TMS320C2x DSP Starter Kit User's Guide

Literature Number: SPRU093
Manufacturing Part Number: 2617630-9741 revision *
March 1993

TEXAS INSTRUMENTS

Printed on Recycled Paper

IMPORTANT NOTICE

Texas Instruments Incorporated (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 current specifications 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.

Please be aware that TI products are not intended for use in life-support appliances, devices, or systems. Use of TI product in such applications requires the written approval of the appropriate TI officer. Certain applications using semiconductor devices may involve potential risks of personal injury, property damage, or loss of life. In order to minimize these risks, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards. Inclusion of TI products in such applications is understood to be fully at the risk of the customer using TI devices or systems.

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.

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.

Read This First

What is This Book About?

This book tells you how to use the DSP (digital signal processing) Starter Kit or 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 how to use the DSK assembler and debugger. This book is divided into three distinct 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 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 2 is analogous to a traditional manual introduction. It lists the key features of the assembler and debugger and tells you the steps you need to take in order to assemble and debug your program.
- **Part II: Assembler Description** contains detailed information about using the assembler.
  - Chapter 3 explains how to create DSK assembler source files and invoke the assembler, while Chapter 4 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 5 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   ——     ——            .ps     0FB00h
00002   ——     ——            .entry
>>>>> ENTRY POINT SET TO fb00
00003   fb00    5588          larp    AR0
00004   fb01    55a9    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. Here is an example of a directive syntax:

  .include "*file name***"**

  .include is the directive. This directive has one parameter, indicated by *file name*. When you use .include, the parameter must be an actual file name, 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:

  LALK *16-bit constant* [*, shift*]

The LALK instruction has two parameters. The first parameter, *16-bit constant*, is required. The second parameter, *shift*, is optional. As this syntax shows, 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 one 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 one.

- 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 **.byte** does not begin in column one.

.word $value_1$ [, ... , $value_n$]

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.

Related Documentation From Texas Instruments

This book describes the TMS320C2x DSP Starter Kit and related support tools. To obtain a copy of this document, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C2x User's Guide (literature number SPRU014) discusses the hardware aspects of the 'C2x of fixed-point digital signal processors. It describes pin assignments, architecture, instruction set, and software and hardware applications. It also includes data sheet information with electrical specifications and package mechanical data for all 'C2x devices. The book features a section with a 'C1x to 'C2x DSP system migration.

Trademarks

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

MS-DOS is a registered trademark of Microsoft Corp.

If You Need Assistance. . .

If you want to. . .	**Do this. . .**
Receive DSK updated software from the electronic bulletinboard system	Call the BBS: **(713) 274–2323** No parity, 8 data bits, 1 stop bit
Ask about product operation or report problems	Call the DSP Hotline: **(713) 274–2320** FAX: **(713) 274–2324**
Request more information about Texas Instruments Digital Signal Processing (DSP) products	Write to: Texas Instruments Incorporated Market Communications Manager, MS 737 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477–8924**
Report mistakes in this document or make suggestions for this or any other TI documentation	Fill out and return the reader response card at the end of this book, or send your comments to: Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251–1443

Contents

Hands-on Information

Assembler Description

Debugger Description

Figures

Tables

Chapter 1

Installing the DSK Assembler and Debugger

This chapter describes how to install the DSP Starter Kit (DSK) on a PC system running under DOS.

Topic **Page**

1.1 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 1.2 megabyte floppy-disk drive
☐	**memory**	Minimum of 640K
☐	**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 under brief power surges.

Notes:

- ❏ 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—do not load more than 50 mA.
- ❏ If you are using an external power supply, be sure you connect it correctly; the DSK is not warranted after you have made modifications to it.

☐	**board**	DSK circuit board
☐	**port**	An 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 5-2.
☐	**miscellaneous materials**	Blank, formatted disks

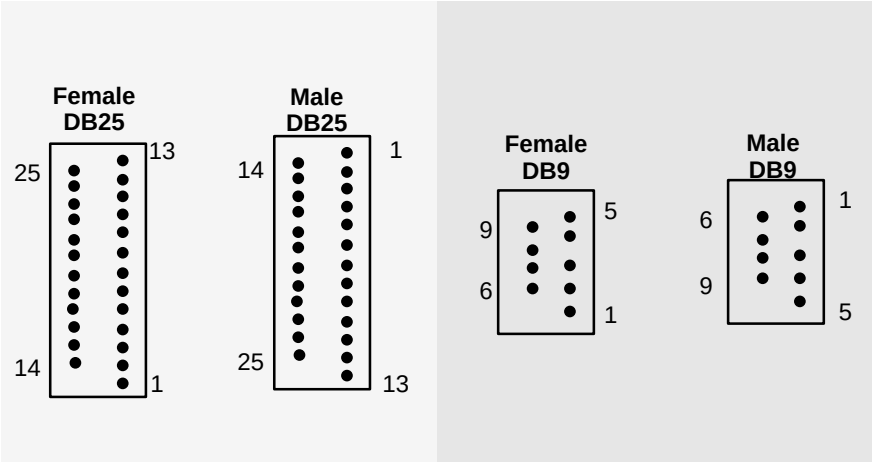
Software checklist

☐	**operating system**	MS-DOS or PC-DOS (version 4.01 or later)
☐	**files**	*dska.exe* is an executable file for the DSK assembler *dskd.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.

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. If you don't have the necessary cables, you can purchase them at your local computer store, or you can make them yourself. If you plan to make your own RS-232 cable, follow the DSK module connections shown in Figure 1–1.

Figure 1–1. The DSK Module Connections for an RS-232 Cable



Signal Name	Pin Assignments	
	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

1.2 Step 1: Connecting the DSK to Your PC

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

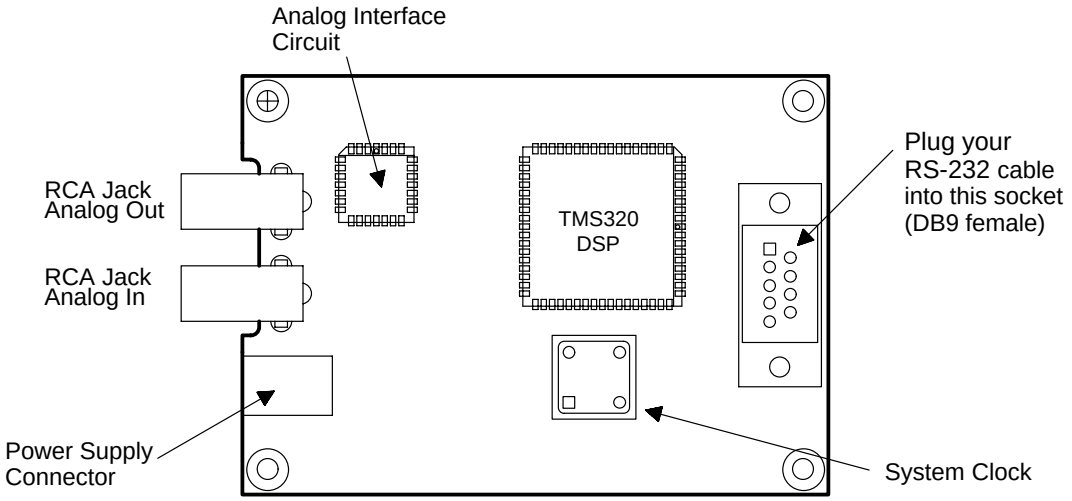
1) Turn off your PC's power.

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 1–2 for details.

Figure 1–2. 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 1–2 for details.

6) Plug the transformer into a wall socket.

7) Turn your PC's power on.

Note:

Suitable RS232 Cables:

- ❑ DB9 Male connects to DB25 Female.
- ❑ DB9 Male connects to DB9 Female.

1.3 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 will contain 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` ↵

1.4 Step 3: Modifying Your CONFIG.SYS File

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

`FILES=20`

Once you have edited your config.sys file and added the line, invoke the file by turning off the PC's power and turning it back on.

1.5 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.*

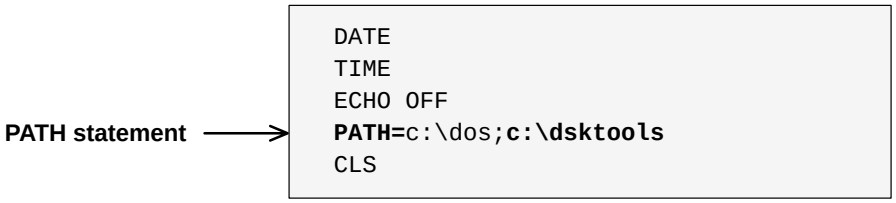
You can accomplish this by entering individual DOS commands, but it's simpler to put the commands in your system's autoexec.bat file. The general format for doing this is:

```
PATH=C:\dsktools;path2;path3;. . .
```

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 1–3.

Figure 1–3. DOS Command Setup for the DSK Environment (Sample autoexec.bat file)



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 ↵
```

1.6 Step 5: Verifying the Installation

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

- If you are using serial communication port 1 (COM1), enter:

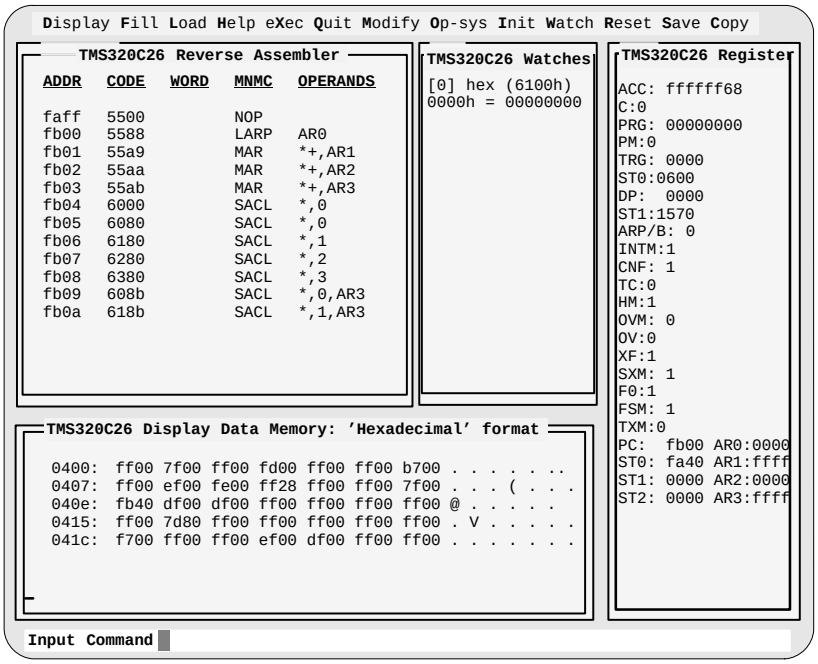
  dskd c1 ↵

- If you are using serial communication port 2 (COM2), enter:

  dskd c2 ↵

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

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



- If you see a display similar to this one, you have correctly installed your DSK board, the assembler, and the debugger.

- If you don't see a display, then your software or cables may not be installed properly. Go back through the installation instructions and be sure that you have followed each step correctly; then re-enter the command above.

Installation errors

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

- ❑ Your baud setting may be incorrect. Some Windows and OS/2 applications, as well as notebook computers, have low-level software limitations that may affect baud settings. Refer to page 5-2 for valid baud settings.
- ❑ You may have used an incorrect communication port (com1 vs. com 2). Refer to page 5-3 for more information on communication ports.
- ❑ Your communication port channel may be interrupted or noisy. If so, try using a lower baud. Refer to page 5-2 for valid bauds.
- ❑ A mouse driver or some 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 5-3 for more information on communication ports.
- ❑ Your RS232 cable and connectors may not be connected snugly.
- ❑ Your 9-V_{ac} transformer may not be plugged in on both ends. If power is getting to the DSK, then the LM7805 voltage regulator will be warm to the touch.

Chapter 2

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.

Topic **Page**

2.1 Description of the DSK Assembler

The DSK assembler is a simple and easy to use interface. Only the most significant 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 will load and run on the DSK.

Key features of the assembler

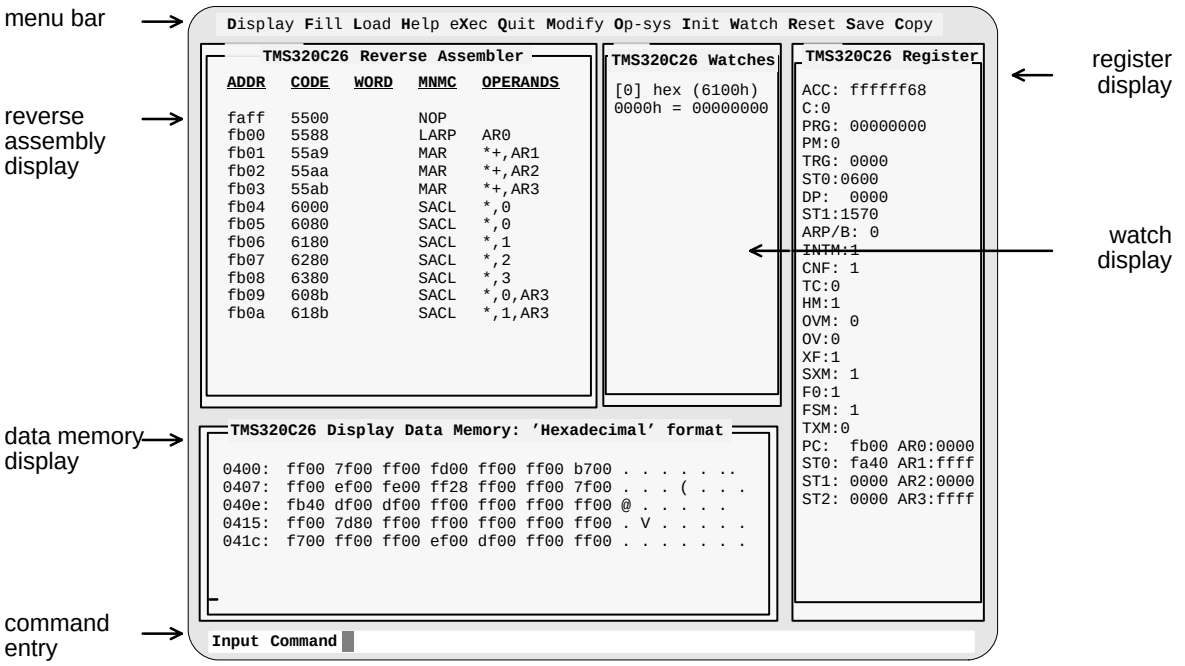
- **Quick.** The DSK assembler differs from many other assemblers in that it does not go through a linker phase to create an output file. Instead, the DSK uses special directives to assemble code at an absolute address during the assembly phase. As a result, you can create small programs quickly and easily.
- **Easy-to-use.** If you want to create larger programs, you can do this by simply chaining files together with the .include directive.

2.2 Description of the DSK Debugger

The debugger is easy to learn and use. Its friendly window- and menu-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 2–1 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 2–1. 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.

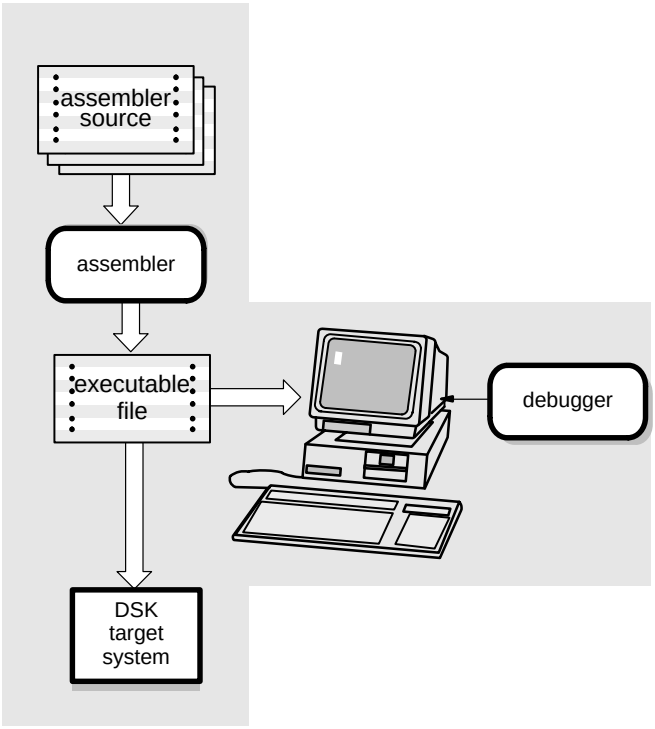
- **Powerful command set.** Unlike many other debugging systems, this debugger doesn't force you to learn a large, intricate command set. The DSK debugger supports a small but powerful command set.

- **Flexible command entry.** There are two main ways to enter commands. You can enter commands at the command line or use the menu bar; choose the method that you like better.

2.3 Developing Code for the DSK

Figure 2–2 illustrates the DSK code development flow.

Figure 2–2. DSK Software Development Flow



The following list describes the tools shown in Figure 2–2.

assembler

The **assembler** translates DSK assembly language source files into machine language object files for the TMS320C2x 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 TI TMS320 fixed-point DSP assembly language tools will also load and run on the DSK.

debugger

The main purpose 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.

2.4 Getting Started

This section provides a quick walkthrough so that you can get started without reading the whole 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.

```
*******************************************************
*  Saw–toothed Wave Generator                         *
*     Ramp rate is determined by interrupt            *
*     rate and step size. Ramp is made by             *
*     numerical rollover.                             *
*******************************************************
; Declare memory–mapped registers and program
; block address
DXR   .set  1     ;location of DXR register
IMR   .set  4     ;location of IMR register
      .ps   0fa0ah
      B     RINT  ;set receive interrupt vector
      .ps   0fb00h
      .entry      ;initial pc address
      LDPK  DXR   ;load data page for DXR (zero)
      LAC   IMR
      ORK   010h  ;turn on receive interrupt RINT)
      SACL  IMR
LOOP: ADDK  10    ;increment ACCU by 10
      SACL  DXR,3 ;shift ACCU left 3 bits when storing
      IDLE        ;wait for D/A interrupt
      B     LOOP
RINT: EINT        ;re–enable interrupts
      RET
;=======================================================
```

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

   dska try1 ↵

   This command invokes the TMS320C2x 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 3.5.

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

3) To see a listing of all errors and warnings that may have occurred during assembly of your program, assemble try1.asm with the –l option (lower-case "L").

   dska 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.

```
DSKA => DSP Starter Kit Assembler Rev 1.00 Thu Feb 18 11:22:20 1993 Copyright
© 1992–1993 Texas Instruments Incorporated

0001  –  ----  ----  ***************************************************
0002  –  ----  ----  *  Saw–toothed Wave Generator                      *
0003  –  ----  ----  *     Ramp rate is determined by interrupt        *
0004  –  ----  ----  *     rate and step size. Ramp is made by         *
0005  –  ----  ----  *     numerical rollover.                        *
0006  –  ----  ----  ***************************************************
0007  –  ----  ----  ; Declare memory–mapped registers and program
0008  –  ----  ----  ; block address
0009  –  ----  ----  DXR   .set  1     ;location of DXR register
0010  –  ----  ----  IMR   .set  4     ;location of IMR register
0011  –  ----  ----        .ps   0fa0ah
0012  0  fa0a  ff80        B     RINT  ;set receive interrupt vector
0013  –  ----  ----        .ps   0fb00h
0014  –  ----  ----        .entry      ;initial pc address
>>>>> ENTRY POINT SET TO fb00
0015  0  fb00  c800        LDPK  DXR   ;load data page for DXR (zero)
0016  0  fb01  2000        LAC   IMR
0017  0  fb02              ORK   010h  ;turn on receive interrupt RINT)
0018  0  fb04              SACL  IMR
0019  0  fb05        LOOP: ADDK  10    ;increment ACCU by 10
0020  0  fb06              SACL  DXR,3 ;shift ACCU left 3 bits when storing
0021  0  fb07              IDLE        ;wait for D/A interrupt
0022  0  fb08              B     LOOP
0023  0  fb0a        RINT: EINT        ;re–enable interrupts
0024  0  fb0b              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:

dskd ↵

6) This command brings up the TMS320C2x DSK debugger on your screen. From here, you can load your try1.dsk sample program by using the LOAD menus. For more information on using the debugger, refer to Chapter 5.

Chapter 3

Using the DSK Assembler

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

Topic **Page**

3.1 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.

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 next several lines show examples of source statements:

```
          .ps    0fb00h    ;initialize PC
sym       .set   2         ;symbol sym=2
Begin:    addk   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 one.
- ❑ 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 8 alphanumeric characters (A–Z, a–z, 0–9, and _). Labels are case-sensitive, and the first character cannot be a number. For example:

```
        .ps      0fb00h    ; Your code can start here
        .entry
Start:  larp  0
        lark  AR0,0
        lack  03fh      ; Turn on all interrupts
        ldpk  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 it's associated with). 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 Begin has the value 0fb00h.

```
.
.
.
00001                       *assume other code was assembled
00002
00003  ____ ____            .ps    0fb00h
00004  fb00 5588   Begin:   .word  0Ah,3,7
00005  fb01 55a9
```

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

```
00019
00020  fb00 5588   Here:
00021  fb01 55a9            .word  3
00022
```

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 to this are the .set directive, which assigns a value to a label, and the LDPK 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 semicolon, or an asterisk.

Using the mnemonic field

The mnemonic field follows the label field. *The mnemonic field cannot start in column 1, or it would 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   0fb00h    ; Your code can start here
        .entry
START:  larp  0
        lark  AR0,0
        lack  03fh      ; Turn on all interrupts
        ldpk  0
```

Refer to your *TMS320C2x User's Guide* for syntax specifications on individual opcodes.

It is generally necessary to resolve *all* fields in an opcode. If an opcode field (such as the shift field in a SACL opcode) is left out, the next field becomes the value for that particular field, and then the next field is not filled in. If you do not specify a field, its value is assumed to be null and is processed accordingly. For example:

```
        LDPK RESULT
        LAC  RESULT
WRONG:  SACL *,AR7    ;becomes SACL *,7 (AR7 used for shift)
RIGHT:  SACL *,0,AR7  ;all fields are specified
OK:     SACL *
```

In the above example, the numeric value of AR7 becomes 7. When this value is placed into the wrong field, the numeric value of AR7 becomes *BR0+.

Note:

Not all improperly specified opcodes generate warnings.

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 3.2) or a symbol (see Section 3.3). 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:

  - * The contents of AR are used as the data memory address.
  - *+ The contents of AR are used as the data memory address and are incremented after the access.
  - *– The contents of AR are used as the data memory address and are decremented after the access.
  - *0+ The contents of AR are used as the data memory address, and the contents of INDX are added to it after the access.
  - *0– The contents of AR are used as the data memory address, and the contents of INDX are subtracted from it after the access.
  - *BR0+ The contents of AR are used as the data memory address, and the contents of AR0 are added to it, with reverse carry (rc) propagation, after the access.
  - *BR0– The contents of AR are used as the data memory address, and the contents of AR0 are subtracted from it, 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 *TMS320C2x User's Guide*.

Commenting your source file

A comment can begin in any column and extends 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   0fb00h
       .entry
START: larp  0
       lark  AR0,0
       lack  03fh      ; Turn on all interrupts
       ldpk  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   0fb00h
       .entry
START: larp  0
       lark  AR0,0
* Turn on all interrupts
       lack  03fh
       ldpk  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.

3.2 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 8 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.

3.3 Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of up to 8 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.

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:

```
          .ps   0fb00h    ;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       ;max_buf = K*2 = 24
          lack  K         ;loads 12
          lack  –K        ’loads –12
          lack  K*2       ;loads 24
          lack  2*K       ;assumes constant, loads 2
          lack  max_buf   ;loads 24
          lack  !BIN      ;loads 0AAh
```

3.4 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 Figure 3–1).

Figure 3–1. Analyzing Expressions With the DSK Using Continuous Strings

(a) Expression Analysis With a COFF Assembler

```
FFT    .set   256
       LRLK   AR0, FFT
       LALK   FFT –1    ;expression analysis
```

(b) Expression Analysis With the DSK Assembler

```
FFT    .set   256
FFT–1  .set   255       ;set string FFT–1 = 255
       LARK   AR0, FFT
       LALK   FFT–1     ;FFT–1 is a complete string
```

In Figure 3–1 *(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:

```
LARK   AR0, 256
LALK   255
```

3.5 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:

dska [*filename*] [*–options*]

dska 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 3–1 lists the assembler options; the following subsections describe the options.

Table 3–1. Summary of Assembler Options

Option	Brief 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
–d	Disassembles the output file as it is created (for debugging purposes)
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 can generate an intermediate listing file containing all unresolved opcodes when you use the –l (Lowercase “L”) option. For example, if you want to assemble a file named test.dsk and create a listing file, enter:

```
dska 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.

Disassembling the output file (–d option)

The disassemble option is most useful when you are trying to find recurring bugs. As the output file is created, a disassembler disassembles the file. Once the file is disassembled, all opcodes contain resolved symbols; therefore, the assembler assigns an absolute value or address to each symbol.

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 this option is:

dska *filename* **asm”***statement”* [**asm”***statement”* ...]

For example:

```
dska test.asm asm”FFT .set  256” asm” .entry 0fb00h”
```

This statement specifies a program entry point (or load address) of 0fb00h and generates the file *test.inc*, in the following format:

```
FFT .SET 256
 .entry 0fb00h
```

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.

```
dska test.asm asm”fft .set 256”
```

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

Chapter 4

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**

4.1 Using the DSK Assembler Directives

Table 4–1 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 4–1. Assembler Directives Summary

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

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

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

Table 4–1. Assembler Directives Summary (Continued)

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

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

4.2 Directives That Define Sections

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

- **.data** identifies portions of code to be placed in data memory. Data memory usually contains initialized data.
- **.ds** functions the same as .data, however, with the .ds directive you can specify an optional address to initialize a new data address.
- **.entry** identifies the starting address of the program counter. By default, the current address is used, or, you can specify an optional address.
- **.ps** identifies portions of code to be placed in program memory. With the .ps directive you can specify an additional address to initialize a new program address.
- **.text** identifies portions of code in the .text section. The .text section usually contains executable code.

Example 4–1 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 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 4–1 is assembled, the sections contain:

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

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

Example 4–1. Sections Directives

```
 1     ____   ____   *********************************
 2     ____   ____   * Initialize section addresses  *
 3     ____   ____   *********************************
 4     ____   ____                .ps      0fb00h
 5     ____   ____                .ds      0fc00h
 7     ____   ____   *********************************
 8     ____   ____   * Start assembling into .text   *
 9     ____   ____   *********************************
10     ____   ____                .text
11     ____   ____                .byte    1,2
11     fb00   0001
11     fb001  0002
12     ____   ____                .byte    3,4
12     fb02   0003
12     fb03   0004
12     ____   ____   *********************************
13                   * Start assembling into .data   *
14                   *********************************
15     ____   ____                .data
16     ____   ____                .byte    9,10
16     fc00   0009
16     fc01   000a
17     ____   ____                .byte    11,12
17     fc02   000b
17     fc03   000c
18                   *********************************
19                   * Resume assembling into .text  *
20                   *********************************
21     ____   ____                .text
22     ____   ____                .byte    5,6
22     fb04   0005
23     fb05   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.

4.3 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 *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.

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.

4.4 Conditional Assembly Directives

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.

.if *expression*	Marks the beginning of a conditional block and assembles code if the .if condition is true (not zero).
.else	Marks a block of code to be assembled if .if is false.
.endif	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.

4.5 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.

- **.byte** places one or more 8-bit values into consecutive words of the current section.
- **.word** places one or more 16-bit values into consecutive words in the current section.
- **.string** places 8-bit characters from one or more character strings into the current section.
- **.long** places one or more 32-bit values into consecutive 32-bit fields in the current section.
- **.int** places one or more 16-bit values into consecutive words in the current section.
- **.qxx** 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.
- **.lqxx** 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.
- **.float** calculates 32-bit IEEE floating-point representations of single precision floating-point value and stores it in two consecutive words in the current section.
- **.bfloat** calculates a 16-bit, signed 2s-complement exponent and a 32-bit, signed 2s-complement mantissa.
- **.efloat** calculates a 16-bit, signed 2s-complement exponent and a 16-bit, signed 2s-complement mantissa.
- **.tfloat** calculates a 32-bit, signed 2s-complement exponent and a 64-bit, signed 2s-complement mantissa.
- **.double** 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.

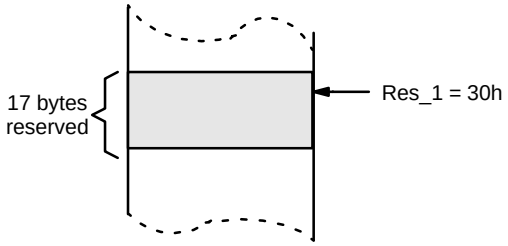
  When you use a label with .space, it points to the *first* word of the reserved block.

  Example 4–2 shows an example of the .space directives. Assume the following code has been assembled for this example:

```
          .ps      0fb00h
          .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.

Example 4–2. The .space Directive



4.6 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.
- The **.listoff** directive overrides the –l option and prohibits source listing.
- The **.liston** directive begins source listing.
- 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
        jmp     bval
```

4.7 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 together on one 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* 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.

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 0h, which is the location of the first initialized byte. The label stry has the value 6h, which is the first byte initialized by the .string directive.

```
0001    ____    ____            .ps     0fb00h
0002    ____    ____    strx:   .byte   10,–1,2,0Ah,”abc”
0002    fb00    000a
0002    fb01    00ff
0002    fb02    0002
0002    fb03    000a
0002    fb04    0061
0002    fb05    0062
0002    fb06    0063
0003    ____    ____    stry:   .string     10,–1,2,0Ah,”abc”
0003    fb07    0aff
0003    fb08    020a
0003    fb09    6162
0003    fb0a    6300
```

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

Syntax

.copy **"***filename***"**

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

(The quote marks enclosing the filename are optional.)

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.

The *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.

Example2

This example shows how the .include directive is used to tell the assembler to read and assemble source statements from other files, then to 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:

```
0001    ____    ____                                    ;filename: source.asm
0002    ____    ____            .space    10h           ;filename: source.asm
0003    ____    ____            .include  ”byte.asm”    ;filename: source.asm
************************************************
*       OPENING INCLUDE FILE byte.asm
************************************************
0001    ____    ____                                    ;filename: byte.asm
0002    ____    ____            .byte     ’a’,0ah,32    ;filename: byte.asm
0002    0001    0061
0002    0002    000a
0002    0003    0020
0003    ____    ____            .include  ”word.asm”    ;filename: byte.asm
************************************************
*       OPENING INCLUDE FILE word.asm
************************************************
0001    ____    ____                                    ;filename: word.asm
0002    ____    ____            .word     0abcdh,56     ;filename: word.asm
0002    0004    abcd
0002    00005   0038
0003    ____    ____
************************************************
*       CLOSING FILE
************************************************
0004    ____    ____            .byte     11,12,13      ;filename: byte.asm
0004    0006    000b
0004    0007    000c
0004    0008    000d
0005    ____    ____                                    ;filename: byte.asm
************************************************
*       CLOSING FILE
************************************************
0004    ____    ____                                    ;filename: source.asm
0005    ____    ____            .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.

The *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      0fb00h    ;set up load and run address's
        .entry
        .include "VECT.ASM"
        .ds      0400h
        .text
setup:  larp     AR0       ;initialize the CPU registers
        lark     AR0,0
        lark     AR1,0
        lark     AR2,0
        lark     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       0FB00h
            .entry
START       NOP
            NOP
            NOP
            B         START
            .end                ;these lines are ignored
LAB         ADDK      5
            sub       1
            B         LAB
```

Listing file:

```
0001     –  ----     ----              .ps      0FB00h
0002     –  ----     ----              .entry
>>>>> ENTRY POINT SET TO fb00
0003     0  fb00     5500     START    NOP
0004     0  fb01     5500              NOP
0005     0  fb02     5500              NOP
0006     0  fb03     ff80              B        START
0007     –  ----     ----              .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     0fb00h
LOOP:   MAR     *+,AR1          ;An infinite loop
        B       LOOP,*+,AR0     ;
        .entry                  ;Start program
        LARP    AR0
        LARK    AR0,0
        LARK    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. The *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

Source file:

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

Listing file:

```
0001      –         ----      ----      .ds       0400h
0002      –         ----      ----      .bfloat   1.5,3,6
0002      1         0400      0000
0002      1         0401      6000
0002      1         0402      0000
0002      1         0403      0000
0002      1         0404      6000
0002      1         0405      0001
0002      1         0406      0000
0002      1         0407      6000
0002      1         0408      0002
0003      –         ----      ----      .bfloat   –1.5,3,6
0003      1         0409      0000
0003      1         040a      a000
0003      1         040b      0000
0003      1         040c      0000
0003      1         040d      6000
0003      1         040e      0001
0003      1         040f      0000
0003      1         0410      6000
0003      1         0411      0002
0004      –         ----      ----      .efloat   1.5,3,6
0004      1         0412      6000
0004      1         0413      0000
0004      1         0414      6000
0004      1         0415      0001
0004      1         0416      6000
0004      1         0417      0002
0005      –         ----      ----
>>>>> LINE 5: .END ENCOUNTERED
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0 WARNINGS:0
```

Syntax

.if *well-defined expression*

code block to execute when the expression is true

.else

code to assemble when the expression is false

.endif

terminate condition block

Description

Three directives provide conditional assembly:

- The **.if** directive marks the beginning of a conditional block. The *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/.end 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     0200h
        .endif
        .if     B1_Dat
        .ds     0400h
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:

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

Syntax

.liston

.listoff

Description

The .liston and .listoff directives can be useful in debugging your code. 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        0fb00h
      lalk       PROG
      lrlk       AR0,DATA
      rptk       3          ;move 4 words from DS to PS
      tblw       *+
loop  nop
      b          loop
      .word      0,0,0,0
```

Listing file:

```
0001    –   ____   ____      ****************************************
0002    –   ____   ____      *       .LISTON/OFF EXAMPLE
0003    –   ____   ____      ****************************************
0004    –   ____   ____              .ds     0400h
0005    –   ____   ____              .listoff
0008    –   ____   ____              .ps     ofbooh
0009    0   fb00   d001              lalk    PROG
0010    0   fb02   d000              lrlk    AR0,DATA
0011    0   fb04   cb03              rptk    3       ;move 4 words from DS to P
0012    0   fb05   59a0              tblw    *+
0013    0   fb06   5500      loop    nop
0014    0   fb07   ff80              b       loop
0015    –   ____   ____      PROG:   .word   0,0,0,0
0015    0   fb09   0000
0015    0   fb0a   0000
0015    0   fb0b   0000
0015    0   fb0c   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. A *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.

```
0001    –   ____   ____              .ps     0fb00h
0002    –   ____   ____      strx:   .long   10000,”String”,’A’
0002    0   fb00   2710
0002    0   fb01   0000
0002    0   fb02   0053
0002    0   fb03   0000
0002    0   fb04   0074
0002    0   fb05   0000
0002    0   fb06   0072
0002    0   fb07   0000
0002    0   fb08   0069
0002    0   fb09   0000
0002    0   fb0a   006a
0002    0   fb0b   0000
0002    0   fb0c   0067
0002    0   fb0d   0000
0002    0   fb0e   0041
0002    0   fb0f   0000

>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS: 0 WARNINGS: 0
```

Syntax

.lqxx *value*$_1$ [**,** *...* **,** *value*$_n$]

.qxx *value*$_1$ [**,** *...* **,** *value*$_n$]

Description

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

Example

Here's an example of the .qxx directive.

```
0001   –  ----  ----  .ds   0x400              ;   |<-- 15 pl -->|
0002   –  ----  ----  .Q15  0.25               ; 0.0100000000000000b
0002   1  0400  2000
0003   –  ----  ----  .Q15  0.5
0003   1  0401  4000
0004   –  ----  ----  .Q15  0.75
0004   1  0402  6000
0005   –  ----  ----  .Q15  –0.25,–0.5,–0.75
0005   1  0403  e000
0005   1  0404  c000
0005   1  0405  a000
0006   –  ----  ----  .LQ24  9,10
0006   1  0406  0000
0006   1  0407  0900
0006   1  0408  0000
0006   1  0409  0a00
>>>>> FINISHED READING ALL FILES

>>>>> ASSEMBLY COMPLETE: ERRORS:0 WARNINGS:0
```

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.

- The *symbol* must appear in the label field.
- The *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
        LACK    zero    ;zero+1 is replaced
        ADDK    zero+1  ;symbol
```

Result

```
LACK    0
ADDK    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:

```
0001  –  ----  ----        **********************************
0002  –  ----  ----        * Begin assembling into .text
0003  –  ----  ----        **********************************
0004  –  ----  ----        .text
0005  –  ----  ----        **********************************
0006  –  ----  ----        * Reserve 15 words in .text
0007  –  ----  ----        **********************************
0008  –  ----  ----        .space              0f0h
0009  –  ----  ----        .word               100h,200h
0009  0  000f  0100
0009  0  0010  0200
0010  –  ----  ----
0011  –  ----  ----
0012  –  ----  ----        **********************************
0013  –  ----  ----        * Begin assembling into .data
0014  –  ----  ----        **********************************
0015  –  ----  ----        .data
0016  –  ----  ----        .string             ".data"
0016  1  0000  2e64
0016  1  0001  6174
0016  1  0002  6100
0017  –  ----  ----
0018  –  ----  ----
0019  –  ----  ----
0020  –  ----  ----
0021  –  ----  ----        **********************************
0022  –  ----  ----        * Reserve 2 words in .data
0023  –  ----  ----        * Res_1 points to the 1st reserved
                          word
0024  –  ----  ----        **********************************
0025  –  ----  ----        .space              020h
0026  –  ----  ----        .word               15
0026  1  0005  000f
0027  –  ----  ----
>>>>> 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 0 if nothing has yet been assembled into the .text or .ps sections. If code has already been assembled into the respective section, the section program counter is restored to its previous value in the section.

The *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 0 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.

```
0001  ____  ____                        **********************************
0002  ____  ____                        * Begin assembling into .data
0003  ____  ____                        **********************************
0004  ____  ____        .ds     0fb00h
0005  ____  ____        .byte   5,6
0005  fb00  0005
0005  fb01  0006
0006  ____  ____                        **********************************
0007  ____  ____                        * Begin assembling into .ps
0008  ____  ____                        **********************************
0009  ____  ____        .ps     7000h
0010  ____  ____        .byte   1
0010  7000  0001
0011  ____  ____        .byte   2,3
0011  7001  0002
0011  7002  0003
0012  ____  ____                        **********************************
0013  ____  ____                        * Resume assembling into .data
0014  ____  ____                        **********************************
0015  ____  ____        .data
0016  ____  ____        .byte   7,8
0016  fb02  0007
0016  fb03  0008
0017  ____  ____                        **********************************
0018  ____  ____                        * Resume assembling into .ps
0019  ____  ____                        **********************************
0020  ____  ____        .ps
0021  ____  ____        .byte   4
0021  7003  0004
>>>>> 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.

The *values* must be absolute. You can use as many *values* as fit on a single line (80 characters). If you use a label, it points to the first word that is initialized.

Example1

This example shows how to use the .word directive to initialize words. The symbol WordX points to the first word that is reserved.

```
0001  ____  ____                .ds     0fe00h
0002  ____  ____  WordX:        .word   3200, 0ffh, 3
0002  fe00  0c80
0002  fe01  00ff
0002  fe02  0003
0003  ____  ____
0004  ____  ____
```

Example2

Here's an example of the .int directive.

```
0005  ____  ____                .ds     0ff00h
0006  ____  ____  LAB1          .int    0,-1, 2, 0ABCDh
0006  ff00  0000
0006  ff01  ffff
0006  ff02  0002
0006  ff03  abcd
>>>>> FINISHED READING ALL FILES
>>>>> ASSEMBLY COMPLETE: ERRORS:0 WARNINGS:0
```

Chapter 5

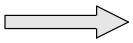
Using the DSK Debugger

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

Topic **Page**

5.1 Invoking the Debugger

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



dskd [*options*]

dskd is the command that invokes the debugger.

options supply the debugger with additional information.

Table 5–1 lists the debugger options; the following subsections describe the options.

Table 5–1. Summary of Debugger Options

Option	Brief description
? or H	Displays a listing of the available options
b *rate*	Selects the valid baud rate
com1, com2 or c1, c2	Selects communication port 1 or 2
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..3]	Sets the configuration mode (default =1)
s	Selects the default screen length of 25

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

You can display the contents of Table 5–1 on your screen by using the ? or H option. For example, enter:

```
dskd ? ↵
```

Selecting the baud (b option)

The valid baud settings are:

- ❑ b4800
- ❑ b9600
- ❑ b19200
- ❑ b38400

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

The c1 or c2 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 COM1. Depending on your serial port connection, replace *serial port* with one of these values:

- If you are using serial communication port 1, enter:

  dskd c1 ↵

- If you are using serial communication port 2, enter:

  dskd c2 ↵

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:

```
dskd e fb00h
```

The above example sets the DSK debugger at an initial address of fb00h.

Selecting a DTR logic level (i option)

Using option i tells the DSKD to invert DTR (data terminal ready) as a reset signal. Usually, the RS232 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 5–2 to switch between screen sizes.

Table 5–2. 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 CONF instruction works. Refer to your *TMS320C2x User's Guide* for more information on the CONF instruction.

5.2 Using Pulldown Menus in the Debugger

Figure 5–1 shows the main menu bar in the DSK debugger.

Figure 5–1. The Main Menu Bar

```
Display Fill Load Help eXec Quit Modify Op-sys Init Watch Reset Save Copy
```

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 one of the submenus for the Display menu:

display submenu →

format submenu →

```
<display> Data Program Version Status Breakpoints Format Memory

<format> Unsgnd Int Char pckdStrng Long Flt Double Q15 Oct heX 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 menu letters form the name of the corresponding debugger command. For example, the highlighted letters in **D**isplay→ **F**ormat→ **C**har show that you press D , F , C , in that order, to display the submenus.

Escaping from the pulldown menus and submenus

If you display a menu and then decide that you don't want to make a selection from this menu, you can press ESC to return to the main menu bar.

Using the Display menu

Table 5–3 lists the submenu selections for the Display menu. The highlighted letters show the keys that you can use for selecting menu choices.

Table 5–3. Menu Selections for Displaying Information

To display this	**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
❑ Insigned integer	→**U**nsgnd
❑ Hexadecimal	→he**X**

Using the Fill menu

Table 5–4 lists the menu selections for filling memory. The highlighted letters show the keys that you can use for selecting menu choices.

Table 5–4. Menu Selections for Filling Memory

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

Using the Load menu

Table 5–5 lists the submenu selections for the Load menu. The highlighted letters show the keys that you can use for selecting menu choices.

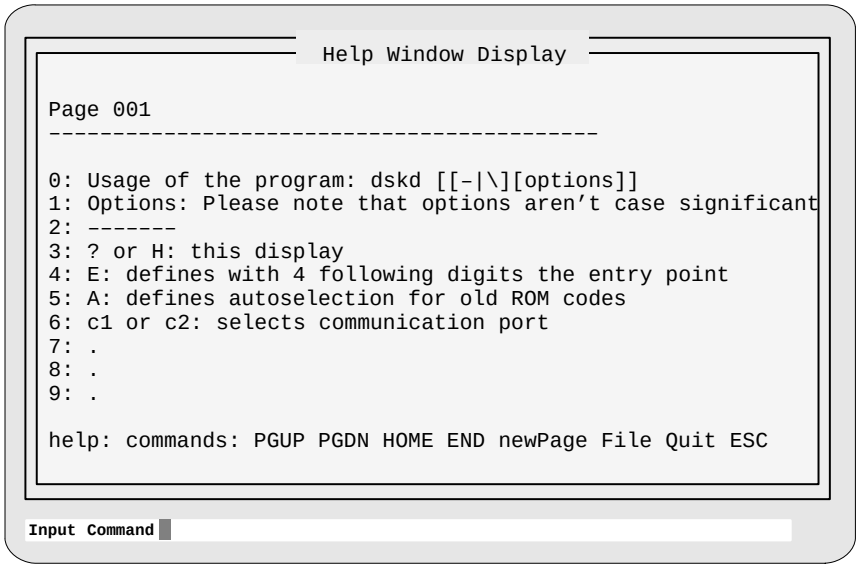
Table 5–5. Menu Selections for Loading Information into Memory

To load this	**L**oad→
COFF file	→**C**OFF
DSK file	→**D**DK
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 menu

You can use the help menu to bring up monitor information; then press ↵ to bring up the help menu shown in Figure 5–2.

Figure 5–2. The Monitor Info Screen



To move through the help window, you can use:

- ❑ 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 menu

Table 5–6 lists the menu selections for executing code. The highlighted letters show the keys that you can use for selecting menu choices.

Table 5–6. Menu Selections for Executing Code

To execute code from	e**X**ec→
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

Using the Quit menu

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 menu

Table 5–7 lists the menu selections for modifying your code. The highlighted letters show the keys that you can use for selecting menu choices.

Table 5–7. Menu Selections for Modifying Your Code

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

Using the Op-sys menu

The debugger provides a simple method of entering DOS commands without explicitly exiting the debugger environment. To do this, use the Op-sys menu 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.

The debugger opens a system shell and displays the DOS prompt. At this point, you can 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 operating-system commands that you can enter from a system shell. For example, you would not be able to invoke another version of the debugger.

Using the Init menu

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

Using the Watch menu

Table 5–8 lists the menu selections for watching your code during program execution. The highlighted letters show the keys that you can use for selecting menu choices.

Table 5–8. Menu Selections for the Watch Menu

To change your watch settings	**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 menu

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 menu

Table 5–9 lists the menu selections for saving code during a debugging session. The highlighted letters show the keys that you can use for selecting menu choices.

Table 5–9. Menu Selections for Saving Code

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

Using the Copy menu

Table 5–10 lists the menu selections for copying information. The highlighted letters show the keys that you can use for selecting menu choices.

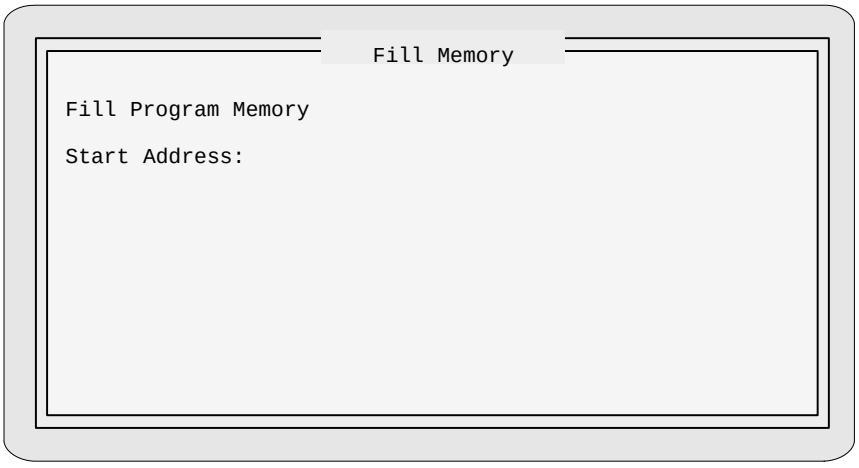
Table 5–10. Menu Selections for Copying Information

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

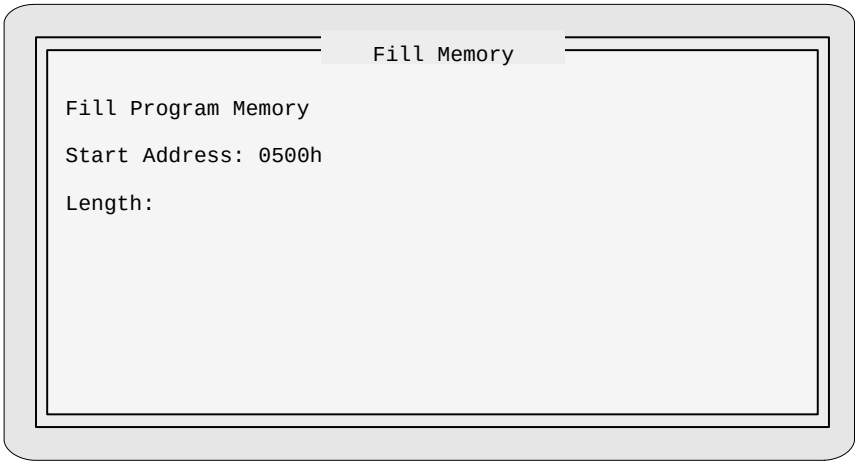
5.3 Using Dialog Boxes

Some of the debugger commands have parameters. When you execute these commands from pulldown menus, 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:

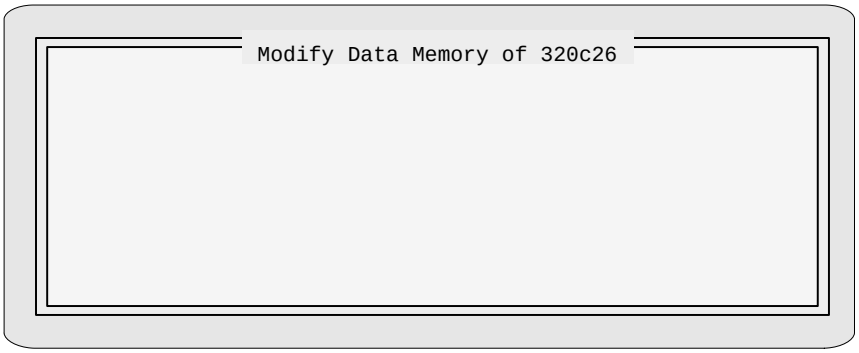


To enter a *start address*, just type, and then press ↵. The next parameter appears in the dialog box:

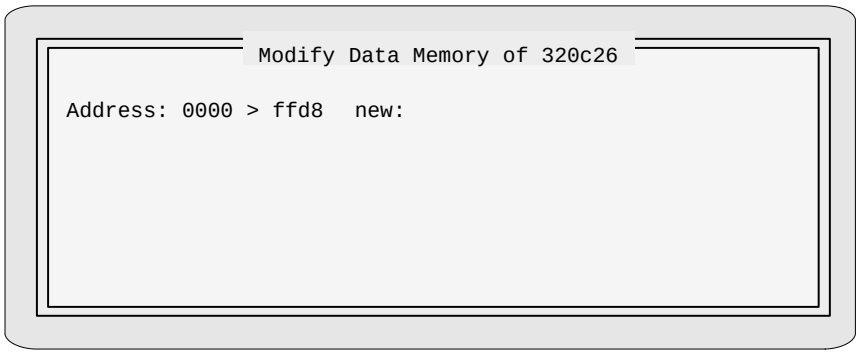


You can omit entries for optional parameters by pressing ↵, but the debugger won't allow you to skip required parameters. When you have entered all appropriate parameter values, *Fill Program Memory finished* appears at the bottom of the dialog box.

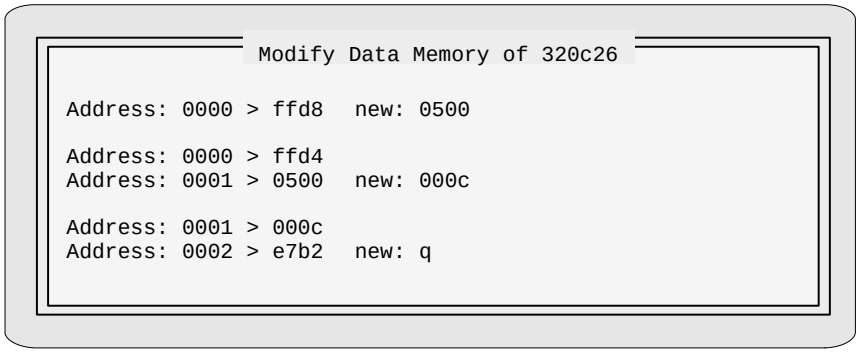
In the case of the Modify menu, when you select Data from its submenu, an empty dialog box appears on the screen:



Press ↵ for the debugger to display the first parameter:



Enter the address you want to modify and press ↵. The next parameter appears in the dialog box.



When you've entered a value for the final parameter, enter **q** and press ↵ at the next prompt; the debugger closes the dialog box and executes the command with the parameter values you supplied.

Closing a dialog box

There are three possible ways to exit from a dialog box:

- ❑ press ESC,
- ❑ press ↵, *or*
- ❑ press Q and ↵.

5.4 Using Software Breakpoints

This section describes the processes of setting and clearing software breakpoints and of 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 the assembly language code. A software breakpoint halts any program execution, whether you're running or single-stepping through code.

Setting a software breakpoint

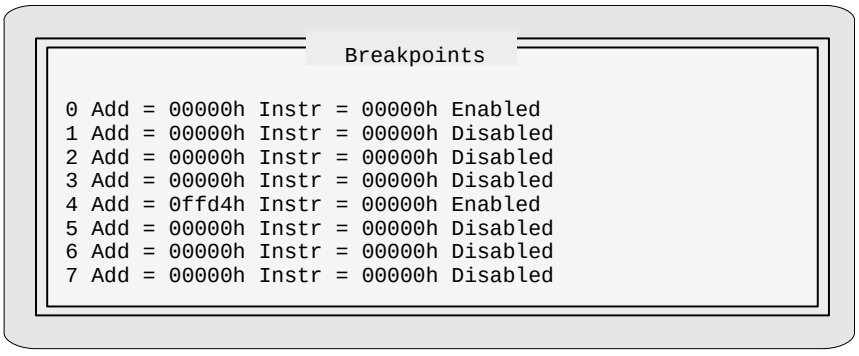
When you set a software breakpoint, the debugger highlights the breakpointed line in a bolder or brighter font. The highlighted statement appears in the reverse assembly window.

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 command or the BE command.

ba If you know the address where you'd like to set a software breakpoint, you can use the BA command. This command is useful because it doesn't require you to search through code to find the desired line. When you enter the BA command, the debugger asks you to enter an absolute address. Once you have entered the address, you are asked to choose the line number you want the breakpoint set on. Figure 5–3 shows a breakpoint set at address ffd4 on line number four. Note that you cannot set multiple breakpoints at the same statement.

Figure 5–3. Setting a software breakpoint



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 5–3, 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, you can use the BD command. Once you enter bd, the Breakpoints box appears on the screen (see Figure 5–3). The debugger then asks you 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 5–3.

5.5 Quick-Reference Guide

The following tables provide a quick-reference guide of the function key definitions, floating-point formats, and register 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 step past calls
F11 or shift + F1	Displays the reverse assembly window
F12 or shift + F2	Turns the trace on or off (this key acts as a toggle switch)

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

Register	Definition	Description
ACCU	accumulator	32 bits with a carry in ST1
PREG	product register	32 bits used for 16 x 16 multiplication
TREG	temporary register	16 bits for multiplication and special instructions
AR*i*	auxiliary register	16 bits with n=0.7 used as a counter and pointer
ST0	status register 0	16 bits
ST1	status register 1	16 bits
STCK*i*	stack register	16 bits with n=0.7 used for hardware stacking (note that the debugger uses one stack level for itself)
DRR	data receive register at address 0	16 bits for use with the serial port
DXR	data transmit register at address 1	16 bits for use with the serial port
TM	timer register at address 2	16 bits
PRD	period register at address 3	16 bits
MSK	interrupt mask register at address 4	6 bits for masking 6 interrupts
GREG	global register at address 5	8 bits to define data memory as global
ARP ST0	auxiliary register pointer	3 bits
ARB ST1	auxiliary register pointer buffer	3 bits
INTM ST0	interrupt mode (enable global interrupt)	1 bit
CNF ST1	internal program/data configuration	2 bits
TXM ST1	FSX mode bit	1 bit
PM ST1	Product register to accumulator shift mode	2 bits
C ST1	carry bit	1 bit
TC ST1	test / control	1 bit
FO ST1	serial port control (8/16 bit mode)	1 bit
HM ST1	hold mode selection	1 bit

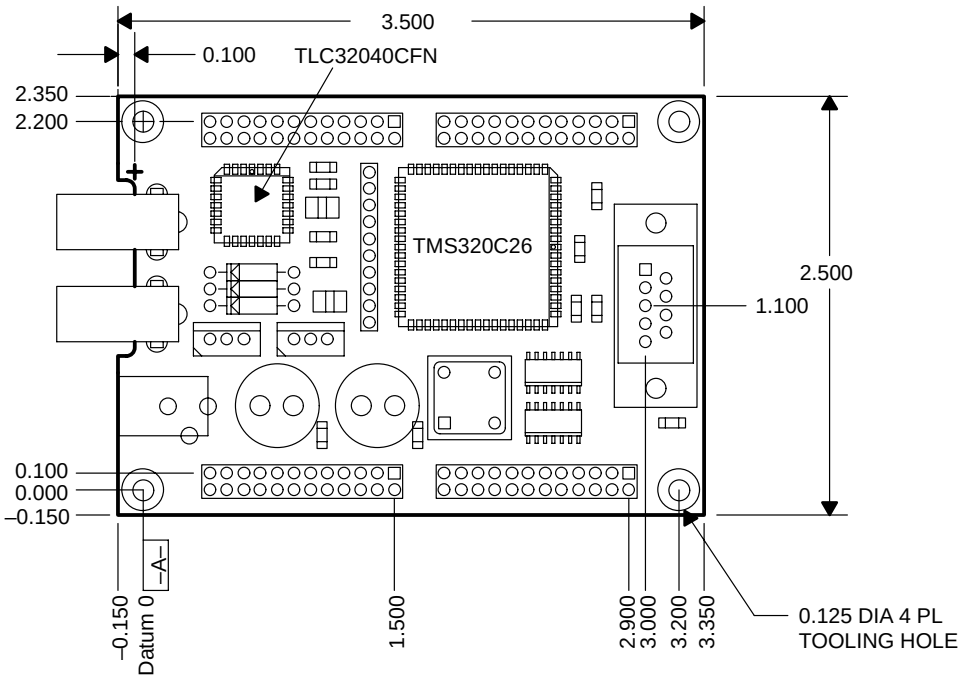
Appendix A

DSK Schematics

This appendix contains the schematics for the DSP Starter Kit.

Topic **Page**

A.1 TMS320C2x DSP Starter Kit



Note:

Dimensions are in inches.

Appendix B

Glossary

A

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

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: One of two different types of files. One type contains DOS commands for the PC to execute. A second type of batch file contains debugger commands for the debugger to execute. The PC doesn't execute debugger batch files, and the debugger doesn't execute PC batch files.

block: A set of declarations and statements that are grouped together with braces.

breakpoint: A point within your program where execution will halt because of a previous request from you.

byte: A sequence of 8 adjacent bits operated upon as a unit.

C

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

command file: A file that contains linker options and names input files for the linker.

command line: The portion of the COMMAND window where you can enter commands.

command-line cursor: A block-shaped cursor 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.

common object file format (COFF): An object file that promotes modular programming by supporting the concept of *sections*.

constant: A numeric value that can be used as an operand.

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

cursor: An icon on the screen (such as a rectangle or a horizontal line) that is used as a pointing device. The cursor is usually under keyboard control.

D

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

debugger: A window-oriented software interface that helps you to debug DSK programs running on a DSK board.

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: Assembly language code formed from the reverse-assembly of the contents of memory.

DSP: Digital signal processing.

DTR: Data terminal ready.

E

EGA: *Enhanced Graphics Adaptor.* An industry standard for video cards.

entry point: The starting execution point in target memory.

expression: A constant, a symbol, or a series of constants and symbols separated by arithmetic operators.

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

F

field: A software-configurable data type whose length can be programmed to be any value in the range of 1–8 bits.

file header: A portion of a 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: A kind of symbol that is either 1) defined in the current module and accessed in another, or 2) accessed in the current module but defined in another.

I

input section: A section from an object file that will be 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.

LSByte: Least significant byte.

M

member: An element or variable 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 found at the top of the debugger display.

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

MSB: Most significant bit.

MSByte: Most significant byte.

N

named section: 1) An initialized section that is defined with a .sect directive, or 2) an uninitialized section that is defined with a .usect directive.

O

object file: A file that has been assembled and contains machine-language object code.

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 the context and where it's used in this book: 1) In installation instructions or information relating to hardware and boards, *PC* means *Personal Computer* (as in IBM PC). 2) In general debugger and program-related information, *PC* means *Program Counter*, which is the register that identifies the current statement in your program.

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.

S

section: A relocatable block of code or data that will ultimately occupy contiguous space in the memory map.

serial port: The serial port that the debugger uses for communicating with the emulator or the applications board. The port address is selected, depending on which communication port the debugger is attached to.

single-step: A form of program execution that allows you to see the effects of each statement. The program is executed statement by statement; the debugger pauses after each statement to update the data-display windows.

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

SPC: Section program counter.

static: A kind of variable whose scope is confined to a function or a program. The values of static variables are not discarded when the function or program is exited; their previous value is resumed when the function or program is re-entered.

string table: A table that stores symbol names that are longer than 8 characters (symbol names of 8 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.

union: A variable that may hold (at different times) objects of different types and sizes.

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

V

VGA: *Video Graphics Array.* An industry standard for video cards.

W

window: A defined rectangular area of virtual space on the display.

word: A 16-bit addressable location in target memory.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W