFh
Memory Address Space
16-Bit Data Bus
16
8
8
TMS320C32
DATA
A14
A13
A12
A1
A0
A-1
STRB0_B2
STRB0_B1
STRB0_B0
A14 DATA
A13
A2
A1
A0
Memory
CS
A14 DATA
A13
A2
A1
A0
Memory
CS
Logical Address Shift = 1 Bit (16-Bit Data Size)
Physical Address Shift = 1 Bit (16-Bit Memory Width)

The amount of shift between logical and physical addresses depends only on the size of data being transferred.

The amount of shift in the physical connection between the ’C32 and the external memory depends only on the width of the memory bank.

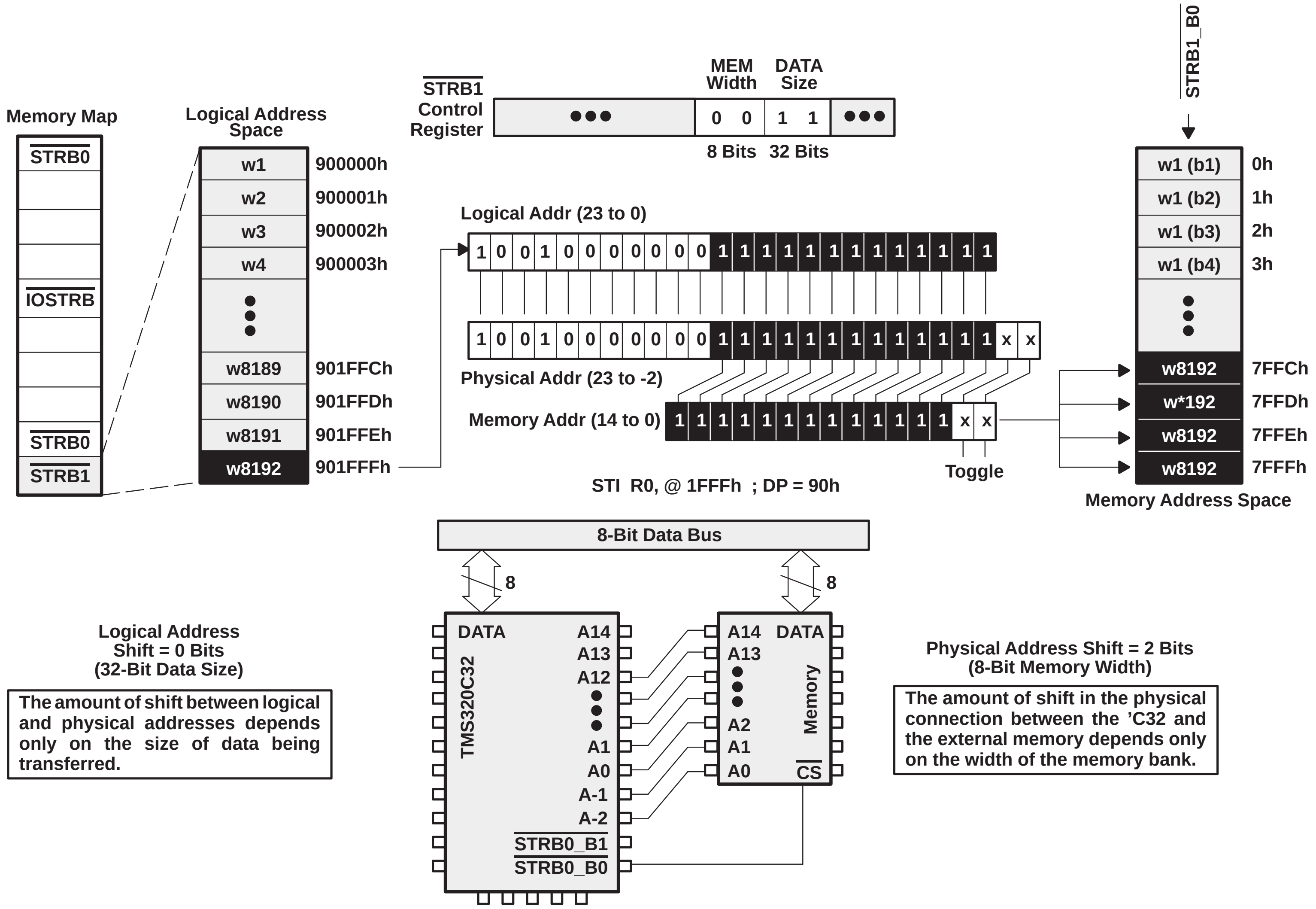
Figure 26. Address Translation for 16-Bit Data Stored in 16-Bit Wide Memory


Memory Map
STRB0
IOSTRB
STRB0
STRB1
Logical Address Space
b1 880000h
b2 880001h
b3 880002h
b4 880003h
b65533 88FFFCh
b65534 88FFFDh
b65535 88FFFEh
b65536 88FFFFh
STRB0 Control Register
STRB Config
0
STRB0
MEM Width
DATA Size
0 1 0 0
16 Bits 8 Bits
STRB Enable 1
0
1
STRB0_B1
STRB0_B0
Logical Addr (23 to 0)
1 0 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Physical Addr (23 to -2)
Memory Addr (14 to 0)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
b2 b1 0h
b4 b3 1h
b6 b5 2h
b8 b7 3h
b65530 b65529 7FFCh
b65532 b65531 7FFDh
b65534 b65533 7FFEh
b65536 b65535 7FFFh
Memory Address Space
STI R0, @ 0FFFFh ; DP = 88h
16-Bit Data Bus
16
8
8
DATA
TMS320C32
A14
A13
A12
A1
A0
A-1
STRB0_B2
STRB0_B1
STRB0_B0
A14 DATA
A13
A2
A1
A0
Memory
CS
A14 DATA
A13
A2
A1
A0
Memory
CS
Logical Address Shift = 2 Bits (8-Bit Data Size)
Physical Address Shift = 1 Bit

The amount of shift between logical and physical addresses depends only on the size of data being transferred.

The amount of shift in the physical connection between the ’C32 and the external memory depends only on the width of the memory bank.

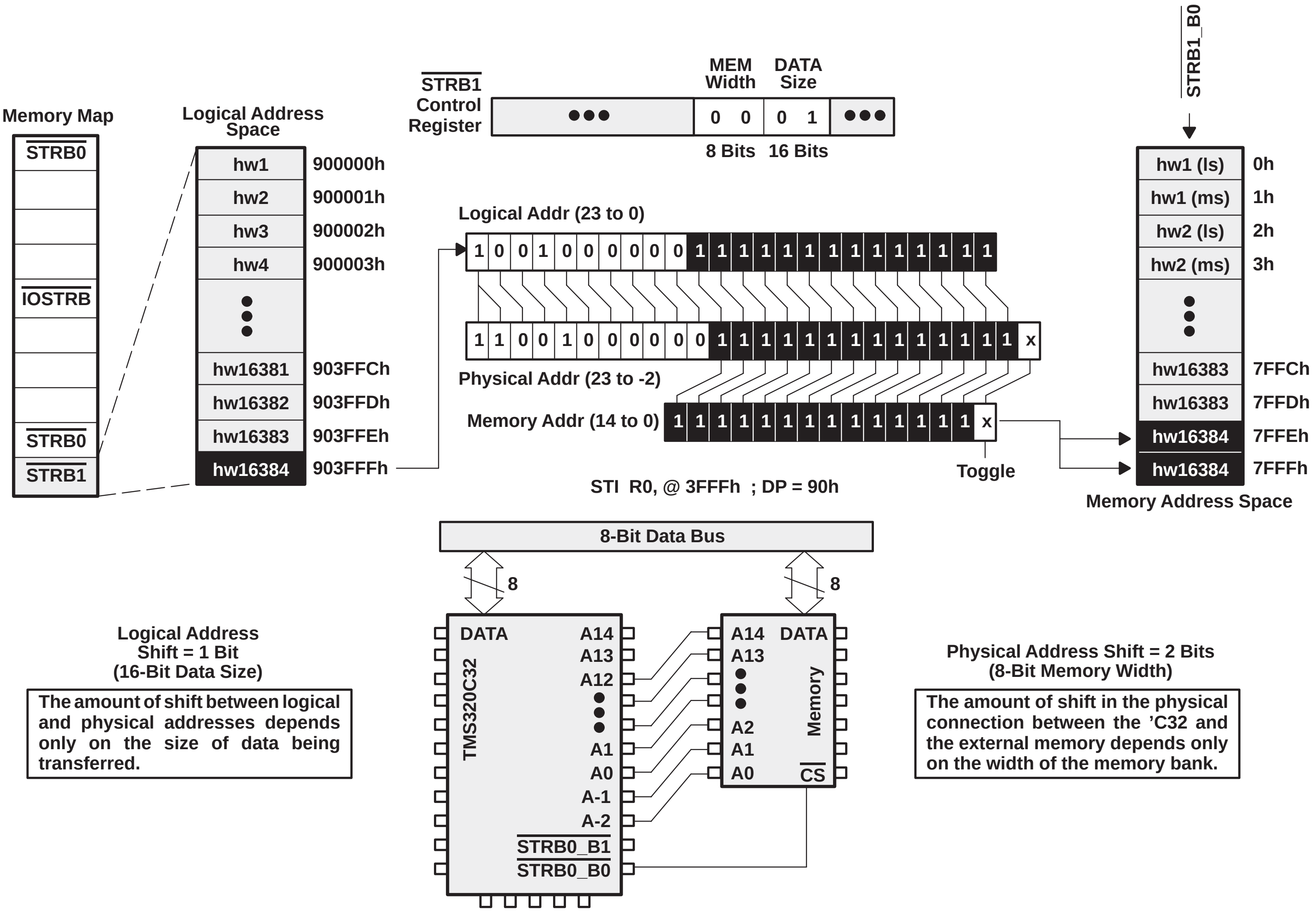
Figure 27. Address Translation for 8-Bit Data Stored in 16-Bit Wide Memory


Memory Map
STRB0
IOSTRB
STRB0
STRB1
Logical Address Space
w1 900000h
w2 900001h
w3 900002h
w4 900003h
w8189 901FFCh
w8190 901FFDh
w8191 901FFEh
w8192 901FFFh
STRB1 Control Register
MEM Width
DATA Size
0 0 1 1
8 Bits
32 Bits
STRB1_B0
Logical Addr (23 to 0)
1 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
1 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 x x
Physical Addr (23 to -2)
Memory Addr (14 to 0)
1 1 1 1 1 1 1 1 1 1 1 1 1 x x
Toggle
STI R0, @ 1FFFh ; DP = 90h
w1 (b1) 0h
w1 (b2) 1h
w1 (b3) 2h
w1 (b4) 3h
w8192 7FFCh
w*192 7FFDh
w8192 7FFEh
w8192 7FFFh
Memory Address Space
8-Bit Data Bus
8
8
TMS320C32
DATA
A14
A13
A12
A1
A0
A-1
A-2
STRB0_B1
STRB0_B0
A14
A13
A2
A1
A0
DATA
Memory
CS
Logical Address Shift = 0 Bits (32-Bit Data Size)
Physical Address Shift = 2 Bits (8-Bit Memory Width)

The amount of shift between logical and physical addresses depends only on the size of data being transferred.

The amount of shift in the physical connection between the ’C32 and the external memory depends only on the width of the memory bank.

Figure 28. Address Translation for 32-Bit Data Stored in 8-Bit Wide Memory


Memory Map
STRB0
IOSTRB
STRB0
STRB1
Logical Address Space
hw1 900000h
hw2 900001h
hw3 900002h
hw4 900003h
hw16381 903FFCh
hw16382 903FFDh
hw16383 903FFEh
hw16384 903FFFh
STRB1 Control Register
MEM Width
DATA Size
0 0
0 1
8 Bits
16 Bits
Logical Addr (23 to 0)
1 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 0 0 1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 x
Physical Addr (23 to -2)
Memory Addr (14 to 0)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 x
Toggle
STI R0, @ 3FFFh ; DP = 90h
STRB1_B0
hw1 (ls) 0h
hw1 (ms) 1h
hw2 (ls) 2h
hw2 (ms) 3h
hw16383 7FFCh
hw16383 7FFDh
hw16384 7FFEh
hw16384 7FFFh
Memory Address Space
8-Bit Data Bus
8
8
TMS320C32
DATA
A14
A13
A12
A1
A0
A-1
A-2
STRB0_B1
STRB0_B0
Memory
A14
A13
A2
A1
A0
DATA
CS

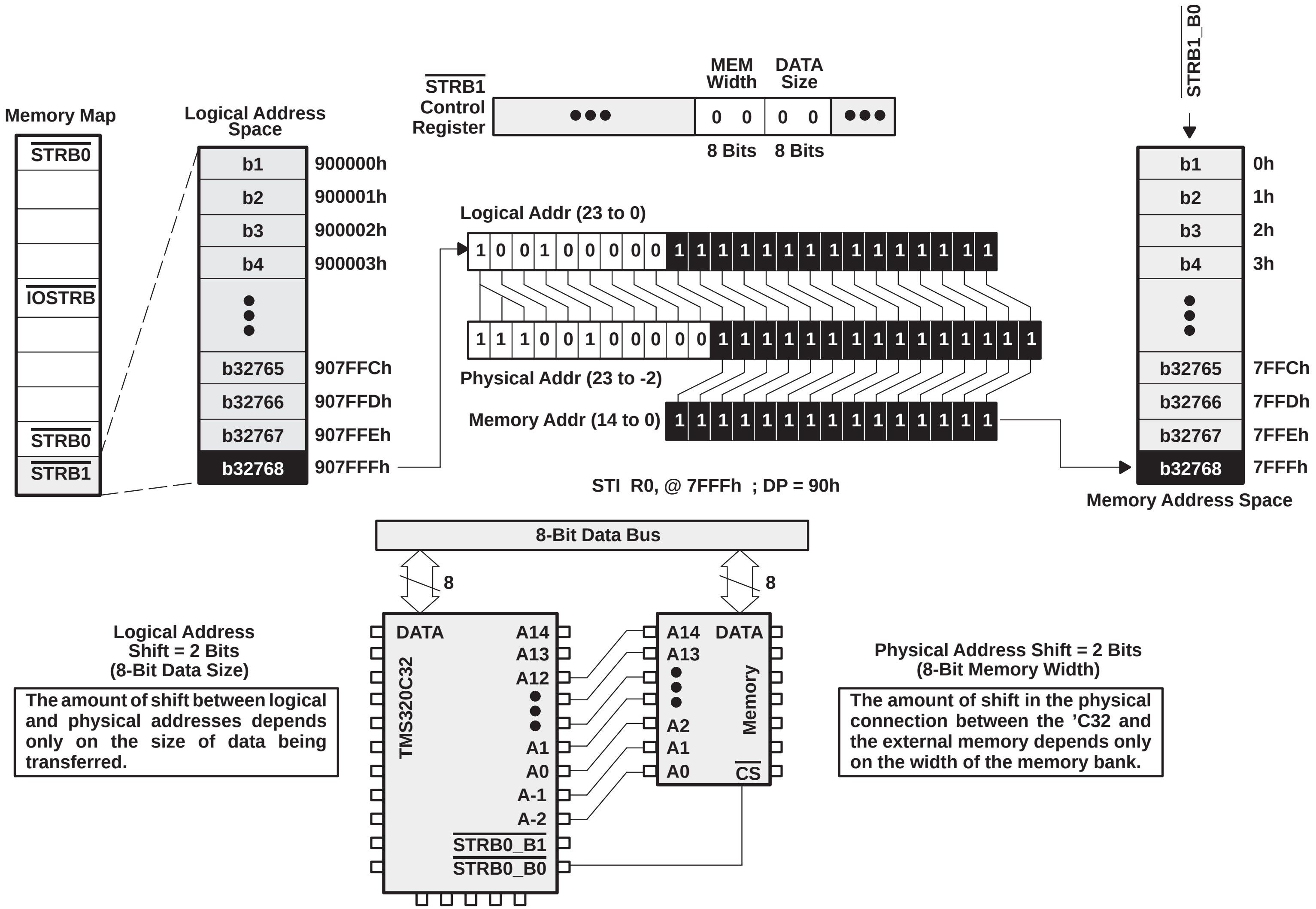
Logical Address Shift = 1 Bit (16-Bit Data Size)

The amount of shift between logical and physical addresses depends only on the size of data being transferred.

Physical Address Shift = 2 Bits (8-Bit Memory Width)

The amount of shift in the physical connection between the ’C32 and the external memory depends only on the width of the memory bank.

Figure 29. Address Translation for 16-Bit Data Stored in 8-Bit Wide Memory


Memory Map
STRB0
IOSTRB
STRB0
STRB1
Logical Address Space
b1 900000h
b2 900001h
b3 900002h
b4 900003h
b32765 907FFCh
b32766 907FFDh
b32767 907FFEh
b32768 907FFFh
STRB1 Control Register
MEM Width
DATA Size
0 0
0 0
8 Bits
8 Bits
Logical Addr (23 to 0)
1 0 0 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 0 0 1 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
Physical Addr (23 to -2)
Memory Addr (14 to 0)
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
STI R0, @ 7FFFh ; DP = 90h
STRB1_B0
b1 0h
b2 1h
b3 2h
b4 3h
b32765 7FFCh
b32766 7FFDh
b32767 7FFEh
b32768 7FFFh
Memory Address Space
8-Bit Data Bus
8
8
DATA
TMS320C32
A14
A13
A12
A1
A0
A-1
A-2
STRB0_B1
STRB0_B0
A14
A13
A2
A1
A0
DATA
Memory
CS
Logical Address Shift = 2 Bits (8-Bit Data Size)
Physical Address Shift = 2 Bits (8-Bit Memory Width)

The amount of shift between logical and physical addresses depends only on the size of data being transferred.

The amount of shift in the physical connection between the ’C32 and the external memory depends only on the width of the memory bank.

Figure 30. Address Translation for 8-Bit Data Stored in 8-Bit Wide Memory