TMS320C80 to TMS320C82 Software Compatibility User's Guide

SPRU154
November 1995

TEXAS INSTRUMENTS

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.

Read This First

About This Manual

The TMS320C80 and the TMS320C82 are the first two members of the ’C8x family of high-performance DSP devices. This guide explains how to port software developed for one of these devices to the other. It also presents a set of software compatibility guidelines for developing software that will run on either device.

How to Use This Manual

This document contains three chapters that deal with different issues related to compatibility:

- Chapter 1 discusses software-related differences between the ’C80 and the ’C82.
- Chapter 2 discusses guidelines and techniques for developing software that can run on both the ’C80 and the ’C82.
- Chapter 3 discusses guidelines and techniques for prototyping ’C82 software using a ’C80.

If you need background information about the differences between the ’C80 and the ’C82, read Chapter 1 before going on to Chapter 2 and Chapter 3.

If you already understand the differences between the ’C80 and the ’C82, then you can go directly to Chapter 2 and use Chapter 1 as reference material.

This user’s guide contains information at a general level to guide you in completing programs. It does not discuss all issues related to the ’C80 and the ’C82; use your best judgement in applying the methods discussed in Chapter 2 and Chapter 3 to your application.

Related Documentation From Texas Instruments

The following books describe the TMS320C8x and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C80 Multimedia Video Processor Data Sheet (literature number SPRS023) describes the features of the ’C80 device and provides pinouts, electrical specifications, and timings for the device.

TMS320C8x System-Level Synopsis (literature number SPRU113) describes the ’C8x features, development environment, architecture, memory organization, and communication network (the crossbar).

TMS320C80 C Source Debugger User’s Guide (literature number SPRU107) describes the ’C8x master processor and parallel processor C source debuggers. This manual provides information about the features and operation of the debuggers and the parallel debug manager; it also includes basic information about C expressions and a description of progress and error messages.

TMS320C80 Code Generation Tools User’s Guide (literature number SPRU108) describes the ’C8x code generation tools. This manual provides information about the features and operation of the linker, the master processor (MP) C compiler and assembler, and the parallel processor (PP) C compiler and assembler. It also describes the common object file format (COFF) and shows you how to link MP and PP code.

TMS320C80 Master Processor User’s Guide (literature number SPRU109) describes the ’C8x master processor (MP). This manual provides information about the MP features, architecture, operation, and assembly language instruction set; it also includes sample applications that illustrate various MP operations.

TMS320C80 Multitasking Executive User’s Guide (literature number SPRU112) describes the ’C8x multitasking executive software. This manual provides information about the multitasking executive’s software features, operation, and interprocessor communications. It also includes a list of task error codes.

TMS320C80 Parallel Processor User's Guide (literature number SPRU110) describes the 'C8x parallel processor (PP). This manual provides information about the PP features, architecture, operation, and assembly language instruction set. It also includes software applications and optimizations.

TMS320C80 Transfer Controller User's Guide (literature number SPRU105) describes the 'C80 transfer controller (TC). This manual provides information about the TC features, functional blocks, and operation. It also includes examples of block-write operations for big- and little-endian modes.

TMS320C80 Video Controller User's Guide (literature number SPRU111) describes the 'C80 video controller (VC). This manual provides information about the VC features, architecture, and operation. It also includes procedures and examples for programming the serial register transfer (SRT) controller and the frame timer registers.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Request more information about Texas Instruments Digital Signal Processing (DSP) products	Call the CRC† hotline: **(800) 336–5236** Or write to: Texas Instruments Incorporated Market Communications Manager, MS 736 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the CRC† hotline: **(800) 336–5236**
Ask questions about product operation or report suspected problems	Call the DSP hotline: **(713) 274–2320**
Report mistakes in this document or any other TI documentation	Fill out and return the reader response card at the end of this book, or send your comments to: Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251–1443

† Texas Instruments Customer Response Center

Contents

Figures

Examples

Chapter 1

Comparison of the ’C80 and ’C82

There are several differences between the ’C80 and the ’C82. As a programmer, you must be aware of these differences to effectively write code that runs on both the ’C80 and the ’C82. This chapter describes important differences to prepare you for the software-related discussions in Chapter 2 and Chapter 3.

In this chapter, you will find information that will help you:

- Understand general differences between the ’C80 and ’C82.
- Understand memory differences between the ’C80 and ’C82.
- Understand the basis for the information in Chapter 2 and Chapter 3.

Topics

1.1 Overview of ’C80 and ’C82 Features

The ’C80 offers processing power equivalent to up to 2 billion RISC operations/second. The ’C82 is a low-cost implementation of the ’C8x architecture that provides high levels of processing power for cost-sensitive applications.

The ’C80 and the ’C82 are very similar in design. They have several features in common, including a crossbar network, parallel processors, a master processor, a transfer controller, and on-chip RAM. The processors are binary compatible and can run each other’s software if the software is written for compatibility. Four main differences relate to compatibility:

- The ’C80 has four parallel processors; the ’C82 has two.
- The on-chip RAM of the ’C80 is divided into 25 2-KB blocks; the on-chip RAM of the ’C82 is divided into 11 4-KB blocks.
- The ’C80 transfer controller has 7 externally initiated packet transfer (XPT) requests; the ’C82 transfer controller has 15.
- The ’C80 has an on-chip video controller; the ’C82 has none.

Table 1–1 compares the features of the ’C80 and the ’C82.

Table 1–1. ’C80 and ’C82 Features

Feature	’C80	’C82
Number of PPs	4	2
On-chip video controllers	1	None
Total on-chip RAM	50 KB	44 KB
Local data RAM per PP	6 KB	8 KB
Parameter RAM per PP	2 KB	4 KB
PP instruction cache size	2 KB	4 KB
MP parameter RAM size	2 KB	4 KB
MP data cache size	4 KB	4 KB
MP instruction cache size	4 KB	4 KB

1.2 Parallel Processor Differences

The most important difference related to the parallel processors (PPs) is the number of PPs per chip: the ’C80 has four PPs and the ’C82 has two.

Since the ’C82 has two PPs, software written to execute on both the ’C80 and ’C82 must not require more than two PPs. Software that can use up to four PPs but that requires only two PPs will run on both devices.

The number of PPs on a chip can be counted through software. This allows you to optimize your software’s use of available PPs. See subsection 2.2.4, *Counting the Number of PPs*, for a description of how your software can determine the number of available PPs.

1.3 Transfer Controller Differences

The differences between the transfer controllers (TCs) on the ’C80 and ’C82 are hardware differences. Generally, these differences have a negligible impact on how you write your software.

The ’C82 and ’C80 TCs differ in two main ways:

- The ’C80 TC supports 7 XPT requests, but the ’C82 TC supports 15 XPT requests.
- The ’C80 TC uses dedicated pins for obtaining memory configuration information, but the ’C82 TC uses a configuration cache.

1.3.1 Number of Externally Initiated Packet Transfer Requests

The number of XPT requests handled by the ’C80 and ’C82 is different. However, because XPT requests are driven by external devices, the seven XPT requests shared by the ’C80 and ’C82 generally aren’t used to emulate each other in different systems with different peripherals. For example, the likelihood is small that the peripheral connected to XPT1 on the ’C80 software development board (SDB) is the same as the peripheral connected to XPT1 on a target ’C82 board. XPTs are normally handled through software drivers that are written for each application. The drivers for the SDB will almost always differ from the drivers used in an actual application.

1.3.2 RAM Bank Configuration Mechanisms

The special RAM bank configuration pins on the ’C80 and the configuration cache on the ’C82 are hardware features that have a negligible impact on software.

1.4 Video Controller Differences

The ’C80 has an on-chip video controller, but the ’C82 does not. If the chips are to be used in video applications, the portion of your software that sets up the video timing registers needs to be written so that an external video controller can be used with the ’C82. This is usually just a matter of switching video drivers, depending on the chip being used.

1.5 Memory Differences

The ’C80 has 50 KB of on-chip RAM, and the ’C82 has 44 KB of on-chip RAM. On-chip RAM is used as data RAM, as parameter RAM, and as instruction and data caches.

The ’C80’s RAM is partitioned into 25 individual 2-KB modules. In contrast, the ’C82’s RAM is partitioned into 11 individual 4-KB modules. Modules can be accessed in parallel over the crossbar during the same clock cycle without contention. However, since RAM organization is not identical in the ’C80 and ’C82, contention may occur in one device without occurring in the other if your program does not compensate for this difference.

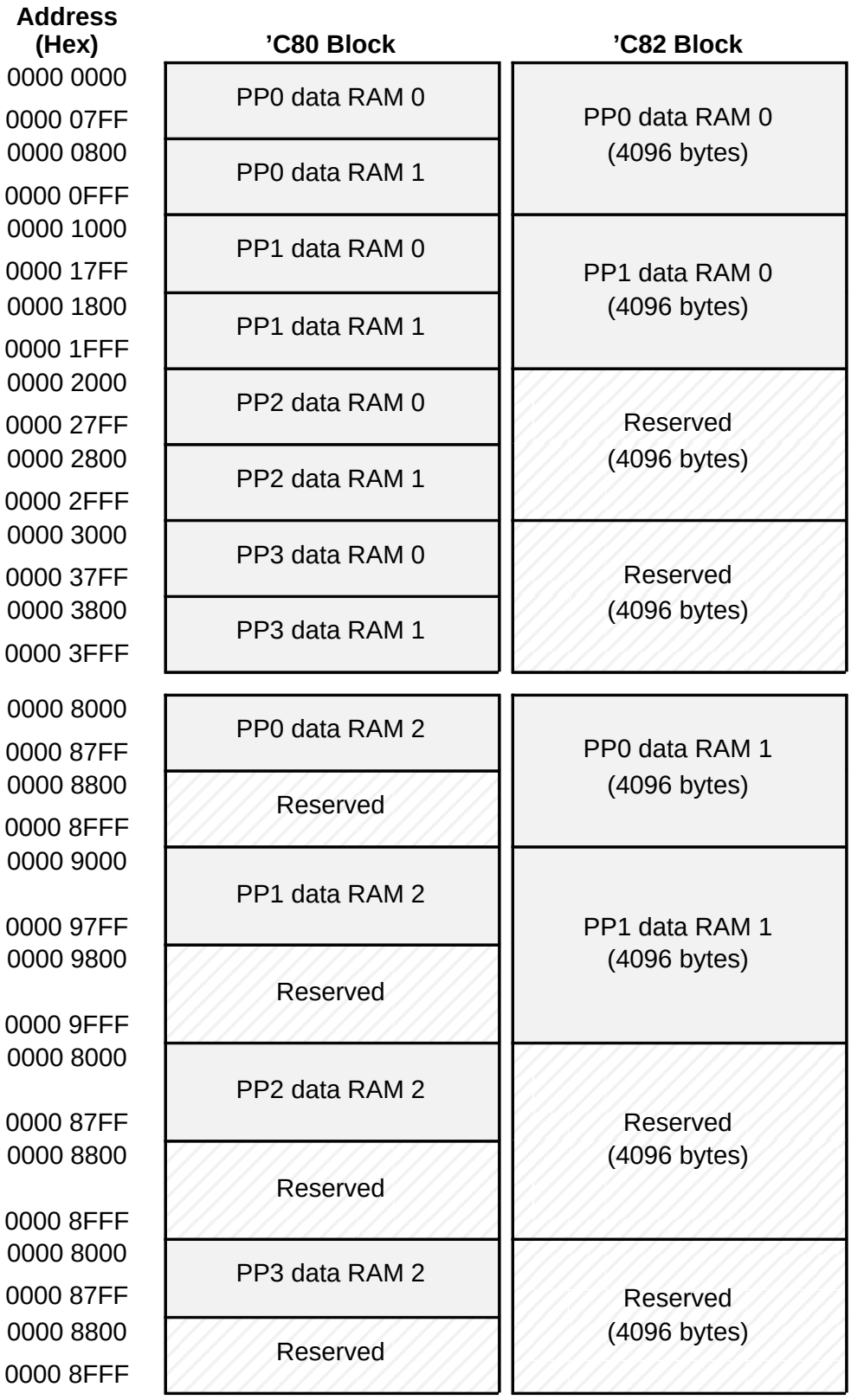
For your reference, the memory maps of the ’C80 and ’C82 are described in Appendix A.

1.5.1 Data RAMs

Data RAMs are standard read/write memory with no caching or special features. They are the main areas in which the PPs store the data they are processing. Any transfer of data to or from data RAMs is done explicitly by the processors, either by a direct read from or write to memory, or by a request to the TC to transfer the data through packet transfers.

As Figure 1–1 illustrates, all data RAM locations corresponding to PP0 and PP1 on the ’C80 are populated with data RAM on the ’C82. Thus, the ’C82’s data-RAM map for PP0 and PP1 is a superset of the ’C80’s. The last half of each of the 4-KB memory spaces occupied by the ’C82’s PP0 data RAM 1 and PP1 data RAM 1 is unpopulated on the ’C80. The memory locations occupied by the ’C80’s data RAMs for PP2 and PP3 are unpopulated on the ’C82.

Figure 1–1. ’C80 and ’C82 Data RAMs in the Memory Map



1.5.2 Parameter RAMs

A parameter RAM is associated with each processor on a ’C8x device. A portion of this RAM is dedicated to hardware parameters, such as the state of suspended packet transfers, buffers for external-to-external packet transfers, and interrupt vectors. The remainder of each parameter RAM is available to software for general-purpose data storage.

Figure 1–2 contrasts the memory maps for the on-chip parameter RAM on the ’C80 and ’C82. All ’C80 memory locations occupied by parameter RAM for PP0, PP1, and the MP are also populated on the ’C82. The memory locations occupied by the ’C80’s parameter RAMs for PP2 and PP3 are unpopulated on the ’C82. Each of the ’C82’s parameter RAMs is 4096 bytes long, which is twice the size of each parameter RAM on the ’C80. The second half of each of the corresponding memory spaces is unpopulated on the ’C80.

Note:

In the ’C80, the MP parameter RAM can be accessed only by the MP and TC. However, in the ’C82, the MP parameter RAM can be accessed by the MP, TC, and PPs.

Figure 1–2. ’C80 and ’C82 Parameter RAMs in the Memory Map

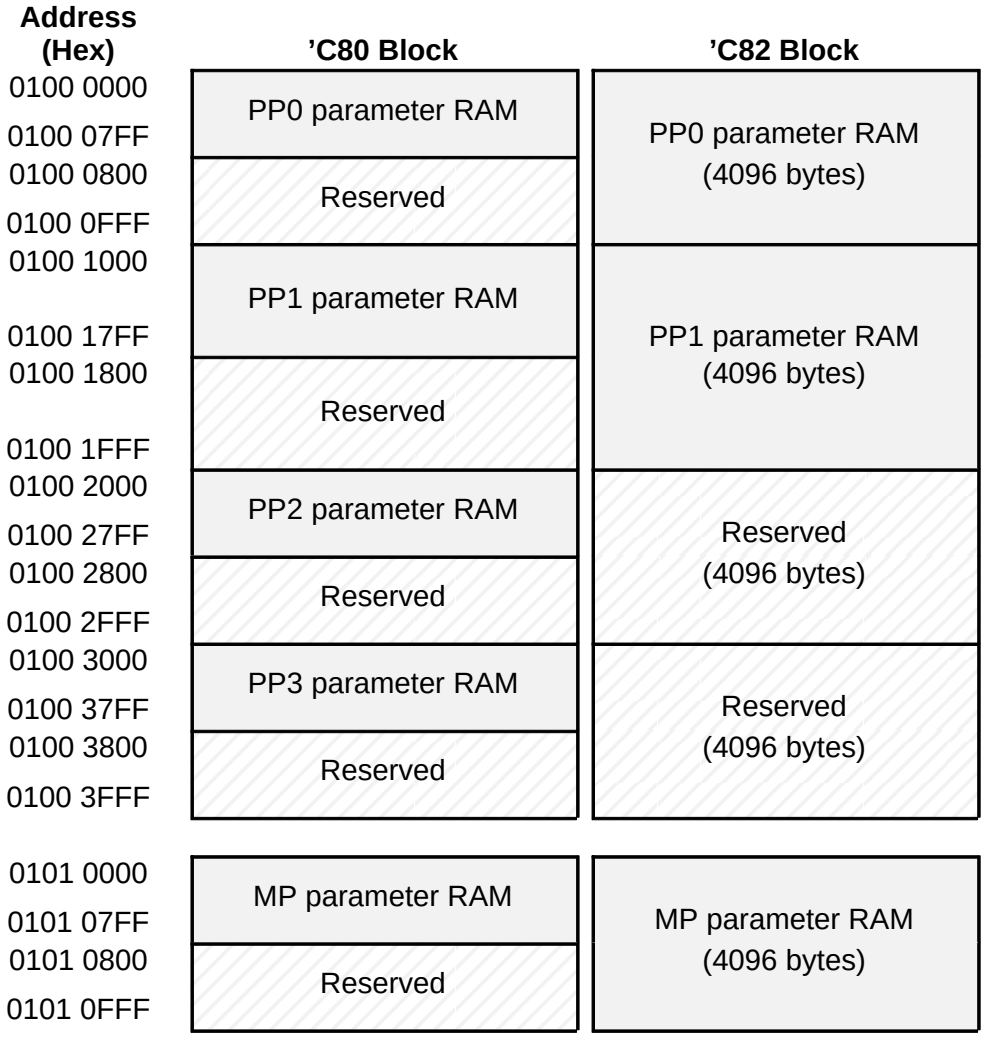


Figure 1–3 shows the MP parameter RAM in the ’C80 and ’C82. The memory maps of the hardware-dedicated areas in the first 672 bytes are nearly identical in the ’C80 and ’C82; the only difference is that the ’C80 dedicates room for seven XPT addresses and the ’C82 dedicates room for 15 XPT addresses. The general-purpose area that begins at address 0x010102A0 is 1376 bytes long in the ’C80 and 3424 bytes long in the ’C82.

Figure 1–3. ’C80 and ’C82 MP Parameter RAM Block Diagram

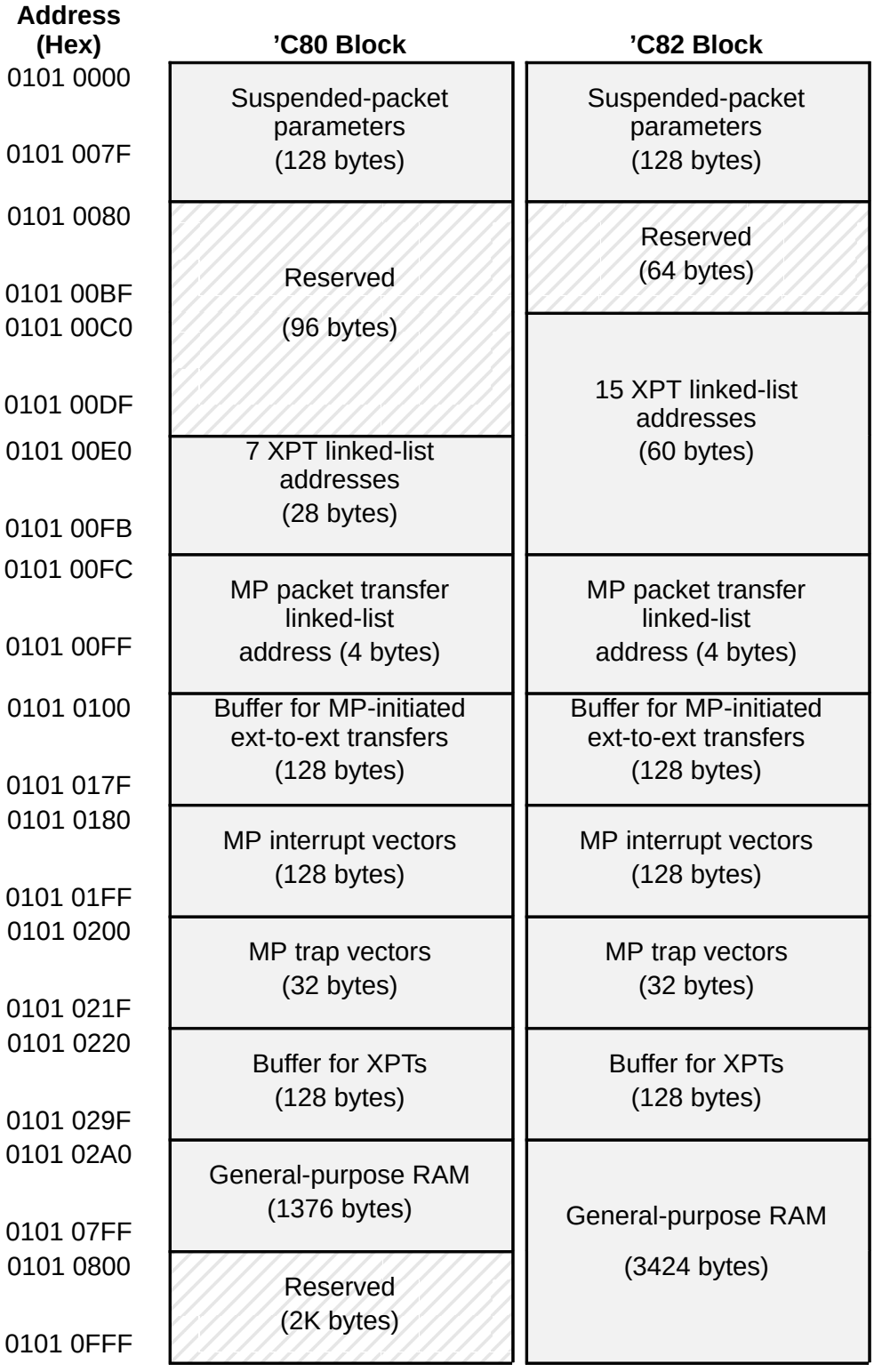
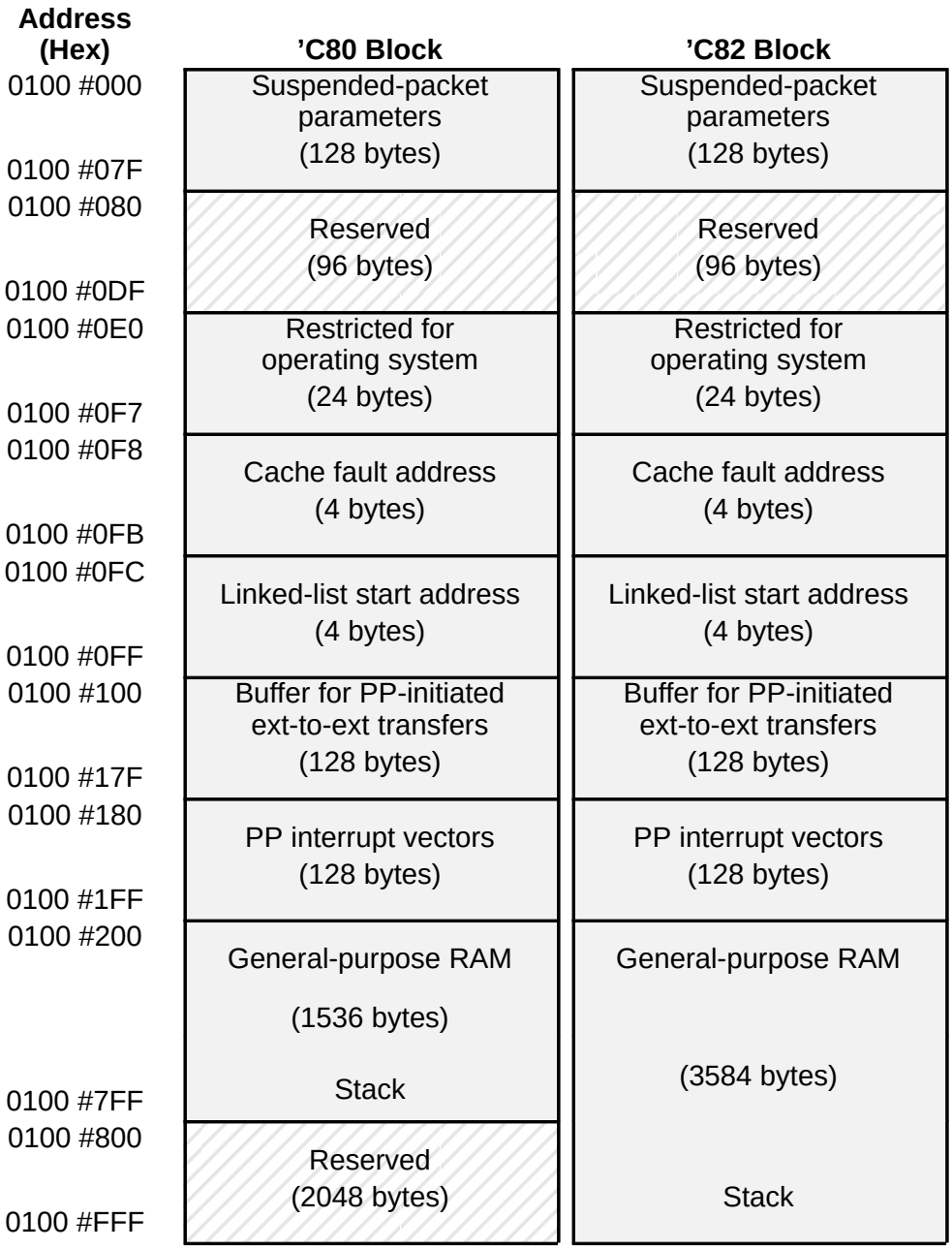


Figure 1–4 shows a typical PP parameter RAM in the ’C80 and ’C82. In the figure, the # sign in each address corresponds to the PP number. The memory maps of the first 512 bytes of the PP parameter RAM are identical in the ’C80 and ’C82. Beginning at address 01001200h is a general-purpose area that is 1536 bytes long in the ’C80 and 3584 bytes long in the ’C82. The PP’s stack occupies the higher addresses in this general-purpose area and grows toward smaller addresses.

Figure 1–4. ’C80 and ’C82 PP Parameter RAM Block Diagram



The four words at the end of the PP parameter RAM are shown in Figure 1–5 (PP1's in this case). The # sign in each address corresponds to the PP number. When a PP is reset, the hardware sets the stack pointer register (SP) to the highest address in the PP's parameter RAM and pushes the before-reset values of the SP, instruction pointer address stage register (IPA), and instruction pointer execute stage register (IPE) onto the stack. As shown in Figure 5, this leaves the SP register pointing to address 010017FCh in the 'C80 and to address 01001FFCh in the 'C82. For most applications, these addresses are likely to be the best choices for the initial SP values because they assign the largest available contiguous block of memory to the combined stack and general-purpose areas in the parameter RAM.

Figure 1–5. PP State Information on the Stack at Reset

'C80 Address (Hex)	'C82 Address (Hex)	Word
0100 #7F0	0100 #FF0	Next word in stack (stack pointer points here after reset)
0100 #7F4	0100 #FF4	IPE value from before reset
0100 #7F8	0100 #FF8	IPA value from before reset
0100 #7FC	0100 #FFC	SP value from before reset

1.5.3 Caches

Generally, cache operation is completely transparent; in most cases you need not worry about cache differences between the ’C80 and the ’C82.

Cache memory is generally not directly accessed by application software, so the locations of the caches are not important. This information is available on the memory maps in Appendix A, for your reference.

Table 1–2 lists the sizes of the caches in the ’C80 and ’C82. Better PP performance may occur in the ’C82, since each ’C82 PP instruction cache is twice as large as its corresponding ’C80 PP instruction cache.

Table 1–2. ’C80 and ’C82 Cache Sizes

Cache	’C80	’C82
PP instruction-cache size (each PP)	2 KB	4 KB
MP data cache-size	4 KB	4 KB
MP instruction-cache size	4 KB	4 KB

Chapter 2

Programming for Compatibility

By following simple compatibility guidelines, you can write code that will run equally well on both the ’C80 and the ’C82. Object code written with these guidelines in mind should run on both devices without recompiling or relinking.

In this chapter, you will find information that will help you:

- ❑ Select the best compatibility technique for your application.
- ❑ Write programs that will run on both the ’C80 and the ’C82.
- ❑ Write code that can test which device it is running on (’C80 or ’C82).
- ❑ Write code that correctly initializes a PP’s stack pointer based upon which device it is running on.

Topics

2.1 Programming Considerations

Despite the differences shown in Chapter 1, the two ’C8x devices are fundamentally quite similar. The similarities allow software developed on one device to be used on the other.

For instance, a ’C80-based board and emulator can serve as a convenient prototyping system for developing code targeted to run on a ’C82. In fact, developing software that will run on both the ’C80 and ’C82 without recompiling or relinking is not difficult.

Writing code that runs on both the ’C80 and ’C82 is straightforward because of the similarities in the two devices’ memory maps.

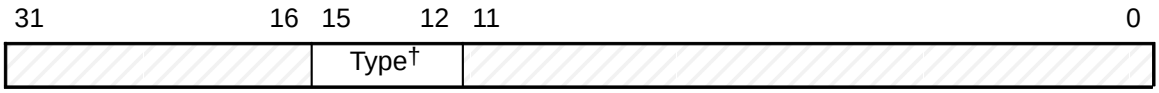
2.2 Obtaining Device-Specific Information

Device-specific information can be obtained by your software. This information can help you effectively use all available PPs and properly initialize each PP's stack pointer.

2.2.1 Determining the Device that is Executing your Software (MP)

An MP-resident program can determine which device it is running on by reading the MP's CONFIG register, which is shown in Figure 2–1. The 4-bit type field in bits 15–12 of the CONFIG register is hardwired to the value 0000_2 in the 'C80, and to the value 0010_2 in the 'C82.

Figure 2–1. The Type Field in the MP CONFIG Register

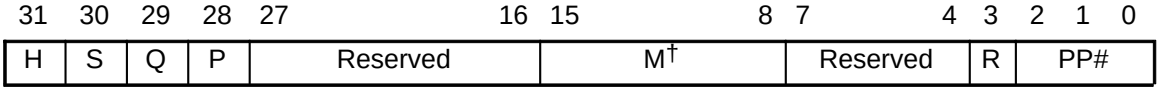


†Type = 0000_2 in the 'C80; Type = 0010_2 in the 'C82

2.2.2 Determining the Device that is Executing your Software (PP)

PP software can determine which device it is running on by reading the M field (bits 15–8) in the comm register, which is shown in Figure 2–2. The M field equals 00000000_2 in the 'C80 and 00000001_2 in the 'C82.

Figure 2–2. The PP's comm Register



†M=00000000_2 in the 'C80; M=00000001_2 in the 'C82

Notes:

1) In order to provide backward compatibility for preproduction 'C80 devices (silicon revisions 1, 2, and 3), your software should write a value of 0 to the comm register to initialize the M field before attempting to read it. In preproduction 'C80 devices, bit 8 of the comm register is a read/write bit whose value is undefined immediately following reset.

2) The M field will be used to identify future versions of the 'C8x, so make sure your software checks all bits in the M field to properly identify the device.

2.2.3 Initializing a PP's Stack Pointer

The PP's initialization software may need to set the SP to its initial value at some time other than immediately after reset. This initial value differs between the 'C80 and 'C82.

One way to initialize the SP is to load it with a PP-relative address constant whose value is specified at link time. If the stack is to begin in different locations in the 'C80 and 'C82, however, this approach requires that separately linked object modules be provided to run on the 'C80 and 'C82.

An approach that does not have this drawback is to link together a single object code module that determines at run time whether it is running on a 'C80 or 'C82 and then sets up the stack accordingly. Typically, the software would set the SP to the same value it would have immediately following reset. This value is 0100#7F0h on the 'C80 and to 0100#FF0h on the 'C82 (the # sign represents the PP number).

Example 2–1 shows a four-instruction code segment, written in PP assembly language, that sets a PP's stack pointer and comm registers to their initial values. The program reads the M bit in the comm register to determine whether it is running on a 'C80 or a 'C82.

Example 2–1. Initializing a PP's Stack Pointer and Comm registers

```
; Sample code for initializing a PP's stack pointer (SP)
; and comm registers to their initial values
;
; For the 'C80, the code initializes the SP to 0100#7f0h.
; For the 'C82, the code initializes the SP to 0100#ff0h.
;
; The # symbol in the previous addresses represents the
; hexadecimal digit that is set to the PP number by the
; first instruction; the first instruction uses the
; keyword pba to specify a PP-relative parameter-RAM
; address offset.
;
;
a7 = &*(sp = pba + 0x7F0)    ; initialize SP for 'C80
comm = 0                     ; fix for 'C80 revs 1–3
a7 = comm & (1 \\ 8)         ; is this a 'C80 or a 'C82?
sp = [nz] sp | (1 \\ 11)     ; if 'C82, modify initial SP
```

2.2.4 Counting the Number of PPs

Counting the number of PPs is essentially a process of checking which device is executing a program and then using that information to determine the number of PPs that are available on the processor. For example, if the device check returns ’C80 as the device type, then the the software should return four as the number of PPs.

Example 2–2 shows an MP C function that checks the MP’s CONFIG register to determine which PPs are present on the ’C8x device. The function returns an 8-bit mask in which each bit indicates whether or not PP0–PP7 are present: bit 0 corresponds to PP0, bit 1 corresponds to PP1, etc. When executed on a ’C80 device, the function returns the value 0Fh to indicate that PP0, PP1, PP2, and PP3 are present. When executed on a ’C82 device, the function returns the value 03h to indicate that PP0 and PP1 are present.

Example 2–2. MP C Function for Counting the Number of PPs

```
#include <mvp.h>
/*
 * Return a mask that specifies the PPs that are
 * present on the ’C8x device. Bits 0 to 7 of the
 * mask represent PPs 0 to 7. If a PP is present,
 * the corresponding mask bit is set to 1.
 *
 * The software checks the device it is running on by
 * checking the type field in the config register.
 */
int PpGetMask(void)
{
  return((0xff03010f >> ((config & 0x3000) >> 9)) & 0xff);
}
```

2.3 Technique #1: Using Only Common RAM

A particularly simple technique for achieving compatibility is to use only the portions of the on-chip RAM that are common to the ’C80 and ’C82.

The ’C80’s memory map for the local RAMs belonging to the MP, PP0, and PP1 is a subset of the ’C82’s memory map. All of the valid local RAM addresses for the ’C80’s MP, PP0, and PP1 correspond to populated local RAM addresses in the ’C82. Therefore, ’C80 code written to use on-chip RAM belonging to the MP, PP0, and PP1 should run on the ’C82 without modification.

2.3.1 Advantages of Technique #1

This technique offers three main advantages:

- It is simple to implement.
- Performance is nearly identical on the ’C80 and ’C82.
- All of the RAM available to the ’C80’s MP, PP0, and PP1 can be used.

2.3.2 Disadvantages of Technique #1

This technique has two main disadvantages:

- All of the RAM available to the ’C82 cannot be used.
- Contention may result in the ’C82 since the space occupied by two RAMs (data RAM 0 and data RAM 1) in the ’C80 is occupied by one RAM (data RAM 0) in the ’C82.

2.3.3 Implementing Technique #1

In order to implement this technique, place three restrictions upon your program:

- Use only RAM that is common to both the ’C80 and ’C82.
- Do not require your program to use RAM belonging to PP2 and PP3 on the ’C80, since the ’C82 doesn’t have these PPs.
- Do not use absolute addresses to access the PP stacks since the PP stacks have different locations in the ’C80 and ’C82.

2.3.4 Considerations When Using Technique #1

Although PP code that uses this technique should produce identical results on the ’C80 and ’C82, the timing and performance of PP-resident programs may be affected by two main differences between the devices:

- Each ’C82 PP instruction cache is larger than each ’C80 PP instruction cache.
- For a given PP, two independently-accessible RAMs (RAM 0 and RAM 1) in the ’C80 occupy the same space as a single RAM (RAM 0) in the ’C82.

First, since the ’C82 PP instruction cache is twice as large as that of the ’C80, PP performance on the ’C82 should be better than PP performance on the ’C80.

Second, the C80 PP’s data RAMs 0 and 1 are independently accessible RAM modules that can be simultaneously accessed over the crossbar without contention. In the ’C82, however, the area of memory corresponding to the ’C80’s data RAMs 0 and 1 is occupied by a single 4-KB RAM module; only a single access of this module can be performed during any one clock cycle. ’C80 applications may run slightly slower on the ’C82 because this area of memory can can be accessed only once per clock cycle. This can occur when software attempts to access data RAMs 0 and 1 simultaneously.

The delays caused by this type of contention are usually negligible unless a critical inner loop of a PP program written for the ’C80 contains instructions that attempt to access data RAMs 0 and 1 in parallel through the PP’s global and local memory ports. Note that this is a performance issue only; code that runs correctly on the ’C80 should produce the same results on the ’C82 despite any increase in processing time due to contention.

Although the ’C80 contains more total on-chip RAM than the ’C82 (50K bytes versus 44K bytes), the ’C82 dedicates a larger amount of local RAM to each of its processors. On the ’C82, the MP has an additional 2048 bytes of parameter RAM; each PP has an additional 2048 bytes of data RAM and an additional 2048 bytes of parameter RAM. The ’C80 addresses corresponding to the ’C82’s additional RAM are unpopulated. Similarly, the ’C82 addresses corresponding to the ’C80’s local RAM for PPs 2 and 3 are unpopulated.

2.4 Technique #2: Using Pointers to Allocate RAM

Technique #1 can be augmented to take advantage of some of the additional blocks of on-chip RAM that lie outside the populated regions common to the ’C80 and ’C82.

Technique #2 uses pointers to access on-chip RAM rather than fixed address constants. With this technique, only the portion of the software that actually allocates the RAM needs to recognize the differences between the memory maps of the ’C80 and ’C82.

2.4.1 Advantages of Technique #2

This technique offers two main advantages:

- Only the portion of the software that actually allocates the RAM needs to recognize the differences between the memory maps of the ’C80 and ’C82.
- More available memory (compared to technique #1) can be used on both devices.

2.4.2 Disadvantages of Technique #2

This technique has a two main disadvantages:

- Technique #2 is more complicated than technique #1.
- One RAM cannot be used since it is needed to store arguments.

2.4.3 Implementing Technique #2

Implement this technique by writing software that performs three tasks:

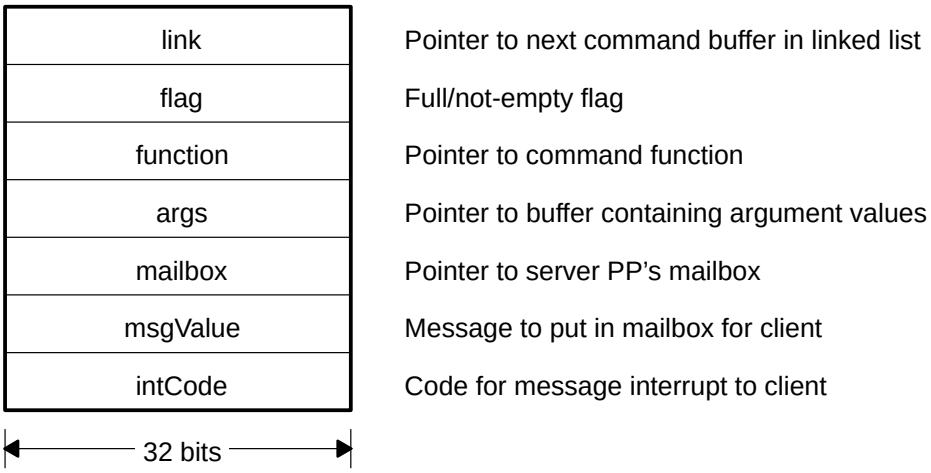
- Access RAM through pointers rather than through fixed address constants.
- Use one block (or part of one block) of RAM as an argument buffer.
- Pass pointers to memory through the argument buffer and use a command buffer to carry the location of the argument buffer from the MP to each PP.

The software running on the MP should determine at system initialization time whether the device it is executing on is a ’C80 or ’C82. From this information, the MP should determine the location of the additional RAM and then pass pointers to this RAM to

each PP. Each PP can then use the RAM without being aware of whether the RAM was allocated from a ’C80 or a ’C82.

The MP should pass a pointer to this RAM through a command buffer, such as the one shown in Figure 2–3 (taken from Section 5.2.6, *Command Buffer*, in the *TMS320C80 Multitasking Executive User’s Guide*). The args field in the command buffer contains a pointer to a buffer in shared memory that contains the argument values for the command. The MP can use one of the additional blocks of RAM as an argument buffer and pass the pointer to the PP through the args field of the command buffer.

Figure 2–3. Structure of a PP Command Buffer



2.4.4 Considerations When Using Technique #2

In the case of the ’C82, the additional RAM is allocated from the local RAM of PP0 and PP1. The PPs can access this RAM over the local-bus connections of crossbar. For the ’C80, however, the additional RAM may not be local to the PP that accesses the RAM. If a PP attempts to access a nonlocal RAM through its local port, however, the only impact is that the access is delayed for one clock cycle. This delay may slightly alter the execution time of the software, but is otherwise transparent to software.

Chapter 3

Prototyping ’C82 Code on the ’C80

The ’C80 can be used to prototype code that is targeted to run on the ’C82 alone.

In this chapter, you will find information that will help you:

- Prototype ’C82 applications with a ’C80.
- Develop ’C82 code on a ’C80 development board.

Topics

3.1 Overview

The ’C80 can be used to prototype code that is targeted to run on the ’C82 alone. To do this, assume that after prototyping is completed, the resulting code is not intended to be used on the ’C80.

The goal in this instance is to make full use of the ’C82’s on-chip RAM, as opposed to using only the subset of the on-chip RAM that is common to the ’C80 and ’C82.

Additionally, since the software will be designed to run primarily on the ’C82, device-specific information can be hard-coded right into the software. This can be accomplished by using linker command files that specify the memory configuration of each device. For software on the prototype board, you would use the linker command file for the ’C80. In the actual target board, you would link your software using the linker command file for the ’C82.

In this way, a ’C80 software development board (SDB) can be used to develop software for the ’C82.

3.2 Emulating the ’C82’s Data RAMs

The PP data RAMs are relatively simple to emulate since the ’C80 has four 4-KB blocks of contiguous memory in the form of data RAMs 0 and 1 for each of its four PPs. The level of emulation provided by this approach should be adequate for many applications.

The technique involves the emulation of four RAMs:

To emulate ’C82:	Use the ’C80:
PP0 data RAM 0	PP0 data RAMs 0 and 1
PP1 data RAM 0	PP1 data RAMs 0 and 1
PP0 data RAM 1	PP2 data RAMs 0 and 1
PP1 data RAM 1	PP3 data RAMs 0 and 1

3.2.1 Emulating the ’C82’s PP0 Data RAM 0 and PP1 Data RAM 0

In the ’C82, each PP data RAM module is 4096 bytes long. The ’C82’s PP0 data RAM 0 occupies the same 4096 bytes of contiguous address space as the ’C80’s PP0 data RAMs 0 and 1.

Thus, the ’C80’s PP0 data RAMs 0 and 1 can directly emulate the ’C82’s PP0 data RAM 0. This approach also works in emulating the ’C82’s PP1 data RAM 0.

3.2.2 Emulating the ’C82’s PP0 Data RAM 1 and PP1 Data RAM 1

Emulating the ’C82’s PP0 data RAM 1 and PP1 data RAM 1 is not quite as straightforward. Only the first half of the 4-KB block of memory occupied by each PP’s data RAM 1 is populated on the ’C80 (by the ’C80’s data RAM 2).

The simplest strategy for emulating the ’C82’s PP0 data RAM 1 is to use the 4096 bytes of contiguous RAM from the ’C80’s PP2 data RAMs 0 and 1. Since the address of this RAM differs from that of the ’C82’s PP0 data RAM 1, accesses to this RAM need to be directed through the use of pointers. Only the portion of the software that initializes the pointer needs to be aware of the difference in addresses between the ’C80 and ’C82. Similarly, the ’C82’s PP1 data RAM 1 can be emulated by the ’C80’s PP3 data RAMs 0 and 1.

Note:

In order for this approach to work, you must keep the ’C80’s PP2 and PP3 in a halted state.

3.3 Emulating the ’C82’s Parameter RAMs

The MP and PP parameter RAMs are a little more difficult to emulate for two reasons. First, only the first half of the 4-KB block of memory occupied by each of the ’C82’s parameter RAMs is populated by parameter RAM on the ’C80. Second, there are no remaining contiguous 4-KB blocks in the ’C80 (assuming that they were used in emulating the data RAM).

The technique involves the emulation of three ’C82 RAMs by using six ’C80 RAMs:

To emulate ’C82:	Use the ’C80:
PP0 parameter RAM	PP0 parameter RAM and PP0 data RAM 2
PP1 parameter RAM	PP1 parameter RAM and PP1 data RAM 2
MP parameter RAM	MP parameter RAM and PP2 parameter RAM

Take three additional actions to ensure successful emulation:

- Reinitialize each PP stack so that it starts at the end of the emulated parameter RAM (the end of the second block).
- Make sure that each PP stack does not exceed the boundaries of its RAM.
- Keep PP2 and PP3 halted in the ’C80.

3.3.1 Emulating the PP Parameter RAMs

The PP parameter RAMs in the ’C82 are twice the size of the PP parameter RAMs in the ’C80. The second half of each parameter RAM is not populated in the ’C80; however, PP0 data RAM 2 and PP1 data RAM 2 are available for your use. In many applications, this RAM may be used in conjunction with the ’C80’s PP0 and PP1 parameter RAMs to emulate the ’C82’s PP parameter RAMs.

To do this, use the ’C80’s PP0 data RAM 2 to emulate the last half of the ’C82’s PP0 parameter RAM, and use the ’C80’s PP1 data RAM 2 to emulate the last half of the ’C82’s PP1 parameter RAM. Access these RAMs through pointers to mask the memory address differences between the ’C80 and ’C82.

This approach is viable as long as your application can tolerate the fact that the RAM modules used for the emulation are not contiguous.

3.3.2 Emulating the MP Parameter RAM

The remaining issue is how to emulate the ’C82’s 4096-byte MP parameter RAM. The last half of the memory address block occupied by this RAM is unpopulated on the ’C80. The ’C80’s PP2 parameter RAM can be mapped into this space and accessed via a pointer. This approach works only if the application allows two noncontiguous blocks of ’C80 RAM to be used to emulate a single block of MP parameter RAM.

This technique uses all 2048 bytes of the ’C80’s PP2 parameter RAM as general-purpose RAM. As long as PP2 remains halted, the hardware functions that can write to the parameter RAM remain inactive and do not alter the contents of the RAM.

3.4 Prototyping Code Using Linker Command Files

Linker command files allow you to put linking information in a file; these files are useful because they allow you to use MEMORY and SECTIONS directives to customize memory allocation for your application. In this case, you can specify the location of the parameter and data RAMs for the processor you are using to execute your software. By using linker command files to develop code for the ’C82 on the ’C80, you do not need to modify your assembly or C code.

This method of emulating the ’C82 on a ’C80 requires that you use different linker command files for the ’C80 and ’C82. Thus, although the code is not binary-compatible, it is made compatible through linking.

The basic procedure for using this method requires three steps:

1) Write your code, taking into account the considerations discussed in this chapter.
2) Link your code with a linker command file for the ’C80 and then test your code on the ’C80.
3) Link your code with a linker command file for the ’C82 when you are ready to use that device.

The software tools for the ’C80 and ’C82 come with template linker command files that you will need to modify according to the requirements of your program.

For a list of linker directives, see Section 13.4, *Linker Command Files*, in the *TMS320C80 (MVP) Code Generation Tools User’s Guide*.

3.4.1 PP-Relative Addressing

One linker command file can be used for linking code so that it can be executed by all PPs in each device. This is possible because PP addresses can be specified using offsets.

In order to make this approach work on the PPs, your code must use PP-relative addressing to access memory.

PP-relative addressing allows the base address for a PP’s local data RAM 0 or local parameter RAM to be used for address generation. PP-relative addressing is specified with the keyword dba, pba, or xba, where dba is the base address beginning at data RAM 0, pba is the base address beginning at the parameter RAM, and xba is determined to be either dba or pba by the linker. A typical line of PP-relative code has the following format:

```
d6 = *(xba+BUF)   ; read a word from memory at BUF
```

For more information on PP-relative addressing, see subsection 8.8.1.5, *PP-Relative Addressing*, in the *TMS320C80 (MVP) Parallel Processor User's Guide*.

3.4.2 Map Files

Map files show where sections are linked in memory. Specifically, they show the names and address ranges of sections and global variables.

You can cause the linker to generate a map file of your code by specifying the –m option on the command line or in the linker command file.

By examining the map file generated by the linker, you can verify that the sections of your program are mapped into the proper locations in memory.

For more information about map files, see subsection 13.3.8, *Create a map file (–m filename option)*, in the *TMS320C80 (MVP) Code Generation Tools User's Guide*.

3.4.3 Linking Your 'C82 Code for Prototyping on a 'C80

Example 3–1 is a sample PP C program that will be used to illustrate how code written for the 'C82 can be made to run on the 'C80 through linking. The program calculates the dot product of two vectors, A and B. The source code needs to compiled only once; the object file then only needs to be linked with the proper linker command file for each device.

Example 3–1. Sample PP C Program

```
/*******************************************************
 * example.c   This example C code calculates the dot
 *             product of vectors A and B.
 *******************************************************/
#include <mvp.h>
   /* allocate space for A in a named section .a_sect */
#pragma DATA_SECTION(A,".a_sect")
   /* allocate space for B in a named section .b_sect */
#pragma DATA_SECTION(B,".b_sect")
   /* define the buffer size for A and B */
#define BUF_SIZE 1536
short A[BUF_SIZE];
short B[BUF_SIZE];

main()
{
     long dot_prod;
     dot_prod = dot_product(A,B,BUF_SIZE);
}

long dot_product(short *A, short *B, int vect_size)
{
     int i;
     long dot_prod;
     dot_prod = 0;           /* set initial value of the
                                accumulator to zero     */
     for (i=0; i<vect_size; i++)
     {
     dot_prod = dot_prod + A[i] * B[i]; /* calculate the
                                           dot product */
     }
     return(dot_prod);
}
```

3.4.3.1 Allocating Memory In Sections

In the code, the vectors are allocated by using the #pragma directive. This directive allocates space in the sections that are defined in the linker command file. A sample #pragma directive is as follows:

```
#pragma DATA_SECTION(A,".a_sect")
```

This sample line allocates space for A in a section named .a_sect; the memory block used for .a_sect is defined in the linker command file.

In assembly language, #pragma is equivalent to the .sect and .usect assembler directives. To allocate the vector A in assembly language, the proper directive would be as follows:

```
A    .usect ".a_sect",1536
```

In this way, you can use the linker to specify where to place variables in memory.

Note:

For more information about the #pragma directive, see Section 2.8, *Pragma Directives* in the *TMS320C80 (MVP) Code Generation Tools User's Guide*. For more information about the .sect and .usect directives, see Section 9.10, *Directives Reference*, in the *TMS320C80 (MVP) Code Generation Tools User's Guide*.

3.4.3.2 The 'C80 Linker Command File

Example 3–2 shows a linker command file for prototyping 'C82 code on a 'C80. This linker command file causes the code to use RAM from other PPs to emulate the RAM in the 'C82.

The method used to create this file is described in Section 3.2, *Emulating the 'C82's Data RAMs* and Section 3.3, *Emulating the 'C82's Parameter RAMs*.

In the SECTIONS area of the linker command file, .a_sect is specified to be placed in DRAM0 and .b_sect is specified to be placed in DRAM1. The directives that accomplish this are as follows:

```
SECTIONS
{
    .a_sect  :  (PASS) > DRAM0
    .b_sect  :  (PASS) > DRAM1
}
```

Note:

For more information about the PASS keyword, see *Section 13.11* in the *TMS320C80 (MVP) Code Generation Tools User's Guide*.

The file in Example 3–2 could be used to create a 'C80 executable file for the program shown in Example 3–1.

Example 3–2.’C80 PP Linker Command File for example.c

```
/*******************************************************
 * c80pp.lnk – PP linker command file for prototyping
 *             ’C82 code on the ’C80.
 *******************************************************/
–pc
–x
–pstack 0x580
–l pp_rts.lib
MEMORY
{
    DRAM0     : o=0x00000004   l = 0x0ffc
    DRAM1     : o=0x00002000   l = 0x1000
    PRAM0     : o=0x01000200   l = 0x00600
    PRAM1     : o=0x00008000   l = 0x00800
    EXTMEM    : o=0x02000000   l = 0x80000
}
SECTIONS
{
    .ptext   :  > EXTMEM
    .pcinit  :  > EXTMEM
    .pbss    :  (PASS) > DRAM0
    .pstack  :  (PASS) > PRAM0
    .a_sect  :  (PASS) > DRAM0
    .b_sect  :  (PASS) > DRAM1
}
```

3.4.3.3 Compiling and Linking example.c for the ’C80

To compile and link example.c for the ’C80, perform two steps:

1) Compile the C source code using the PP compiler.
2) Link the output of the compiler with the linker command file shown in Example 3–2.

Both of these steps can be accomplished using a single command:

```
ppcl –g example.c –z –o c80ex.out –m c80ex.map c80pp.lnk
```

3.4.3.4 The ’C80 Map File

The map file shows where each variable was linked into memory. In this case, the linker command file specifically assigns A to DRAM0 and B to DRAM1. Example 3–3 shows the lines from the map file that show where A and B were placed in memory. The entire file is not shown in that example; dots (..) are used to show sections that were deleted.

In this example, .a_sect was placed at address 00000140h in DRAM0 and .b_sect was placed at address 00002000h in DRAM1.

To see a complete map file, see Section 14.5, *The Example Linker Map Files*, in the *TMS320C80 (MVP) Code Generation Tools User’s Guide*.

Example 3–3.’C80 Map File Lines for Vector A and Vector B

```
************************************************
MVP COFF Linker             Version 1.10
************************************************
Tue Oct  3 10:00:06 1995

OUTPUT FILE NAME:   <c80ex.out>
ENTRY POINT SYMBOL: ″$c_int00″  address: 020006b8

MEMORY CONFIGURATION
           name      origin     length     attributes     fill
         --------   --------   ---------   ----------   --------
         DRAM0      00000004   000000ffc      RWIX
         DRAM1      00002000   000001000      RWIX
         PRAM1      00008000   000000800      RWIX
         PRAM0      01000200   000000600      RWIX
         EXTMEM     02000000   000080000      RWIX

SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
   ..      ..       ..          ..              ..
   ..      ..       ..          ..              ..
.a_sect    0    00000140    00000c00     PASS SECTION
                 00000140    00000c00      example.o (.a_sect)
.b_sect    0    00002000    00000c00     PASS SECTION
                 00002000    00000c00      example.o (.b_sect)
   ..      ..       ..          ..              ..
   ..      ..       ..          ..              ..
GLOBAL SYMBOLS

address  name
-------- ----
00000140 $A
00002000 $B
   ..      ..       ..          ..              ..
   ..      ..       ..          ..              ..
```

3.4.4 Linking your Code for a ’C82

After prototyping your code on the ’C80, you can make it execute on the ’C82 by simply linking the object file using a ’C82 linker command file. In this case, you can use the linker command file in this section to link the object file generated by compiling Example 3–1 to create an executable file for the ’C82.

The linker command files for the ’C80 and the ’C82 differ mainly in the MEMORY section; the locations of the data RAMs, parameter RAMs, and external memory are defined in that section.

The entries in the SECTIONS area need to use RAMs that are defined in the MEMORY section; thus, the entries in the SECTIONS area may differ in the two files. This is the case with the parameter RAM, since the ’C80 uses two noncontiguous RAMs to emulate each single ’C82 PP parameter RAM. Thus, in the ’C80 linker command file, the parameter RAM is defined as

PRAM0 and PRAM1 instead of just PRAM, as is the case in the ’C82 linker command file.

3.4.4.1 The ’C82 Linker Command File

The linker command file for the ’C82 specifies the location of DRAM0, DRAM1, and PRAM, and external memory (EXTMEM).

Example 3–4.’C82 PP Linker Command File

```
/*******************************************************
 * c82pp.lnk – ’C82 PP linker command file.
 *******************************************************/
–pc
–x
–pstack 0x580
–l pp_rts.lib
MEMORY
{
    DRAM0     : o=0x00000004   l = 0x0ffc
    DRAM1     : o=0x00008000   l = 0x1000
    PRAM      : o=0x01000200   l = 0x00e00
    EXTMEM    : o=0x02000000   l = 0x80000
}
SECTIONS
{
    .ptext   :  > EXTMEM
    .pcinit  :  > EXTMEM
    .pstack  :  (PASS) > PRAM
    .pbss    :  (PASS) > DRAM0
    .a_sect  :  (PASS) > DRAM0
    .b_sect  :  (PASS) > DRAM1
}
```

3.4.4.2 Compiling and Linking example.c for the ’C82

To produce an executable for the ’C82, link the object file generated when you last compiled example.c with the linker command file shown in Example 3–4.

This step can be accomplished using a single command:

```
ppcl –z –o c82ex.out –m c82ex.map example.o c82pp.lnk
```

If you do not have an object file to link, you can compile and link with a single command:

```
ppcl –g example.c –z –o c82ex.out –m c82ex.map c82pp.lnk
```

3.4.4.3 The ’C82 Map File

The ’C82 linker command file specifically assigns A to DRAM0 and B to DRAM1, as was the case for the ’C80 linker command file. Example 3–5 shows the lines from the map file that show where A and B were placed in memory. The entire file is not shown in that example; dots (..) are used to show sections that were deleted.

Example 3–5.’C82 Map File Lines for Vector A and Vector B

```
*****************************************************
MVP COFF Linker              Version 1.10
*****************************************************
Tue Oct  3 10:04:01 1995

OUTPUT FILE NAME:   <c82ex.out>
ENTRY POINT SYMBOL: ”$c_int00”  address: 020006b8

MEMORY CONFIGURATION
           name      origin     length     attributes     fill
         --------   --------   ---------   ----------   --------
         DRAM0      00000004   000000ffc      RWIX
         DRAM1      00008000   000001000      RWIX
         PRAM       01000200   000000e00      RWIX
         EXTMEM     02000000   000080000      RWIX

SECTION ALLOCATION MAP

 output                                  attributes/
section   page    origin      length       input sections
--------  ----  ----------  ----------   ----------------
   ..      ..       ..          ..              ..
   ..      ..       ..          ..              ..
.a_sect    0    00000140    00000c00     PASS SECTION
                  00000140    00000c00       example.o (.a_sect)
.b_sect    0    00008000    00000c00     PASS SECTION
                  00008000    00000c00       example.o (.b_sect)
   ..      ..       ..          ..              ..
   ..      ..       ..          ..              ..
GLOBAL SYMBOLS

address  name
-------- ----
00000140 $A
00008000 $B
   ..      ..       ..          ..              ..
   ..      ..       ..          ..              ..
```

In this example, .a_sect was placed at address 00000140h in DRAM0 and .b_sect was placed at address 00008000h in DRAM1.

To see a complete map file, see Section 14.5, *The Example Linker Map Files*, in the *TMS320C80 (MVP) Code Generation Tools User’s Guide*.

3.4.5 MP Linker Command Files

MP linker command files function in the same way as PP linker command files. Example 3–6 shows the command file for prototyping ’C82 code on a ’C80. Example 3–7 show the command file for linking ’C82 code to execute on a ’C82.

Example 3–6.’C80 MP Linker Command File

```
/*******************************************************
 * c80mp.lnk – MP linker command file for prototyping
 *             ’C82 code on a ’C80
 *******************************************************/
–c
–x
–heap  0x2000
–stack 0x2000
–l mp_rts.lib
MEMORY
{
    EXTMEM    : o=0x02000000   l = 0x80000
    MPPRAM0   : o=0x01010000   l = 0x800
    MPPRAM1   : o=0x01002000   l = 0x800
}
SECTIONS
{
    .text    :  > EXTMEM
    .ptext   :  > EXTMEM
    .bss     :  > EXTMEM
    .const   :  > EXTMEM
    .switch  :  > EXTMEM
    .sysmem  :  > EXTMEM
    .stack   :  > EXTMEM
    .cinit   :  > EXTMEM
    .pcinit  :  > EXTMEM
}
```

Example 3–7.’C82 MP Linker Command File

```
/*******************************************************
 * c82mp.lnk – ’C82 MP linker command file
 *******************************************************/
–c
–x
–heap  0x2000
–stack 0x2000
–l mp_rts.lib
MEMORY
{
    EXTMEM    : o=0x02000000   l = 0x80000
    MPPRAM    : o=0x01010000   l = 0x1000
}
SECTIONS
{
    .text    :  > EXTMEM
    .ptext   :  > EXTMEM
    .bss     :  > EXTMEM
    .const   :  > EXTMEM
    .switch  :  > EXTMEM
    .sysmem  :  > EXTMEM
    .stack   :  > EXTMEM
    .cinit   :  > EXTMEM
    .pcinit  :  > EXTMEM
}
```

3.4.6 Considerations when Using Linker Command Files

There are two basic considerations when prototyping ’C82 code on a ’C80 using linker command files:

- In some cases, code may run more quickly on the ’C80 than on the ’C82. The probability of contention is lower in the ’C80 because each PP DRAM0, PP DRAM1, PP PRAM, and MP PRAM is emulated by two RAMs in the ’C80.
- In some cases, code may run more slowly on the ’C80 than on the ’C82. Code that uses the local port to access DRAM1 could cause a 1-cycle stall during each access, since DRAM1 is not local to the ’C80 PPs.

3.4.6.1 Contention Differences in the ’C80 and ’C82

In the ’C82, contention exists between accesses to the first half and last half of a RAM. In the ’C80, however, this contention does not exist. A ’C80 PP can access the first and second halves of a given RAM at the same time. On the other hand, a ’C82 PP can only access either the first or second half of a given RAM in one cycle. In the case of the ’C82, one access will be stalled until the other access is complete.

This situation is one that you should avoid when you prototype ’C82 code on the ’C80; in the very least, take this into consideration when estimating performance on the ’C82.

3.4.6.2 Global and Local Ports in the ’C80 and ’C82

In the ’C82, each PP’s data is local to that PP; however, on the ’C80, each PP uses data RAMs from another PP to emulate DRAM1. Since the borrowed RAM is not local to each PP, any access to that RAM will be forced over the PP’s global port, regardless of whether the local port or global port was specified in the code.

If the PP’s DRAM1 is accessed using the PP’s local port, and a global port access is made in parallel, the instruction would stall for one cycle. This would cause the execution time on the ’C80 to be longer than the execution time on the ’C82.

Example 3–8 is a C PP example that calls a PP assembly language function to calculate the dot product of two vectors, A and B. Example 3–9 is the assembly language function called by the C program. These two examples together perform the same function as example.c, the sample program used in the previous example.

This example illustrates how to use the PP’s address units so that ’C82 code that is prototyped on the ’C80 can equivalent in terms of timing and functionality. In other words, the most important consideration in this case is to write the assembly code to avoid different execution times for each processor.

In the example, A is accessed (loaded) using the local port, since it is specifically linked to a portion of memory that is local to the PP that is executing the code. B, on the other hand, is accessed using the global port, since it is linked to a RAM that is global when it runs on the ’C80 (but not when it runs on the ’C82).

In the code, the access to vector B takes place through the global port. The following assembly language line shows how the port is specified using a ppca (register allocator/compactor) directive:

```
Ga_B_ptr .reg  ga
```

The access to vector A is specified as taking place through the local port. The following line of assembly language code shows how the port was specified using a ppca directive:

```
La_A_ptr .reg  la
```

The following lines of assembly language code show how the accesses to A and B are specified in parallel:

```
|| B =h *Ga_B_ptr++      ; load next B element
|| A =h *La_A_ptr++      ; load next A element
```

In your code, you can avoid timing differences by using the technique shown in the example: use the global port when accessing RAM that is global to a ’C80 PP, and use the local port when accessing RAM that is local to a ’C80 PP.

Example 3–8. Sample PP C Program With an Assembly Language Function

```
/*******************************************************
 * ex_asm.c   This example C code calculates the dot
 *            product of vectors A and B.
 *
 *            The code uses an assembly language function
 *            to calculate the dot product. The assembly
 *            language code has the filename c80ex.p.
 *******************************************************/
#include <mvp.h>
/* allocate space for A in a named section .asect */
#pragma DATA_SECTION(A,".a_sect")
/* allocate space for B in a named section .bsect */
#pragma DATA_SECTION(B,".b_sect")
/* define a the buffer size for A and B */
#define BUF_SIZE 1536
short A[BUF_SIZE];
short B[BUF_SIZE];
extern long dot_product(short *, short *, int);
main()
{
     long dot_prod;
     dot_prod = dot_product(A,B,BUF_SIZE);
}
```

Example 3–9. Assembly Language Function to Calculate the Dot Product of Two Vectors

```
a_addr       .set  d1                ; pointer to A[] passed in d1
b_addr       .set  d2                ; pointer to B[] passed in d2
vect_size    .set  d3                ; vector size is passed in d3
dot_product  .set  d5                ; the dot product of A and B is
                                     ; returned in d5.
A            .reg  d                 ; ppca determines the registers
B            .reg  d                 ; these variables will be assigned to.
prod         .reg  d
La_A_ptr     .reg  la
Ga_B_ptr     .reg  ga

      .lock d6, d7, a4, a12          ; tell ppca not to use
                                     ; these registers
      .entry a_addr, b_addr, vect_size     ; tell ppca that registers
                                     ; are live on entry (these
                                     ; are the arguments passed
                                     ; from the C calling function).

      .system $dot_product           ; define function entry point
      .system _dot_product           ; so that it is visible
                                     ; to both MP and PP C.
$dot_product:
_dot_product:
     La_A_ptr = a_addr               ; initialize a local address
                                     ; register to vector A.
     Ga_B_ptr = b_addr               ; initialize global address
                                     ; register to vector B.

     lrse2 = vect_size – 1           ; Use the fast initialization
                                     ; form to set up a single
                                     ; instruction loop.  The PP
                                     ; executes the instruction
                                     ; at DOT_PROD_LOOP vect_size
                                     ; times.

     prod = 0                        ; clear prod.
       || A =h *La_A_ptr++           ; load first element of A,
                                     ; using the local port
                                     ; and postincrement A’s
                                     ; pointer.

     dot_product = 0                 ; clear dot_product
       || B =h *Ga_B_ptr++           ; load first element of B,
                                     ; using the global port
                                     ; and postincrement B’s
                                     ; pointer.
DOT_PROD_LOOP:
     prod = A * B                    ; calculate the product of
                                     ; A and B
           || dot_product=dot_product+prod   ; calculate sum of products
        || B =h *Ga_B_ptr++          ; load next B element (global port)
        || A =h *La_A_ptr++          ; load next A element (local port)

      .cjump DOT_PROD_LOOP           ; tell ppca that the code conditionally
                                     ; loops to DOT_PROD_LOOP.
     br = iprs                       ; return to calling function.
     dot_product = dot_product + prod        ; calculate last sum.
     nop
      .uexit                         ; tell ppca that this is the end of
                                     ; this function.
```

Appendix A

The ’C80 and ’C82 Memory Maps

The TMS320C8x is a byte-addressable device with a single 4-GB memory space common to its processors. Each address refers to a specific byte in the address space. Addresses less than 02000000h are reserved for on-chip memory, and addresses from 02000000h to FFFFFFFFh are assigned to off-chip memory. The memory map is shown in Figure A–1 for the ’C80 and in Figure A–2 for the ’C82.

Figure A–1. TMS320C80 Memory Map

Starting Address (hex)	Ending Address (hex)	Bank Size (bytes)	Memory or Device
0000 0000	0000 07FF	2K	PP0 Data RAM 0
0000 0800	0000 0FFF	2K	PP0 Data RAM 1
0000 1000	0000 17FF	2K	PP1 Data RAM 0
0000 1800	0000 1FFF	2K	PP1 Data RAM 1
0000 2000	0000 27FF	2K	PP2 Data RAM 0
0000 2800	0000 2FFF	2K	PP2 Data RAM 1
0000 3000	0000 37FF	2K	PP3 Data RAM 0
0000 3800	0000 3FFF	2K	PP3 Data RAM 1
0000 4000	0000 7FFF	16K	Reserved
0000 8000	0000 87FF	2K	PP0 Data RAM 2
0000 8800	0000 8FFF	2K	Reserved
0000 9000	0000 97FF	2K	PP1 Data RAM 2
0000 9800	0000 9FFF	2K	Reserved
0000 A000	0000 A7FF	2K	PP2 Data RAM 2
0000 A800	0000 AFFF	2K	Reserved
0000 B000	0000 B7FF	2K	PP3 Data RAM 2
0000 B800	00FF FFFF	16M†	Reserved
0100 0000	0100 07FF	2K	PP0 Parameter RAM
0100 0800	0100 0FFF	2K	Reserved
0100 1000	0100 17FF	2K	PP1 Parameter RAM
0100 1800	0100 1FFF	2K	Reserved
0100 2000	0100 27FF	2K	PP2 Parameter RAM
0100 2800	0100 2FFF	2K	Reserved
0100 3000	0100 37FF	2K	PP3 Parameter RAM
0100 3800	0100 FFFF	50K	Reserved

† Block sizes have been rounded to the nearest unit size.

Figure A–1. TMS320C80 Memory Map (Continued)

Starting Address (hex)	Ending Address (hex)	Bank Size (bytes)	Memory or Device
0101 0000	0101 07FF	2K	MP Parameter RAM
0101 0800	0180 17FF	8M†	Reserved
0180 1800	0180 1FFF	2K	PP0 Instruction Cache
0180 2000	0180 37FF	6K	Reserved
0180 3800	0180 3FFF	2K	PP1 Instruction Cache
0180 4000	0180 57FF	6K	Reserved
0180 5800	0180 5FFF	2K	PP2 Instruction Cache
0180 6000	0180 77FF	6K	Reserved
0180 7800	0180 7FFF	2K	PP3 Instruction Cache
0180 8000	0180 FFFF	32K	Reserved
0181 0000	0181 07FF	2K	MP Data Cache 0
0181 0800	0181 0FFF	2K	MP Data Cache 1
0181 1000	0181 7FFF	28K	Reserved
0181 8000	0181 87FF	2K	MP Instruction Cache 0
0181 8800	0181 8FFF	2K	MP Instruction Cache 1
0181 9000	0181 FFFF	28K	Reserved
0182 0000	0182 01FF	512	Memory-Mapped TC Registers
0182 0200	0182 03FF	512	Memory-Mapped VC Registers
0182 0400	01FF FFFF	8M†	Reserved
0200 0000	FFFF FFFF	4G†	External Memory

† Block sizes have been rounded to the nearest unit size.

Figure A–2. TMS320C82 Memory Map

Starting Address (hex)	Ending Address (hex)	Bank Size (bytes)	Memory or Device
0000 0000	0000 0FFF	4K	PP0 Data RAM 0
0000 1000	0000 1FFF	4K	PP1 Data RAM 0
0000 2000	0000 7FFF	24K	Reserved
0000 8000	0000 8FFF	4K	PP0 Data RAM 1
0000 9000	0000 9FFF	4K	PP1 Data RAM 1
0000 A000	00FF FFFF	16M†	Reserved
0100 0000	0100 0FFF	4K	PP0 Parameter RAM
0100 1000	0100 1FFF	4K	PP1 Parameter RAM
0100 2000	0100 FFFF	56K	Reserved
0101 0000	0101 0FFF	4K	MP Parameter RAM
0101 1000	0180 0FFF	8M†	Reserved
0180 1000	0180 1FFF	4K	PP0 Instruction Cache
0180 2000	0180 2FFF	4K	Reserved
0180 3000	0180 3FFF	4K	PP1 Instruction Cache
0180 4000	0180 FFFF	48K	Reserved
0181 0000	0181 0FFF	4K	MP Data Cache
0181 1000	0181 7FFF	28K	Reserved

† Block sizes have been rounded to the nearest unit size.

Figure A–2. TMS320C82 Memory Map (Continued)

Starting Address (hex)	Ending Address (hex)	Bank size (bytes)	Memory or Device
0181 8000	0181 8FFF	4K	MP Instruction Cache
0181 9000	0181 FFFF	28K	Reserved
0182 0000	0182 01FF	512	Memory-Mapped TC Registers
0182 0200	01FF FFFF	8M†	Reserved
0200 0000	FFFF FFFF	4G†	External Memory

† Block sizes have been rounded to the nearest unit size.