TMS320C32

How TMS320 Tools Interact With the TMS320C32's Enhanced Memory Interface

Pedro R. Gelabert
Digital Signal Processing Products – Semiconductor Group

SPRA048
November 1995

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.

Contents

List of Illustrations

List of Tables

Abstract

This application report describes how to use the TMS320 floating-point digital signal processor (DSP) optimizing C compiler and assembly language tools with the variable memory width and data sizes supported by the TMS320C32's enhanced memory interface. The author provides an overview of the 'C32's strobe control registers as well as a description of how these registers are configured for compiler, linker, and debugger usage. This report also contains examples of different memory configurations that demonstrate the flexibility of the 'C32's memory interface.

Introduction

The TMS320C32 DSP is an enhanced, low-cost version of the TMS320C3x DSP devices. The following CPU enhancements have been incorporated into the ’C32:

- Variable-width memory interface
- Faster instruction cycle time
- Power-down modes
- Relocatable interrupt vector table
- Edge- or level-triggered interrupts

This report describes the capability of the ’C32 to support variable memory widths and data sizes. Topics include the ’C32’s external memory interface and C compiler/linker/debugger interaction with the ’C32. Additionally, examples are provided which demonstrate how to do the following:

- Allocate buffers dynamically and statically using C code
- Build link files to allocate code in a desired configuration
- Configure a debugger to handle ’C32 memory (as configured)

External Memory Interface

The ’C32’s memory interface accesses external memory through one 24-bit address bus and one 32-bit data bus. The data bus is shared by three mutually-exclusive strobes: STRB0, STRB1, and IOSTRB. Depending upon the address accessed, the ’C32 activates one of these strobes as indicated by the memory map shown in Figure 1.

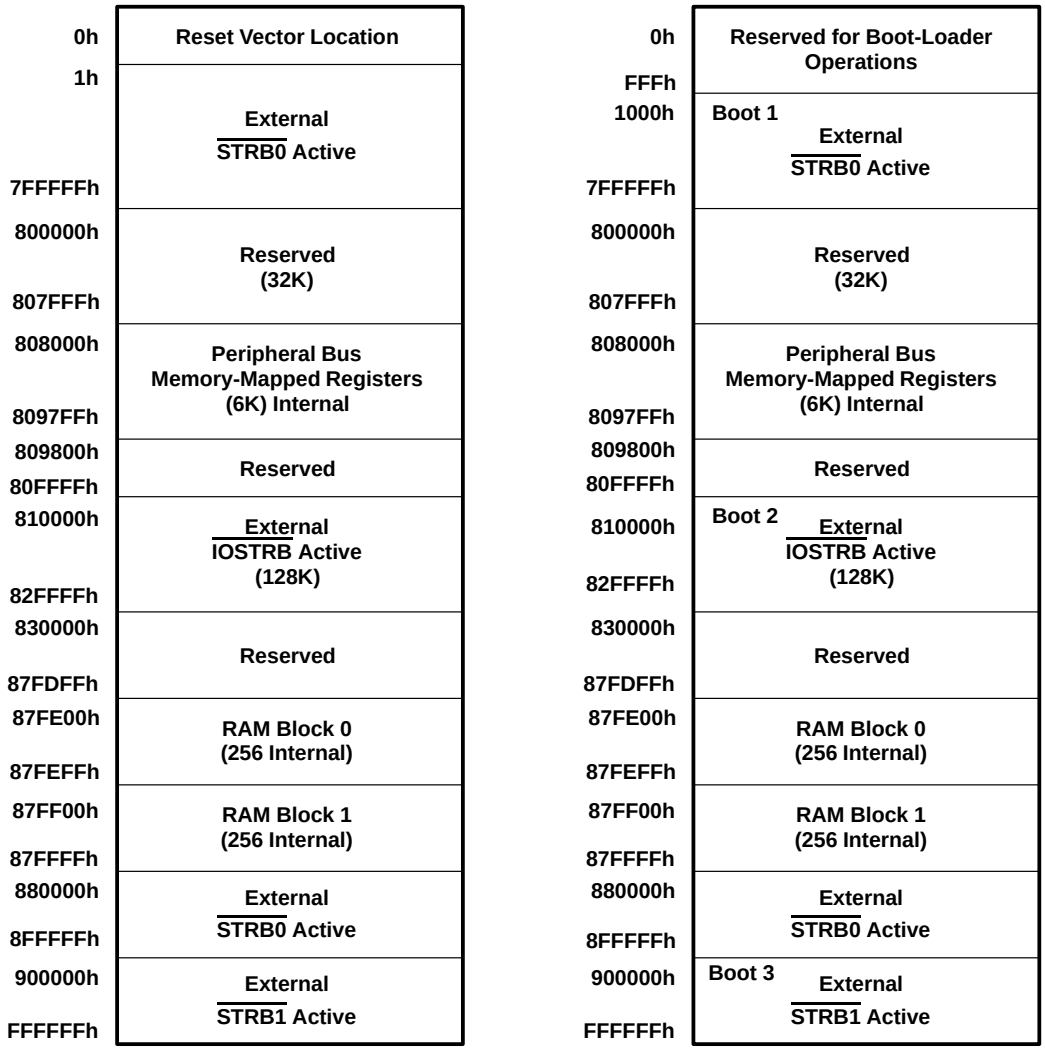

Figure 1. TMS320C32 Memory Map

$\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ can access 8-, 16-, or 32-bit data quantities from 8-, 16-, or 32-bit-wide memory. Access is achieved by four signals within each strobe. These signals are as follows:

- $\overline{\text{STRBx\_B3}}$/A$_{-1}$
- $\overline{\text{STRBx\_B2}}$/A$_{-2}$
- $\overline{\text{STRBx\_B1}}$
- $\overline{\text{STRBx\_B0}}$

The listed signals serve as byte-enable pins for accessing a byte, half-word, or full-word from external memory. The first two signals also serve as additional address pins when performing two or four consecutive accesses in 8- or 16-bit-wide external memory. The data accessed is truncated, packed, or unpacked accordingly, with no additional overhead. The following list shows the behavior of these pins as dictated by the data size and memory-width bit fields of a strobe control register.

Memory width (default value dependent upon the program memory width select [PRGW] pin level):

- 8-bit wide memory
  - $\overline{\text{STRBx\_B3}}$/A$_{-1}$ and $\overline{\text{STRBx\_B2}}$/A$_{-2}$ are address pins.
  - $\overline{\text{STRBx\_B0}}$ is a byte-enable/chip-select signal.
  - $\overline{\text{STRBx\_B1}}$ is not used.
- 16-bit wide memory
  - $\overline{\text{STRBx\_B3}}$/A$_{-1}$ are address pins.
  - $\overline{\text{STRBx\_B1}}$ and $\overline{\text{STRBx\_B0}}$ are byte-enable signals.
  - $\overline{\text{STRBx\_B2}}$/A$_{-2}$ are not used.
- 32-bit wide memory
  - $\overline{\text{STRBx\_B3}}$/A$_{-1}$, $\overline{\text{STRBx\_B2}}$/A$_{-2}$, $\overline{\text{STRBx\_B1}}$, and $\overline{\text{STRBx\_B0}}$ are byte-enable signals.

Data size:

- 8-bit data
  - The physical address is the logical address shifted right by two.
- 16-bit data
  - The physical address is the logical address shifted right by one.
- 32-bit data
  - The physical address is the logical address.

$\overline{\text{IOSTRB}}$ can access 32-bit data from 32-bit-wide memory. However, $\overline{\text{IOSTRB}}$ does not have the flexibility of $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ because it is composed of a single signal, i.e., $\overline{\text{IOSTRB}}$. $\overline{\text{IOSTRB}}$ bus cycles differ from $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ bus cycles. (See section 7.4 of the TMS320C32 User's Guide [literature number SPRU132] for additional information regarding $\overline{\text{IOSTRB}}$ bus cycles.) This timing difference accommodates slower I/O peripherals.

The 'C32 also supports program execution from 16- and 32-bit external memory widths. Execution is controlled through the status of the PRGW pin. When this pin is *pulled high*, the 'C32 executes from 16-bit-wide memory. When the PRGW pin is *pulled low*, the 'C32 executes from 32-bit-wide memory. For 16-bit-wide, zero wait-state memory, the 'C32 takes two instruction cycles to fetch a single 32-bit instruction. The lower 16 bits of the instruction are obtained during the first cycle; during the second cycle, the upper 16 bits are retrieved and concatenated with the lower 16 bits. The 'C32's 32-bit memory fetches are identical to those of the TMS320C30 and TMS320C31.

In summary, the 'C32 memory interface parallel bus implements three mutually-exclusive address spaces that are distinguished through the use of three separate control signals (see Figure 2). $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ support 8-, 16-, and 32-bit data access in 8-, 16-, and 32-bit-wide external memory and 32-bit program access in 16/32-bit wide external memory. $\overline{\text{IOSTRB}}$ address space supports 32-bit data/program access in

32-bit-wide external memory. Internally, the ’C32 has a 32-bit architecture, hence the memory interface accordingly packs and unpacks the data accessed. Three strobe control registers manipulate the variable-width memory interface of the ’C32. The following subsections describe these registers.

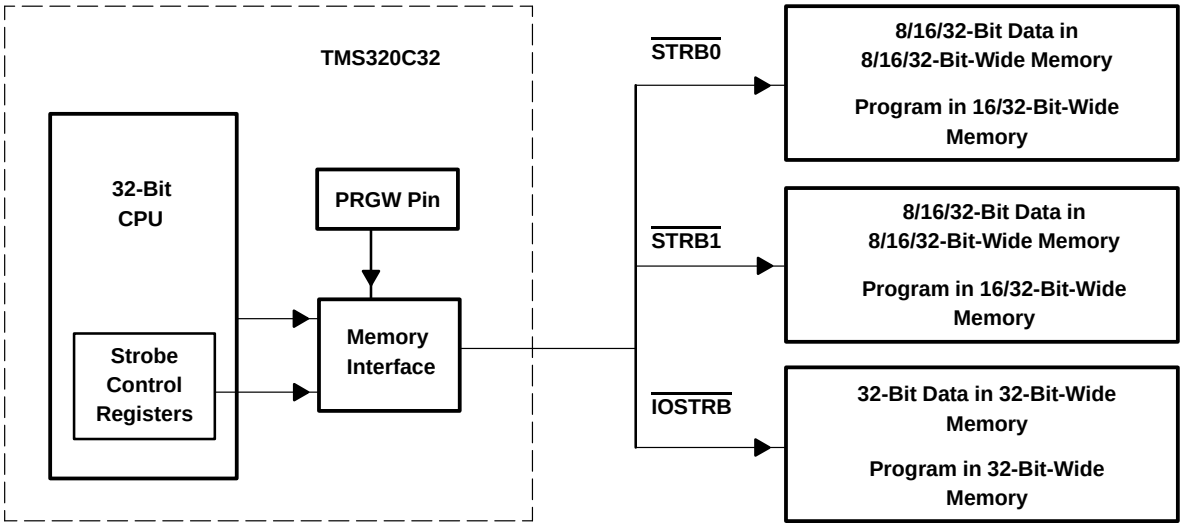

Figure 2. TMS320C32 Memory Address Spaces

$\overline{\text{STRB0}}$ Control Register

The $\overline{\text{STRB0}}$ control register (see Figure 3) is a 32-bit register that contains control bits for the portion of external bus memory space that is mapped to $\overline{\text{STRB0}}$. Figure 3 lists the register bits, bit names, and functions. (Shaded entries denote bit fields present only in the ’C32.) At system reset, 0F10F8h is written to the $\overline{\text{STRB0}}$ control register if the PRGW pin is *logic low*; 0710F8h is written to this register when the PRGW pin is *logic high*.

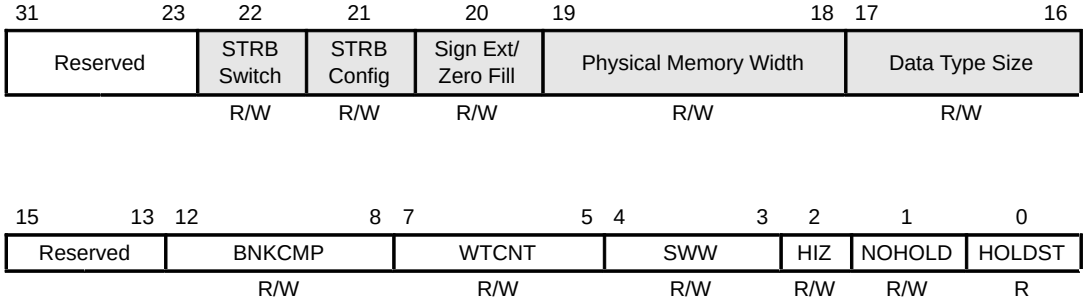

Figure 3. $\overline{\text{STRB0}}$ Control Register

$\overline{\text{STRB1}}$ Control Register

The $\overline{\text{STRB1}}$ control register (see Figure 4) is a 32-bit register that contains control bits for the portion of external bus memory space that is mapped to $\overline{\text{STRB1}}$. Figure 4 lists the register bits, bit names, and functions. (Shaded entries denote bit fields present only in the 'C32.) At system reset, 0F10F8h is written to the $\overline{\text{STRB1}}$ control register if the PRGW pin is *logic low*; 0710F8h is written to this register when the PRGW pin is *logic high*.

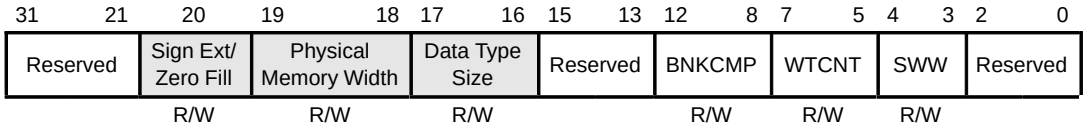

Figure 4. $\overline{\text{STRB1}}$ Control Register

$\overline{\text{IOSTRB}}$ Control Register

The $\overline{\text{IOSTRB}}$ control register (see Figure 5) is a 32-bit register that contains control bits for the portion of external bus memory space that is mapped to $\overline{\text{IOSTRB}}$. Unlike $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$, there is no byte-enable signal for the $\overline{\text{IOSTRB}}$ register. Data access through $\overline{\text{IOSTRB}}$ is always 32 bits. Figure 5 shows the register bits, bit names, and functions. At system reset, 0F8h is written to the $\overline{\text{IOSTRB}}$ control register. $\overline{\text{IOSTRB}}$ timing is identical to that of the 'C30's $\overline{\text{IOSTRB}}$ control register.

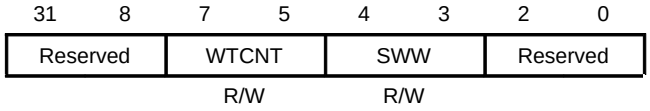

Figure 5. $\overline{\text{IOSTRB}}$ Control Register

Data-Type-Size Field

The data-type-size fields indicate the size of the data type that has been written to memory. This field can have the values listed in Table 1.

Table 1. Data Type Size Field

BIT 17	BIT 16	DATA TYPE SIZE
0	0	8 bits
0	1	16 bits
1	0	Reserved
1	1	32 bits

NOTE: The shaded entry indicates the reset value.

Physical-Memory-Width Field

The physical-memory-width field indicates the size of the physical memory connected to the device. The reset value is dependent upon the status of the PRGW pin. If the PRGW pin is *logic low*, the physical memory width is configured to 32 bits (bit 18 = 1, bit 19 = 1). If the PRGW pin is *logic high*, the physical-memory-width is configured to 16 bits (bit 18 = 1, bit 19 = 0). These fields can have the following values listed in Table 2.

Table 2. Physical-Memory-Width Field

BIT 19	BIT 18	PHYSICAL MEMORY WIDTH
0	0	8 bits
0	1	16 bits
1	0	Reserved
1	1	32 bits

NOTE: The shaded entries indicate reset values. The reset value is dependent upon the state of the PRGW pin.

Setting the physical-memory-width field of the $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$ control registers changes the functionality of the $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$ signals as follows:

- When the physical-memory-width field is configured to 32 bits, the corresponding $\overline{\text{STRBx\_B0}}$ – $\overline{\text{STRBx\_B3}}$ signals are configured as byte-enable pins.
- When the physical-memory-width field is configured to 16 bits, the corresponding $\overline{\text{STRBx\_B3}}$/A_1 signal is configured as an address pin while $\overline{\text{STRBx\_B0}}$ and $\overline{\text{STRBx\_B1}}$ signals are configured as byte-enable pins.
- When the physical-memory-width field is configured to 8 bits, the $\overline{\text{STRBx\_B3}}$/A_1 and $\overline{\text{STRBx\_B2}}$/A_2 signals are configured as addresses while $\overline{\text{STRBx\_B0}}$ is configured as byte enable.
- Once a $\overline{\text{STRBx\_Bx}}$ signal is configured as an address pin, it is active for any external memory access, i.e., $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, $\overline{\text{IOSTRB}}$, and external program fetches.

Sign Ext/Zero Fill Field

The sign ext(ension)/zero fill field indicates the conversion method to be applied upon 8- and 16-bit integer data transferred from external memory to an internal register or memory location. This field can contain the values listed in Table 3.

Table 3. Sign Ext/Zero Fill Function

BIT 20	SIGN EXT/ZERO FILL FUNCTION DESCRIPTION
0	Sign-extend 8- or 16-bit integers to 32 bits.
1	Zero-fill 8- or 16-bit integers to make 32-bit numbers.

NOTE: The shaded entry indicates the reset value.

Integer loads (8- and 16-bits) are stored in the least significant bits of 'C32 registers/memory. The most significant bits are sign-extended or zero-filled according to the setting of this bit field.

STRB Config Field

The STRB Config field indicates if the $\overline{\text{STRB0\_Bx}}$ signals are active when data is being accessed from either $\overline{\text{STRB0}}$ or $\overline{\text{STRB1}}$ memory space. This mode is useful when accessing a single external memory bank that stores two different data types, each mapped to a different STRB. This field can have the values listed in Table 4.

Table 4. STRB Configuration Function

BIT 21 (STRB0 ONLY)	STRB CONFIG FUNCTION DESCRIPTION
0	$\overline{\text{STRB0\_Bx}}$ signals are active for address locations 0h–7FFFFFh and 880000h–8FFFFFh. $\overline{\text{STRB1\_Bx}}$ signals are active for address locations 900000h– FFFFFFh.
1	$\overline{\text{STRB0\_Bx}}$ signals are active for address locations 0h–7FFFFFh, 880000h–8FFFFFh, and 900000h– FFFFFFh. $\overline{\text{STRB1\_Bx}}$ signals are active for address locations 900000h– FFFFFFh.

NOTE: The shaded entry indicates the reset value.

STRB Switch Field

The STRB switch field indicates whether or not a single cycle is inserted between back-to-back reads when crossing $\overline{\text{STRB0}}$-to-$\overline{\text{STRB1}}$ or $\overline{\text{STRB1}}$-to-$\overline{\text{STRB0}}$ boundaries (switching STRBs). The extra cycle toggles the strobe signal during back-to-back reads; otherwise, the strobe remains low. This field can contain the values listed in Table 5.

Table 5. STRB Switch Function

BIT 22 (STRB0 ONLY)	STRB SWITCH FUNCTION DESCRIPTION
0	Does not insert a single cycle between back-to-back reads that switch from $\overline{\text{STRB0}}$ to $\overline{\text{STRB1}}$ or vice versa.
1	Inserts a single cycle between back-to-back reads when switching from $\overline{\text{STRB0}}$ to $\overline{\text{STRB1}}$ or vice versa.

NOTE: The shaded entry indicates the reset value.

C Compiler Interaction With the TMS320C32 Memory Interface

The ’C32s internal 32-bit architecture allows the C compiler’s data types to remain at 32 bits. However, the C compiler’s runtime-support library includes pragma directives and new dynamic-allocation routines (malloc, realloc, calloc, bmalloc, free, etc.) that support the creation of data sections. These data sections serve as memory pools, or heaps, for storing 8- and 16-bit data. These sections can reside in 8-, 16-, and 32-bit-wide memory. The programmer must ensure that the appropriate strobe control register is loaded with the correct data size and memory width. The ’C32’s memory interface truncates, packs, or unpacks the data in the manner specified by the settings of the strobe control register. Table 6 lists the data sizes supported by the different sections created by the C compiler.

Table 6. Data Sizes Supported by Sections Created by the C Compiler

	DATA SIZE		
SECTION TYPE	**32 BITS**	**16 BITS**	**8 BITS**
Initialized	.text .cinit .const *.user_section*	*.user_section*	*.user_section*
Uninitialized	.bss .stack .sysmem *.user_section*	.sysm16 *.user_section*	.sysm8 *.user_section*

The contents of the above-named sections are as follows:

- **.text**: executable code and/or string literals
- **.cinit**: tables for variable and constant initialization
- **.const**: string literals and switch tables
- **.bss**: global variables and statically-allocated variables
- **.stack**: system stack used to pass function arguments and to allocate local function variables
- **.sysmem**: memory pool for dynamic allocation of 32-bit data
- **.sysm16**: memory pool for dynamic allocation of 16-bit data
- **.sysm8**: memory pool for dynamic allocation of 8-bit data
- **.user_section**: section created using the #pragma DATA_SECTION directive

The following section describes the C compiler’s preprocessor pragma. Subsequent sections discuss modules that were incorporated into the runtime-support library, which supports 8- and 16-bit memory pools. 32-bit memory pools are handled through the standard minit( ), malloc( ), calloc( ), realloc( ), and free( ) routines which operate on the .sysmem section.

DATA_SECTION Pragma Directive

To support additional memory pools, the C compiler utilizes a data section pragma directive. This directive instructs the C compiler to allocate space for *symbol_name* in the section specified by *section_name* of size *symbol_size*. Refer to section 3.4 in the *TMS320 Floating-Point DSP Optimizing C Compiler User’s Guide* (literature number SPRU034E) for additional information. The syntax for DATA_SECTION is as follows:

```
#pragma DATA_SECTION(symbol_name, “section_name”)
type symbol_name;
```

For example, define a new section called .mydata as an array of 1K integer values in the following manner:

```
#pragma DATA_SECTION(dataBuf, “.mydata”)
```

```
int dataBuf[1024];
```

MEMORY8.C

The MEMORY8.C module contains functions that implement dynamic memory management routines for using 8-bit data with the ’C32. The following subsections describe each of these functions. (Refer to the appendix for additional information on 8-bit runtime-support functions.)

Pragma Directive

MEMORY8.C contains a pragma directive that defines a .sysm8 section. The size of this memory pool in words (system memory or heap) is set at link time by using the -heap8 option. If the -heap8 option is not used, the compiler does not allocate an 8-bit system memory area. If arguments are not used in conjunction with this switch, the size of the 8-bit system memory area defaults to 1K 8-bit words.

minit8() Function

The minit8( ) function initializes and resets the 8-bit dynamic memory management system. This function is analogous to minit( ) with the exception that minit8( ) operates in the 8-bit .sysm8 section.

malloc8() Function

The malloc8( ) function allocates 8-bit words from the 8-bit memory pool and returns a pointer to the allocated space. This function is analogous to malloc( ) with the exception that malloc8( ) operates in the 8-bit .sysm8 section.

calloc8() Function

The calloc8( ) function allocates 8-bit words from the 8-bit memory pool, clears allocated memory locations, and returns a pointer to the allocated space. This function is analogous to calloc( ) with the exception that calloc8( ) operates in the 8-bit .sysm8 section.

realloc8() Function

The realloc8( ) function reallocates 8-bit words from previously unallocated areas in the 8-bit memory pool; a pointer to the allocated space is also returned. This function is analogous to the realloc( ) function with the exception that realloc8( ) operates in the 8-bit .sysm8 section.

free8() Function

The free8( ) function frees previously allocated space from the 8-bit memory pool. This function is analogous to free( ) with the exception that free8( ) operates in the 8-bit .sysm8 section.

bmalloc8() Function

The bmalloc8( ) function allocates 8-bit words from the 8-bit memory pool. The allocated words are aligned to a boundary that is suitable for the ’C32’s circular- and bit-reversed buffers; a pointer to the allocated space is also returned. This function is analogous to bmalloc( ) with the exception that bmalloc8( ) operates in the 8-bit .sysm8 section.

_SYSMEM8_SIZE Label

_SYSMEM8_SIZE is an external label that contains the size, in words, of the 8-bit system memory pool.

MEMORY16.C

The MEMORY16.C module contains functions that implement dynamic memory management routines for the ’C32’s 16-bit data. The following subsections describe each of these functions. (Refer to the appendix for additional information on 16-bit runtime-support functions.)

Pragma Directive

MEMORY16.C contains a pragma directive that defines a .sysm16 section. The size of this memory pool in words (system memory or heap) is set at link time by using the -heap16 option. If the -heap16 option is not used, the compiler does not allocate a 16-bit system memory area. If arguments are not used in conjunction with this switch, the size of the 16-bit system memory area defaults to 1K 16-bit words.

minit16() Function

The minit16( ) function initializes and resets the 16-bit dynamic memory management system. This function is analogous to minit( ) with the exception that minit16( ) operates in the 16-bit .sysm16 section.

malloc16() Function

The malloc16( ) function allocates 16-bit words from the 16-bit memory pool and returns a pointer to the allocated space. This function is analogous to malloc( ) with the exception that malloc16( ) operates in the 16-bit .sysm16 section.

calloc16() Function

The calloc16( ) function allocates 16-bit words from the 16-bit memory pool, clears allocated memory locations, and returns a pointer to the allocated space. This function is analogous to calloc( ) with the exception that calloc16( ) operates in the 16-bit .sysm16 section.

realloc16() Function

The realloc16( ) function reallocates 16-bit words from previously unallocated areas in the 16-bit memory pool; a pointer to the allocated space is also returned. This function is analogous to the realloc( ) function with the exception that realloc16( ) operates in the 16-bit .sysm16 section.

free16() Function

The free16( ) function frees previously allocated space from the 16-bit memory pool. This function is analogous to free( ) with the exception that free16( ) operates in the 16-bit .sysm16 section.

bmalloc16() Function

The bmalloc16( ) function allocates 16-bit words from the 16-bit memory pool. The allocated words are aligned to a boundary that is suitable for the ’C32’s circular- and bit-reversed buffers; a pointer to the allocated space is also returned. This function is analogous to bmalloc( ) with the exception that bmalloc16( ) operates in the16-bit .sysm16 section.

SYSMEM16 SIZE Label

_SYSMEM16_SIZE is an external label that contains the size, in words, of the 16-bit system memory pool.

Memory Pool Limitations

The ’C32 has only three strobes: $\overline{\text{STRB0}}$, $\overline{\text{STRB1}}$, and $\overline{\text{IOSTRB}}$. Therefore, a programmer cannot have more than three memory pools, i.e., one memory pool assigned to each strobe. $\overline{\text{IOSTRB}}$ can hold only 32-bit data and can only accommodate the 32-bit memory pool .sysmem. Conversely, $\overline{\text{STRB0}}$ and $\overline{\text{STRB1}}$ can hold 8-, 16-, and 32-bit data and can accommodate the 8-, 16-, and 32-bit memory pools .sysm8, .sysm16, and .sysmem.

All pointers and constants must be stored in memory configured to hold 32-bit data. Hence, .bss, .stack, .cinit, and .const sections must reside in memory with data size configured to 32 bits.

C Compiler and Assembler Switch

To create code for the 'C32, the assembler and C compiler utilize the -v32 version specification switch. The following example demonstrates the use of this switch with the assembler and C compiler, respectively:

```
asm30  -v32   myfile.asm
cl30   -v32   myfile.c
```

Linker Switches

To support the 'C32's 8- and 16-bit memory pools, the linker utilizes the following switches: -heap8, -heap16, and -heap. These switches set the size, in words, of the respective 8-, 16-, and 32-bit memory system areas .sysm8, .sysm16, and .sysmem. The user must link these sections into the appropriate addresses thereby activating strobes that are configured to access 8-, 16-, or 32-bit data.

The following example demonstrates the link-time sizing of an 8-bit memory pool to 256K words:

```
lnk30 -heap8 0x4000
```

The linker creates these memory system areas using an input file that contains the .sysmem, .sysm8, and .sysm16 data-section definitions. If the input file does not exist, the linker is unable to perform memory area processing.

The linker also creates the global symbols _SYSMEM_SIZE, _SYSMEM8_SIZE, and _SYSMEM16_SIZE and subsequently assigns each a value equal to the respective -heap, -heap8, and -heap16 size. The default size for each memory system area is 1K words (word size is dependent upon system memory width).

Debugger Configuration

For the debugger to properly disassemble and read/write external memory, the user must configure the strobe control registers before loading and executing code. Because the 'C32 supports code execution from 16- or 32-bit memory, the debugger may need to temporarily set the strobe control register to a 32-bit data size in order to write an instruction (either by loading code or patching code) or to read an instruction with the objective of disassembling a range of program memory.

To support code execution from 16- and 32-bit memory, the Memory Map Add command includes a new *type* parameter that directs the debugger to treat .text sections as 32-bit data. While reading or writing .text sections, the debugger does the following:

- Temporarily stores the configuration of the appropriate strobe control register
- Temporarily sets the data size to 32 bits
- Reads or writes the targeted portion of the .text section
- Restores the strobe control register to its previous value

The syntax for the Memory Map Add command is:

ma *address, length, type*

address: defines the starting address of a range of memory

length: defines the length of the memory range

type: identifies the read/write characteristic of the memory range depending upon one or more of the following keywords:

- **R**: read only
- **W**: write only
- **WR** or **RAM**: read/write
- **PROTECT**: no-access memory
- **TX**: memory that stores *.text* (code) section

TMS320C32 Configuration Examples

The following subsections demonstrate potential ’C32 memory interface configurations. Also included are instructions on how to allocate buffers, build link files, and configure the debugger for each memory configuration.

Example 1. Two External Memory Banks

The ’C32’s external memory interface allows the use of two zero wait-state external memory banks with different widths without requiring additional logic or incurring access penalty costs. These external memory banks provide the programmer with flexibility in trading performance versus system cost, i.e., amount of memory chips. For example, the programmer could execute code from 32-bit wide memory

while storing data in 8-bit memory (see Figure 6). This approach would be advantageous for applications with large amounts of 8-bit data that require execution at the fastest speed of the device.

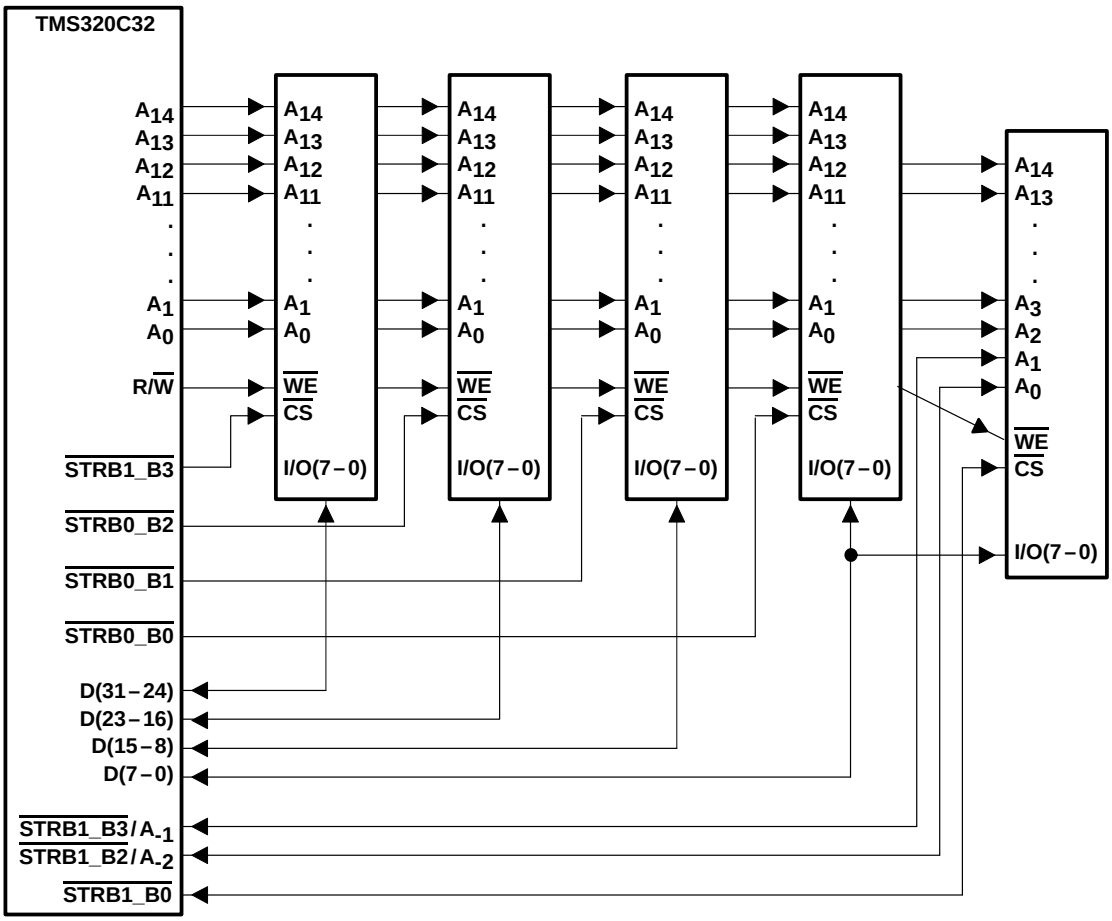

Figure 6. Zero Wait-State Interface for 32- and 8-Bit SRAM Banks

In Figure 6, a bank of 32K × 32 bits is mapped to $\overline{\text{STRB0}}$, and a bank of 32K × 8 bits is mapped to $\overline{\text{STRB1}}$. For this configuration, the programmer must set the following:

- $\overline{\text{STRB0}}$ control register physical memory width to 32 bits and the data type size to 32 bits
- STRB Config bit field to zero, i.e., $\overline{\text{STRB0}}$ control register = 000F0000h (banks are separate)
- $\overline{\text{STRB1}}$ control register physical memory width to 8 bits and the data type size to 8 bits, i.e., $\overline{\text{STRB1}}$ control register = 00000000h

Additionally, the PRGW pin must be *pulled low* to indicate 32-bit program memory width.

Figure 6 also maps the 32-bit wide bank's external memory address pins, $A_{14}A_{13}...A_1A_0$, to the 'C32's $A_{14}A_{13}A_{12}...A_1A_0$ pins. Conversely, the 8-bit wide bank's memory address pins, $A_{14}A_{13}...A_1A_0$, are mapped to the 'C32s $A_{12}...A_1A_0A_{-1}$ pins. Because $\overline{\text{STRB1}}$ is configured for 8-bit memory width, the external address presented on 'C32 pins is shifted right by two bits. As a result of this mapping, external memory accesses in the range 0h through 7FFFh read or write 32-bit data to the 32-bit-wide bank ($\overline{\text{STRB0}}$). Memory accesses in the range 900000h through 907FFFh read or write 8-bit data to the 8-bit-wide bank ($\overline{\text{STRB1}}$).

Two banks of different memory widths should not be connected to the same STRB without external decode logic. Different memory widths require STRBx_Bx signals to be configured as address pins. These address pins are active for any external memory access, i.e., STRB0, STRB1, IOSTRB, and program fetches.

Dynamic Memory Allocation

Examples of 8-bit dynamic buffer allocation, linker configuration, and a debugger batch file are provided in the following subsections.

C Code

The following C code demonstrates the allocation of two buffers (1K and 4K, 8-bit words) using the 8-bit dynamic memory allocation routines.

```
void main( )
{
   int    *buffer1;
   float  *buffer2;
   /* Configure the STRB0 control register for 32-bit wide memory, 32-bit
   data size. */
   *0x808064 = 0xF0000;
   /* Configure the STRB1 control register for 8-bit wide memory, 8-bit data
   size. */
   *0x808068 = 0x00000;
   /* Allocate 1K 8-bit words in the 8-bit memory pool. */
   buffer1 = malloc8(1024 * sizeof(int) );
   /* Allocate 4K 8-bit floats in the 8-bit memory pool. */
   buffer2 = malloc8(4096 * sizeof(float) );
   /* Process buffers. */
   callDSPoperation(buffer1, buffer2);
   /* Free buffers. */
   free8(buffer2);
   free8(buffer1);
}
```

NOTE: The TMS320 floating-point C compiler *sizeof* function returns **1** for both integers and float data types.

Linker Command File

The following linker command file allocates sections of the above code into the desired memory configuration.

```
sample.obj             /* Input filename                */
–heap8 32768           /* Set 8-bit memory pool size.   */
–stack 8704            /* Set C system stack size.      */
–o sample.out          /* Specify output file.          */
```

```
–m sample.map          /* Specify map file.             */
MEMORY
{
   PRGRAM    :  org = 0x0000,       len = 0x2000
   STRB0RAM  :  org = 0x2000,       len = 0x6000
   ONCHIRAM  :  org = 0x87Fe00,     len = 0x200
   STRB1RAM  :  org = 0x900000,     len = 0x8000
}
SECTIONS
{
   .text  > PRGRAM        /* 32-bit data section             */
   .cinit > STRB0RAM      /* 32-bit data section             */
   .const > STRB0RAM      /* 32-bit data section             */
   .bss   > STRB0RAM      /* 32-bit data section             */
   .stack > STRB0RAM      /* 32-bit data section             */
   .sysm8 > STRB1RAM      /* 8-bit memory pool mapped to STRB1 */
}
```

Debugger Batch File

The following debugger batch file executes initialization commands that configure the C source debugger to handle a ’C32 with the memory configuration shown in Figure 6.

```
mr
sconfig init.clr
;   Define memory configuration.
ma 0x0000, 0x2000, R|W|TX   ; Inform debugger that this section holds code
                              (.text).
ma 0x2000, 0x6000, RAM      ; No code here, STRB0
ma 0x87FE00, 0x200, RAM     ; On-chip
ma 0x808000, 0x10, RAM      ; Peripheral Bus Control – DMA
ma 0x808020, 0x20, RAM      ; Peripheral Bus Control – Timers
ma 0x808040, 0x10, RAM      ; Peripheral Bus Control – Serial Port 0
ma 0x808060, 0x10, RAM      ; Peripheral Bus Control – External Memory Interface
ma 0x900000, 0x8000, RAM    ; STRB1
;
reset
map on                      ; Make emulator aware of this memory configuration.
;
?*0x808064 = 0xF0000        ; Set STRB0 control register to 32-bit memory width,
                            ; 32-bit data size.
?*0x808068 = 0x00000        ; Set STRB1 control register to 8-bit memory width,
                            ; 8-bit data size.
```

```
;
load sample.out              ; Configure STRB0 and STRB1 control registers before
                             ; loading code.
```

Static Memory Allocation

Examples of 8-bit static buffer allocation and associated linker configuration are provided in the following subsections. The debugger batch file (not shown) is identical to the batch file previously listed under Dynamic Memory Allocation.

C Code

The following C code demonstrates the static allocation of two buffers (1K and 4K, 8-bit words) by defining a user section called .mydata8. This section is used to hold a structure consisting of two arrays of data values.

```
#pragma DATA_SECTION(buffer8, ".mydata8")
struct bufferStruct {
   in[1024];
   out[4096];
} buffer8;
void main( )
{
   /* Configure the STRB0 control register for 32-bit wide memory, 32-bit
   data size. */
   *0x808064 = 0xF0000;
   /* Configure the STRB1 control register to 8-bit wide memory, 8-bit data
   size. */
   *0x808068 = 0x00000;
   /* Process buffers. */
   callDSPoperation(buffer8.in, buffer8.out);
}
```

Linker Command File

The following linker command file allocates sections of the above C code into the desired memory configuration.

```
sample.obj             /* Input filename             */
–stack 8704            /* Set C system stack size.   */
–o sample.out          /* Specify output file.       */
–m sample.map          /* Specify map file.          */
MEMORY
{
   PRGRAM    :  org = 0x0000,    len = 0x2000
   STRB0RAM  :  org = 0x2000,    len = 0x6000
   ONCHIRAM  :  org = 0x87Fe00,  len = 0x200
```

```
   STRB1RAM  :  org = 0x900000,  len = 0x8000
}
SECTIONS
{
   .text  > PRGRAM         /* 32-bit data section              */
   .cinit > STRB0RAM       /* 32-bit data section              */
   .const > STRB0RAM       /* 32-bit data section              */
   .bss   > STRB0RAM       /* 32-bit data section              */
   .stack > STRB0RAM       /* 32-bit data section              */
   .mydata8 > STRB1RAM     /* 8-bit memory pool mapped to STRB1 */
}
```

Example 2. Single External Memory Bank

Consider the case of a typical audio compression application written in C that requires 32-bit data for the system stack and 16-bit data for the audio buffers. In this case, the programmer could interface the ’C32 as shown in Figure 7. This example assumes 32K 32-bit words of external memory. This memory is further defined as containing 8.5K 32-bit words of stack and 8K 32-bit words of program space; both areas are mapped to STRB0 (program space includes constants and global/static variables) Also, external memory contains 32K of 16-bit word data buffers that are mapped into STRB1.

Due to this mapping, the programmer must set the following:

- STRB0 control register physical memory width to 32 bits and the data type size to 32 bits
- STRB Config bit field to one, i.e., STRB0 control register = 002F0000h
- STRB1 control register physical memory width to 32 bits and the data type size to 16 bits, i.e., STRB1 control register = 000D0000h

Additionally, the PRGW pin must be *pulled low* to indicate 32-bit program memory width.

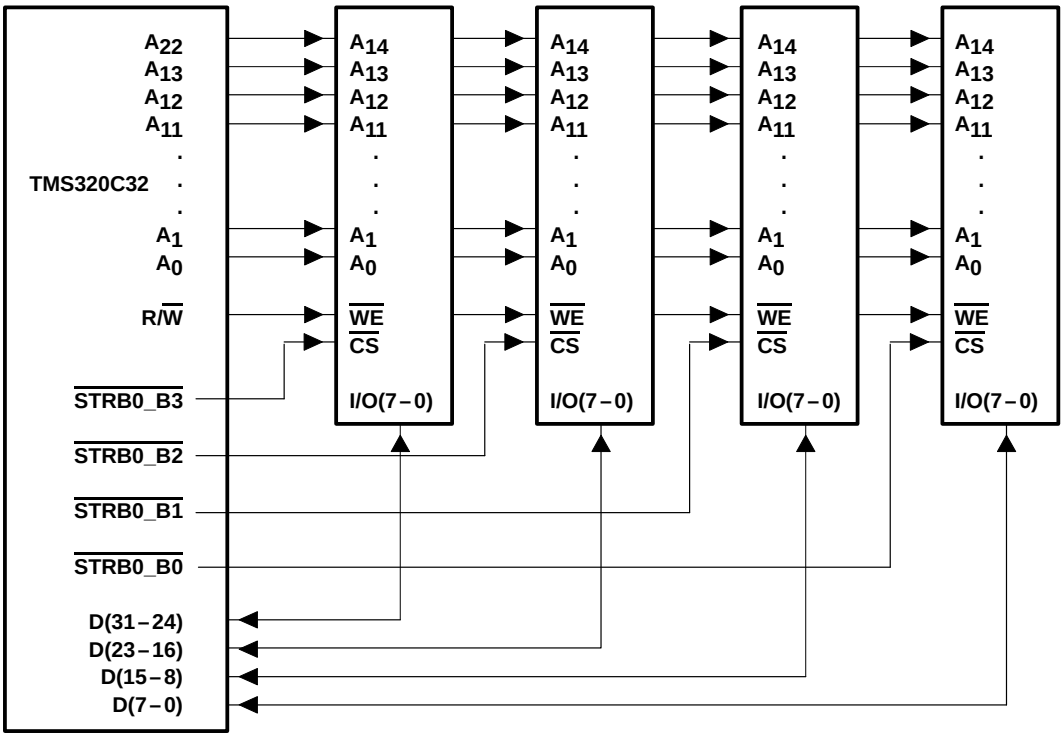

Figure 7. Zero Wait-State Interface for 32-Bit SRAMs with 16- and 32-Bit Data Accesses

The external memory address pins $A_{14}A_{13}...A_1A_0$ are mapped to the 'C32's $A_{22}A_{13}A_{12}...A_1A_0$ pins. This mapping was selected to position the system stack immediately after the 'C32's internal RAM. Performance is consequently improved as the top of the stack resides in internal RAM, and the stack is allowed to grow into external RAM. With this mapping, external memory accesses in the range 4000h through 7FFFh read or write 16-bit data; memory accesses in the range 0h through 3FFFh read or write 32-bit data. The PRGW pin controls the program fetches.

Figure 8 shows the contents of external memory. Due to the address shift of the ’C32’s external memory interface, the memory map seen by the ’C32 CPU is slightly different (see Figure 9).

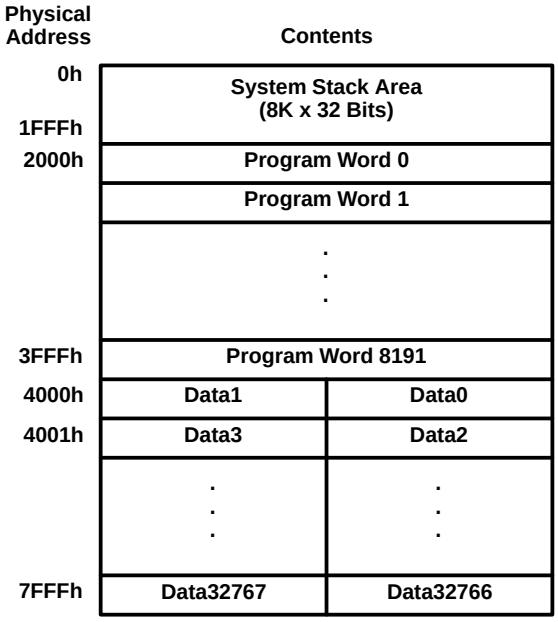

Figure 8. External Memory Map

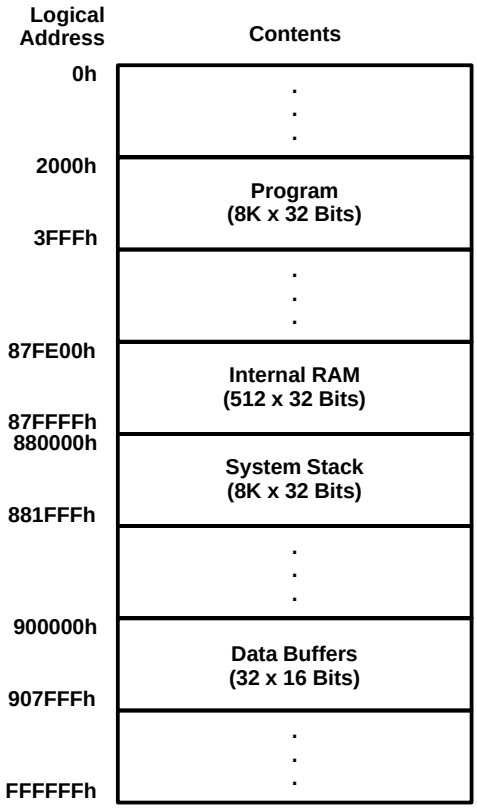

Figure 9. TMS320C32 Memory Map

NOTE: For 32-bit data, physical address = logical address. For 16-bit data, physical address = logical address shifted left by one.

Dynamic Memory Allocation

Examples of 16-bit dynamic buffer allocation, linker configuration, and a debugger batch file are provided in the following subsections.

C Code

The following C code demonstrates the allocation of two buffers (1K and 4K, 16-bit words) using the 16-bit dynamic memory allocation routines provided by the runtime-support library.

```
# include <bus30.h>
void main( )
{
   int    *buffer1;
   float  *buffer2;
   /* Configure the STRB0 control register to STRB0 and STRB1 overlay. */
   /* 32-bit wide memory, 32-bit data size */
```

```
   /* If using the PRTS30 headers,
      BUS_ADDR->STRB0_gcontrol = STRB0_1_CNFG | MEMW_32 | DATA_32; */
   *0x808064 = 0x2F0000;
   /* Configure STRB1 control register to 32-bit wide memory, 16-bit data
   size. */
   /* If using the PRTS30 headers,
      BUS_ADDR->STRB1_gcontrol = MEMW_32 | DATA_16; */
   *0x808068 = 0xD0000;
   /* Allocate 1K 16-bit words in the 16-bit memory pool. */
   buffer1 = malloc16(1024 * sizeof(int) );
   /* Allocate 4K 16-bit floats in the 16-bit memory pool. */
   buffer2 = malloc16(4096 * sizeof(float));
   /* Process buffers. */
   callDSPoperation(buffer1, buffer2);
   /* Free buffers. */
   free16(buffer2);
   free16(buffer1);
}
```

Linker Command File

The following linker command file allocates sections of the above C code into the memory configuration depicted by Figure 8.

```
sample.obj             /* Input filename                */
-heap16 32768          /* Set 16-bit memory pool size.  */
-stack 8704            /* Set C system stack size.      */
-o sample.out          /* Specify output file.          */
-m sample.map          /* Specify map file.             */
MEMORY
{
   STRB0RAM  :  org = 0x2000,  len = 0x2000
   STACKRAM  :  org = 0x87Fe00, len = 0x2200
   STRB1RAM  :  org = 0x900000, len = 0x8000
}
SECTIONS
{
   .text  > STRB0RAM      /* 32-bit data section                */
   .cinit > STRB0RAM      /* 32-bit data section                */
   .const > STRB0RAM      /* 32-bit data section                */
   .bss   > STRB0RAM      /* 32-bit data section                */
```

```
    .stack > STACKRAM       /* 32-bit data section                  */
    .sysm16 > STRB1RAM      /* 16-bit memory pool mapped to STRB1   */
}
```

Debugger Batch File

The following debugger batch file executes initialization commands that configure the C source debugger to handle a ’C32 with the memory configuration shown in Figure 8.

```
mr
sconfig init.clr
;   Define memory configuration.
ma 0x2000, 0x2000, R|W|TX     ; Inform debugger that this section holds code (.text).
ma 0x87FE00, 0x2000, RAM
ma 0x900000, 0x8000, RAM
map on                        ; Make emulator aware of this memory configuration.
?*0x808064 = 0x2F0000         ; Set STRB0 control register to STRB0 and STRB1 overlay.
                              ; 32-bit memory width, 32-bit data size
                              ;
?*0x808068 = 0xD0000          ; Set STRB1 control register.
                              ; 32-bit memory width, 16-bit data size
                              ;
load sample.out               ; Configure STRB0/STRB1 control registers before loading
                                code.
```

References

1. TMS320C32 Microcontroller User's Guide, Texas Instruments, 1995, literature number SPRU132, pp. 7–3, 7–8.
2. TMS320 Floating-Point DSP Optimizing C Compiler User's Guide, Texas Instruments, 1995, literature number SPRU034E, p. 5–35.
3. TMS320 Floating-Point DSP Assembly Language Tools User's Guide, Texas Instruments, 1990, literature number SPRU035, p. 8–5.
4. TMS320C3x C Source Debugger User's Guide, Texas Instruments, 1993, literature number SPRU053, p. 11–25.

Appendix

MEMORY8.C Runtime Support Functions

minit8() Reset Dynamic Memory Pool

Syntax: #include <stdlib.h>

Defined in: void minit8(void);

Description: The minit8( ) function resets all of the 8-bit memory pool that previously was allocated by calls to the malloc8( ), calloc8( ), and realloc8( ) functions.

NOTE: Calling the minit8( ) function makes all of the heap8 memory space available again. Any objects previously allocated will be lost, i.e., can no longer be accessed.

minit8( ) uses memory from a special memory pool, or heap, that is defined in the uninitialized .sysm8 section in MEMORY8.C. The linker sets the size of this section from the value specified by the -heap8 option. The default heap size is 1K words. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* (literature number SPRU034E).

malloc8( ) Allocate Memory

Syntax: #include <stdlib.h>

Defined in: void *malloc8(size_t size);

Description: The malloc8( ) function allocates size 8-bit words from the 8-bit memory pool and returns a pointer to the allocated space. This function does not modify memory that it allocates. If malloc8( ) cannot allocate space, i.e., there is no available memory, it returns a null pointer (0).

malloc8( ) uses memory from a special memory pool, or heap, that is defined in the uninitialized .sysm8 section in MEMORY8.C. The linker sets the size of this section from the value specified by the -heap8 option. The default heap size is 1K words. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* (literature number SPRU034E).

Example: This example allocates free space for a structure.

```
struct xyz *p;
p = malloc8(sizeof (struct xyz));
```

calloc8() Allocate and Clear Memory

Syntax: #include <stdlih.h>

Defined in: void *calloc8(size_t nmemb, size_t size);

Description: The calloc8() function allocates size 8-bit words from the 8-bit memory pool for each of nmemb objects and returns a pointer to the allocated space. Allocated memory is initialized to all 0s. If calloc8() cannot allocate memory, i.e., there is no available memory, it returns a null pointer (0).

calloc8() uses memory from a special memory pool, or heap, that is defined in the uninitialized .sysm8 section in MEMORY8.C. The linker sets the size of this section from the value specified by the -heap8 option. The default heap size is 1K words. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* (literature number SPRU034E).

Example: This example uses the calloc8() routine to allocate and clear twenty 8-bit words.

```
ptr = calloc8(10,2);   /*Allocate and clear twenty 8-bit words. */
```

realloc8() Change Heap Size

Syntax: #include <stdlib.h>

Defined in: void *realloc8(void *ptr, size_t size);

Description: The realloc8() function changes the size of the allocated memory pointed to by ptr to the number of 8-bit words specified by size. The contents of the memory space (up to the lesser of the old and new sizes) are not changed.

- If ptr is 0, realloc8() behaves like malloc8().
- If ptr points to unallocated space, the function takes no action and returns.
- If space cannot be allocated, memory is not changed, and realloc8() returns 0.
- If size = 0 and ptr is not null, realloc8() frees the space pointed to by ptr.

When an entire object must be moved in order to allocate more space, realloc8() returns a pointer to the new space. Any memory freed by this operation is deallocated. If an error occurs, realloc8() yields a null pointer (0).

realloc8() uses memory from a special memory pool, or heap, that is defined in the uninitialized .sysm8 section in MEMORY8.C. The linker sets the size of this section from the value specified by the -heap8 option. The default heap size is 1K words. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* (literature number SPRU034E).

free8() Deallocate Memory

Syntax: #include <stdlib.h>

Defined in: void free8(void *ptr);

Description: The free8( ) function deallocates memory space (pointed to by ptr) that previously was allocated by a malloc8( ), calloc8( ), or realloc8( ) call. This deallocation makes the memory space available again. free8( ) will not take action involving a request to free unallocated space, i.e., will only return to the point of the call. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* (literature number SPRU034E).

Example:

This example allocates ten 8-bit words and then frees them.

```
char *x
x = malloc8(10);      /* Allocate ten 8-bit words.*/
free8(x);             /* Free ten 8-bit words bytes. */
```

MEMORY16.C Runtime Support Functions

minit16() Reset Dynamic Memory Pool

Syntax: #include <stdlib.h>

Defined in: void minit16(void);

Description: The minit16( ) function resets all of the 16-bit memory pool that previously was allocated by calls to the malloc16( ), calloc16( ), and realloc16( ) functions.

NOTE: Calling the minit16( ) function makes all of the heap16 memory space available again. Any objects previously allocated will be lost, i.e., can no longer be accessed.

minit16( ) uses memory from a special memory pool, or heap, that is defined in the uninitialized .sysm16 section in MEMORY16.C. The linker sets the size of this section from the value specified by the -heap16 option. The default heap size is 1K words. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* (literature number SPRU034E).

malloc16() Allocate Memory

Syntax: #include <stdlib.h>

Defined in: void *malloc16(size_t size);

Description: The malloc16( ) function allocates size 16-bit words from the 16-bit memory pool and returns a pointer to the allocated space. This function does not modify memory that it allocates. If malloc16( ) cannot allocate space, i.e., there is no available memory, it returns a null pointer (0).

malloc16( ) uses memory from a special memory pool, or heap, that is defined in the uninitialized .sysm16 section in MEMORY16.C. The linker sets the size of this section from the value specified by the -heap16 option. The default heap size is 1K words. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* (literature number SPRU034E).

Example: This example allocates free space for a structure.

```
struct xyz *p;
p = malloc16(sizeof (struct xyz));
```

calloc16() Allocate and Clear Memory

Syntax: #include <stdlih.h>

Defined in: void *calloc16(size_t nmemb, size_t size);

Description: The calloc16( ) function allocates size 16-bit words from the 16-bit memory pool for each of nmemb objects and returns a pointer to the allocated space. Allocated memory is initialized to all 0s. If calloc16( ) cannot allocate memory, i.e., there is no available memory, it returns a null pointer (0).

calloc16( ) uses memory from a special memory pool, or heap, that is defined in the uninitialized .sysm16 section in MEMORY16.C. The linker sets the size of this section from the value specified by the -heap16 option. The default heap size is 1K words. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* (literature number SPRU034E).

Example: This example uses the calloc16( ) routine to allocate and clear ten 16-bit words.

```
ptr = calloc16(10,1); /*Allocate and clear ten 16-bit words. */
```

realloc16() Change Heap Size

Syntax: #include <stdlib.h>

Defined in: void *realloc16(void *ptr, size_t size);

Description: The realloc16( ) function changes the size of the allocated memory pointed to by ptr to the number of 16-bit words specified by size. The contents of the memory space (up to the lesser of the old and new sizes) are not changed.

- If ptr is 0, realloc16( ) behaves like malloc16( ).
- If ptr points to unallocated space, the function takes no action and returns.
- If space cannot be allocated, memory is not changed, and realloc16( ) returns 0.
- If size = 0 and ptr is not null, realloc16( ) frees the space pointed to by ptr.

When an entire object must be moved in order to allocate more space, realloc16( ) returns a pointer to the new space. Any memory freed by this operation is deallocated. If an error occurs, realloc16( ) yields a null pointer (0).

realloc16( ) uses memory from a special memory pool, or heap, that is defined in the uninitialized .sysm16 section in MEMORY16.C. The linker sets the size of this section from the value specified by the -heap16 option. The default heap size is 1K words. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* (literature number SPRU034E).

free16( ) Deallocate Memory

Syntax: #include <stdlib.h>

Defined in: void free16(void *ptr);

Description: The free16( ) function deallocates memory space (pointed to by ptr) that previously was allocated by a malloc16( ), calloc16( ), or realloc16( ) call. This deallocation makes the memory space available again. free16( ) will not take action involving a request to free unallocated space, i.e., will only return to the point of the call. For more information, refer to subsection 4.1.3, Dynamic Memory Allocation, on page 4-4 of the *TMS320 Floating-Point Optimizing C Compiler User's Guide* (literature number SPRU034E).

Example: This example allocates ten 16-bit words and then frees them.

```
char *x
x = malloc16(10);    /* Allocate ten 16-bit words. */
free16(x);           /* Free ten 16-bit words. */
```