TMS320C5x
DSP Starter Kit
User's Guide

PRINTED WITH
SOY INK™

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

This book describes the DSP (digital signal processor) Starter Kit (DSK) and how to use the DSK with these tools:

❑ The DSK assembler
❑ The DSK debugger

How to Use This Manual

The goal of this book is to help you learn to use the DSK assembler and debugger. This book is divided into three parts:

❑ **Part I: Hands-On Information** is presented first so that you can start using your DSK the same day you receive it.

   ■ Chapter 1 describes the features and provides an overview of the TMS320C5x DSP Starter Kit.

   ■ Chapter 2 contains installation instructions for your assembler and debugger. It lists the hardware and software tools you'll need to use the DSK and tells you how to set up its environment.

   ■ Chapter 3 lists the key features of the assembler and debugger and tells you the steps you need to take to assemble and debug your program.

❑ **Part II: Assembler Description** contains detailed information about using the assembler.

   ■ Chapter 4 explains how to create DSK assembler source files and invoke the assembler.

   ■ Chapter 5 discusses the valid directives and gives you an alphabetical reference to these directives.

❑ **Part III: Debugger Description** contains detailed information about using the debugger. Chapter 6 explains how to invoke the DSK debugger and use its pulldown menus, dialog boxes, and debugger commands.

Notational Conventions

This document uses the following conventions.

❏ Program listings, program examples, interactive displays, filenames, and symbol names are shown in a special typeface similar to a typewriter's. Examples use a **bold version** of the special typeface for emphasis; interactive displays use a **bold version** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

Here is a sample program listing:

```
00001  ——    0a00          .ps    0a00h
00002  ——    ——            .entry
>>>>> ENTRY POINT SET TO 0a00
00003  0a00  8b88          mar    *,AR0
00004  0a01  8ba9   loop   mar    *+, AR1
```

Here is an example of a system prompt and a command that you might enter:

```
C:> dska testfile.asm
```

❏ In syntax descriptions, the instruction, command, or directive is in a **bold face** font and parameters are in an *italics*. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Syntax that is entered on a command is centered in a bounded box. Syntax that is used in a text file is left-justified in an unbounded box. Here is an example of a directive syntax:

> **.include** **"***filename***"**

.include is the directive. This directive has one parameter, indicated by *filename*. When you use .include, the parameter must be an actual filename, enclosed in double quotes.

❏ Square brackets (**[** and **]**) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you don't enter the brackets themselves. Here's an example of an instruction that has an optional parameter:

> **LACC** *16-bit constant* [*, shift*]

The LACC instruction has two parameters. The first parameter, *16-bit constant*, is required. The second parameter, *shift*, is optional. If you use the optional second parameter, you must precede it with a comma.

❏ Braces ({ and }) indicate a list. The symbol **|** (read as *or*) separates items within the list. Here's an example of a list:

{ * | *+ | *– }

This provides three choices: *, *+, or *–.

Unless the list is enclosed in square brackets, you must choose one item from the list.

❏ In assembler syntax statements, column 1 is usually reserved for the first character of an *optional* label or symbol. If a label or symbol is a *required* parameter, the symbol or label will be shown starting against the left margin of the shaded box as in the example below. No instruction, command, directive, or parameter, other than a symbol or label, should begin in column one.

symbol **.set** *symbol value*

In the above example, the symbol is required for the .set directive and must begin in column 1.

❏ Some directives can have a varying number of parameters. For example, the .word directive can have several parameters. The syntax for this directive is:

Note that .word does not begin in column 1.

.word   0abcdh,56

This syntax shows that .word must have at least one value parameter, but you have the option of supplying a label or additional value parameters, separated by commas.

Information About Cautions and Warnings

This book may contain warnings.

This is an example of a warning statement.

A warning statement describes a situation that could potentially cause harm to <u>you</u>.

Related Documentation From Texas Instruments

The following books describe the TMS320C5x and related support tools. To obtain a copy of any of these documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C5x User's Guide (literature number SPRU056) describes the 'C5x 16-bit, fixed-point, general-purpose digital signal processors. Covered are its architecture, internal register structure, instruction set, pipeline, specifications, DMA, I/O ports, and on-chip peripherals.

TMS320C1x/C2x/C2xx/C5x Assembly Language Tools User's Guide (literature number SPRU018) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C1x, 'C2x, 'C2xx, and 'C5x generations of devices.

TMS320C2x/C2xx/C5x Optimizing C Compiler User's Guide (literature number SPRU024) describes the 'C2x/C2xx/C5x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C2x, 'C2xx, and 'C5x generations of devices.

TMS320C5x C Source Debugger User's Guide (literature number SPRU055) tells you how to invoke the 'C5x emulator, evaluation module, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320 DSP Designer's Notebook: Volume 1 (literature number SPRT125) presents solutions to common design problems using 'C2x, 'C3x, 'C4x, 'C5x, and other TI DSPs.

If You Need Assistance . . .

If you want to . . .	Contact Texas Instruments at . . .	
Visit TI online	World Wide Web:	http://www.ti.com
Receive general information or assistance	World Wide Web:	http://www.ti.com/sc/docs/pic/home.htm
	North America, South America:	(214) 644–5580
	Europe, Middle East, Africa	
	Dutch:	33–1–3070–1166
	English:	33–1–3070–1165
	French:	33–1–3070–1164
	Italian:	33–1–3070–1167
	German:	33–1–3070–1168
	Japan (Japanese or English)	
	Domestic toll-free:	0120–81–0026
	International:	81–3–3457–0972 or 81–3–3457–0976
	Korea (Korean or English):	82–2–551–2804
	Taiwan (Chinese or English):	886–2–3771450
Ask questions about Digital Signal Processor (DSP) product operation or report suspected problems		(713) 274–2320
	Fax:	(713) 274–2324
	Fax Europe:	+33–1–3070–1032
	Email:	4389750@mcimail.com
	World Wide Web:	http://www.ti.com/dsps
	Bulletin Board Service (BBS) North America:	(713) 274–2323 8–N–1
	BBS Europe:	+44–2–3422–3248
	320 BBS Online:	ftp.ti.com:/mirrors/tms320bbs (192.94.94.53)
Request tool updates	Software:	(214) 638–0333
	Software fax:	(214) 638–7742
	Hardware:	(713) 274–2285
Order Texas Instruments documentation (see Note 1)	Literature Response Center:	(800) 477–8924
Make suggestions about or report errors in documentation (see Note 2)	Email:	comments@books.sc.ti.com
	Mail:	Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251–1443

Notes: 1) The literature number for the book is required; see the lower-right corner on the back cover.

2) Please mention the full title of the book, the literature number from the lower-right corner of the back cover, and the publication date from the spine or front cover.

FCC Warning

This equipment is intended for use in a laboratory test environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to subpart J of part 15 of FCC rules, which are designed to provide reasonable protection against radio frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case the user at his own expense will be required to take whatever measures may be required to correct this interference.

Trademarks

IBM, PC, and PC-DOS are registered trademarks of International Business Machines Corp.

MS-DOS is a registered trademark of Microsoft Corp.

AT is a trademark of International Business Machines Corp.

Contents

Part I: Hands-On Information

Part II: Assembler Description

Part III: Debugger Description

Figures

Tables

Examples

Introduction

This chapter provides an overview of the TMS320C5x DSP Starter Kit (DSK). The 'C5x DSK is a low-cost, simple, stand-alone application board that lets you experiment with and use 'C5x DSPs for real-time signal processing. The DSK has a 'C50 onboard to allow full-speed verification of the 'C5x code. The DSK also gives you the freedom to create your own software to run on the DSK board, allows you to build new boards, or to expand the system in many ways. The supplied debugger is windows-oriented, which simplifies code development and debugging capabilities.

1.1 Key Features

This section describes the key features of the TMS320C5x DSK:

❏ Industry standard 'C50 fixed-point DSP

❏ 50 ns instruction cycle time

❏ 32K-byte PROM (programmable read-only memory)

❏ Voice quality analog data acquisition via the TLC32040 AIC (analog interface circuit)

❏ Standard RCA connectors for analog input and output that provide direct connection to microphone and speaker

❏ XDS510 emulator connector

❏ I/O expansion bus for external design

1.2 DSK Overview

Figure 1–1 depicts the basic block diagram of the 'C50. It shows the interconnections, which include the host interface, analog interface, and emulation interface. PC communications are via the RS-232 port on the DSK board. The 32K bytes of PROM contain the kernel program for boot loading.

All pins of the 'C50 are connected to the external I/O interfaces. The external I/O interfaces include four 24-pin headers, a 4-pin header, and a 14-pin XDS510 header.

The TLC32040 AIC interfaces to the 'C50 serial port. Two RCA connectors provide analog input and output on the board.

Figure 1–1. 'C5x DSK Block Diagram
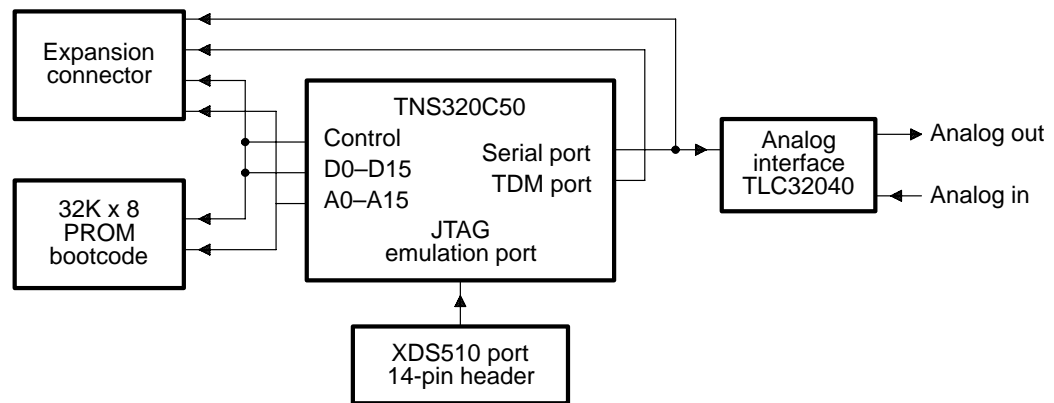
1.3 Memory

The 'C5x DSK is one of the simplest 'C5x DSP application boards. Even though no external memory is available on the board, the 10K on-chip RAM of the 'C50 provides enough memory for most DSP application programs. The kernel program is contained in the 32K, 8-bit PROM. The PROM is only for DSK boot loading and cannot be accessed after boot loading, as this portion of the on-chip memory is reserved for the kernel program. Figure 1–2 shows the memory map of the 'C5x DSK.

Figure 1–2. Memory Map of the 'C5x DSK

Program	Data
0000h — Bootloader (on-chip) ROM	Memory map registers
0800h — Interrupt vectors†	0060h — Reserved by kernel B2
0840h — Debugger kernel program	0080h — Reserved
0980h — User's program	0100h — B0‡
2C00h — External space	0300h — B1
FE00h — B0‡	0500h — Reserved
FFFFh	0800h — Reserved by debugger kernel
	0980h — User's space
	2C00h — External space
	FFFFh
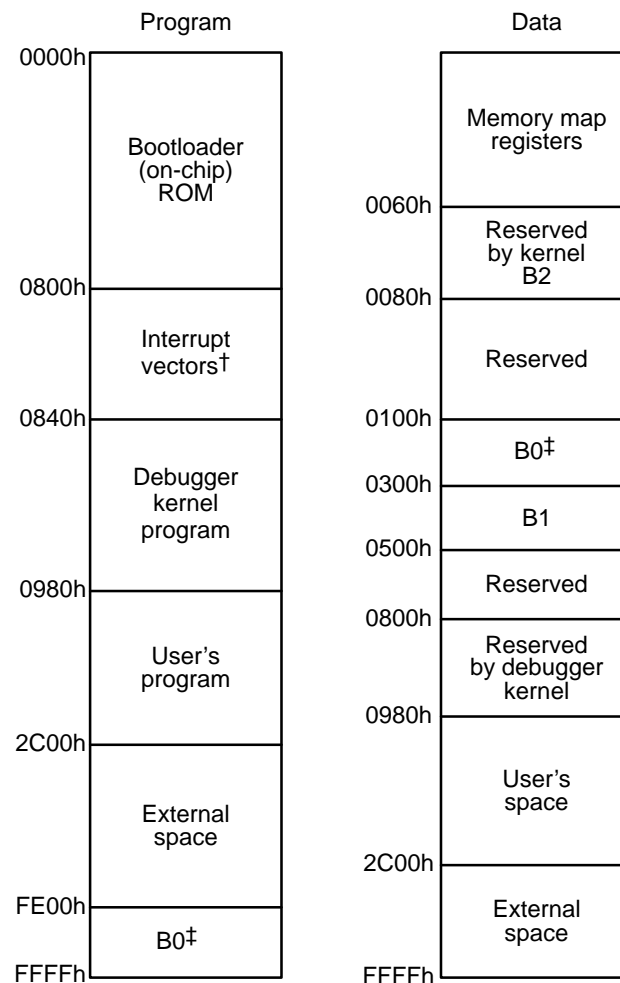
† INT2 of TRAP is reserved by the DSK.

‡ B0 may be configured as either program or data memory, depending on the value of the CN bit in status register ST1.

The on-chip, dual-access, random-access-memory (DARAM) B2 is reserved as a buffer for the status registers. The single-access, random-access-memory (SARAM) is configured as program and data memory. The kernel program is stored in this area from 0x840h–0x980h. If the kernel program performs an overwrite, a reset signal is required to let the DSK reload the kernel program. Since the kernel program is stored in the SARAM, this on-chip memory cannot be configured as data memory only (RAM = 0). The interrupt vectors are allocated, starting from 0x800h. The IPTR in the PMST register should not be modified (refer to subsection 3.6.3, Status and Control Registers of the *TMS320C5x User's Guide*).

The TLC32040 AIC on the board provides a single-channel, input/output, voice-quality analog interface with the following features:

❑ Single-chip digital-to-analog (D/A) and analog-to-digital (A/D) conversion with 14 bits of dynamic range

❑ Variable D/A and A/D sampling rate and filtering

The AIC interfaces directly to the 'C50 serial port. The master input clock to the AIC is provided by a 10-MHz timer output from the 'C50.

The AIC is hard-wired for 16-bit word mode operation. The reset pin of the AIC is connected to the $\overline{BR}$ pin of the 'C50.

DSK analog capabilities are suited to many applications, including audio data processing. You can directly connect most preamplified microphones and speakers to the DSK analog input and output. For more information concerning the AIC, refer to the AIC data sheet, literature number SLAS014E.

The DSK provides six headers, including the XDS510, to help you to design your own external hardware. The majority of the 'C50 and other integrated circuit (IC) signals to the board are connected to these headers. The XDS510 header allows the DSK to become a portable XDS510 target system.

The 'C5x DSK has its own windows-oriented debugger that makes it easy to develop and debug software code. The DSK communicates with the PC using the XF and $\overline{BIO}$ pins through the RS-232 serial port. Figure 1–3 shows the module connections between the RS-232 serial port of the PC and the DSK.

Figure 1–3. DSK to RS-232 Connections
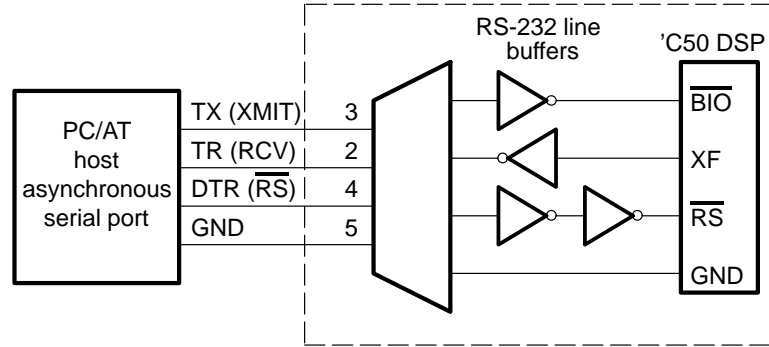
The DSK has its own assembler. Refer to Chapters 4, 5, and 6 for a description of the assembler and the debugger and their uses.

Installing the
DSK Assembler and Debugger

This chapter describes how to install the DSP Starter Kit (DSK) on an IBM PC ™ system running under PC-DOS ™ or MS-DOS ™.

2.1   What You'll Need

The following checklists detail items that are shipped with the DSK assembler and debugger and any additional items you'll need to use these tools. The DSK module connections for an RS-232 cable are also discussed in this section.

Hardware checklist

☐	**host**	An IBM PC, AT™, or 100% compatible PC with a hard disk system and a 1.2M-byte floppy-disk drive
☐	**memory**	Minimum of 640K bytes
☐	**display**	Monochrome or color (color recommended)
☐	**power requirements**	A 9 V_{ac} @ 250 mA (or greater) power supply with a 2.1-mm power jack connector, which is common to most wall-mounted AC transformers. A low-current UL transformer is recommended, because it is designed to hold up during brief power surges.

Notes:

1) You may want to use the DSK's on-board power supply and regulators for external circuits. If so, you must not overload the circuit. External loads cause the regulators to operate at a higher temperature. Loads > 50 mA are not recommended.

2) If you are using an external power supply, be sure you connect it correctly; the DSK is not warrantied after you have made modifications to it.

WARNING

To minimize risk of electric shock and fire hazard, the power supply adapter should be rated class 2 or safety extra-low voltage. The adapter and personal computer providing energy to this product should be certified by one or more of the following: UL, CSA, VDE, TUV.

☐	**board**	DSK circuit board
☐	**port**	Asynchronous RS-232 serial communications link
☐	**cable**	RS-232 with a DB9 interface

☐   **optional hardware**   An EGA- or VGA-compatible graphics display card and a large monitor. The debugger has two options that allow you to change the overall size of the debugger display. To use a larger screen size, you must invoke the debugger with the −l option. For more information about debugger options, refer to page 6-2.

☐   **miscellaneous materials**   Blank, formatted disks

Software checklist

☐   **operating system**   MS-DOS or PC-DOS (version 4.01 or later)

☐   **files**   *dsk5a.exe* is an executable file for the DSK assembler.

               dsk5d.exe is an executable file needed for running the DSK debugger interface.

☐   **miscellaneous files**   Other files are included in your DSK package, such as sample source files and additional documentation. You can find a brief description of these files in the Readme file included on your disk. Be sure to check the Readme file for the latest information on software changes and DSK operation.

Note:

Other applications for the DSK can also be downloaded from the TMS320 BBS or Internet file transfer protocol (FTP) site. See the *If You Need Assistance* subsection on page vii, for the Internet address.

DSK module connections

You need an RS-232 cable to connect your PC to your DSK board. The DSK is designed with a DB9 RS-232 connection mounted on the board. Figure 2–1 shows the DSK module connections.

Figure 2–1. The DSK Module Connections for an RS-232 Cable
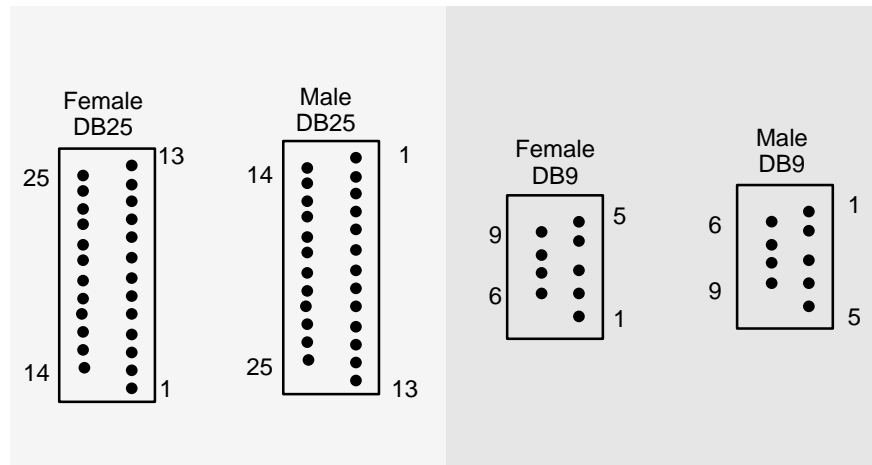
	Pin Assignments	
Signal Name	**DB25**	**DB9**
Protective ground	1	
Transmit data	2	3[†]
Receive data	3	2[†]
Request to send	4	7
Clear to send	5	8
Data set ready	6	6
Signal ground	7	5[†]
Carrier detect	8	1
Data terminal ready	20	4[†]
Ring indicator	22	9

[†] These signals are used by the DSK.

2.2 Step 1: Connecting the DSK to Your PC

Follow these steps to connect your DSK board to your PC:

1) Turn off your PC's power.

2) Connect your RS-232 cable to either communication port 1 or 2 on your PC.

3) Connect your RS-232 cable to a 25-to-9 pin adapter, if necessary.

4) Plug the RS-232 cable (or adapter) into the DSK board. Refer to Figure 2–2 for details.

Figure 2–2. Connecting Your RS-232 Cable Into Your DSK Board
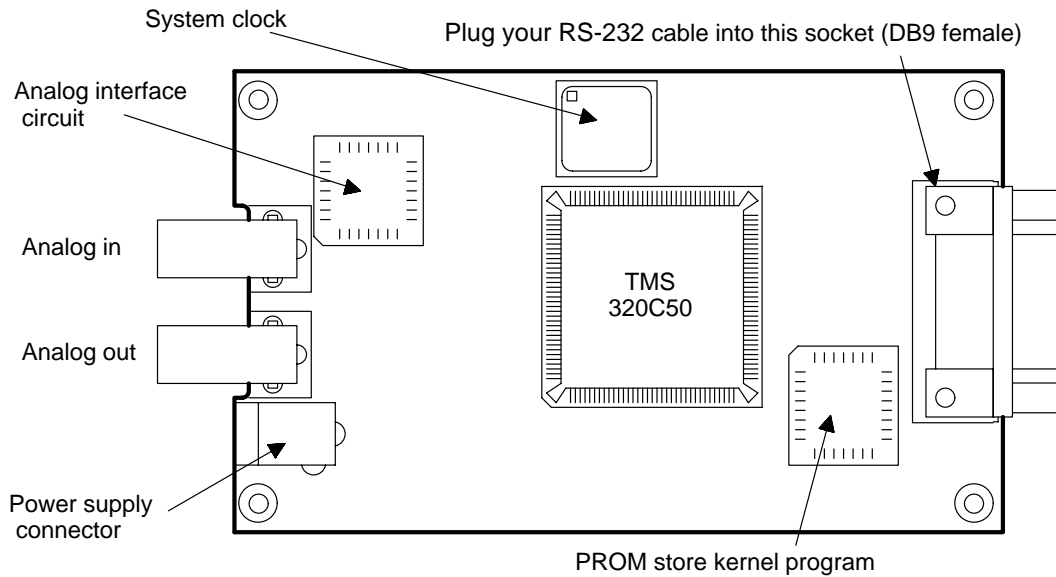
For schematics and more detail on the DSK board, refer to Appendix A.

5) Connect the 9-V_{ac} transformer onto the DSK board. Refer to Figure 2–2 for details.

6) Plug the transformer into a wall socket.

7) Turn your PC's power on.

Note:

The following are suitable RS-232 connections:

❑ DB9 male connected to DB25 female
❑ DB9 male connected to DB9 female

2.3   Step 2: Installing the DSK Software

This section explains the process of installing the debugger software on a hard disk system.

1) Make a backup copy of the product disk. (If necessary, refer to the DOS manual that came with your computer.)

2) On your hard disk or system disk, create a directory named *dsktools*. This directory is for the DSK assembler and debugger software. To create this directory, enter:

   md c:\dsktools ⏎

3) Insert your product disk into drive A. Copy the contents of the disk:

   copy a:*.* c:dsktools*.* /v ⏎

2.4 Step 3: Modifying Your config.sys File

When using the debugger, you can have only 20 files open or active at one time. To tell the system not to allow more than 20 active files, you must add the following line to your config.sys file:

FILES=20

Once you edit your config.sys file and add the line, invoke the file by turning off the PC's power and turning it on again.

2.5   Step 4: Modifying the PATH Statement

To ensure that your debugger works correctly, you must modify the PATH statement to identify the dsktools directory. Not only must you do this before you invoke the debugger for the first time, *you must do it any time you power up or reboot your PC.*

You can accomplish this by entering individual DOS commands, but it's simpler to put the commands in your system's autoexec.bat file. The general format for doing this is:

PATH=C:\dsktools;*pathname2***;***pathname3***;. . .**

This allows you to invoke the debugger without specifying the name of the directory that contains the debugger executable file.

If you are modifying your autoexec.bat file and it already contains a PATH statement, simply include *;C:\dsktools* at the end of the statement as shown in Figure 2–3.

Figure 2–3. DOS Command Setup for the DSK Environment (Sample autoexec.bat File)

```
                              DATE
                              TIME
                              ECHO OFF
PATH statement  ──────────>   PATH=c:\dos;c:\dsktools
                              CLS
```
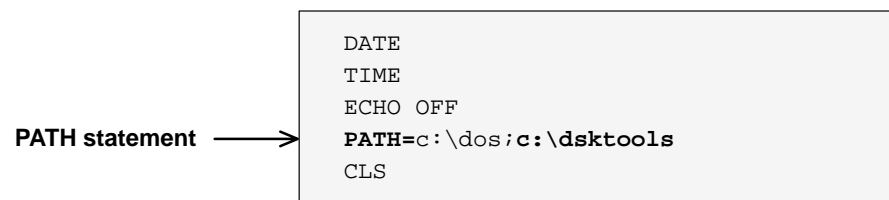
If you modify the autoexec.bat file, be sure to invoke it before invoking the debugger for the first time. To invoke this file, enter:

autoexec ⏎

2.6 Step 5: Verifying the Installation

To ensure that you have correctly installed your DSK board, assembler, and debugger, enter one of the following commands at the system prompt:

❑ If you are using serial communication port 1 (com1), enter:

dsk5d c1 ⏎

❑ If you are using serial communication port 2 (com2), enter:

dsk5d c2 ⏎

❑ If you are using serial communication port 3 (com3), enter:

dsk5d c3 ⏎

❑ If you are using serial communication port 4 (com4), enter:

dsk5d c4 ⏎

Use c1, c2, c3, or c4 to identify the serial port that the debugger uses for communicating with your PC. The default setting is c1.

After entering the dsk5d command, you should see a display similar to this one:

Figure 2–4. The Basic Debugger Display

```
 Display Fill Load Help eXec Quit Modify Break Init Watch Reset Save Copy Pc

┌──── TMS320C50 Reverse Assembler ────┐ ┌─TMS320C50 Watches─┐ ┌─TMS320C50 Register─┐
                                                                
 ADDR CODE WORD MNMC OPERANDS  FIELD    [0] hex (1010h)       ACC  :00000000  C:0
                                        edffh = 0166777o      ACCB :00000000 OV:0
 0a00 be41      SETC INIM                                     PRG  :00000000 PM:0
 0a01 bcic      LDP  #28                                      TRG0 :0000TRG1:0000
 0a02 bf80 0000 LACC #0000h,0                                 TRG2 :0000  DP:0000
 0a04 c000      MPY  #0000h                                   ST0:0600  ST1: 01fc
 0a05 bf09 0e00 LAR  AR1,#0e00h                               PC: 0a00  AR0: 0000
 0a07 8b89      MAR  *,AR1                                    St0: 0000 AR1: 000a
 0a08 bb06      RPT  #6                                       St1: 0000 AR2: 0000
 0a09 a2a0 0e00 MAC  0e00h,*+                                 St2: 0000 AR3: 0000
 0a0b be04      APAC                                          St3: 0000 AR4: 0000
 a0ac 90a0      SACL *+,0                                     St4: 0000 AR5: 0000
 0a0d 1100      LACC 0000h,1                                  St5: 0000 AR6: 0000
 0a0e 90a0      SACL *+,0                                     St6: 0000 AR7: 0000
 0a0f 1101      LACC 0001h,1                                  DRR: 0000 DXR: 0000
 0a10 90a0      SACL *+,0                                     TIM: 2932 PRD: ffff
                                                             IMR: 0002 IFR: 000b
                                                             PMST:0834 INDX:0000
                                                             DBMR:0000 BMAR:0000
┌─TMS320C50 Display Data Memory: 'Hexadecimal' format ─┐     CWSR:0000 GRG: ff00
                                                             SPCR:0800 TCR: 0000
 1000:  0000 ffef feef bfdf 0100 0008 0000. ∩∩▶ . . . .
 1007:  0800 ffff ffff fffd fffe 0001 1200. . .2▪(). . .
 100e:  2000 0000 edff 7ffe ffbf ffff 0240  . .▪⎤ .@. .
 1015:  0000 0000 0010 fdff 7fff ffff fef7. .▶ . . . .
 101c:  1101 0000 2841 0000 ff7d ffff ffff.() A.}. . . .

 INPUT COMMAND ▌
```

If you see a display similar to this one, you have correctly installed your DSK board, assembler, and debugger. If you don't see a display, your software or cables may not be installed properly. Go through the installation instructions again and make sure that you have followed each step correctly; then reenter the command above.

Installation errors

If you still do not see a display, one or more of the following conditions may be the cause:

❑ Your baud setting may be incorrect. Some Windows and OS/2 applications and notebook computers have low-level software limitations that may affect baud settings. Refer to page 6-2 for valid baud settings.

❑ You may have used an incorrect communication port (com1 versus com 2). Refer to page 6-3 for more information on communication ports.

❑ Your communication port channel may be interrupted or noisy. If so, try using a lower baud rate. Refer to page 6-2 for valid baud rates.

❑ A mouse driver or other software may be using the same communication port you are attempting to use with the DSK. If so, try another communication port for the DSK. Refer to page 6-3 for more information on communication ports.

❑ Your RS-232 cable and connectors may not be connected snugly.

❑ Your 9-V_{ac} transformer may not be plugged in on both ends. When the DSK is receiving power the LM7805 voltage regulator is warm to the touch.

Overview of a Code Development and Debugging System

The DSP Starter Kit (DSK) lets you experiment with and use a DSP for real-time signal processing. The DSK gives you the freedom to create your own software to run on the board as is or to build new boards and expand the system in any number of ways.

The DSK assembler and debugger are software interfaces that help you to develop, test, and refine DSK assembly language programs.

This chapter provides an overview of the assembler and debugger and describes the overall code development process.

3.1 Description of the DSK Assembler

The DSK assembler is a simple and easy-to-use interface. Only the most signifi-cant features of an assembler have been incorporated. Note that this is not a COFF assembler; however, you can create object files by using the TI TMS320 fixed-point DSP assembly language tools that load and run on the DSK.

Key features of the assembler

❏ **Quick.** The DSK assembler differs from many other assemblers in that it does not go through a linker phase to create an output file. Instead, the DSK uses special directives to assemble code at an absolute address during the assembly phase. As a result, you can create small programs quickly and easily.

❏ **Easy to use.** If you want to create larger programs, you can do so by simply chaining files together with the .include directive.

3.2 Description of the DSK Debugger

The debugger is easy to learn about and to use. Its friendly, window-oriented interface reduces learning time and eliminates the need to memorize complex commands. The debugger is capable of loading and executing code with single-step, breakpoint, and run-time halt capabilities.

Figure 3–1 identifies several features of the debugger display. When you invoke the debugger, you should see a display similar to this one (it may not be exactly the same, but it should be close).

Figure 3–1. The Basic Debugger Display
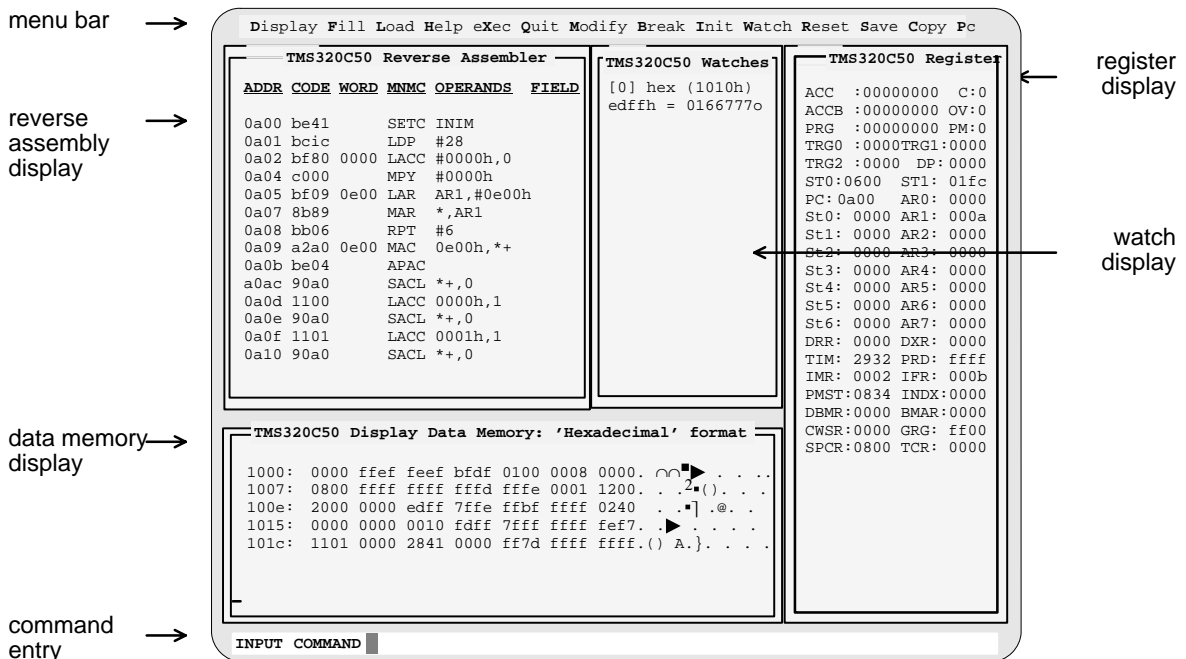
Key features of the debugger

❏ **Easy to use, window-oriented interface.** The DSK debugger separates code, data, and commands into manageable portions.

❏ **Powerful command set.** Unlike many other debugging systems, this debugger doesn't force you to learn a large, intricate command set. The DSK debugger supports a small but powerful command set.

❏ **Flexible command entry.** There are two main ways to enter commands: at the command line or by using the menu bar. Choose the method that you like better.

3.3   Developing Code for the DSK

Figure 3–2 illustrates the DSK code development flow.

Figure 3–2. DSK Software Development Flow
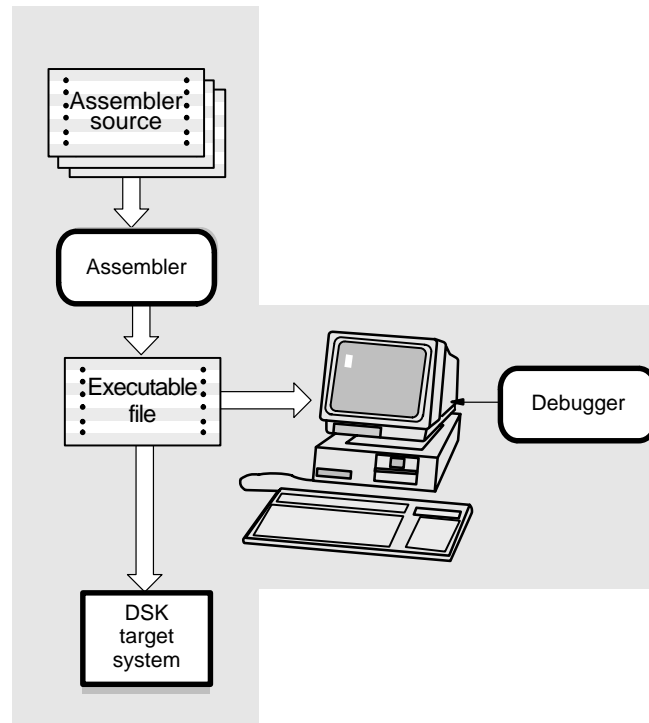
The following list describes the tools shown in Figure 3–2.

assembler

The **assembler** translates DSK assembly language source files into machine language object files for the TMS320C5x family of processors. Only the most essential features of an assembler have been incorporated. This is *not* a COFF assembler, although executable object files created by the TMS320 fixed-point DSP assembly language tools will also load and run on the DSK.

debugger

The goal of the development process is to produce a module that can be executed in a **DSK target system.** You can use the **debugger** to refine and correct your code.

3.4 Getting Started

This section provides a quick walkthrough so that you can get started without reading the entire user's guide. These examples show the most common methods for invoking the assembler and debugger.

1) Create a short source file to use for the walkthrough; call it try1.asm.

Example 3–1. Source File try1.asm

```
******************************************************************************
* Saw-toothed Wave Generator                                                 *
*  Ramp rate is determined by interrupt rate and step size. Ramp is made by  *
*  numerical rollover. No AIC initialization, using the default value.       *
******************************************************************************
; Declare memory-mapped registers and
; program block address
        .mmregs              ; Include memory map reg
        .ps      0080ch      ;
        B        XINT        ; Set transmit interrupt vector
        .ps      00a00h      ;
        .entry               ; Initial PC address
        LDP      #0          ; Load Data Page for DXR (Zero)
        CALL     SP_init     ; Call serial port initialize function
        LAMM     IMR         ;
        OR       #20h        ; Unmask receive interrupt (XINT)
        SAMM     IMR         ;
        SPLK     #0ffffh, IFR; Clear pending interrupt
LOOP:   ADD      #10         ; Increment ACCU by 10
        SACL     DXR,3       ; Shift ACCU left 3 bits when storing
        IDLE                 ; Wait for D/A interrupt
        B        LOOP        ;
XINT:   RETE                 ; Reenable interrupts
SP_init:
        SPLK     #01h,PRD    ; Generate 10 MHz clock from TOUT to
        SPLK     #20h,TCR    ; support AIC master clock
        MAR      *,AR0       ;
        LACC     #0080h      ; Set 00000080h => ACC
        SACH     DXR         ; Clear DXR
        SACL     GREG        ; Set GREG = 80h, >8000h memory = Global
        LAR      AR0,#0FFFFh ; AR0 point to global memory
        RPT      #9999       ; Bring the BR low for 10000 cycles
        LACC     *,0,AR0     ; (.5ms at 50ns)
        SACH     GREG        ; Disable global memory
        LACC     #0008h      ; Put serial port in reset and configure as
        SACL     SPC         ; burst mode, FSX input, and data length 16 bits
        LACC     #80c8h      ;
        SACL     SPC         ; Bring the serial port out of reset
        RET
```

2) Enter the following command to assemble try1.asm:

dsk5a try1 ⌨

This command invokes the TMS320C5x DSK assembler. If the input file extension is .asm (for example, try1.asm), you don't have to specify the extension; the assembler uses .asm as the default. For more information about invoking the assembler, refer to Section 4.5.

When you enter this command, the debugger creates an executable file called try1.dsk.

3) To see a listing of errors and warnings that may have occurred during assembly, assemble try1.asm with the –l option (lowercase L).

dsk5a try1 –l ⌨

4) This time, the assembler not only creates an executable file, it creates a listing file called try1.lst. The listing file is helpful because it contains a list of all unresolved symbols and opcodes.

Example 3–2. Assembler Created List File try1.lst

```
00001 ----  ----  *****************************************************************************
00002 ----  ----  *Saw-toothed Wave Generator                                                 *
00003 ----  ----  *  Ramp rate is determined by interrupt rate and step size. Ramp is made  *
00004 ----  ----  * by numerial rollover. No AIC initialization, using the default value.*
00005 ----  ----  *****************************************************************************
00006 ----  ----  ; Declare memory-mapped registers and
00007 ----  ----  ; program block address
00008 ----  0000          .mmregs             ; Include memory map reg
00009 ----  080c          .ps     0080ch      ;
00010 080c  7980          B       XINT        ; Set transmit interrupt vector
      080d  0000
00011 ----  0a00          .ps     00a00h      ;
00012 ----  0000          .entry              ; Initial PC address
>>>>> ENTRY POINT SET TO 0a00
00013 0a00  bc00          LDP     #0          ; Load Data Page for DXR (Zero)
00014 0a01  7a80          CALL    SP_init     ; Call serial port initialize function
      0a02  0000
00015 0a03  0804          LAMM    IMR         ;
00016 0a04  bfc0          OR      #20h        ; Unmask receive interrupt (XINT)
      0a05  0020
00017 0a06  8804          SAMM    IMR         ;
00018 0a07  ae06          SPLK    #0ffffh,IFR ; Clear pending interrupt
      0a08  ffff
00019 0a09  b80a LOOP:    ADD     #10         ; Increment ACCU by 10
0020  0a0a  9321          SACL    DXR,3       ; Shift ACCU left 3 bits when storing
00021 0a0b  be22          IDLE                ; Wait for D/A interrupt
00022 0a0c  7980          B       LOOP        ;
      0a0d  0a09
00023 0a0e  be3a XINT:    RETE                ; Re-enable interrupts
00024 ----  ---- SP_init:
00025 0a0f  ae25          SPLK    #01h,PRD    ; Generate 10 MHz clock from TOUT to
      0a10  0001
00026 0a11  ae26          SPLK    #20h,TCR    ; support AIC master clock
      0a12  0020
00027 0a13  8b88          MAR     *,AR0       ;
00028 0a14  bf80          LACC    #0080h      ; Set 00000080h => ACC
      0a15  0080
00029 0a16  9821          SACH    DXR         ; Clear DXR
00030 0a17  9005          SACL    GREG        ; Set GREG = 80h, >8000h memory = Global
00031 0a18  bf08          LAR     AR0,#0FFFFh ; AR0 point to global memory
      0a19  ffff
00032 0a1a  bec4          RPT     #9999       ; Bring the BR low for 10000 cycles
      0a1b  270f
00033 0a1c  1088          LACC    *,0,AR0     ; (.5ms at 50ns)
00034 0a1d  9805          SACH    GREG        ; Disable global memory
00035 0a1e  bf80          LACC    #0008h      ; Put serial port in reset and configure as
      0a1f  0008
00036 0a20  9022          SACL    SPC         ; burst mode, FSX input, and data length 16 bits
00037 0a21  bf80          LACC    #80c8h      ;
      0a22  80c8
00038 0a23  9022          SACL    SPC         ; Bring the serial port out of reset
00039 0a24  ef00          RET
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0  WARNINGS:0
```

5) Now you are ready to debug your program. Enter the following command to invoke the debugger:

    dsk5d ⏎

6) This command brings up the TMS320C5x DSK debugger on your screen. Now you can load your try1.dsk sample program by using the LOAD command. For more information on using the debugger, refer to Chapter 6.

Using the DSK Assembler

This chapter tells you how to use the DSK assembler and describes valid DSK source files.

4.1 Creating DSK Assembler Source Files

To create a DSK assembler source file, you can use almost any ASCII program editor. Be careful using word processors; these files contain various formatting codes and special characters which word processors may alter.

DSK assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, and comments. Your source statement lines can be up to 80 characters per line.

The following example shows several lines of source statements:

```
          .ps    0a00h      ;initialize PC
sym       .set   2          ;symbol sym=2
Begin:    add    #sym       ;add sym (5) to accumulator
          .word  016h       ;initialize a word with 016h
          sacl   sym        ;store accumulator-location sym(5)
LAB_1:
LAB_2:    b      LAB_1
LAB_3:    b      LAB_2      ;location of LAB_1 & LAB_2 are same
          b      LAB_3      ;LAB_3 is at next address
```

Your source statement can contain four ordered fields. The general syntax for source statements is as follows:

[*label*][**:**]   *mnemonic*   [*operand list*]   [**;** *comment*]

Follow these guidelines:

❏ All statements must begin with a label, a blank, an asterisk, or a semicolon.

❏ Labels are optional; if you use them, they must begin in column 1.

❏ One or more blanks must separate each field. Note that tab characters are equivalent to blanks.

❏ Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column *must* begin with a semicolon.

Using valid labels

Labels are optional for all assembly language instructions and for most (but not all) assembler directives. When you use them, a label *must* begin in column 1 of a source statement. A label can contain up to 16 alphanumeric characters (A–Z, a–z, 0–9, and _). Labels are case sensitive and the first character cannot be a number. For example:

```
        .ps      0a00h    ; Your code can start here
        .entry
Start:  mar      *,AR0
        lar      AR0,#0
        lacl     #03fh    ; Turn on all interrupts
        ldp      #0       ;IMR located in page 0
        sacl     4        ;store mask to IMR
```

In the preceding example, the colon is optional. The DSK assembler does not require a label terminator.

When you use a label, its value is the current value of the section program counter (the label points to the statement with which it is associated). If, for example, you use the .word directive to initialize several words, a label would point to the first word. In the following example, the label *Begins* has the value 0a00h.

```
00001 ---- ----   *assume other code was assembled
00002 ---- ----
00003 ---- 0a00            .ps    0a00h
00004 0a00 000a  Begins:   .word 0Ah,3,7
      0a01 0003
      0a02 0007
```

When a label appears on a line by itself, it points to the instruction on the next line:

```
00018 ---- fb00            .ds    0fb00h
00019 ---- ---- Here:
00020 fb00 000a            .word 0Ah,3,7
00021 ---- ----
```

When an opcode or directive references a label, the label is substituted with the address of the label's location in memory. The only exceptions are the .set directive, which assigns a value to a label, and the LDP opcode, which loads the nine most significant bits (MSB) of the address.

If you don't use a label, the first character position must contain a blank, a semi-colon, or an asterisk.

Using the mnemonic field

The mnemonic field follows the label field. *The mnemonic field must not start in column 1 or it will be interpreted as a label.* The mnemonic field can contain one of the following opcodes:

❑ Machine-instruction mnemonic (such as ADD, MPY, POP)
❑ Assembler directive (such as .data, .set, .entry)

If you have a label in the first column, a space, colon, or tab must separate the mnemonic field (opcode) from the label. For example:

```
        .ps     0a00h    ; Your code can start here
        .entry
START:  mar     *,AR0
        lar     AR0,#0
        lacl    #03fh    ; Turn on all interrupts
        ldp     #0
```

Refer to your *TMS320C5x User's Guide* for syntax specifications on individual opcodes.

Using the operand field

The operand field is a list of operands that follow the mnemonic field. An operand can be a constant (see Section 4.2) or a symbol (see Section 4.3). You must separate operands with commas.

The assembler allows you to specify that a constant, symbol, or expression should be used as an address, an immediate value, or an indirect value. The following rules apply to the operands of instructions.

❏ **No prefix — the operand is a well-defined immediate value**. The assembler expects a well-defined immediate value, such as a register symbol or a constant. This is an example of an instruction that uses operands without prefixes:

```
Label:ADD A3
```

The assembler adds the contents of address A3 to the contents of the accumulator.

❏ *** prefix — the operand is a register indirect address.** If you use the ***** sign as a prefix, the assembler treats the operand as an indirect address; that is, it uses the operand as an address. For example:

```
Label:ADD *+,AR3
```

The following symbols are used in indirect addressing, including bit-reversed (BR) addressing. A * indicates that the contents of AR are used as the data memory address plus the functions indicated.

Table 4–1. Indirect Addressing

Operand	Additional Functions
*	
*+	Incremented after the access
*–	Decremented after the access
*0+	The contents of INDX are added to AR after the access
*0–	The contents of INDX are subtracted from AR after the access
*BRO+	The contents of AR0 are added to AR with reverse carry (rc) propagation after the access
*BRO–	The contents of AR0 are subtracted from AR with reverse carry (rc) propagation after the access

For more information on indirect addressing and bit-reversed addressing, refer to *Memory Addressing Modes* in the *TMS320C5x User's Guide*.

Commenting your source file

A comment can begin in any column and extend to the end of the source line. A comment can contain any ASCII character, including blanks. Comments are printed in the assembly source listing, but they do not affect the assembly.

You can comment your source file in one of two ways. The most common way is to place a semicolon anywhere on the line you want to comment. All text placed after the semicolon is ignored by the DSK assembler. For example:

```
; Your code can start here
      .ps   0a00h
      .entry
START: mar   *,AR0
      lar   AR0,#0
      lacl  #03fh    ; Turn on all interrupts
      ldp   #0
```

Another way to comment your source file is to use an asterisk *in the first column* of your code.

```
* Your code can start here
      .ps   0a00h
      .entry
START: mar   *,AR0
      lar   AR0,#0
* Turn on all interrupts
      lacl  #03fh
      ldp   #0
```

If the asterisk is not in the first column, the assembler assumes it is part of your code and may generate an error.

A source statement that contains only a comment is valid.

4.2 Constants

The assembler supports four types of constants:

❑ Decimal integer constants
❑ Hexadecimal integer constants
❑ Binary integer constants
❑ Character constants

The assembler maintains each constant internally as a 32-bit quantity. Constants *are not sign extended*. For example, the constant 0FFh is equal to 00FF (base 16) or 255 (base 10); it *does not* equal −1.

Decimal integers

A decimal integer constant is a string of decimal digits, ranging from −2 147 483 647 to 4 294 967 295. Examples of valid decimal constants are:

1000   Constant equal to 1000_{10} or $3E8_{16}$
−32768   Constant equal to $-32\ 768_{10}$ or 8000_{16}
25   Constant equal to 25_{10} or 19_{16}

Hexadecimal integers

A hexadecimal integer constant is a string of up to eight hexadecimal digits followed by the suffix H (or h). Hexadecimal digits include the decimal values 0–9 and the letters A–F or a–f. *A hexadecimal constant must begin with a decimal value (0–9).* These are examples of valid hexadecimal constants:

78h   Constant equal to 120_{10} or 0078_{16}
0Fh   Constant equal to 15_{10} or $000F_{16}$
37ACh   Constant equal to $14\ 252_{10}$ or $37AC_{16}$

Binary integers

A binary integer constant is a string of 0s and 1s followed by the suffix B (or b). Examples of valid binary constants include:

0101b   Constant equal to 5
10101b   Constant equal to 21
−0101b   Constant equal to −5

Character constants

A character constant is a single character enclosed in *double* quotes. The characters are represented as 8-bit ASCII characters.

4.3   Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of up to 16 alphanumeric characters (A–Z, a–z, 0–9, $, –, and +); symbols cannot contain embedded blanks. The *first character* in a symbol cannot be a number or special character. The symbols you define are case sensitive; for example, the assembler recognizes *ABC*, *Abc*, and *abc* as three unique symbols.

Labels

Symbols that are used as labels become symbolic addresses that are associated with locations in the program. A label must be unique. Do not use register names as labels.

```
.................the label Begins has the value of a00h.
00001 ---- ----   *assume other code was assembled
00002 ---- ----
00003 ---- 0a00              .ps    0a00h
00004 0a00 000a   Begins:    .word  0Ah,3,7
      0a01 0003
      0a02 0007
```

Constants

Symbols can be set to constant values. By using constants, you can equate meaningful names with constant values. The .set directive enables you to set constants to symbolic names. Symbolic constants *cannot* be redefined. The following example shows how these directives can be used:

```
;==========================================================
; Example showing valid symbols, labels and references
;==========================================================
        .ps   0a00h    ; Initialize PC
K       .set  12       ; constant definition K = 12
K*2     .set  24       ; constant definition K*2 = 24
BIN     .set  01010101b; BIN = 055h
max_buf .set  K*2      ; constant definition max_buf = K*2
= 24
+A      .set  10       ; constant definition << Incorrect
        lacl  #K       ; loads 12
        lacl  #-K      ; loads -12
        lacl  #K*2     ; loads 24
        lacl  max_buf  ; loads 24
        lacl  !BIN     ; loads 0AAh
```

4.4   **Using Symbols as Expressions**

Unlike other assemblers, the DSK assembler is not capable of analyzing numerical or logical expressions. However, by removing all of the spaces within a field so that the expression is a continuous string, you can set the entire string to a specific value (see Example 4–1).

Example 4–1. Analyzing Expressions With the DSK by Using Continuous Strings

(a) Expression analysis with a COFF assembler

```
FFT    .set   256
       LAR    AR0, #FFT
       LACC   #FFT –1      ;expression analysis
```

(b) Expression analysis with the DSK assembler

```
FFT    .set   256
FFT–1  .set   255      ;set string FFT–1 = 255
       LAR    AR0, #FFT
       LACC   #FFT-1   ;FFT-1 is a complete string
```

In Example 4–1 (b) , FFT–1 is a continuous string. The .set directive equates the value 256 to the symbol FFT and 255 to the symbol FFT–1; these symbols can now be used in place of their values. The two opcodes now contain the following:

```
LAR  AR0, #256
LACC  #255
```

4.5   Assembling Your Program

Before you attempt to debug your programs, you must first assemble them. Here's the command for invoking the assembler when preparing a program for debugging:

dsk5a   [*filename(s)*]   [*–options*]

dsk5a   is the command that invokes the assembler.

filenames   are one or more assembly language source files. Filenames are not case sensitive.

–options   affect the way the assembler processes input files.

Options and filenames can be specified in any order on the command line.

Table 4–2 lists the assembler options; the following subsections describe the options.

Table 4–2. Summary of Assembler Options

Option	Description
–k	Generates an output file regardless of errors or warnings
–l	Generates a temporary file containing a list of any unresolved opcodes or symbols
asm	Allows you to define assembler statements from the command line

Generating an output file (–k option)

By default, the DSK deletes a file corrupted with errors. For debugging purposes, the –k option tells the DSK assembler to generate an output file despite any errors or warnings found.

Creating a temporary object file (–l option)

The DSK assembler generates an intermediate listing file containing all un-resolved opcodes when you use the –l (lowercase L) option. For example, if you want to assemble a file named test.asm and create a listing file, enter:

```
dsk5a test –l
```

The above example creates the file test.lst from the file test.asm. Any unresolved symbols are resolved after the DSK assembler has read the entire assembly file.

Defining assembler statements from the command line (asm option)

The asm option allows you to define assembler statements from the command line. Since the DSK does not have a linker, using the asm option allows you to specify constants and load addresses. The general format for the command containing this option is:

dsk5a *filename* **asm"***statement***"** [**asm"***statement***"** ...]

For example:

dsk5a test.asm asm"FFT .set 256" asm" .entry 0a00h"

This statement specifies a program entry point (or load address) of 0a00h and generates the file test.inc, in the following format:

```
FFT .SET 256
 .entry 0a00h
```

All asm statements are written to an include file named file.inc, overwriting the previous file.

The asm statement is also useful for controlling parameter values such as .set, or controlling conditional assembler execution by using such directives as the .if/.else/.endif.

dsk5a test.asm asm"fft .set 256"

In this example, the asm statement is assigning a value of 256 to the symbol fft.

Assembler Directives

Assembler directives supply program data and control the assembly process. They allow you to do the following:

❏ Assemble code and data into specified sections
❏ Reserve space in memory for uninitialized variables
❏ Control the appearance of listings
❏ Initialize memory
❏ Assemble conditional blocks
❏ Define global variables

Topic	Page

5.1 Using the DSK Assembler Directives

Table 5–1 summarizes the assembler directives. Note that all source statements that contain a directive may have a label and a comment. To improve readability, they are not shown as part of the directive syntax.

Table 5–1. Assembler Directives Summary

(a) Directives that define sections

Mnemonic and Syntax	Description
.data	Assemble into data memory
.ds [*address*]	Assemble into data memory (initialize data address)
.entry [*address*]	Initialize the starting address of the program counter when loading a file
.ps [*address*]	Assemble into program memory (initialize program address)
.text	Assemble into program memory

(b) Directives that reference other files

Mnemonic and Syntax	Description
.copy [*"*]*filename*[*"*]	Include source statements from another file
.include [*"*]*filename*[*"*]	Include source statements from another file

(c) Conditional assembly directives

Mnemonic and Syntax	Description
.else	Optional conditional assembly
.endif	End conditional assembly
.if *well-defined expression*	Begin conditional assembly

Table 5–1. Assembler Directives Summary (Continued)

(a) Directives that initialize constants (data and memory)

Mnemonic and Syntax	Description
.bfloat $value_1$ [,..., $value_n$]	Initialize a 16-bit, 2s-complement exponent and a 32-bit, 2s-complement mantissa—an unpacked floating-point number
.byte $value_1$ [,..., $value_n$]	Initialize one or more successive words in the current section
.double $value_1$ [,..., $value_n$]	Initialize a 64-bit, IEEE double-precision, floating-point constant
.efloat $value_1$ [,..., $value_n$]	Initialize a 16-bit, 2s-complement exponent and a 16-bit, 2s-complement mantissa—a less accurate unpacked floating-point number
.float $value_1$ [,..., $value_n$]	Initialize a 32-bit, IEEE single-precision, floating-point constant
.int $value_1$ [,..., $value_n$]	Initialize one or more 16-bit integers
.long $value_1$ [,..., $value_n$]	Initialize one or more 32-bit integers
.lqxx $value_1$ [,..., $value_n$]	Initialize a 32-bit, signed 2s-complement integer whose decimal point is displaced *xx* places from the LSB
.qxx $value_1$ [,..., $value_n$]	Initialize a 16-bit, signed 2s-complement integer whose decimal point is displaced *xx* places from the LSB
.space *size in bits*	Reserve *size* bits in the current section; note that a label points to the beginning of the reserved space
.string *"string$_1$"* [,..., *"string$_n$"*]	Initialize one or more text strings
.tfloat $value_1$ [,..., $value_n$]	Initialize a 32-bit, 2s-complement exponent and a 64-bit, 2s-complement mantissa; note that the initialized integers are in unpacked form
.word $value_1$ [,..., $value_n$]	Initialize one or more 16-bit integers

(b) Miscellaneous directives

Mnemonic and Syntax	Description
.end	Program end
.listoff	End source listing (overrides the −l assembler option)
.liston	Restart the source listing (overrides the −l assembler option)
.set	Equate a value with a local symbol
.mmregs	Enter memory-map registers into symbol table

5.2   Directives That Define Sections

Five directives associate the various portions of an assembly language program with the appropriate sections:

❑ The **.data** directive identifies portions of code to be placed in data memory. Data memory usually contains initialized data.

❑ The **.ds** directive functions like .data; however, with .ds you can specify an optional address to initialize a new data address.

❑ The **.entry** directive identifies the starting address of the program counter. The current address is used by default, but you can specify an optional address.

❑ The **.ps** directive identifies portions of code to be placed in program memory. With .ps you can specify an additional address to initialize a new program address.

❑ The **.text** directive identifies portions of code in the .text section. The .text section usually contains executable code.

Example 5–1 shows how you can use sections directives to associate code and data with the proper sections. This is an output listing; column 1 shows line numbers, and column 2 shows the section program counter (SPC) values. (Each section has its own section program counter, or SPC. When code is first placed in a section, its SPC equals 0. When you resume assembling into a section, its SPC resumes counting as if there had been no intervening code.

After the code in Example 5–1 is assembled, the sections contain:

.text   Initialized bytes with the values 1, 2, 3, 4, 5, and 6

.data   Initialized bytes with the values 9, 10, 11, and 12

Example 5–1. Sections Directives

```
00001 ---- ----  *********************************
00002 ---- ----  * Initialize section addresses  *
00003 ---- ----  *********************************
00004 ---- 0a00                  .ps     0a00h
00005 ---- 0e00                  .ds     0e00h
00006 ---- ----  *********************************
00007 ---- ----  * Start assembling into .text   *
00008 ---- ----  *********************************
00009 ---- ----                  .text
00010 0a00 0001                  .byte   1,2
      0a01 0002
00011 0a02 0003                  .byte   3,4
      0a03 0004
00012 ---- ----  *********************************
00013 ---- ----  * Start assembling into .data   *
00014 ---- ----  *********************************
00015 ---- ----                  .data
00016 0e00 0009                  .byte   9,10
      0e01 000a
00017 0e02 000b                  .byte   11,12
      0e03 000c
00018 ---- ----  *********************************
00019 ---- ----  * Resume assembling into .text  *
00020 ---- ----  *********************************
00021 ---- ----                  .text
00022 0a04 0005                  .byte   5,6
      0a05 0006
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0   WARNINGS:0
```

Note:

You can use the .ps and .ds directives to assemble your code to the same memory locations. This won't cause an assembly error; however, it is possible to overwrite previously defined memory blocks.

5.3   Directives That Reference Other Files

The **.copy** and **.include** directives tell the assembler to read source statements from another file. This is the syntax for these directives:

❑   **.copy** *"filename"*

❑   **.include** *"filename"*

The .copy and .include directives tell the assembler to begin reading source statements from another file. When the assembler finishes reading the source statements in the copy/include file, it resumes reading source statements from the current file. The statements read from the copied or included files are printed in the listing file.

The *filename* names a copy/include file that the assembler reads statements from. The *filename* can be a complete pathname, a partial pathname, or a filename with no path information. The assembler searches for the file in the directory that contains the current source file. The current source file is the file being assembled when the .copy or .include directive is encountered.

5.4   Directives that Enable Conditional Assembly

The **.if/.else/.endif** directives tell the assembler to conditionally assemble a block of code according to the evaluation of an expression. Note that you cannot nest .if statements.

❑ The **.if** *expression* directive marks the beginning of a conditional block and assembles code if the .if condition is true (not zero).

❑ The **.else** directive marks a block of code to be assembled if .if is false.

❑ The **.endif** directive marks the end of a conditional block and terminates the block.

The *expression* parameter can be either a numeric value or a previously defined symbol.

5.5 Directives That Initialize Memory

Each of these directives, with the exception of the .byte and .string directives, aligns the object to a 16-bit word boundary.

❑ The **.byte** directive places one or more 8-bit values into consecutive words of the current section.

❑ The **.word** directive places one or more 16-bit values into consecutive words in the current section.

❑ The **.string** directive places 8-bit characters from one or more character strings into the current section.

❑ The **.long** directive places one or more 32-bit values into consecutive 32-bit fields in the current section.

❑ The **.int** directive places one or more 16-bit values into consecutive words in the current section.

❑ The **.qxx** directive places one or more 16-bit, signed 2s-complement values into consecutive words in the current section. Note that the decimal point is displaced *xx* places from the LSB.

❑ The **.lqxx** directive places one or more 32-bit, signed 2s-complement values into consecutive 32-bit fields in the current section. Note that the decimal point is displaced *xx* places from the LSB.

❑ The **.float** directive calculates 32-bit IEEE floating-point representations of single precision floating-point value and stores it in two consecutive words in the current section.

❑ The **.bfloat** directive calculates a 16-bit, signed 2s-complement exponent and a 32-bit, signed 2s-complement mantissa.

❑ The **.efloat** directive calculates a 16-bit, signed 2s-complement exponent and a 16-bit, signed 2s-complement mantissa.

❑ The **.tfloat** directive calculates a 32-bit, signed 2s-complement exponent and a 64-bit, signed 2s-complement mantissa.

❑ The **.double** directive calculates a 64-bit IEEE floating-point representation of a double precision floating-point value and stores it in four consecutive words in the current section.

❑ The **.space** directive reserves a specified number of bits in the current section. The assembler advances the SPC and skips the reserved words.

When you use a label with .space, it points to the *first* word of the reserved block.

Figure 5–1 shows an example of the .space directives. Assume the following code has been assembled for this example:

```
        .ps     0a00h
        .word   100h, 200h
RES_1:  .space  30h             ;Reserve 48 bits or 3 words
        .word   15
```

Res_1 points to the first byte in the space reserved by .space.

Figure 5–1. The .space Directive

3 words reserved

Res_1 = 0a02h
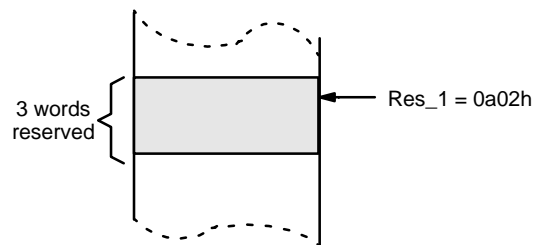
5.6 Miscellaneous Directives

This section discusses miscellaneous directives.

❏ The **.end** directive terminates assembly. It should be the last source statement of a program. This directive has the same effect as an end-of-file character.

❏ The **.listoff** directive overrides the –l option and prohibits source listing.

❏ The **.liston** directive begins source listing.

❏ The **.mmregs** directive defines symbolic names for the memory-mapped register. Using .mmregs is the same as executing a .set for all memory-mapped registers—for example, greg .set 4—and makes it unnecessary to define these symbols. See Table 5–2, page 5-25, for a list of memory-mapped registers.

❏ The **.set** directive equates meaningful symbol names to constant values or strings. The symbol is stored in the symbol table and cannot be redefined; for example:

```
 bval     .set    0100h
          .byte   bval
          B       bval
```

5.7   Directives Reference

The remainder of this chapter is a reference. Generally, the directives are organized alphabetically, one directive per page; however, related directives (such as .if/.else/.endif) are presented on the same page. Here's an alphabetical table of contents for the directive reference:

Syntax

.byte $value_1$ [, ... , $value_n$]

.string $string_1$ [, ... , $string_n$]

Description

The **.byte** and **.string** directives place one or more 8-bit values into consecutive bytes of the current section. A value or a string can be either:

❑ An expression that the assembler evaluates and treats as an 8-bit signed number, or

❑ A character string enclosed in double quotes. Each character in a string represents a separate value.

The .byte directive places one or more 8-bit values into consecutive words of the current section.

Unlike the .byte directive, the .string directive places the 8-bit values into memory in a packed form in the order they are encountered. If a word is not filled, the remaining bits are filled with zeros.

Example

This example shows several 8-bit values placed into consecutive bytes in memory. The label strx has the value a00h, which is the location of the first initialized byte. The label stry has the value a07h, which is the location of the first byte initialized by the .string directive.

```
00001 ---- 0a00           .ps      0a00h
00002 0a00 000a  strx:    .byte    10,-1,2,0Ah,"abc"
      0a01 00ff
      0a02 0002
      0a03 000a
      0a04 0061
      0a05 0062
      0a06 0063
00003 0a07 0aff  stry:    .string  10,-1,2,0Ah,"abc"
      0a08 020a
      0a09 6162
      0a0a 6300
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0    WARNINGS:0
```

In the above example, abc is converted into three ASCII characters.

Syntax	**.copy** **"***filename***"**
	.include **"***filename***"**

Description The **.copy** and **.include** directives tell the assembler to read source statements from a different file. The assembler:

1) Stops assembling statements in the current source file

2) Assembles the statements in the copied/included file

3) Resumes assembling statements in the main source file, starting with the statement that follows the .copy or .include directive

filename is a required parameter that names a source file; the filename must be enclosed in double quotes and must follow operating system conventions. You can specify a full pathname (for example, c:\dsktools\file1.asm). If you do not specify a full pathname, the assembler searches for the file in the current directory.

The statements that are assembled from an included file are printed in the assembly listing, depending on the .liston/.listoff directives and –l option.

The .copy and .include directives can be nested within a file being copied or included. The assembler limits this type of nesting to eight levels; the host operating system may set additional restrictions.

Example This example shows how the .include directive is used to tell the assembler to read and assemble source statements from other files, and then resume assembling into the current file.

Source file: (source.asm)

```
                                ;filename: source.asm
.space          10h             ;filename: source.asm
.include        "byte.asm"      ;filename: source.asm
                                ;filename: source.asm
.space          20h             ;filename: source.asm
```

First copy file: (byte.asm)

```
                                ;filename: byte.asm
.byte           'a', 0ah, 32    ;filename: byte.asm
.include        "word.asm"      ;filename: byte.asm
.byte           11,12,13        ;filename: byte.asm
                                ;filename: byte.asm
```

Second copy file: (word.asm)

```
                                ;filename: word.asm
.word           0abcdh, 56      ;filename: word.asm
                                ;filename: word.asm
```

Listing file:

```
00001 ---- 0e00    .ds      0e00h      ;filename:source.asm
00002 ---- ----                        ;filename:source.asm
00003 0e00 0010    .space   10h        ;filename:source.asm
00004 ---- ----    .include "byte.asm" ;filename:source.asm
*******************************************
*    OPENING INCLUDE FILE  byte.asm
*******************************************
00001 ---- ----                        ;filename:byte.asm
00002 0e01 0061    .byte    "a",0ah,32  ;filename:byte.asm
      0e02 000a
      0e03 0020
00003 ---- ----    .include "word.asm" ;filename:byte.asm
*******************************************
*    OPENING INCLUDE FILE  word.asm
*******************************************
00001 ---- ----                        ;filename:word.asm
00002 0e04 abcd    .word    0abcdh,56   ;filename:word.asm
      0e05 0038
00003 ---- ----                        ;filename:word.asm
>>>>> FINISHED READING ALL FILES
*******************************************
*   CLOSING FILE   word.asm
*******************************************
00004 0e06 000b    .byte    11,12,13    ;filename:byte.asm
      0e07 000c
      0e08 000d
*******************************************
*    CLOSING FILE  byte.asm
*******************************************
00005 ---- ----                        ;filename:source.asm
00006 0e09 0020    .space   20h         ;filename:source.asm
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0   WARNINGS:0
```

Syntax	**.data**
	.ds [*address*]

Description   The **.data** and **.ds** directives tell the assembler to begin assembling source code into data memory. The .data and .ds sections are normally used to contain tables of data or preinitialized variables.

address is an optional parameter that specifies a 16-bit address. Normally, the section program counter is set to 0 the first time the .data or .ds section is assembled; you can use this parameter to assign an initial value to the SPC.

Note that the assembler assumes that .text is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you specify a section control directive.

Example   This example shows the assembly of code into the .data and .text sections.

```
        .ps     0a00h           ;set up load and run addresses
        .entry
        .include "VECT.ASM"
        .ds     0400h
        .text
setup:  mar     *,AR0           ;initialize the CPU registers
        lar     AR0,#0
        lar     AR1,#0
        lar     AR2,#0
        lar     AR3,#0

        .data
val_1   .int    0,1,2,3,4,5,6,7 ;init. integer values
:
        .text                   ;continue with some code
loop:   mar     *+,AR1
        mar     *+,AR2
        mar     *+,AR3
        b       loop,*+,AR0

        .data
val_2   .float  0,1,2,3,4,5,6,7 ;init. flt-pt values
:
```

Syntax   **.end**

Description   The **.end** directive is an optional directive that terminates assembly. It should
                        be the last source statement of a program. The assembler ignores any source
                        statements that follow an .end directive.

Example   This example shows how the .end directive terminates assembly.

                        Source file:

```
            .ps       0a00h
            .entry
START       NOP
            NOP
            NOP
            B         START
            .end
LAB         ADD       #5
            sub       #1
            B         LAB
```

Listing file:

```
00001 ---- 0a00              .ps      0a00h
00002 ---- 0000              .entry
>>>>> ENTRY POINT SET TO 0a00
00003 0a00 8b00    START     NOP
00004 0a01 8b00              NOP
00005 0a02 8b00              NOP
00006 0a03 7980              B        START
      0a04 0a00
00007 ---- ----              .end
>>>>> LINE:7  .END ENCOUNTERED
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0   WARNINGS:0
```

Syntax   **.entry** [*value*]

Description   The **.entry** directive tells the assembler the address of the program counter
                       when a file is loaded. If you do not use the *value* parameter, the current program
                       memory address, determined by the .ps or .text section, becomes the starting
                       address. If you have more than one .entry directive in your file, then the last
                       .entry directive encountered becomes the starting address of your code.

Example   Here is an example of the .entry directive.

```
            .ps     0a00h
 LOOP:      MAR     *+,AR1              ;An infinite loop
            B       LOOP,*+,AR0         ;
            .entry                      ;Start program
            MAR     *,AR0
            LAR     AR0,#0
            LAR     AR1,#0
            B       LOOP                ;call the routine
```

Syntax

.float   *value* [,..., *value_n*]

.bfloat   *value* [,..., *value_n*]

.double   *value* [,..., *value_n*]

.efloat   *value* [,..., *value_n*]

.tfloat   *value* [,..., *value_n*]

Description

The **.float** directive places the floating-point representation of a single floating-point constant into two words in the current section. *value* must be a floating-point constant. Each constant is converted to a floating-point value in 32-bit IEEE floating-point format.

The IEEE floating-point format consists of three fields:

❏ A 1-bit sign field (*s*)
❏ An 8-bit biased exponent (*exponent*)
❏ A 23-bit normalized mantissa (*mantissa*)
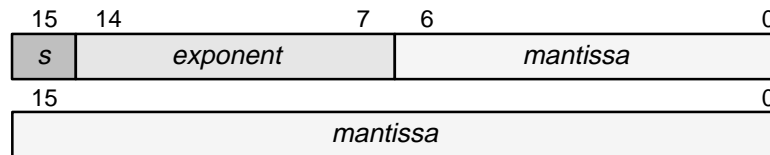
The **.bfloat** directive format is slightly different in that it has a 16-bit exponent and both a high and low mantissa:
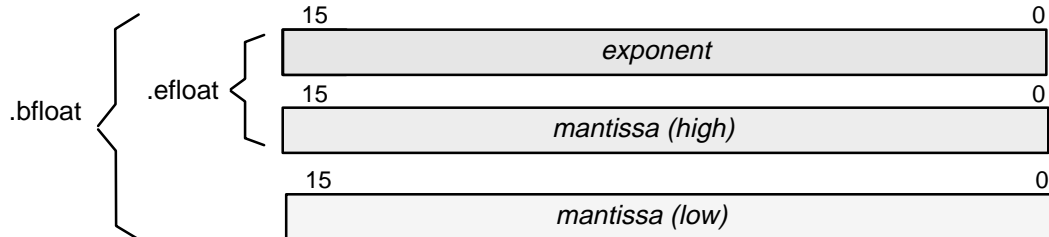
Example

Here is an example of floating-point directives.

Source file:

```
.ds          0400h
.bfloat      1.5,3,6
.bfloat      −1.5,3,6
.efloat      1.5,3,6
.end
```

Listing file:

```
00001 ---- 0400           .ds     0400h
00002 0400 0000           .bfloat 1,5,3,6
      0401 6000
      0402 0000
      0403 0000
      0404 6000
      0405 0001
      0406 0000
      0407 6000
      0408 0002
00003 0409 0000           .bfloat -1,5,3,6
      040a a000
      040b 0000
      040c 0000
      040d 6000
      040e 0001
      040f 0000
      0410 6000
      0411 0002
00004 0412 6000           .efloat 1,5,3,6
      0413 0000
      0414 6000
      0415 0001
      0416 6000
      0417 0002
00005 ---- ----           .end
>>>>> LINE:5  .END ENCOUNTERED
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0   WARNINGS:0
```

Syntax

> **.if**   *well-defined expression*
>
> **.else**
>
> **.endif**

Description   Three directives provide conditional assembly:

❏ The **.if** directive marks the beginning of a conditional block. The *well-defined expression* is a required parameter.

■ If the expression evaluates to *true* (nonzero), the assembler assembles the code that follows it (up to an .else or an .endif).

■ If the expression evaluates to *false* (0), the assembler assembles code that follows an .else (if present) or an .endif.

❏ The **.else** directive identifies a block of code that the assembler assembles when the .if expression is false (0). This directive is optional in the conditional block; if an expression is false and there is no .else statement, the assembler continues with the code that follows the .endif.

❏ The **.endif** directive terminates a conditional block.

Nested .if/.else/.endif directives are not valid.

Example   Here are some examples of conditional assembly:

```
yes      .set     1
no       .set     0
B0_Dat   .set     no
B1_Dat   .set     yes
If_1:    .if      B0_Dat
         .ds      0100h
         .endif
         .if      B1_Dat
         .ds      0300h
If_2:    .endif
```

Note:

In this instance, the asm option can be particularly useful in turning on the .if conditional statement from the command line. For example, you could enter:

```
dsk5a test asm"B0_Dat .set 1"
```

Syntax	**.liston**
	.listoff

Description   The **.liston** and **.listoff** directives can be useful in debugging. They override the –l assembler option, which turns on the output listing. The source listing is always written to a file with an extension of .lst.

Example   Here's an example of a source file and its output listing file.

Source file:

```
**************************************
*     .liston/off example
**************************************
      .ds       0400h
      .listoff
DATA  .word     1,2,3,4,5   ;Do not want this listed!
      .liston               ;Note this line isn't listed
      .ps       0a00h
      lacl      #PROG
      lar       AR0,#DATA
      rpt       #3          ;move 4 words from DS to PS
      tblw      *+
loop  nop
      b         loop
      .word     0,0,0,0
```

Listing file:

```
00001 ---- ----   *****************************************
00002 ---- ----   *       .liston/off example
00003 ---- ----   *****************************************
00004 ---- 0400           .ds             0400h
00005 ---- ----           .listoff
00008 ---- 0a00           .ps             0a00h
00009 0a00 bf80           lacc            #PROG
      0a01 0a09
00010 0a02 bf08           lar             AR0,#DATA
      0a03 0400
00011 0a04 bb03           rpt             #3
00012 0a05 a7a0           tblw            *+
00013 0a06 8b00           loop            nop
00014 0a07 7980           b               loop
      0a08 0000
00015 0a09 0000   PROG    .word           0,0,0,0
      0a0a 0000
      0a0b 0000
      0a0c 0000
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0   WARNINGS:0
```

Syntax   **.long** $value_1$ [, ... , $value_n$]

Description   The **.long** directive places one or more 32-bit values into consecutive words
                        of the current section. *value* can be either:

                        ❏   An expression that the assembler evaluates and treats as a 32-bit signed
                            number, or

                        ❏   A character string enclosed in double quotes. Each character in a string
                            represents a separate value.

                        If you use a label, it points to the location at which the assembler places the
                        first byte.

Example   This example shows several 32-bit values placed into consecutive bytes in
                        memory. The label strx has the value 0FB00h, which is the location of the first
                        initialized byte.

```
00001 ---- fb00          .ps    0fb00h  ;
00002 fb00 2710  strx:   .long  10000,"String","A"
      fb01 0000
      fb02 0053
      fb03 0000
      fb04 0074
      fb05 0000
      fb06 0072
      fb07 0000
      fb08 0069
      fb09 0000
      fb0a 006a
      fb0b 0000
      fb0c 0067
      fb0d 0000
      fb0e 0041
      fb0f 0000
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0   WARNINGS:0
```

Syntax   .**lqxx** *value₁* [, ... , *valueₙ*]

                          .**qxx** *value₁* [, ... , *valueₙ*]

Description   The **.qxx** and **.lqxx** directives generate signed, 2s-complement fractional integers and long integers whose decimal points are displaced *xx* places from the LSB.

Example   Here's an example of the .qxx directive.

```
00001 ———— 0400              .ds     0x400
00002 0400 2000              .Q15    0.25
00003 0401 4000              .Q15    0.5
00004 0402 6000              .Q15    0.75
00005 0403 e000              .Q15    −0.25,−0.5,−0.75
      0404 c000
      0405 a000
00006 0406 0000              .LQ24   9,10
      0407 0900
      0408 0000
      0409 0a00
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0   WARNINGS:0
```

Syntax   **.mmregs**

Description   The **.mmregs** directive defines global symbolic names for the TMS320 registers and places them in the global symbol table. It is equivalent to executing greg .set 5, imr .set 4, etc. The symbols are local and absolute. Using the .mmregs directive makes it unnecessary to define these symbols. The symbols are placed as shown in Table 5–2.

Table 5–2. Memory-Mapped Registers

	Address		
Name	**DEC**	**HEX**	**Description**
	0–3	0–3	Reserved
IMR	4	4	Interrupt mask register
GREG	5	5	Global memory allocation register
IFR	6	6	Interrupt flag register
PMST	7	7	Processor mode status register
RPTC	8	8	Repeat counter register
BRCR	9	9	Block repeat counter register
PASR	10	A	Block repeat program address start register
PAER	11	B	Block repeat program address end register
TREG0	12	C	Temporary register used for multiplicand
TREG1	13	D	Temporary register used for dynamic shift count
TREG2	14	E	Temporary register used as bit pointer in dynamic bit test
DBMR	15	F	Dynamic bit manipulation register
AR0	16	10	Auxiliary register 0
AR1	17	11	Auxiliary register 1
AR2	18	12	Auxiliary register 2
AR3	19	13	Auxiliary register 3
AR4	20	14	Auxiliary register 4
AR5	21	15	Auxiliary register 5

Table 5–2. Memory-Mapped Registers (Continued)

Name	Address DEC	HEX	Description
AR6	22	16	Auxiliary register 6
AR7	23	17	Auxiliary register 7
INDX	24	18	Index register
ARCR	25	19	Auxiliary register compare register
CBSR1	26	1A	Circular buffer 1 start register
CBER1	27	1B	Circular buffer 1 end register
CBSR2	28	1C	Circular buffer 2 start register
CBER2	29	1D	Circular buffer 2 end register
CBCR	30	1E	Circular buffer control register
BMAR	31	1F	Block move address register
DRR	32	20	Data receive register
DXR	33	21	Data transmit register
SPC	34	22	Serial port control register
	35	23	Reserved
TIM	36	24	Timer register
PRD	37	25	Period register
TCR	38	26	Timer control register
	39	27	Reserved
PDWSR	40	28	Program S/W wait-state register
IOWSR	41	29	I/O S/W wait-state register
CWSR	42	2A	S/W wait-state control register
	43–47	2B–2F	Reserved
TRCV	48	30	TDM data receive register
TDXR	49	31	TDM data transmit register
TSPC	50	32	TDM serial port control register

Table 5–2. Memory-Mapped Registers (Continued)

Name	Address		Description
	DEC	**HEX**	
TCSR	51	33	TDM channel select register
TRTA	52	34	Receive/transmit address register
TRAD	53	35	Received address register
	54–79	36–4F	Reserved

Syntax   *symbol*   **.set**   *value*

Description   The **.set** directive equates a constant value to a symbol. The symbol can then
                     be used in place of a value in assembly source. This allows you to equate
                     meaningful names with constants and other values.

                     ❑ *symbol* must appear in the label field.

                     ❑ *value* must be a well-defined expression; that is, all symbols in the expression
                        must be previously defined in the current source module.

Example   This example shows how symbols can be assigned with .set.

```
zero      .set      0
zero+1    .set      1          ;zero + 1 is a symbol
          LACL      zero       ;zero + 1 is replaced
          ADD       #zero+1    ;symbol
```

Result   LACL   0
                     ADD   1

Syntax   **.space**   *size in bits*

Description   The **.space** directive reserves *size* number of bits in the current section. The
                          SPC is incremented to point to the word following the reserved space.

                          When you use a label with the .space directive, it points to the first word reserved.

Example   This example shows how the .space directive reserves memory.

                          Source file:

```
*************************************
*   Begin assembling into .text
*************************************
                .text
*************************************
*   Reserve 15 words in .text
*************************************
                .space 0f0h
                .word 100h, 200h


*************************************
*   Begin assembling into .data
*************************************
                .data
                .string ".data"
*********************************************
*   Reserve 2  words in .data;
*   Res_1 points to the first reserved  word
*********************************************
                .space 020h
                .word  15
```

Listing file:

```
00001 ---- ----    *****************************************
00002 ---- ----    *   Begin assembling into .text
00003 ---- ----    *****************************************
00004 ---- 0000             .text
00005 ---- ----    *****************************************
00006 ---- ----    *   Reserve 15 words in .text
00007 ---- ----    *****************************************
00008 0a00 00f0             .space    0f0h
00009 0a0f 0100             .word     100h,200h
      0a10 0200
00010 ---- ----
00011 ---- ----    *****************************************
00012 ---- ----    *   Begin assembling into .data
00013 ---- ----    *****************************************
00014 ---- 0000             .data
00015 0e00 2e64             .string   ".data"
      0e01 6174
      0e02 6100
00016 ---- ----
00017 ---- ----
00018 ---- ----
00019 ---- ----    *****************************************
00020 ---- ----    *   Reserve 2 words in .data
00021 ---- ----    *   Res_1 points to the 1st reserved word
0022  ---- ----    *****************************************
00023 0e03 0020    Res_1 .space    020h
00024 0e05 000f             .word     15
00025 0e06 0e03             .word     Res_1
00026 ---- ----
00027 ---- ----
>>>> FINISHED READING ALL FILES
>>>> ASSEMBLY COMPLETE: ERRORS:0    WARNINGS:0
```

Syntax

        .text

        .ps [*address*]

Description

The **.text** and **.ps** directives tell the assembler to begin assembling into the .text or .ps sections (program memory), which usually contain executable code. The section program counter is set to a00h if nothing has yet been assembled into the .text or .ps sections. If code has already been assembled into the respective sections, the section program counter is restored to its previous value in the section.

address is an optional parameter for the .ps directive that specifies a 16-bit address. This address sets the initial value of the SPC. If no address is specified, a default value of a00h is used.

Note that the assembler assumes that .text is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you specify one of the other sections directives (.data, .ps, .ds, .entry).

Example   This example shows code assembled into the .text and .data sections. The .data section contains integer constants, and the .text section contains character strings.

```
00001 ---- ----    ********************************
00002 ---- ----    * Begin assembling into .data   *
00003 ---- ----    ********************************
00004 ---- fb00                    .ds     0fb00h
00005 fb00 0005                    .byte   5,6
      fb01 0006
00006 ---- ----    ********************************
00007 ---- ----    * Begin assembling into .ps     *
00008 ---- ----    ********************************
00009 ---- 7000                    .ps     7000h
00010 7000 0001                    .byte   1
00011 7001 0002                    .byte   2,3
      7002 0003
00012 ---- ----    ********************************
00013 ---- ----    * Begin assembling into .data   *
00014 ---- ----    ********************************
00015 ---- ----                    .data
00016 fb02 0007                    .byte   7,8
      fb03 0008
00017 ---- ----    ********************************
00018 ---- ----    * Begin assembling into .ps     *
00019 ---- ----    ********************************
00020 ---- ----                    .ps
00021 7003 0004                    .byte   4
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0   WARNINGS:0
```

Syntax	.**word** value$_1$ [, ... , value$_n$]
	.**int** value$_1$ [, ... , value$_n$]
Description	The **.int** and **.word** directives place one or more 16-bit values into consecutive words in the current section.
	value must be absolute. You can use as many values as fit on a single line (80 characters). If you use a label, it must point to the first initialized word.
Example 1	This example shows how to use the .word directive to initialize words. The symbol WordX points to the first reserved word.

```
00001 ---- fe00              .ds    0fe00h
00002 fe00 0c80  WordX:      .word  3200,0ffh,3
      fe01 00ff
      fe02 0003
00003 ---- ----
00004 ---- ----
```

Example 2   Here's an example of the .int directive.

```
00005 ---- ff00              .ds    0ff00h
00006 ff00 0000  LAB1        .int   0,-1,2,0ABCDh
      ff01 ffff
      ff02 0002
      ff03 abcd
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0   WARNINGS:0
```

Using the DSK Debugger

This chapter tells you how to invoke the DSK debugger and use its pulldown menus.

6.1 Invoking the Debugger

Here's the basic format for the command that invokes the debugger:

dsk5d [*options*]

dsk5d   is the command that invokes the debugger.

options   supply the debugger with additional information.

Table 6–1 lists the debugger options; the following subsections describe the options.

Table 6–1. Summary of Debugger Options

Option	Description
? or H	Displays a listing of the available options
b*rate*	Selects the valid baud rate
com# or c#	Selects serial communication port 1, 2, 3, or 4
e*address*	Defines a program entry point
i	Selects a logic level for DTR (data terminal ready) reset; note that the default DTR is inverse
l	Selects the EGA/VGA screen sizes
m [0,1]	Sets the configuration of CNF bit (default = 0)
s	Selects the default screen length of 25

Displaying a list of available options (? or H option)

You can display the contents of Table 6–1 on your screen by using the ? or H option. For example, enter:

dsk5d ? ⏎

Selecting the baud (b option)

The valid baud settings are:

❑ b4800
❑ b9600
❑ b19200
❑ b38400
❑ b57600

Identifying the serial port (com# or c# option)

The c1, c2, c3, or c4 option identifies the serial port that the debugger uses for communicating with your PC. The default setting, c1, is used when your serial port is connected to serial communication port 1 (com1). Depending on your serial port connection, replace serial port with one of these values:

❏ If you are using com1, enter:

    dsk5d c1 ⏎

❏ If you are using com2, enter:

    dsk5d c2 ⏎

❏ If you are using com3, enter:

    dsk5d c3 ⏎

❏ If you are using com4, enter:

    dsk5d c4 ⏎

Defining an entry point (e option)

Use option e to set the initial program entry address. The address you select must be a four-digit hexadecimal value. For example:

dsk5d ea00h

The above example sets the DSK debugger at an initial address of 0a00h.

Selecting a data terminal ready (DTR) logic level (i option)

Using option i tells the dsk5d to invert DTR as a reset signal. Usually, the RS-232 DTR line is high and pulses low for a reset signal. However, if you use the i option (inverse), the DTR line is low and pulses high for a reset signal.

Selecting the screen size (l and s options)

By default, the debugger uses an 80-character-by-25-line screen. You can use one of the options in Table 6–2 to switch between screen sizes.

Table 6–2. Screen Size Options

Option	Description
l	80 characters by 43 lines
s	80 characters by 25 lines (default)

Setting the configuration mode for memory (m option)

Use the m option to configure memory sections in the same way the SETC/ CLRC instruction works. Refer to your *TMS320C5x User's Guide* for more information on the SETC instruction.

6.2 Using Pulldown Menus in the Debugger

Figure 6–1 shows the main menu bar in the DSK debugger.

Figure 6–1. The Main Menu Bar

Display **F**ill **L**oad **H**elp e**X**ec **Q**uit **M**odify **B**reak **I**nit **W**atch **R**eset **S**ave **C**opy **P**C

Many of the debugger's pulldown menus have additional submenus. A submenu is indicated by a main menu selection enclosed in < > characters. For example, here's the Display submenu and Format, which is a submenu of Display:

display submenu →   <display> **D**ata **P**rogram **V**ersion **S**tatus **B**reakpoints **F**ormat **M**emory

format submenu → <format> **U**nsgnd **I**nt **C**har **P**ckdstrng **L**ong **D**ouble **Q**15 **O**ct he**X** **B**-**E**-**T** float
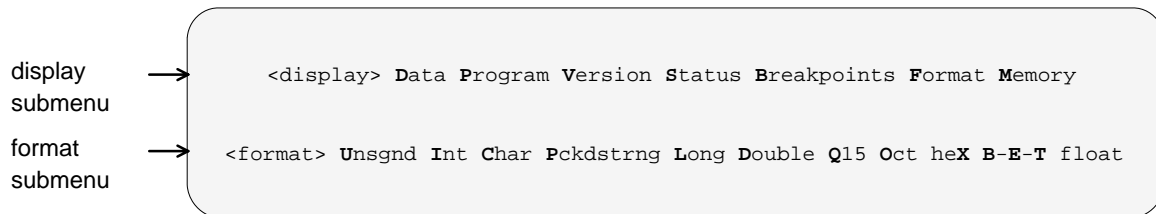
Because the DSK debugger supports over 50 commands, it's not practical to discuss the commands associated with all of the submenu choices. Here's a tip to help you with the DSK commands: the highlighted letters show the key to press for the corresponding debugger command. For example, the highlighted letters in **D**isplay→ **F**ormat→ **C**har show that you press Ⓓ, Ⓕ, Ⓒ , in that order, to display the submenus.

Escaping from the pulldown menus and submenus

If you display a submenu and then decide that you don't want to make a selection press ⌈ ESC ⌉ to return to the main menu bar.

Using the Display submenu

Table 6–3 lists the submenu selections for Display submenu. The highlighted letters show the keys that you can use to select choices.

Table 6–3. Submenu Selections for Displaying Information

To display this . . .	Select $\rightarrow$ **D**isplay
Data memory	**D**ata
Program memory	**P**rogram
Current version of the debugger	**V**ersion
Register status	**S**tatus
List of set breakpoints	**B**reakpoints
Format	**F**ormat
Unsigned integer	**U**nsgnd
Integer	**I**nt
Character	**C**har
String	pckd**S**trng
Long	**L**ong
Floating-point number	**F**lt
Double	**D**ouble
Signed Q15	**Q**15
Octal	**O**ct
Hexadecimal	he**X**
Big floating-point number (exponent = 16; mantissa = 32/Q30)	**B**-E-T float
Short floating-point number (exponent = 16; mantissa = 16/Q14)	B-**E-**T float
Long floating-point number (exponent = 32; mantissa = 64/Q62)	B-E-**T** float
Memory	**M**emory
Set a new address	**A**ddress
Big (exponent = 16;mantissa = 32/Q30)	**B**-E-T float
Double	**D**ouble
Short floating-point number	B-**E**-T float
Long floating-point number	B-E-**T** float
Floating-point number	**F**loat
Integer	**I**nt
Long	**L**ong
Octal	**O**ct
Q15	**Q**15
Unsigned integer	**U**nsgnd
Hexidecimal	he**X**

Using the Fill submenu

Table 6–4 lists the selections for filling memory. The highlighted letters show the keys you can use to select choices.

Table 6–4. Submenu Selections for Filling Memory

To fill this . . .	Select → **F**ill
Data memory	**D**ata
Program memory	**P**rogram

Using the Load submenu

Table 6–5 lists the selections for the Load submenu. The highlighted letters show the keys you can use to select choices.

Table 6–5. Submenu Selections for Loading Information into Memory

To load this . . .	Select → **L**oad
COFF file	**C**OFF
DSK file	**D**SK
List of set breakpoints	**B**reakpoints
Format	**F**ormat
Unsigned integer	**U**nsgnd
Integer	**I**nt
Character	**C**har
String	pckd**S**trng
Long	**L**ong
Floating-point number	**F**lt
Double	**D**ouble
Signed Q15	**Q**15
Octal	**O**ct
Hexadecimal	he**X**
Big floating-point number (exponent = 16; mantissa = 32/Q30)	**B**-E-T float
Short floating-point number (exponent =16; mantissa = 16/Q14)	B-**E-**T float
Long floating-point number (exponent = 32; mantissa = 64/Q62)	B-E-**T** float
Program counter	**P**rogramcounter

Using the Help submenu

You can press F1 or H to bring up the Help Window Display shown in Figure 6–2. Choose from the menu selections listed below to find additional information.

Figure 6–2. The Monitor Information Screen

```
                          Help Window Display

 Usage of the program: dskd [[-|\][options]
 Options: Please note that options are not case significant
 --------
 ? or H : this display
 Bxxxxx : xxxxx selects the baudrates! Work as today:
                4800.9600.19200[default].38400.57600
 COMx   : comport: x=1 [default] or x=2, 3 and 4, and
          optional 'C1,' 'C2,' 'C3' or 'C4' can be used
 Exxxx  : defines with 4 following digits the entry point
                format e|E[0z]xxxx[h]
 I      : selects logic level for DIR->Reset (default->
                inverse)
 L      : selects the EGA/VGA scree length (43 or 50)
 S      : selects the default screen length (25)

 Function Key definitions
 ========================
 F1    Help Information
```

```
     PGUP  PGDN  HOME  END  newPage  File  Quit  ESC          01/17
```

To move through the Help Window Display, you can use the following submenu selections:

- ❏ PGUP to move ahead a page
- ❏ PGDN to move back a page
- ❏ HOME to return to the first page of the help menu
- ❏ END to go to the last page of the help menu
- ❏ newPage to go to a specific page number in the help menu
- ❏ File to print the file help.txt
- ❏ Quit to exit the help menu and return to the debugger
- ❏ ESC to exit the help menu and return to the debugger

Using the eXec submenu

Table 6–6 lists the selections for executing code. The highlighted letters show the keys that you can use to select choices.

Table 6–6. Submenu Selections for Executing Code

To execute code from . . .	Select → eXec
The beginning of your program	**G**o
A particular address	**A**ddress
One line of code to the next	**S**inglestep/ret/blank
One line number to the next	**N**um_steps
The beginning of a certain function	**F**unction
The beginning of your program with break point disabled (free run)	**R**un

Using the Quit submenu

To exit the debugger and return to the operating system, enter this command:

q ⏎

If a program is running or a submenu is displayed, press ESC before you quit the debugger to halt program execution or return to the main menu.

Using the Modify submenu

Table 6–7 lists the selections for modifying your code. The highlighted letters show the keys you can use to select choices.

Table 6–7. Submenu Selections for Modifying Code

To modify . . .	Select → Modify
A register	**R**egister
Your program	**P**rogram
Data	**D**ata
An in or out port	**I**n/out ports

Using the Break submenu

Table 6–8 lists the selections for setting software breakpoints in your program. The highlighted letters show the keys you can use to select choices.

Table 6–8. Submenu Selections for Handling Breakpoints

To perform the following . . .	Select → **B**reak
Set a breakpoint for a specific address	ba
Set a breakpoint for an unknown address	be
Clear a breakpoint	bd
Find breakpoints	bl

Refer to Section 6.4, Using Software Breakpoints for more information.

Using the Init submenu

Using the Init submenu initializes the CPU registers and entry point of your program.

Using the Watch submenu

Table 6–9 lists the selections for watching your code during program execution. The highlighted letters show the keys you can use to select choices.

Table 6–9. Submenu Selections for Watching Data

To change your watch settings . . .	Select → **W**atch
Add a variable/value to watch	**A**dd
Delete a variable/value to watch	**D**elete
Format the variables/values you are watching	**F**ormat
Modify the variables/values you are watching	**M**odify

Using the Reset submenu

To reset the DSK board, enter this command:

r ⏎

If a submenu is displayed, press ⌷ESC⌷ to return to the main menu before you reset the board.

Using the Save submenu

Table 6–10 lists the menu selections for saving code during a debugging session. The highlighted letters show the keys you can use to select choices.

Table 6–10. Submenu Selections for Saving Code

To save . . .	Select → **S**ave
A register value	**R**egister
Data	**D**ata
Your program	**P**rogram
A certain format	**F**ormat

Using the Copy submenu

Table 6–11 lists the menu selections for copying information. The highlighted letters show the keys you can use to select choices.

Table 6–11. Submenu Selections for Copying Information

To copy from . . . to . . .	Select → **C**opy
Data to data	**D**ata to data
One program to another program	**P**rogram to program
Data to your program	d**A**ta to program
Your program to data	p**R**ogram to data

Using the Op-sys submenu

The debugger provides a simple method for entering DOS commands without explicitly exiting the debugger environment. To do this, use the Op-sys submenu. Op-sys is not displayed in the main menu bar, but you may use it by entering this command:

o ⏎

If a submenu is displayed, press ⎡ESC⎤ to return to the main menu before attempting to enter the operating system (op-sys).

The debugger opens a system shell and displays the DOS prompt. At this point, enter any DOS command.

When you are finished entering commands and are ready to return to the debugger environment, enter:

exit ⏎

Note:

Available memory may limit the Op-sys commands that you can enter from a system shell. For example, you cannot invoke another version of the debugger.

6.3 Using Dialog Boxes

Some of the debugger commands have parameters. When you execute these commands from pulldown submenus, you must have some way of providing parameter information. The debugger allows you to do this by displaying a *dialog box* that asks for this information.

Entering text in a dialog box is much like entering commands in the operating system. For example, when you select Program from the Fill submenu, the debugger displays a dialog box that asks you for parameter information. The dialog box looks like this:

```
                        Fill Program Memory

Fill Program Memory

Start Address:
```
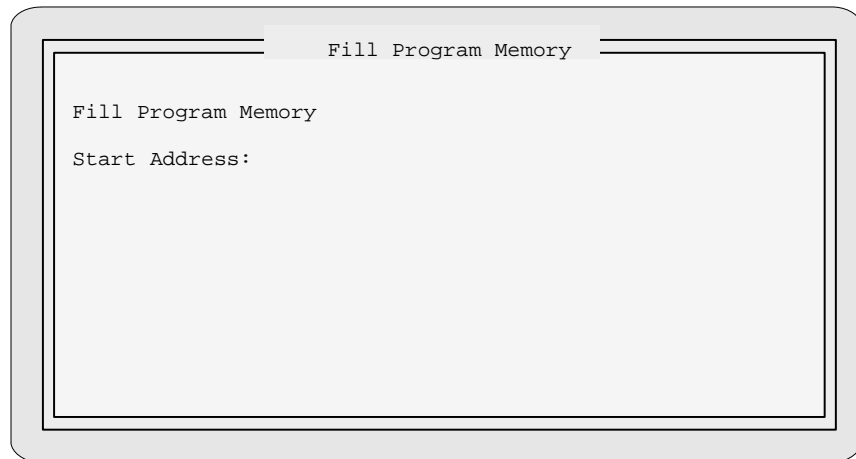
To enter a *start address*, simply type it in and press ⏎. The next parameter appears in the dialog box:

```
                        Fill Program Memory

Fill Program Memory

Start Address: 0500h

Length:
```
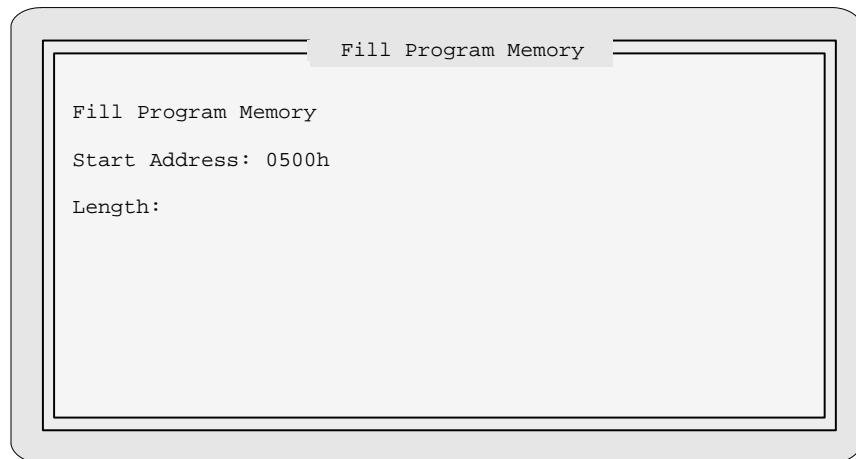
You can omit entries for optional parameters by pressing ⌨, but the debugger won't allow you to skip required parameters. When you have entered all appropriate parameter values, *Fill Program Memory finished* appears at the bottom of the dialog box.

In the case of the Modify menu when you select Data from its submenu, an empty dialog box appears on the screen:

```
                   Modify Data Memory of 320c50

   Startadd
```
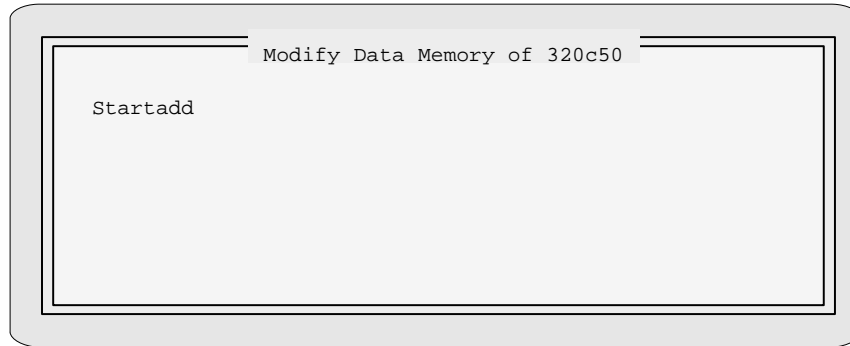
Press ⌨ for the debugger to display the first parameter:

```
                   Modify Data Memory of 320c50

   Startadd      1000
   Address: 1000 > efff   new:
```
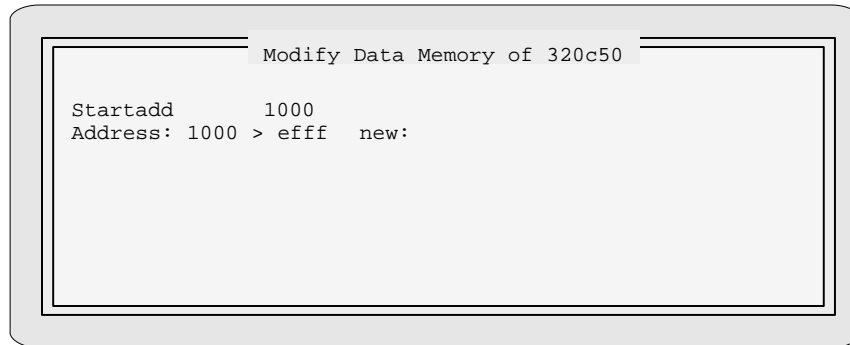
Enter the address you want to modify and press ⌨. The next parameter appears in the dialog box.

```
                   Modify Data Memory of 320c50

   Startadd       1000

   Address: 1003 > ffff   new  q
   Address: 0002 > 0003
   Address: 1001 > 0002
   Address: 1000 > 0001
```
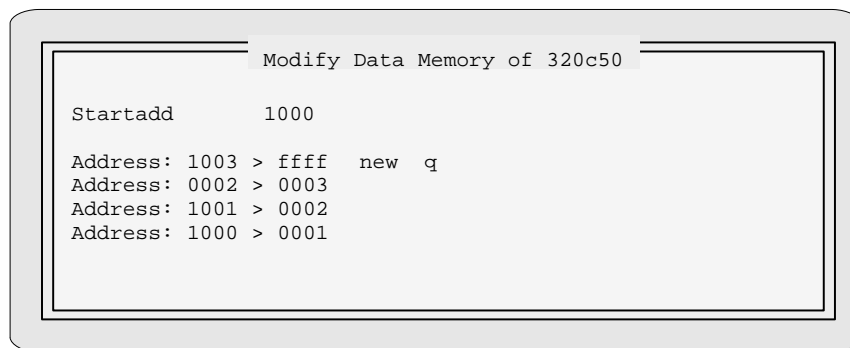
Closing a dialog box

When you've entered a value for the final parameter, there are three ways to exit from the dialog box:

❑ Press ⎣ESC⎦

❑ Press ⮐

❑ Press Ⓠ and ⮐

Performing the last of these options at the prompt causes the debugger to close the dialog box and execute the command with the parameter values you supplied.

6.4 Using Software Breakpoints

This section describes the processes for setting and clearing software break-points and for obtaining a listing of all the breakpoints that are set.

During the debugging process, you may want to halt execution temporarily so that you can examine the contents of selected variables, registers, and memory locations before continuing with program execution. You can do this by setting software breakpoints in assembly language code. A software break-point halts any program execution, whether you're running or single-stepping through code.

Setting a software breakpoint
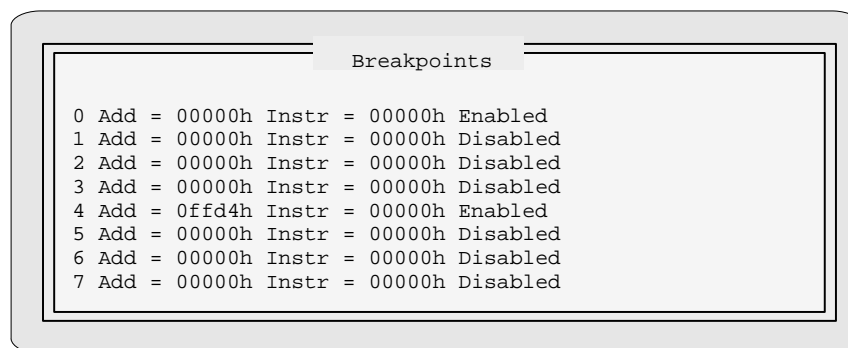
When you set a software breakpoint, the debugger highlights the breakpointed line in a bolder or brighter font. The highlighted statement appears in the reverse assembly window.

After execution is halted by a breakpoint, you can continue program execution by reissuing any of the run or single-step commands.

You can set a software breakpoint by entering either the ba or the be command.

ba If you know the address where you'd like to set a software breakpoint, you can use ba. This command is useful because it doesn't require you to search through code to find the desired line. When you enter ba , the debugger asks you to enter an absolute address. Once you have entered the address, you are asked to choose the line number where you want the breakpoint set. Figure 6–3 shows a breakpoint set at address ffd4 on line number 4. Note that you cannot set more than one breakpoint at the same statement.

Figure 6–3. Setting a Software Breakpoint

```
                      Breakpoints

0 Add = 00000h Instr = 00000h Enabled
1 Add = 00000h Instr = 00000h Disabled
2 Add = 00000h Instr = 00000h Disabled
3 Add = 00000h Instr = 00000h Disabled
4 Add = 0ffd4h Instr = 00000h Enabled
5 Add = 00000h Instr = 00000h Disabled
6 Add = 00000h Instr = 00000h Disabled
7 Add = 00000h Instr = 00000h Disabled
```

be If you don't know a specific address, you can enter the be (breakpoint enable/ disable) command. The debugger displays a list of addresses as shown in Figure 6–3, and asks you what line number you want to set a breakpoint on.

Clearing a software breakpoint

bd   If you'd like to clear a breakpoint, use the bd command. When you enter bd, the Breakpoints box appears on the screen (see Figure 6–3). The debugger then asks which line number contains the breakpoint you want to delete. When you enter the line number, the breakpoint is disabled.

Finding the software breakpoints that are set

bl   Sometimes you may need to know where software breakpoints are set. The bl command provides an easy way to get a complete listing of all the software breakpoints that are currently set in your program. The bl command displays the Breakpoints box shown in Figure 6–3.

6.5 Quick-Reference Guide

Table 6-12, Table 6-13, and Table 6-14 provide a quick-reference guide to the debugger function key definitions, floating-point formats, and register definitions.

Table 6–12. Debugger Function Key Definitions

Function Key	Description
F1	Displays help information
F2	Prints the contents of the screen to a file named *screen.srn*
F3	Displays the directory
F4	Not used
F5	Executes your program to the next breakpoint
F6	Not used
F7	Not used
F8	Single-steps your program
F9	Not used
F10	Single-steps your program and steps past calls
F11 or SPACE F1	Displays the reverse assembly window
F12 or SPACE F2	Turns the trace on or off (this key acts as a toggle switch)

Table 6–13. Debugger Floating-Point Formats

Floating-Point Format	Description
.float	32-bit IEEE standardized floating-point format
.double	64-bit IEEE standardized double floating-point format
.bfloat	16-bit exponent + 32-bit mantissa (exponent is 2s complement / mantissa = Q30)
.tfloat	32-bit exponent + 64-bit mantissa (exponent is 2s complement / mantissa = Q62)
.efloat	16-bit exponent + 16-bit mantissa (exponent is 2s complement / mantissa = Q14)

Table 6–14. Debugger Register Definitions

Register	Definition	Description
ACCU	Accumulator	32 bits with a carry in ST1
ACCB	Accumulator buffer	32 bits to temporarily store ACCU
PREG	Product register	32 bits used for 16 x 16 bit multiplication
TRG0	Temporary multiplicand	16 bits for multiplication and special instructions
TRG1	Temporary register 1	5 bits for dynamic shift
TRG2	Temporary register 2	4 bits for bit pointer in bit test
ARi	Auxiliary register	16 bits with i = 0, 7 used as a counter and pointer
ST0	Status register 0	16 bits
ST1	Status register 1	16 bits
PMST	Status register	16 bits
SPC	Serial port control register	16 bits
STCKi	Stack register	16 bits with i = 0,7 used for hardware stacking[†]
DRR	Data receive register at address 32	16 bits for the serial port
DXR	Data transmit register at address 33	16 bits for the serial port
TIM	Timer register at address 36	16 bits
PRD	Period register at address 37	16 bits
IMR	Interrupt mask register at address 4	9 bits for masking 9 interrupts
GREG	Global register at address 5	8 bits to define data memory as global
ARP (ST0)	Auxiliary register pointer	3 bits
OV (ST0)	Overflow flag	1 bit
OVM (ST0)	Overflow mode	1 bit
INTM (ST0)	Interrupt mode (enable global interrupt)	1 bit
DP (ST0)	Data page pointer	9 bits
ARB (ST1)	Auxiliary register pointer buffer	3 bits
CNF (ST1)	DARAM program/data configuration	1 bit
TC (ST1)	Test/control flag	1 bit
SXM (ST1)	Sign-extension mode enable	1 bit
C (ST1)	Carry bit	1 bit
HM (ST1)	Hold mode selection	1 bit

[†] The debugger uses one stack level for itself.

Table 6-14.Debugger Register Definitions (Continued)

Register	Definition	Description
PM (ST1)	PREG to ACCU shift mode	2 bits
IPTR (PMST)	Interrupt vector pointer	5 bits
OVLY (PMST)	Data SARAM enable	1 bit
RAM (PMST)	Program SARAM enable	1 bit
MP/MC (PMST)	Micro processor/computer mode	1 bit
TXM (SPC)	FSX mode bit	1 bit
MCM (SPC)	CLKX clock mode (internal/external)	1 bit
FSM (SPC)	Frame synch mode (burst/continuous)	1 bit
FO (SPC)	Format bit (8/16 bit mode)	1 bit

DSP Starter Kit (DSK) Circuit Board Dimensions and Schematic Diagrams

This appendix contains the circuit board dimensions and the schematic diagrams for the TMS320C5x DSK.

Figure A−1. TMS320C5x DSK Circuit Board Dimensions
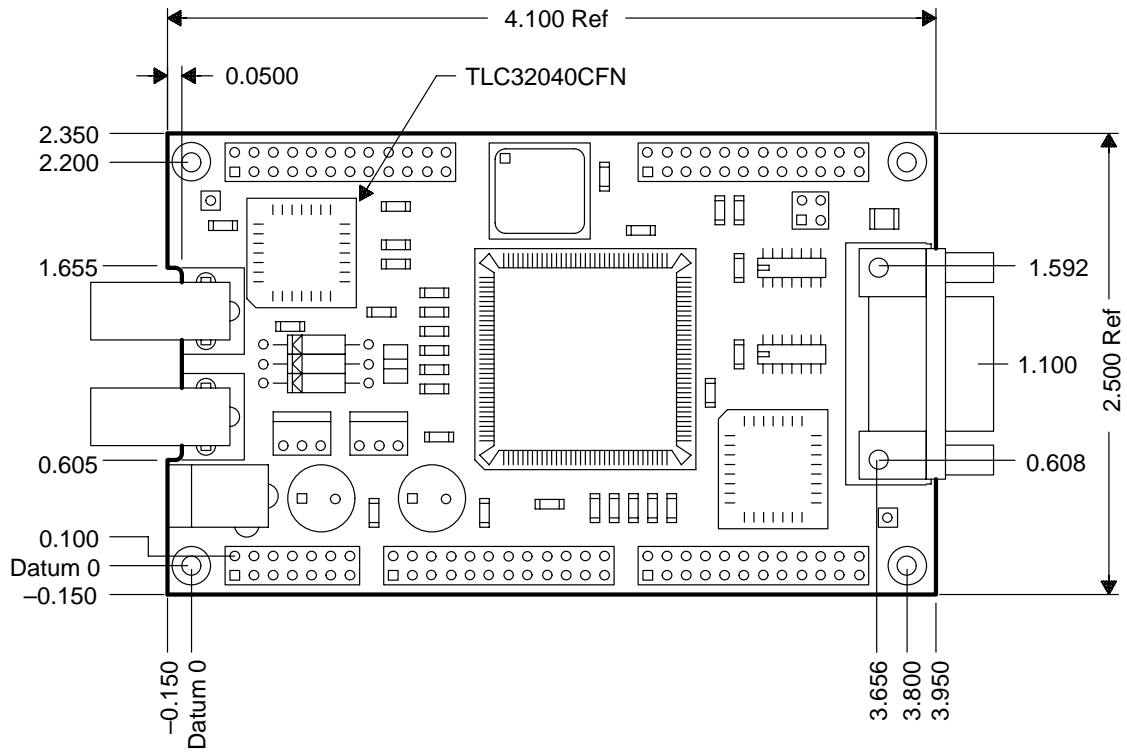
Note: Dimensions are in inches.

VCC

R13 4.7K R12 4.7K R11 4.7K R10 4.7K R9 4.7K

VCC

U4

TMS320C50_QFP

INT1\ 38 INT1
INT2\ 39 INT2
INT3\ 40 INT3
INT4\ 41 INT4
NMI\ 42 NMI
IACK\ 112 IACK
RS\ 127 RS
MP/MC\ 5 MP/MC−
BIO\ 130 BIO
XF 109 XF
TOUT 122 TOUT
CLKR 46 CLKR
CLKX 124 CLKX
FSR 45 FSR
FSX 104 FSX
DR 43 DR
DX 106 DX
TCLKR 126 TCLKR
TCLKX 123 TCLKX
TFSR 125 TFSR/TADD
TFSX 105 TFSX/TFRM
TDR 44 TDR
TDX 107 TDX
97 X1
96 X2/CLKIN
CLKIN2 95 CLKIN2
CLKMD1 71 CLKMD1
CLKMD2 103 CLKMD2
CLKOUT 110 CLKOUT1

A0 55 A0
A1 56 A1
A2 57 A2
A3 58 A3
A4 59 A4
A5 60 A5
A6 61 A6
A7 62 A7
A8 63 A8
A9 64 A9
A10 72 A10
A11 73 A11
A12 74 A12
A13 75 A13
A14 76 A14
A15 77 A15
IS 90 IS\
PS 91 PS\
DS 89 DS\
STRB 93 STRB\
R/W 92 R/W\
RD 82 RD\
WE 83 WE\
READY 128 READY
HOLDA 108 HOLDA\
HOLD 129 HOLD\
BR 94 BR\
IAQ 1 IAQ\
MSC 111
TRST 2 TRST\
TMS 31 TMS
TDI 67 TDI
TCK 34 TCK
TDO 100 TDO
EMU0 118 EMU0
EMU1/OFF 119 EMU1

D[0..15] A[0..15]

R3 4.7K R14 4.7K R15 4.7K

U5 CLK NC 3 1 40MHz

GND

TEXAS INSTRUMENTS
Title TMS320C5x DSP STARTERS KIT
Size A Document Number 2617687 REV *
Date: January 21, 1994 Sheet 2 of 5

VCC

R6
4.7K

9

1
2

GND

10   U7C   8   13   U7D   11   RS\
      75189   75189

DTR   DTR

2

R8
0 OHM

1   U7A   3   INT2\
     75189   BIO\

P1

5
9
4
8
3
7
2
6
1

TX

TR

U8A

3   2   TX
              XF
     75188

TR

CONNECTOR DB9

AIN+

AUXIN+
AUXIN−
OUT+
OUT−

U3

26   IN+   RESET   2   BR\
25   IN−   EODR   3   EODR\
24   AUXIN+   FSR   4   FSR
23   AUXIN−   DR   5   DR
22   OUT+   EODX   11   EODX\
21   OUT−   DX   12   DX
8   REF   FSX   14   FSX
20   VCC+   MCLK   6   TOUT
19   VCC−   SCLK   10   CLKX
18   AGND   BYTE   13   CLKR
17   AGND   VDD   7
1   NU   DGND   9

R16
1K

R17
1K

VCC

VCC−

R7
0 OHM

TLC32040_PLCC

GND   VCC

2   1   J1
          RCA
     3

2   1   J2
          RCA
     3

GND

TEXAS INSTRUMENTS		
Title		
TMS320C5x DSP STARTERS KIT		
Size	Document Number	REV
A	2617687	A
Date: January 17, 1994	Sheet 3 of 5	

D[0..15]

A[0..15]

D[0..15]

VCC

U6

A0	11	A0	O0	13	D0
A1	10	A1	O1	14	D1
A2	9	A2	O2	15	D2
A3	8	A3	O3	18	D3
A4	7	A4	O4	19	D4
A5	6	A5	O5	20	D5
A6	5	A6	O6	21	D6
A7	4	A7	O7	22	D7
A8	29	A8			
A9	28	A9			
A10	24	A10			
A11	27	A11			
A12	3	A12			
A13	30	A13			
A14	31	A14			

R5
4.7K

R4
4.7k

JP2

	1	2	
READY	3	4	HOLD\
BIO\	5	6	MP/MC\
GND	7	8	GND
D15	9	10	D14
D13	11	12	D12
D11	13	14	D10
D9	15	16	D8
D7	17	18	D6
D5	19	20	D4
D3	21	22	D2
D1	23	24	D0

HEADER 12X2

	23	CE	NU	
BR\	25	OE	NU	
RD\	2	VPP	NC	
			NC	

27PC256FM

VCC

A[0..15]

JP3

	1	2	
INT1\	3	4	INT2\
INT3\	5	6	INT4\
NMI\	7	8	GND
GND	9	10	GND
A0	11	12	A1
A2	13	14	A3
A4	15	16	A5
A6	17	18	A7
A8	19	20	A9
A10	21	22	A11
A12	23	24	A13
A14			A15

HEADER 12X2

VCC

R1
4.7K

R2
4.7K

JP1

	1	2	
TMS	3	4	TRST\
TDI	5	6	PIN 6 (KEY)
TDO	7	8	
	9	10	
TCK	11	12	
EMU0	13	14	EMU1

XDS510 HEADER

GND

TEXAS INSTRUMENTS

Title

TMS320C5x DSP STARTERS KIT

Size	Document Number				REV
A	2617687				*
Date:	January 21, 1994	Sheet	4	of	5

TEXAS INSTRUMENTS

Title				
TMS320C5x DSP STARTERS KIT				
Size	Document Number			REV
A	2617687			*
Date:	January 21, 1994	Sheet	5 of 5	

Glossary

A

absolute address: An address that is permanently assigned to a memory location.

A/D: Analog-to-digital. Conversion of continuously variable electrical signals to discrete or discontinuous electrical signals.

AIC: Analog interface circuit. Integrated circuit that performs serial A/D and D/A conversions.

assembler: A software program that creates a machine-language program from a source file that contains assembly language instructions, directives, and macro directives. The assembler substitutes absolute operation codes for symbolic operation codes, and absolute or relocatable addresses for symbolic addresses.

assignment statement: A statement that assigns a value to a variable.

autoexec.bat: A batch file that contains DOS commands for initializing your PC.

B

batch file: A file containing an accumulation of data to be processed. This data may be either DOS commands for the PC to execute or debugger commands for the debugger to execute.

BBS: Bulletin board service. Computer program which may be accessed by remote users, allowing them to post questions and view responses.

block: A set of declarations and statements grouped together in braces and treated as an entity.

breakpoint: A place in a computer program, usually specified by an instruction, where its execution may be interrupted by external intervention.

byte: A sequence of eight adjacent bits operated upon as a unit.

C

code-display windows: Windows that show code, text files, or code-specific information.

COFF: Common object file format. A system of object files configured according to a standard developed by AT&T. These files are relocatable in memory space.

command: A character string you provide to a system, such as an assembler, that represents a request for system action.

command file: A file created by the user which names initialization options and input files for the linker or the debugger.

command line: The portion of the COMMAND window where you can enter instructions to the system.

command-line cursor: An on-screen marker that identifies the current character position on the command line.

comment: A source statement (or portion of a source statement) that is used to document or improve readability of a source file. Comments are not assembled.

constant: A fixed or invariable value or data item.

cross-reference listing: An output file created by the assembler that lists the symbols that were defined, the line they were defined on, the lines that referenced them, and their final values.

cursor: An on-screen marker that identifies the current character position.

D

D/A: Digital-to-analog. Conversion of discrete or discontinuous electrical signals to continuously variable signals.

DARAM: Dual-access, random-access memory. Memory that can be altered twice during each cycle.

D_DIR: An environment variable that identifies the directory containing the commands and files necessary for running the debugger.

debugger: A software interface that permits the user to identify and eliminate mistakes in a program.

directive: Special-purpose commands that control the actions and functions of a software tool (as opposed to assembly language instructions, which control the actions of a device).

disassembly: The process of translating the contents of memory from machine language to assembly language. Also known as reverse assembly.

DSK: Digital signal processor starter kit. Tools and documentation provided to new DSP users to enable rapid use of the product.

DSP: Digital signal processor. DSPs process or manipulate digital signals, which are discrete or discontinuous electrical impulses.

DTR: Data terminal ready. A signal defined by the RS-232 standard that allows a data source, such as a computer or terminal, to indicate that it is ready for transmission.

E

EGA: Enhanced graphics array. An industry-standard video card.

entry point: A point in target memory where the program begins execution.

expression: One or more operations in assembler programming represented by a combination of symbols, constants, and paired parentheses separated by arithmetic operators.

external symbol: A symbol that is used in the current program module but defined in another program module.

F

field: A software-configurable data type which can be programmed to be from one to eight bits long.

file header: A portion of the COFF object file that contains general information about the object file, such as the number of section headers, the type of system the object file can be downloaded to, the number of symbols in the symbol table, and the symbol table's starting address.

G

global symbol: A symbol that is either defined in the current module and accessed in another or accessed in the current module but defined in another.

I

IC: Integrated circuit. A tiny wafer of substitute material upon which is etched or imprinted a complex of electronic components and their interconnections.

input section: A section from an object file that is linked into an executable module.

L

label: A symbol that begins in column 1 of a source statement and corresponds to the address of that statement.

listing file: An output file created by the assembler that lists source statements, their line numbers, and any unresolved symbols or opcodes.

LSB: Least significant bit. The binary digit, or bit, in a binary number that has the least influence on the value of the number.

LSByte: Least significant byte. The byte in a multibyte word that has the least influence on the value of the word.

M

member: An element of a structure, union, or enumeration.

memory map: A map of target system memory space that is partitioned into functional blocks.

menu bar: A row of pulldown menu selections at the top of the debugger display.

mnemonic: An instruction name that the assembler translates into machine code.

MSB: Most significant bit. The binary digit, or bit, in a binary number that has the most influence on the value of the number.

MSByte: Most significant byte. The byte in a multibyte word that has the most influence on the value of the word.

N

named section: Either an initialized section that is defined with a .sect directive, or an uninitialized section that is defined with a .usect directive.

O

object file: A set of related records treated as a unit that is the output of an assembler or compiler and is input to a linker.

operand: The arguments or parameters of an assembly language instruction, assembler directive, or macro directive.

options: Command parameters that allow you to request additional or specific functions when you invoke a software tool.

P

PC: Personal computer or program counter, depending on context and where it's used. In this book, in installation instructions, or in information relating to hardware and boards, PC means personal computer (as in IBM PC). In general debugger and program-related information, PC means program counter, which is the register that identifies the current statement in your program.

PROM: Programmable read-only memory. An integrated circuit on which information can be programmed by the user. This circuit can be read from but not written to.

pulldown menu: A command menu that is accessed by name from the menu bar at the top of the debugger display.

R

raw data: Executable code or initialized data in an output section.

reverse assembly: The process of translating the contents of memory from machine language to assembly language. Also known as disassembly.

S

SARAM: Single-access, random-access memory. Memory that can be altered only once during each cycle.

section: A relocatable block of code or data that ultimately occupies a space adjacent to other blocks of code in the memory map.

serial port: An access point that the debugger uses to sequentially transmit and receive data to and from the emulator or the applications board. The port address represents the communication port to which the debugger is attached.

single step: A form of program execution in which the program is executed statement by statement. The debugger pauses after each statement to update the data-display window.

source file: A file that contains C code or assembly language code that will assembled to form a temporary object file.

SPC: Section program counter. A specific register that holds the address of the section where the following directive is to be obtained.

static variable: A variable that is allocated before execution of a program begins and remains allocated for the duration of the program.

string table: A table that stores symbol names that are longer than eight characters. Symbol names of eight characters or longer cannot be stored in the symbol table; instead, they are stored in the string table. The name portion of the symbol's entry points to the location of the string in the string table.

structure: A collection of one or more variables grouped together under a single name.

symbol: A string of alphanumeric characters that represents an address or a value.

T

tag: An optional type name that can be assigned to a structure, union, or enumeration.

U

unconfigured memory: Memory that is not defined as part of the memory map and cannot be loaded with code or data.

unsigned value: A value that is treated as a positive number, regardless of its actual sign.

V

VGA: Video graphics array. An industry-standard video card.

W

word: A character or bit string considered as an entity.

Index

? debugger option 6-2

A

absolute address, definition B-1
AC transformer, power requirements 2-2
asm assembler option 4-11
assembler 3-4, 4-1 to 4-12
 –l option 3-5
 constants 4-7
 definition B-1
 description of 3-2
 directives 5-1 to 5-11
 executable file 2-3
 expressions 4-9
 key features 3-2
 options 4-10
 source, listings 4-2
 source statement format 4-2
 symbols 4-8
assembler directives. *See* directives
assembling your program 4-10
assembly-time constants E-28
assigning a value to a symbol E-28
assignment statement, definition B-1
autoexec.bat 2-8
autoexec.bat file, definition B-1

B

b debugger option 6-2
ba command 6-16
backup, of product disk 2-6
batch files
 config.sys 2-7
 definition B-1
 invoking, autoexec.bat 2-8

baud rate 6-2
 error 2-10
bd command 6-17
.bfloat 6-18
 assembler directive 5-8, E-18
binary integers 4-7
bl command 6-17
block, definition B-1
breakpoints. *See* software breakpoints
bulletin board, updating DSK software, vii
.byte, assembler directive 5-8, E-12
byte, definition B-1

C

c or com debugger option 6-3
c1/c2/c3/c4
 debugger option 2-9
 error 2-10
cable, requirements 2-2, 2-4
character, constants 4-7
CLRC, clear 6-4
code-display windows, definition B-2
code, developing 3-4
com1/com2/com3/com4
 debugger option 2-9
 error 2-10
command file, definition B-2
command line
 defining assembler statements 4-11
 definition B-2
command-line cursor, definition B-2
comments 4-6 to 4-7
 definition B-2
communication, link between PC and DSK 2-2
communication port, error 2-10

Q

R

S

T

tag, definition   B-6

temporary object file, creating   4-10

.text
   assembler directive   5-4, E-31
   section   5-4

.tfloat   6-18
   assembler directive   5-8, E-18

trace, turning on/off   6-18

transformer, power requirements   2-2

U

unconfigured memory, definition   B-6
unsigned, definition   B-6

V

VGA, definition   B-6

W

warranty information   2-2
Watch submenu   6-10
.word, assembler directive   5-8, E-33
word, definition   B-6