NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS
MAIL STATION 640
P. O. BOX 1443
HOUSTON, TX 77251–9879

PERF

BIND THIS EDGE

PERF

PERF

BIND THIS EDGE

PERF

Reader Response Card: TMS320C54x DSKplus User's Guide

Texas Instruments wants to provide you with the best documentation possible. Please help us meet this goal by answering these questions and returning this card.

What is your primary use for the information in this manual?

- ❑ Designing 'C54x-based hardware
- ❑ Designing 'C54x-based software

How have you used this manual?

- ❑ To look up specific information or procedures when needed (as a reference)
- ❑ To read chapters about subjects of interest
- ❑ To read from front to back before using the information

Please describe any mistakes or unclear information in this manual (include page numbers).

_____

_____

_____

_____

Which topics should be described in greater detail?

_____

_____

_____

_____

Please list topics that were difficult to find, and why (for example, the topic was not in a logical location).

_____

_____

_____

_____

Please list any other suggestions for improving this book.

_____

_____

_____

_____

Name_____Title_____
Company_____
Address_____
City_____State_____Zip/Country_____
Phone number_____

Thank you. *October, 1996*

TEXAS INSTRUMENTS

TMS320C54x DSKplus

DSP Starter Kit

User's Guide

1996

Digital Signal Processing Solutions

Printed in U.S.A., October 1996
SDS

SPRU191

User's Guide

TMS320C54x DSKplus
DSP Starter Kit

1996

TMS320C54x DSKplus User's Guide

DSP Starter Kit

Literature Number: SPRU191
January 1998

Printed on Recycled Paper

IMPORTANT NOTICE

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

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

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

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

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

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

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

Preface

Read This First

About This Manual

This book describes the TMS320C54x digital signal processing (DSP) enhanced starter kit (DSKplus) and how to use the DSKplus with these tools:

- The DSKplus assembler
- The DSKplus debugger
- The DSKplus application loader

How to Use This Manual

The following table summarizes the information contained in this book:

If you are looking for information about:	Turn to these chapters:
Application loader	Chapter 3, *DSKplus Debugger and Application Loader Software*
Assembler directives	Chapter 5, *Assembler Description,* and Appendix C, *Assembler Directives Reference*
Code Explorer™	Chapter 3, *DSKplus Debugger and Application Loader Software*
DSKplus assembler	Chapter 5, *Assembler Description*
DSKplus debugger	Chapter 3, *DSKplus Debugger and Application Loader Software*
Hardware and software installation	Chapter 2, *Installing the DSKplus Assembler and Debugger*
Initialization of devices	Chapter 7, *Initialization Routines*
Overview of the DSKplus	Chapter 1, *Introduction*
Programming the DSP or host PC	Chapter 4, *Software Considerations*

If you are looking for information about:	Turn to these chapters:
Using a PAL® device	Chapter 6, *Hardware*
Using the XDS510™ with DSKplus	Chapter 6, *Hardware*

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface.` 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:

```
0011  0005  0001         .field    1, 2
0012  0005  0003         .field    3, 4
0013  0005  0006         .field    6, 3
0014  0006               .even
```

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

```
C:  csr –a /user/ti/simuboard/utilities
```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** font and parameters are in an *italic typeface*. 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:

  .sect "*section name***",** *address*

  .sect is the directive. This directive has two parameters, indicated by *section name* and *address*. When you use .sect, the first parameter must be an actual section name, enclosed in double quotes; the second parameter must be an address.

- 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.

- Some directives can have a varying number of parameters. For example, the .byte directive can have up to 100 parameters. The syntax for this directive is:

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

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

Information About Cautions

This book contains cautions.

> **This is an example of a caution statement.**
>
> **A caution statement describes a situation that could potentially damage your software or equipment.**

The information in a caution is provided for your protection. Please read each caution carefully.

Related Documentation From Texas Instruments

The following books describe the ’54x and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477-8924. When ordering, please identify the book by its title and literature number.

The ***TMS320C54x DSP Reference Set*** (literature number SPRU210) is composed of four volumes of information, each with its own literature number for individual ordering.

TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals (literature number SPRU131) describes the TMS320C54x 16-bit, fixed-point, general-purpose digital signal processors. Covered are its architecture, internal register structure, data and program addressing, the instruction pipeline, DMA, and on-chip peripherals. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

TMS320C54x DSP Reference Set, Volume 2: Mnemonic Instruction Set (literature number SPRU172) describes the TMS320C54x digital signal processor mnemonic instructions individually. Also includes a summary of instruction set classes and cycles.

TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set (literature number SPRU179) describes the TMS320C54x digital signal processor algebraic instructions individually. Also includes a summary of instruction set classes and cycles.

TMS320C54x DSP Reference Set, Volume 4: Applications Guide (literature number SPRU173) describes software and hardware applications for the TMS320C54x digital signal processor. Also includes development support information, parts lists, and design considerations for using the XDS510 emulator.

TLC320AC01C Fixed-Point Digital Signal Processors (literature number SLAS057) data manual contains the electrical and timing specifications, as well as parameter measurement information for the TLC320AC01C.

TMS320C54x, TMS320LC54x, TMS320VC54x Fixed-Point Digital Signal Processors (literature number SPRS039) data sheet contains the electrical and timing specifications for these devices, as well as signal descriptions and pinouts for all of the available packages.

TMS320C54x Assembly Language Tools User's Guide (literature number SPRU102) describes the assembly language tools (assembler, linker, and other tools used to develop assembly language code), assembler directives, macros, common object file format, and symbolic debugging directives for the 'C54x generation of devices.

TMS320C5xx C Source Debugger User's Guide (literature number SPRU099) tells you how to invoke the 'C54x emulator, EVM, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C54x Code Generation Tools Getting Started Guide (literature number SPRU147) describes how to install the TMS320C54x assembly language tools and the C compiler for the 'C54x devices. The installation for MS-DOS™, OS/2™, SunOS™, Solaris™, and HP-UX™ 9.0x systems is covered.

TMS320C54x Evaluation Module Technical Reference (literature number SPRU135) describes the 'C54x EVM, its features, design details and external interfaces.

TMS320C54x Optimizing C Compiler User's Guide (literature number SPRU103) describes the 'C54x C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C54x generation of devices.

TMS320C5x Simulator Getting Started (literature number SPRU137) describes how to install the TMS320C5x simulator and the C source debugger for the 'C5x. The installation for MS-DOS®, PC-DOS™, SunOS™, Solaris™, and HP-UX™ systems is covered.

TMS320 Third-Party Support Reference Guide (literature number SPRU052) alphabetically lists over 100 third parties that provide various products that serve the family of '320 digital signal processors. A myriad of products and applications are offered—software and hardware development tools, speech recognition, image processing, noise cancellation, modems, etc.

TMS320 DSP Development Support Reference Guide (literature number SPRU011) describes the TMS320 family of digital signal processors and the tools that support these devices. Included are code-generation tools (compilers, assemblers, linkers, etc.) and system integration and debug tools (simulators, emulators, evaluation modules, etc.). Also covered are available documentation, seminars, the university program, and factory repair and exchange.

Trademarks

Code Explorer is a trademark of GoDSP Corporation.

HP-UX is a trademark of Hewlett-Packard Company.

MS-DOS is a registered trademark of Microsoft Corporation.

OS/2 is a trademark of International Business Machines Corporation.

PC is a trademark of International Business Machines Corporation.

PAL® is a registered trademark of Advanced Micro Devices, Inc.

Solaris is a trademark of Sun Microsystems, Inc.

TI, XDS510, and 320 Hotline On-line are trademarks of Texas Instruments Incorporated.

Windows is a trademark of Microsoft Corporation.

Pentium is a trademark of Intel Corporation.

If You Need Assistance. . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.html

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/mirrors/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32	Email: epic@ti.com
Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68		
English	+33 1 30 70 11 65		
Francais	+33 1 30 70 11 64		
Italiano	+33 1 30 70 11 67		
EPIC Modem BBS	+33 1 30 70 11 99		
European Factory Repair	+33 4 93 22 25 40		
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10	

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: comments@books.sc.ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Figures

Tables

Examples

Chapter 1

Introduction

The TMS320C54x DSKplus is a low-cost DSP starter kit with enhanced architecture. The development kit contains a stand-alone application board that you connect to your PC™ and that enables you to explore the architecture and operation of the ’C54x CPU and its peripherals. The DSKplus board executes your custom ’C54x code in real time while the Windows™-based debugger analyzes it line-by-line, displaying internal DSP register information in multiple windows, also in real time. The board’s communication interface lets you create your own ’C54x DSP code and host PC code. The hardware enables the use of expansion boards for adding memory, peripherals such as codecs, interface logic, other DSPs, or microcontrollers.

Topic **Page**

1.1 Kit Contents and Features

The DSKplus development kit contains these parts:

- The DSKplus development board
- PC parallel port cable (DB-25M to DB-25F)
- Universal power supply (Input: 100–250 V, 50–60 Hz; Output: 5-V dc, 3.3 A)
- GoDSP's Windows™-based Code Explorer™ debugger
- TMS320C54x ('C54x) algebraic assembler
- Self-test program
- Various application programs
- *TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals*
- *TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set*
- *TMS320C54x DSP Data Sheet*
- *TLC320AC01C Single-Supply Analog Interface Circuit Data Manual*

Several features of the DSKplus board enable MIP-intensive, low-power applications:

- One TMS320C542 ('C542) enhanced fixed-point DSP
- 40 MIPS (25-ns instruction cycle time)
- 10K words of dual-access RAM (DARAM)
- 2K words boot ROM
- One time-division-multiplexed (TDM) serial port
- One buffered serial port (BSP)
- One host port interface (HPI) for PC-to-DSP communications
- One on-chip timer
- Three power-down modes on the 'C542
- Programmable, voice-quality TLC320AC01 (DAC, ADC interface circuit)
- Socketed PAL22V10 for board customization
- Socketed oscillator
- Phase-locked-loop (PLL) clock generator
- XDS510 emulator header
- I/O expansion bus and control signals for external designs
- Standard 1/8-inch mono mini-jacks for analog I/O (microphone and multi-media speakers)

1.2 What You Need

Make sure that you have the appropriate hardware and software.

1.2.1 Hardware Requirements

In addition to kit contents, you need the following equipment to use the DSKplus board:

☐	**Host**	A 386, 486, or Pentium PC with a 1.44M-byte 3.5″ floppy disk drive
☐	**Port**	DSKplus supports 4-bit parallel ports and 8-bit bidirectional parallel ports. DSKplus does not support Enhanced Printer Port and Extended Capabilities Port functionality. However, DSKplus can operate in standard mode of these ports.
☐	**Memory**	Minimum of 4M bytes
☐	**Monitor**	Color VGA

1.2.2 Software Requirements

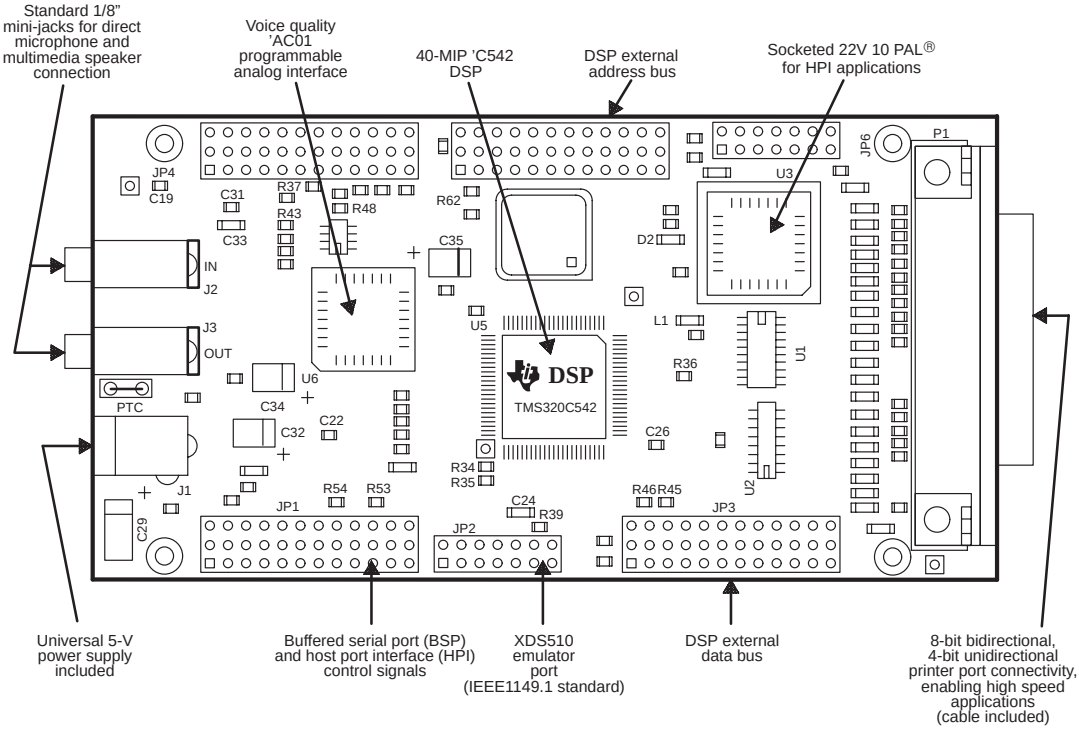
In addition to the provided software, you need the following applications to use the DSKplus board:

☐	**Windows**	Windows 3.1 or Windows 95
☐	**ASCII editor**	

1.3 Functional Overview

The diagram of the DSKplus development board is shown in Figure 1–1. It identifies the analog interface circuit TLC320AC01 (’AC01), IEEE.1149.1 emulation port (XDS510), host port interface (HPI), CPU, and peripherals.

Figure 1–1. DSKplus Board Diagram



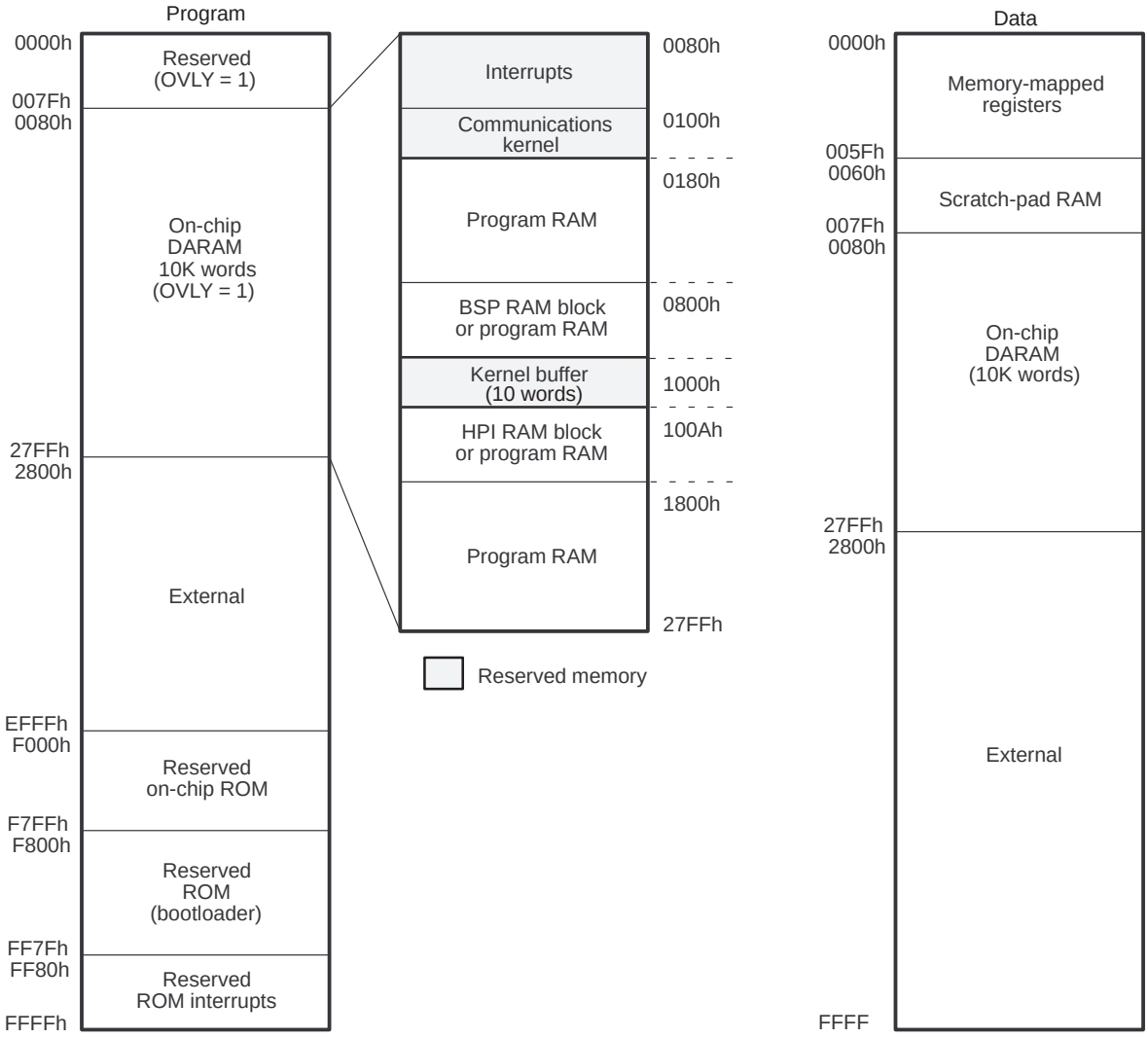
The host port interface logic is an on-board PAL® device that operates as the main interface between the host PC’s parallel port and the ’C542 host port interface (HPI). As a result, the interface logic gives the host PC direct control of the ’C542’s HPI and DSP reset signal, and it can configure the board to operate with different PC parallel ports (that is, 4-bit and 8-bit printer ports).

When you power the board and start the debugger, the debugger software initializes the host interface logic and configures the board to the correct parallel port mode, either 4-bit or 8-bit. At this time, the communication link between the DSKplus board and host PC is ready for operation.

For the DSKplus and the host to communicate properly, the DSP must follow a common communication protocol defined by the host. Therefore, the host PC downloads the protocol to the DSP communication software, which resides in DSP memory at addresses 80h–17Fh. The protocol also uses a mutual com-

munication buffer in DSP memory at 1000h–1009h. The communication buffer is the memory used for communicating between the host and DSP. All memory locations from 80h –17Fh and 1000h –1009h are reserved and must never be written over. See the DSKplus memory map in Figure 1–2 for information on reserved memory.

Figure 1–2. DSKplus Memory Map



The ’AC01 analog interface circuit provides a single channel of voice-quality data acquisition. The ’AC01 has the following features:

- Single-chip solution A/D and D/A conversions with 14 bits of dynamic range
- Built-in, programmable antialiasing filter
- Software-programmable sampling rates
- Software-programmable reset, gain, and loopback
- Software-programmable power-down mode
- 2-channel analog input summing
- Software-selectable auxiliary input
- Configurable as master/slave to allow cascading

The ’AC01 interfaces directly to the ’C542 TDM serial port. The ’AC01 generates the required shift clock (SCLK) and frame sync (FS) pulses used to send data to/from the ’AC01. These pulses are a function of software-programmable registers and the ’AC01 master clock. The master clock is generated by the on-board oscillator. See Chapter 4, *Software Considerations*, for instructions on how to program the ’AC01 or see the *TLC320AC01C Single-Supply Analog Interface Circuit Data Manual.*

The DSKplus board provides six headers, including an XDS510 emulator header, to aid in the design of daughter boards. The XDS510 emulator header allows the board to act as an XDS emulator target board with robust, nonintrusive debugging capabilities. The XDS is the advanced emulator from TI available through your local sales office.

The on-board 10-MHz oscillator provides a clock to the ’C542, the ’AC01, and the PAL® device. The ’C542 PLL option is set to 4, creating a 40-MHz internal clock oscillator. The ’AC01 runs at 10 MHz. The ’AC01 data manual includes information for operation at 10.368 MHz; the data from the ’AC01 tables and graphs must be interpolated to 10 MHz.

The GoDSP Code Explorer provides a Windows-based debugging interface and a manageable development environment. The debugger interface can display and modify all of the internal registers of the DSP. Some common functions of the debugger are single-stepping code, setting breakpoints in code, setting up watch windows, and managing file I/O. Additional debugger features and information can be found in Chapter 3, *DSKplus Debugger and Application Loader Software.* These tools allow you to fully develop and debug DSP code in a real-time environment.

The DSKplus software includes a Windows-based real-time debugger, DSKplus application loader, and an absolute algebraic assembler. The DSKplus algebraic assembler enables you to program in assembly language without having extensive knowledge of the mnemonic instruction set. Take, for example, a case in which you want to multiply two numbers, x × y, and place the result in accumulator B. With a mnemonic-based assembler, you must know the function of each mnemonic instruction:

```
STM    #y, AR2          ; Address of first multiplicand
STM    #X, AR3          ; Address of second multiplicand
::
MPY    *AR3, *AR2, B    ;
```

With an algebraic assembler, you use simpler mathematical notation:

```
AR2 = #y
AR3 = #x
::
B = *AR2 * *AR3
```

For more information regarding the algebraic instruction set, see *TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set.*

The DSKplus algebraic assembler converts the source file with a .asm extension to an object-based COFF file with a .obj extension. As a result, code addressing is fully resolved at assembly time using in-line assembler directives, so there is no need for a linker stage. The code is ready to be loaded into the DSP.

Chapter 2

Installing the DSKplus Assembler and Debugger

Before you use the DSKplus board, verify that your equipment meets the requirements described in the previous chapter. Next, you must install the hardware and the software on your PC.

This chapter provides instructions for connecting the DSKplus board, installing the DSKplus software, and running the self-test program.

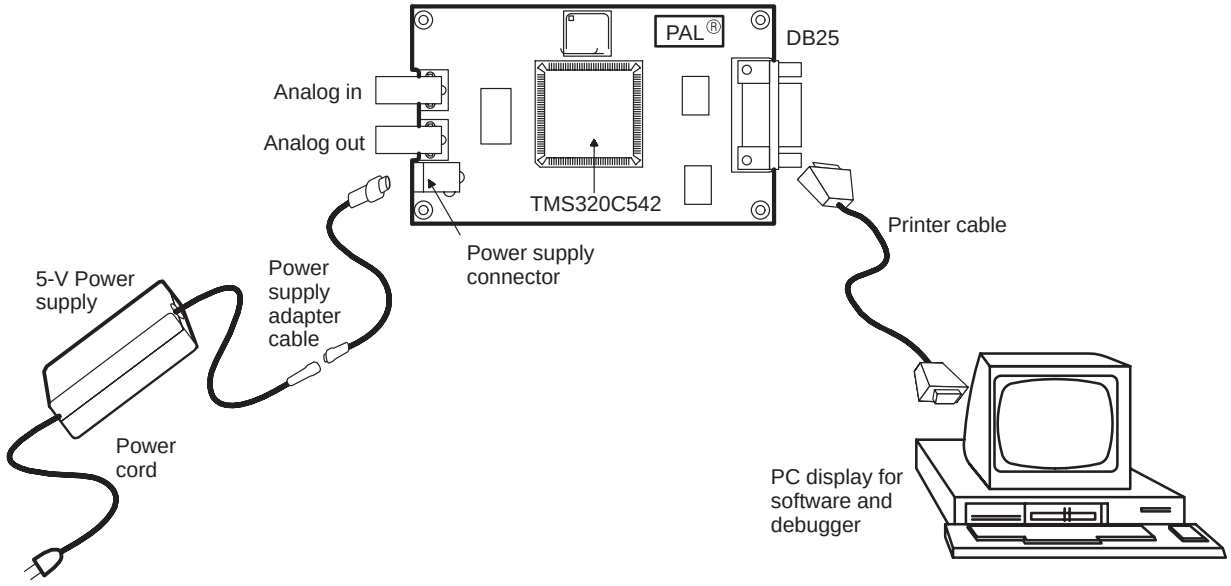
Topic **Page**

2.1 Connecting the DSKplus Board

Follow these steps to ensure a proper connection between the DSKplus board and the PC host. (See Figure 2–1)

1) Turn off the PC's power.
2) Connect the DB25 printer cable to the PC's parallel port.
3) Connect the DB25 printer cable to the DSKplus board.
4) Connect the power cord to the 5-V dc power supply.
5) Plug the transformer into the wall outlet.
6) Connect the 5-pin DIN-to-5.5-mm power supply adapter cable to the power supply's 5-pin DIN connector.
7) Connect the 5.5-mm power supply adapter cable into the power supply connector on the DSKplus board.
8) Turn on the PC's power.

Figure 2–1. Connection Diagram



2.2 Installing the DSKplus Software

From the *File* menu item in the Windows program manager, click on the RUN command (or in Windows 95, click on the Start button and select Run...) and then type:

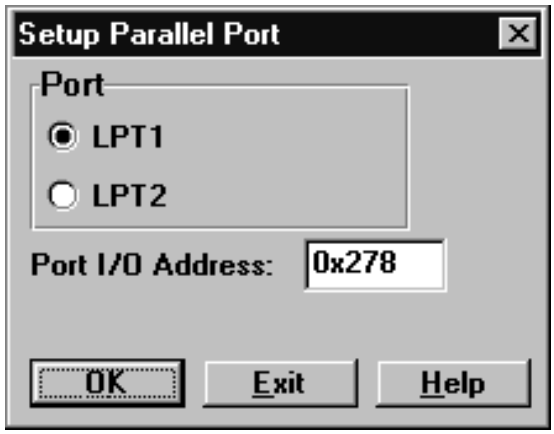
```
A:\SETUP.EXE
```

By default the installation program installs the software to the C:\DSKplus directory. If you like, change this directory when you are prompted to confirm the destination directory.

After running the install program, a Windows program group icon appears called Code Explorer. It has two program icons: Code Explorer and 'C54x Help. The absolute assembler, application loader and self-test are not members of this group because they are DOS programs and are accessed through the Windows DOS shell.

To test the setup, click on the Code Explorer icon. The port dialog box appears, as shown in Figure 2–2.

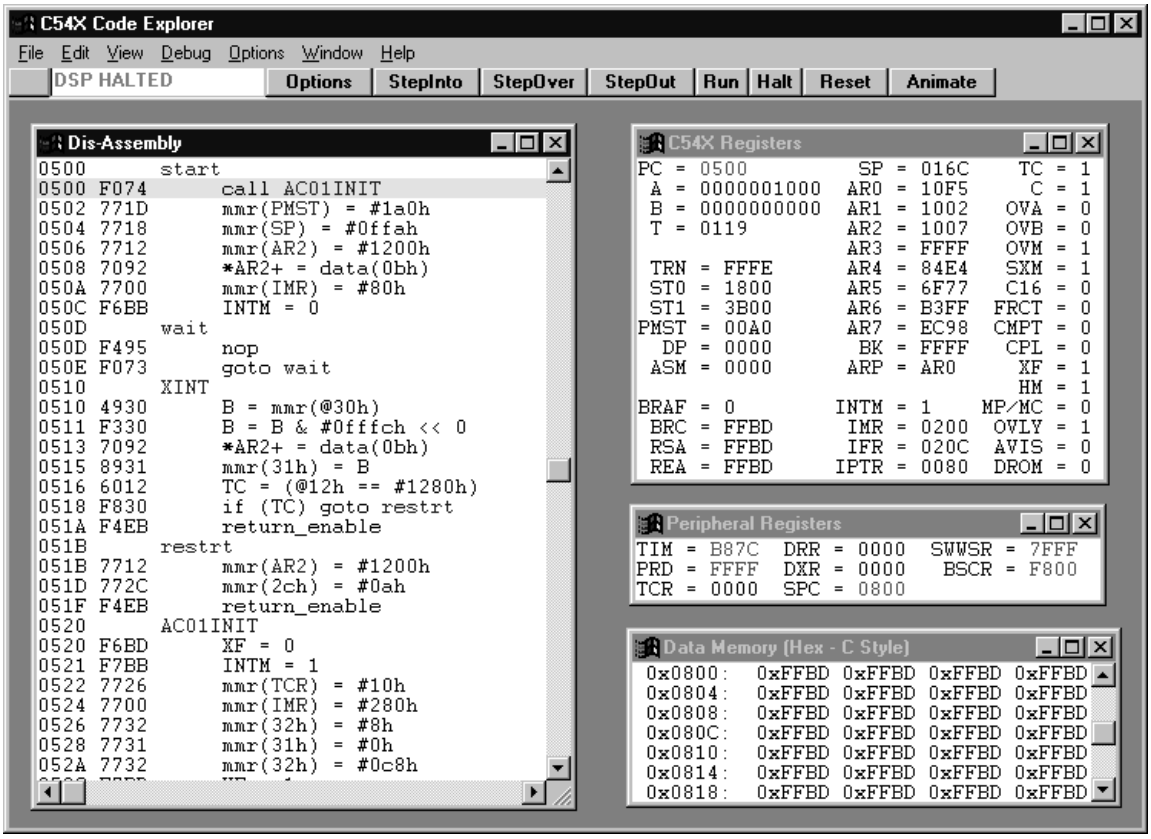
Figure 2–2. Code Explorer Port Selection Dialog Box



The debugger lists all available parallel ports in the dialog box. You may select one of the ports listed or type in the desired port I/O address to override the existing address. The correct port will start the debugger interface. See Section 3.1, *Code Explorer Debugger,* page 3-2 for more information.

Provided you have properly installed the hardware and software, the Code Explorer debugger interface is displayed on your screen as shown in Figure 2–3.

Figure 2–3. Code Explorer Debugger Interface



If an error occurs when you attempt to start the debugger, it may be due to the hardware setup. To test the hardware setup, run the self-test program.

2.3 Running the Self-Test Program

The self-test program helps you to determine the cause of errors. The self-test program performs several tests on the parallel port, the DSKplus interface logic, the ’C54x HPI, the ’C54x DSP, and the ’AC01.

The tests performed, in order, are as follows:

1) **Port locator.** Checks all parallel ports to determine which are connected to the DSKplus board.

2) **Continuity check.** Checks for open data lines and shorts between data lines.

3) **PAL® state machine test.** Checks nibble mode functionality and the PAL® clock.

4) **Latch mode test.** Verifies that the latch mode of the PAL® is operating correctly, and brings the PAL® out of 3-state mode.

5) **HPIC verification.** Checks the HPI control register configuration.

6) **HPIA verification.** Checks the address in the HPI address register and HPIA mode.

7) **DATA (increment) verification.** Checks the data increment mode of the HPI.

8) **DATA (static) verification.** Checks the no-increment mode of the HPI.

9) **Port mode analysis.** Determines the parallel port’s configuration (4-bit unidirectional or 8-bit bidirectional).

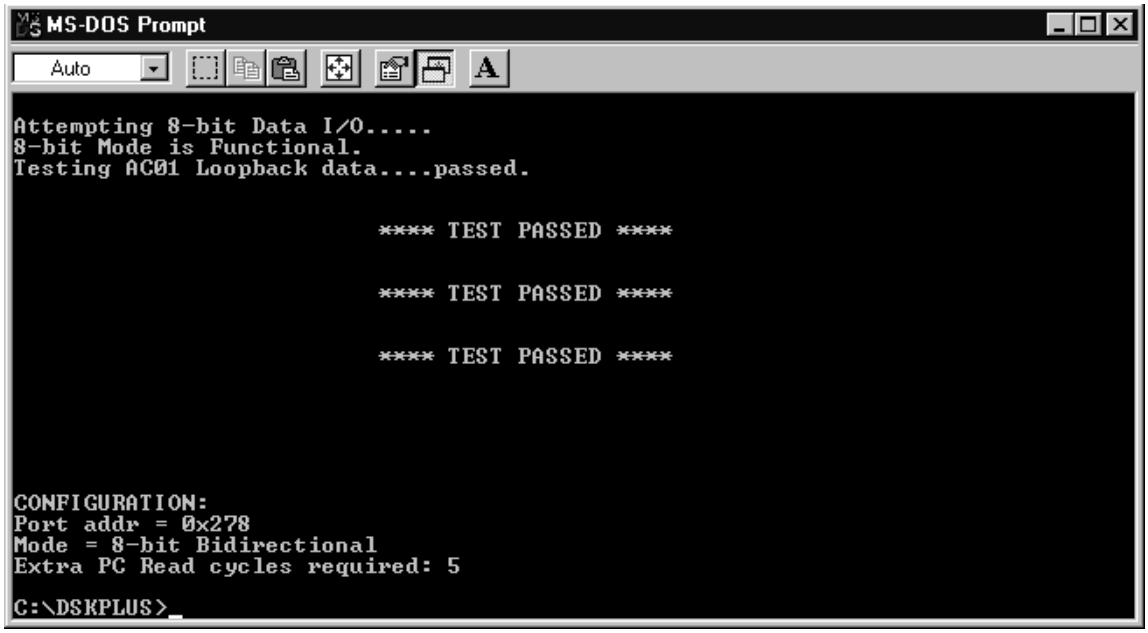
10) **’AC01 test.** Performs ’AC01 register programming. Checks the analog final output data via loopback mode.

To start the self-test, click on the DOS icon to access a DOS prompt and type:

```
SELFTEST.EXE
```

The testing script appears on the screen as shown in Figure 2–4 on page 2-6. If any errors occur the execution is halted.

Figure 2–4. Self-Test Script



If an error occurs during the self-test, read the error script completely and confirm the following:

- ❑ DSKplus power is on, indicated by illumination of the green LED.
- ❑ DSKplus is firmly connected to the PC via the printer cable.

The self-test program is somewhat redundant to test for several different causes of errors. For example, the port locator writes a 0xF0 to the data register and looks for the bit (high nibble) in the status register. If this case is true, it loads the data register with 0x0 and examines the status once again. If this case passes, it assumes the DSKplus board is attached to the port. If it fails, it will try the next port. However, a false reply of NO CONNECT occurs if any of the high four bits are open or shorted to ground. When the test passes and a port is located, it is still not known if any of the data lines are shorted to one another. The continuity check performs adjacent data line continuity testing.

The system setup can be responsible for problems connecting to the DSKplus board. For example, in Windows 95 the DSKplus software does not work if the parallel port is being “captured” by Windows. You must go into the system setup and make sure the port is *not* captured. A common error is to have a printer set up to print from DOS-based programs. This captures the port, making it inaccessible to DSKplus applications.

Another source of errors is the PC port configuration. It is suggested that you reboot your PC and enter the BIOS setup routine. Confirm that the BIOS parallel port setup does not specify extended capabilities port (ECP) or enhanced parallel port (EPP). If either is specified, change it to a standard port.

Accesses to the parallel port can vary in speed from machine to machine. The self-test program ends with information of which you may want to take note if you plan to write custom host PC applications. This information includes the port base address, the operating mode (either 4-bit unidirectional or 8-bit bidirectional) and additional CPU cycles needed for reading from the port. The extra CPU cycles may be needed for reads, because the data lines become valid after the RC time constant on the data lines. Self-test calculates how many host PC CPU cycles are required.

Chapter 3

DSKplus Debugger and Application Loader Software

The DSKplus lets you experiment with and use a DSP for real-time signal processing. The DSKplus 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 a number of ways.

The DSKplus debugger works with the assembler and application loader to help you develop, test, and refine DSKplus assembly language programs. This chapter describes the features of the debugger and how to use the application loader software.

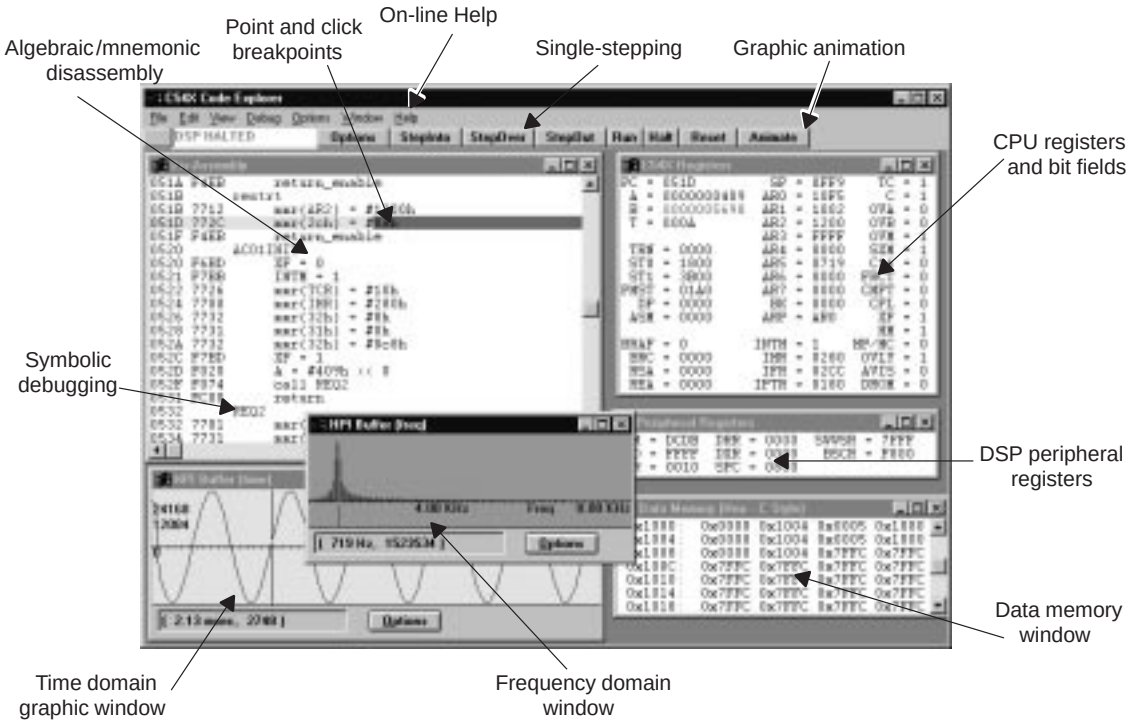
Topic **Page**

3.1 Code Explorer Debugger

The Windows-based debugger included with the DSKplus is a windowed debugger interface developed by GoDSP and is shown in Figure 3–1. It contains four default windows: disassembly, CPU registers, peripheral registers, and data memory windows.

The disassembly window shows the DSP code and address location. The location of the DSP program counter (PC) is highlighted by a yellow line overlaying the code. The interface also supports symbolic debugging, which makes debugging code much easier. You can reference locations in code and code variables by the assembly name or label, so you do not need to know the physical address. Breakpoints can be added or deleted by pointing and clicking on the instruction for the operation you would like to break. Disassembly window properties can be changed using your menu and select buttons.

Figure 3–1. Debugger Overview



Note: Watch windows can be set up to watch variables, system stack, or any other memory location. Files can be connected to probe points within your code.

The CPU registers window allows you to view the internal registers and important bit fields of the DSP. To change a value of a register, point and click on the register and type in the new value.

The peripheral register window is like the CPU register window, except that it includes only the registers that are used for the DSP peripherals, such as the serial ports.

The data memory window is a default data memory window. The starting address and length can be defined using your select button. Multiple data memory windows can be displayed, allowing you to view any variable, such as the system stack or assembly variables. Using the data memory window properties screen you can rename the window to reflect the variable name.

The tool bar on top of the screen includes buttons for single-stepping, running, and resetting the DSKplus board. These buttons allow you to step over or into functions. The animation button supports a graphical representation of a variable or buffer. The data can be viewed in either the time domain or the frequency domain.

Code Explorer probe points are used to connect hard-disk drive files to points within your application code. Once connected, these files can be used as inputs or outputs to your code. To set a probe point, position the cursor over the instruction and click your right mouse button. To set/reset the probe point select *toggle probe point*. After the probe point is set, specific attributes must be assigned in the probe point window.

The debugger's on-line help is accessed through a button on the interface. It can be helpful in providing answers to common questions you may have while you are using the tool.

3.2 Using the Application Loader

The application loader, called LoadApp, loads your application code to the DSP memory and starts executing it. LoadApp loads the kernel to 0x80h – 0x17Fh and then proceeds to load the application code. The general form of the command to load an application is:

loadapp –a c:\path\appfile.obj –e label [options]

This command loads appfile.obj to the DSP and begins executing the code at *label*. The label must be a valid label in the assembly source files (.asm).

Available command line options:

–a specifies an application file. Immediately followed by a space and the path and filename.

–bx Specifies the port MODE: x = 4: 4-bit mode
x = 8: 8-bit mode

–px Forces the line printer (LPT) port to a specific number where x is either 1, 2, or 3.

–e Specifies the starting location for execution in the DSP. The option is followed by a space and a valid label or address. If you use an address, it can be in either 0x0 or 0h format. Labels are case sensitive.

–k loads an alternate kernel. The kernel must be one contiguous section (no multiple-section kernels) and cannot exceed 7F6h in length. The kernel path and filename must directly follow this command line option.

–? lists the options on the screen.

After the code has been loaded, the system exits LoadApp and restores the DOS prompt. The loader returns the PAL® device to an uninitialized state, so you must make sure you reinitialize the PAL® device if you have a PC-based application interfacing with the DSKplus board.

Chapter 4

Software Considerations

DSP code is the program you create and eventually load into the resident DSP processor. To create DSP code, you must have an ASCII text editor and the *TMS320C54x DSP Reference Set, Volume 3: Algebraic Instruction Set.* The assembly source code you create in the editor must be assembled using the ’C54x DSKplus algebraic assembler. The algebraic assembler converts your source code (.asm) file to machine code (.obj file) that only the DSP can use.

Host PC application code is a program you create with one of the many PC-based C or C++ compilers. These compilers generate machine code for the PC CPU (PC resident); this machine code will not run on the DSP. The debugger, and application loader are perfect examples of executable PC code and are used to load DSP code to the DSP CPU.

Normally, a DSP application begins with the creation of the DSP code, followed by the creation of the host PC application code (if needed). This chapter describes software considerations you must make before creating DSP and/or host PC application code for the ’C54x DSKplus board.

Topic **Page**

4.1 DSP Software

When creating software applications for the DSP processor, you need an ASCII text editor to create source code, the DSKplus algebraic assembler to generate DSP object file (machine code), and the debugger interface to examine the results. This section illustrates the process of combining the DSP application code with the DSKplus board and assembler.

The DSKplus software includes a simple application that takes data from the ’AC01 and places it in a buffer. To view the source code, open the file named firstapp.asm located in the firstapp subdirectory of the DSKplus software and load it into your ASCII editor.

The source code contains two sections: .text and vectors. The .text section includes all of the executable code that gets data from the ’AC01 and places it into the buffer. The second section, called vectors, contains the vector location where the DSP should receive data from the ’AC01. Each time the ’AC01 transfers a data word to the DSP, the DSP goes to the vector for the serial port interrupt service routine associated with the ’AC01.

By using the .setsect directive, you can set the code section to a particular address and page. The .text section resides at 500h and the vectors section resides at 180h, as shown in the file’s statements:

```
.setsect ”.text”,   0x500,0
.setsect ”vectors”, 0x180,0
```

The page determines which memory space the section will be loaded to. The page indicator is either 0 or 1, corresponding to program or data space, respectively.

The .copy directive copies the source code from the file name enclosed in double quotes. For the following code, shown in full on your PC editor screen it copies vectors.asm into the vectors section and appends the ac01init.asm file to the .text section. Therefore, the assembly source code actually contains code from the three files: firstapp.asm, vectors.asm, and ac01init.asm.

```
          .sect ″vectors″
          .copy ″c:\dskplus\inits\vectors.asm″

start:
          .text
          <initialize DSP>
          <wait>

XINT:
          <interrupt service routine>
          <return>

          .copy ″C:\dskplus\inits\ac01init.asm″
```

The program begins at the label start, where it initializes the DSP and continues in a wait routine. The initialization routine must always set up the ’AC01(if you plan to use it), the interrupt table pointer (IPTR), the stack pointer (SP), and the interrupt mask register (IMR). In this code, the ar2 register is also initialized to the beginning of the data buffer at 1200h. The interrupt mask bit (INTM) is set to 0 when initialization is complete and the DSP is ready to receive data.

```
start:
          call AC01INIT
          pmst = #01a0h       ; set up iptr
          sp = #0ffah         ; init stack pointer
          ar2 = #1200h        ; pointer to receive buffer
          *ar2+ = data(#0bh)  ; store to rcv buffer
          imr = #280h
          intm = 0            ; ready to rcv int’s

          wait      nop
                    goto   wait
```

When the DSP receives an interrupt, it proceeds to the vectors section and reads the vector location. The vector informs the DSP to go to the XINT subroutine (also known as the XINT interrupt service routine). As a result, the code in the XINT routine is executed.

```
XINT:
          b = trcv            ; load acc b with input
          b = #0FFFCh & b
          *ar2+ = data(#0bh)  ; store to rcv buffer
          tdxr = b            ; transmit the data.
          TC = (@ar2 == #01280h)
          if (TC) goto restrt ; stop if rcv buffer is at 1280h
          return_enable
restrt
          ar2 = #1200h
          return_enable       ; used only when not using
                              debugger
          .end
```

The interrupt service routine gathers data and copies it to the data buffer and transmits it back to the ’AC01 until the buffer is full. When the buffer is full, the DSP enters the routine restrt and initializes the ar2 buffer pointer to 1200h to begin again.

You can use the DSKplus algebraic assembler to assemble this code by typing the following at a DOS prompt:

```
dskplasm c:\dskplus\firstapp\firstapp.asm –l
```

This creates a file firstapp.obj, which you can load into the debugger and examine. The data values received via the ’AC01 are loaded into the DSP buffer at location 1200h.

4.2 DSP Programming Tips

The following tips can help you develop your application code faster and more efficiently.

1) The stack pointer (SP) must be initialized. Choose a memory location that allows the SP to grow with your application.

2) The interrupt mask register (IMR) must always have the HPI interrupt enabled so that the debugger can communicate with the DSP's communication kernel (IMR = 200h only if you are using the debugger or the loader).

3) Memory from 80h–17Fh is reserved for the kernel. Memory from 1000h–1009h is reserved for the communication buffer.

4) Always have INT2 masked in the IMR register. The HINT pin is tied to INT2 to perform an HPI boot at power up. Enable this interrupt *only* if you want the DSP to interrupt itself when the DSP sets the HINT bit in the HPIC. Otherwise, keep it 0 in the IMR.

5) TRAP 2 is reserved for the kernel (only when using the debugger). There are many software interrupts to choose from.

4.3 Host PC Software

Creating host PC software requires that you have a PC-based C or C++ compiler capable of generating machine code for your PC. This code can be used to create various applications that allow you to see the results of your DSP code. In this section, you take firstapp.obj to the next step: you load the data buffer back to the PC and display the data on the PC screen. To accomplish this task, the host application must know how to communicate with the DSP using the communication protocol. Transferring data to and from the PC is quite simple using the host interface library functions. This library contains all of the required functions to communicate to the DSKplus board via the parallel port and is found in the C54XHIL directory. The C++ source code hostapp.cpp is located in the firstapp subdirectory.

To compile the hostapp.cpp code correctly, load hostapp.cpp into your C or C++ editor and make sure your project or make file includes the following files:

- HI54X.H (HIL include file)
- HI54X.CPP (HIL linkable function files)

After compiling the hostapp.cpp source file, you can write a batch file to run the DSP and host PC code simultaneously. First, load the DSP code into the DSP memory using the LoadApp program, then start the hostapp.exe program. The batch file would be similar to:

```
loadapp –a c:\path\firstapp\firstapp.obj –e start
hostapp.exe
```

Be sure to reassemble firstapp.asm with instruction *hpic = #0ah* added into the source file as follows:

```
restrt
          ar2 = #1200h
          hpic = #0ah
          return_enable
```

By adding this line, the DSP generates a DSP-to-host interrupt when the buffer is full. This is the same interrupt the debugger uses to communicate with DSKplus board. Therefore, it cannot be used with the debugger software.

4.4 Host Programming Tips

When you are using C54XHIL, be sure to keep the following variables external. These variables affect functions throughout the C54XHIL.

- extern int pport, portmode, Readdelay
- extern int datareg[pport]
- extern int statreg[pport]
- extern int ctrlreg[pport]

The first three of these variables are used to set up the port number (pport), the parallel port mode (portmode), and the delay for 8-bit reads (Readdelay). The port number, pport, is 1, 2, or 3 to select the corresponding port; portmode is either 0 or 1, to identify 4-bit or 8-bit mode, respectively. The Readdelay variable is needed in cases where the host PC can read data from the data register before it is validated from the DSKplus board. Readdelay is the value of PC CPU cycles required before the information on the DSKplus data lines is valid.

The next three variables are the data, status, and control register addresses of the three common parallel ports. The datareg is the data register where data is loaded to and from the PC and DSKplus. The statreg is the status register and is used by the host PC to read data in 4-bit mode and receive DSP-to-host interrupts. The ctrlreg is the control register and is used to control the DSKplus board via the host interface logic and send host-to-DSP interrupts.

See the C54XHIL.DOC file in the C54XHIL subdirectory for a complete reference list of the host interface library functions.

Chapter 5

DSKplus Assembler Description

The DSKplus assembler translates assembly language source (.asm) files into machine language object (.obj) files. Source files can contain the following assembly language elements:

- Assembler directives
- Assembly language instructions

This chapter explains how to invoke the assembler and discusses source statement format, valid constants and expressions, and assembler output.

Topic **Page**

5.1 DSKplus Assembler Overview

The DSKplus assembler is a two-pass program that performs the following tasks:

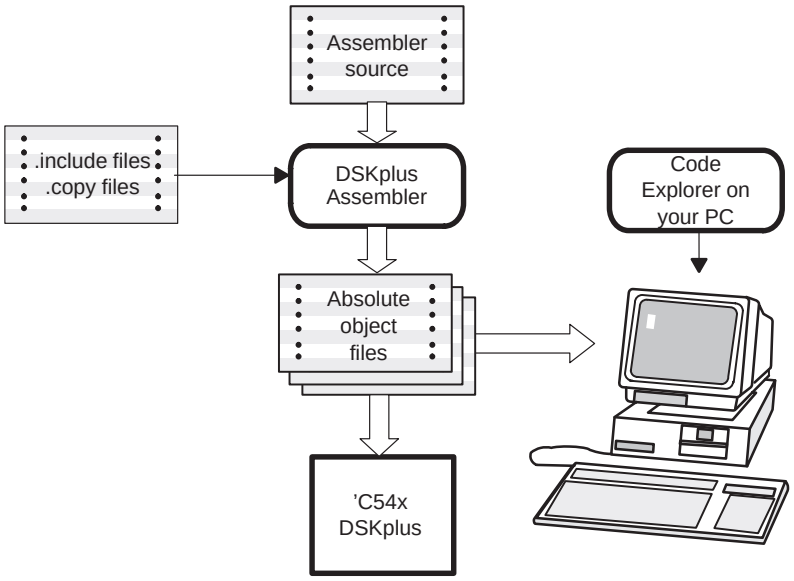
- Processes the source statements in a text file to produce an absolute object file
- Produces a source listing (if requested) and provides you with control over this listing. The section program counter (SPC) is the absolute address of that opcode.
- Defines and references symbols
- Assembles conditional blocks

This assembler allows you to segment your code into sections and maintain an SPC for each section of object code.

5.2 DSKplus Assembler Development Flow

Figure 5–1 illustrates the assembler's role in the software development flow. The DSKplus assembler accepts assembly language source files as its input and produces executable code (object files) as its output.

Figure 5–1. DSKplus Assembler in the Software Development Flow



5.3 Invoking the DSKplus Assembler

To invoke the DSKplus assembler, enter the following:

dskplasm [*input file* [*object file* [*listing file*]]] [*–options*]

dskplasm is the command that invokes the assembler.

input file names the assembly language source file. If you do not supply an extension, the assembler uses the default extension *.asm*. If you do not supply an input filename, the assembler prompts you for one.

object file names the object file that the assembler creates. If you do not supply an extension, the assembler uses *.obj* as a default. If you do not supply an object file, the assembler creates a file that uses the input filename with the *.obj* extension.

listing file names the optional listing file that the assembler can create. If you do not supply a listing filename, *the assembler does not create one* unless you use the –l (lowercase L) option. In this case, the assembler uses the input filename. If you do not supply an extension, the assembler uses *.lst* as a default.

options identifies the assembler options that you want to use. Options are not case sensitive and can appear anywhere on the command line following the command. Precede each option with a hyphen. You can combine single-letter options without parameters: for example, –lc is equivalent to –l –c. Options that have parameters, such as –i, must be specified separately.

- **–c** makes case insignificant in the assembly language files. For example, –c will make the symbols ABC and abc equivalent. *If you do not use this option, case is significant* (default).
- **–d** **–d***name* [=*value*] sets the *name* symbol. This is equivalent to inserting *name* **.set** [*value*] at the beginning of the assembly file. If *value* is omitted, the symbol is set to 1. For more information, see subsection 5.8.2, page 5-14.
- **–i** **–i***pathname* specifies a directory where the assembler can find files named by the .copy or .include statements. You can specify up to 10 directories in this manner; each pathname must be preceded by the –i option. For more information, see subsection 5.4.1, page 5-5.
- **–l** (lowercase L) produces a listing file.
- **–q** (quiet) suppresses the banner and all progress information.

5.4 Naming Alternate Directories for Assembler Input

The .copy and .include directives tell the assembler to use code from external files and to read source statements from another file. The syntax for these directives is:

```
.copy "filename"
.include "filename"
```

The *filename* names a copy/include file that the assembler reads statements from. The filename may be a complete pathname, a partial pathname, or a filename with no path information. The assembler searches for the file in the following location in the order listed:

1) 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.
2) Any directories named with the –i assembler option
3) Any directories set with the environment variable A_DIR

You can augment the assembler's directory search algorithm by using the –i assembler option or the A_DIR environment variable.

5.4.1 –i Assembler Option

The –i assembler option names an alternate directory that contains copy/include files. The format of the –i option is as follows:

```
dskplasm –ipathname
```

You can use up to 10 –i options per invocation; each –i option names one pathname. In assembly source, you can use the .copy or .include directive without specifying path information. If the assembler doesn't find the file in the directory that contains the current source file, it searches the paths designated by the –i options.

For example, assume that a file called source.asm is in the current directory; source.asm contains the following directive statement:

```
.copy "copy.asm"
```

Assume that the file is stored in the directory

c:\320tools\files\copy.asm

Your assembly invocation is:

```
dskplasm –ic:\320tools\files\source.asm
```

The assembler first searches for copy.asm in the current directory because source.asm is in the current directory. Then the assembler searches in the directory named with the –i option.

5.4.2 A_DIR Environment Variable

An environment variable is a system symbol that you define and assign a string to. The assembler uses the environment variable **A_DIR** to name alternate directories that contain copy or include files. The command for assigning the environment variable is:

```
set A_DIR=pathname;another pathname ...
```

The *pathnames* are directories that contain copy or include files. You can separate the pathnames with a semicolon or with blanks. In assembly source, you can use the .copy or .include directives without specifying path information in these statements and use the A_DIR path list to give the assembler the alternate paths. If the assembler doesn't find the copy/include file in the directory that contains the current source file or in directories named by –i, it searches the paths named by the environment variable.

For example, assume that a file called source.asm contains these statements:

```
.copy "copy1.asm"
.copy "copy2.asm"
```

Assume that the files are stored in the following directories:

c:\320tools\files\copy1.asm
c:\dsys\copy2.asm

You could set up the search path with these commands:

```
set A_DIR=c:\dsys
DSKPLASM –ic:\320tools\files source.asm
```

The assembler first searches for copy1.asm and copy2.asm in the current directory because source.asm is in the current directory. Then the assembler searches in the directory named with the –i option and finds copy1.asm. Finally, the assembler searches the directory named with A_DIR and finds copy2.asm.

The environment variable remains set until you reboot the system or reset the variable by entering:

```
set A_DIR =
```

5.5 Source Statement Format

TMS320C54x assembly language source programs consist of source statements that can contain assembler directives, assembly language instructions, and comments. The format of the source file determines how long the source statement can be, but the assembler reads a maximum of 200 characters per line, so keep your source statements to 200 characters or less. If a statement contains more than 200 characters, the assembler truncates the line and issues a warning. If you are not concerned with the comments in your source file, you can allow them to be truncated, but the operational portion of the statements must be kept to a maximum of 200 characters.

Follow these guidelines in writing assembly language source statements:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional; if used, they must begin in column 1.
- One or more blanks must separate each field. 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.

The following are examples of source statements:

```
SYM1          .set      2             ; Symbol SYM1 = 2.
Begin:         AR1 = #SYM1            ; Load AR1 with 2.
              .word     016h          ; Initialize word (016h).
```

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

[*label*] [:] instruction [;*comment*]

5.5.1 Label Field

Labels are optional for all assembly language instructions and for most assembler directives (except for .set and .equ, which require labels). When a label is used, it must begin in column 1 of a source statement. A label can contain up to 32 alphanumeric characters (A–Z, a–z, 0–9, _, and $). Labels are case sensitive unless –c is used in invoking the assembler, and the first character cannot be a number. A label can be followed by a colon (:); the colon is not treated as part of the label name. If you don't use a label, the first character position must contain a blank, a semicolon, or an asterisk.

When you use a label, its value is the current value of the section program counter (the label points to the statement it is associated with). For example,

if you use the .word directive to initialize several words, a label would point to the first word. In the following example, the label Start has the value 40h.

```
 .      .           .            .
 .      .           .            .
 .      .           .            .
 9     003F         ; Assume some other code was assembled.
10     0040   000A  Start:  .word 0Ah,3,7
       0041   0003
       0042   0007
```

A label on a line by itself is a valid statement. The label assigns the current value of the section program counter to the label; this is equivalent to the following directive statement:

```
label .set $  ; $ provides the current value of the SPC.
```

When a label appears on a line by itself, it points to the instruction on the next line (the SPC is not incremented):

```
3   0050            Here:
4   0050   0003        .word 3
```

5.5.2 Instruction Field

The instruction field follows the label field. The instruction field must not start in column 1; if it does, it will be interpreted as a label. The instruction field can contain one of the following opcodes:

- Algebraic instruction (such as B = B + 4123h)
- Assembler directive (such as .data, .list, .set)

5.5.3 Operands

An operand can be a constant (see Section 5.6, page 5-10), a symbol (see Section 5.8, page 5-13), or a combination of constants and symbols in an expression (see Section 5.9, page 5-15).

- **Operand prefixes for instructions**

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

  - **# prefix — the operand is an immediate value**. If you use the # sign as a prefix, the assembler treats the operand as an immediate value. This is true even when the operand is a register or an address; the assembler treats the address as a value instead of using the contents of the address. This is an example of an instruction that uses an operand with the # prefix:

    ```
    Label:  B = B + #123
    ```

The operand #123 is an immediate value. The assembler adds 123 (decimal) to the contents of accumulator B.

- **@ prefix — the operand is direct-memory addressed**. When using the @ prefix, the operand is addressed via the direct-addressing mode. The address is a function of data pointer or stack pointer. The instruction below adds 10 to AR2 only if DP = 0.

  ```
  Label:  @AR2 += #10
  ```

- *** prefix — the operand is an indirect address.** If you use the ***** sign as a prefix, the assembler treats the operand as an indirect address; that is, it uses the contents of the operand as an address. This is an example of an instruction that uses an operand with the * prefix:

  ```
  Label:  A = *AR4
  ```

  The instruction directs the assembler to go to the address specified by the contents of register AR4 and move the contents of that location to accumulator A.

❑ **Immediate value for directives**

The immediate value mode uses the # character in front of the immediate value and is primarily used with instructions. In some cases, it can also be used with the operands of directives.

It is not usually necessary to use the immediate value mode for directives. Compare the following statements:

```
A = A + #10
.byte 10
```

In the first statement, the immediate value mode is necessary to tell the assembler to add the value 10 to accumulator A. In the second statement, however, immediate value is not used; the assembler expects the operand to be a value and initializes a byte with the value 10.

5.5.4 Comment Field

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.

A source statement that contains only a comment is valid. If it begins in column 1, it can start with a semicolon (;) or asterisk (*). Comments that begin anywhere else on the line must begin with a semicolon. The asterisk identifies a comment only if it appears in column 1.

5.6 Constants

The assembler supports six types of constants:

- Binary integer
- Octal integer
- Decimal integer
- Hexadecimal integer
- Character
- Assembly time

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.

5.6.1 Binary Integers

A binary integer constant is a string of up to 16 binary digits (0s and 1s) followed by the suffix B (or b). If fewer than 16 digits are specified, the assembler right-justifies the value and zero-fills the unspecified bits. These are examples of valid binary constants:

00000000B	Constant equal to 0_{10} or 0_{16}
0100000b	Constant equal to 32_{10} or 20_{16}
01b	Constant equal to 1_{10} or 1_{16}
11111000B	Constant equal to 248_{10} or $0F8_{16}$

5.6.2 Octal Integers

An octal integer constant is a string of up to six octal digits (0 through 7) followed by the suffix Q (or q). Octals can also be any constant prefixed with a '0' without a suffix. These are examples of valid octal constants:

10Q	Constant equal to 8_{10} or 8_{16}
100000Q	Constant equal to $32\ 768_{10}$ or $8\ 000_{16}$
226q	Constant equal to 150_{10} or 96_{16}

5.6.3 Decimal Integers

A decimal integer constant is a string of decimal digits, ranging from –32 768 to 32 767 or 0 to 65 535. These are examples of valid decimal constants:

1000	Constant equal to 1000_{10} or $3E8_{16}$
–32 768	Constant equal to $–32\ 768_{10}$ or $8\ 000_{16}$
25	Constant equal to 25_{10} or 19_{16}

5.6.4 Hexadecimal Integers

A hexadecimal integer constant is a string of up to four hexadecimal digits followed by the suffix H (or h). Hexadecimal digits include the decimal values 0–9 and the letters A–F and a–f. A hexadecimal constant must begin with a decimal value (0–9), it may also be prefixed with 0x. If fewer than four hexadecimal digits are specified, the assembler right-justifies the bits. 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 $37AC_{16}$
0x37AC	Constant equal to 14 252_{10} or $37AC_{16}$

5.6.5 Character Constants

A character constant is a string of one or two characters enclosed in *single* quotes. The characters are represented internally as 8-bit ASCII characters. Two consecutive single quotes are required to represent each single quote that is part of a character constant. A character constant consisting only of two single quotes is valid and is assigned the value 0. If only one character is specified, the assembler right-justifies the bits. These are examples of valid character constants:

'a'	Represented internally as 61_{16}
'C'	Represented internally as 43_{16}
''D'	Represented internally as 2744_{16}

Note the difference between character constants and character strings (Section 5.7, page 5-12, discusses character strings). A character constant represents a single integer value; a string is a list of characters.

5.6.6 Assembly-Time Constants

If you use the .set directive to assign a value to a symbol, the symbol becomes a constant. To use this constant in expressions, the value that is assigned to it must be absolute. For example:

```
shift3    .set   3
           A = #shift3
```

You can also use the .set directive to assign symbolic constants for register names. In this case, the symbol becomes a synonym for the register:

```
AuxR1     .set   AR1
           SP = AuxR1
```

5.7 Character Strings

A character string is a string of characters enclosed in *double* quotes. A double quote that is part of a character string is represented by two consecutive double quotes. The maximum length of a string varies and is defined for each directive that requires a character string. Characters are represented internally as 8-bit ASCII characters.

These are examples of valid character strings:

”sample program”	defines the 14-character string *sample program.*
”PLAN ””C””””	defines the 8-character string *PLAN ”C”.*

Character strings are used for the following:

- Filenames, as in .copy ”filename”
- Section names, as in .sect ”section name”
- Section address initializers, as in .setsect ”section name”
- Data initialization directives, as in .byte ”charstring”
- Operands of .string directives

5.8 Symbols

Symbols are used as labels, constants, and substitution symbols. A symbol name is a string of up to 32 alphanumeric characters (A–Z, a–z, 0–9, $, and _). The first character in a symbol cannot be a number, and symbols cannot contain embedded blanks. The symbols you define are case sensitive; for example, the assembler recognizes ABC, Abc, and abc as three unique symbols. You can override case sensitivity with the –c assembler option. A symbol is valid only during the assembly in which it is defined.

5.8.1 Labels

Symbols used as labels become symbolic addresses associated with locations in the program. Labels used locally within a file must be unique. Opcodes and assembler directive names (without the . prefix) are valid label names.

Labels can also be used as the operands of the .bss directive; for example:

```
          .bss   label1, 1

label2    NOP
          B = B + @label1
          goto label2

K      .set   1024            ;constant definitions
maxbuf .set   2*K
```

The assembler also has several predefined symbolic constants; these are discussed in the next section.

5.8.2 Defining Symbolic Constants (–d Option) on the Command Line

The –d option equates a constant value with a symbol. The symbol can then be used in place of a value in assembly source. The format of the –d option is as follows:

dskplasm –d*name*=[*value*]

The *name* is the name of the symbol you want to define. The *value* is the value you want to assign to the symbol. If the *value* is omitted, the symbol will be set to 1.

Within assembler source, you may test the symbol with the following directives:

Type of Test	Directive Usage
Existence	`.if $isdefed("name")`
Nonexistence	`.if $isdefed("name") = 0`
Equal to value	`.if name = value`
Not equal to value	`.if name != value`

The argument to the $isdefed built-in function must be enclosed in quotes. The quotes cause the argument to be interpreted literally rather than as a substitution symbol.

5.8.3 Predefined Symbolic Constants

The assembler has several predefined symbols, including the following:

- **$**, the dollar sign character, represents the current value of the section program counter (SPC).
- **Register symbols**, including AR0 – AR7

5.9 Expressions

An expression is a constant, a symbol, or a series of constants and symbols separated by arithmetic operators. The range of valid expression values is –32 768 to 32 767. Three main factors influence the order of expression evaluation:

Parentheses

Expressions that are enclosed in parentheses are always evaluated first.

8 / (4 / 2) = 4, but 8 / 4 / 2 = 1

You *cannot* substitute braces ({ }) or brackets ([ ]) for parentheses.

Precedence groups

The 'C54x algebraic assembler uses similar precedence as does the C language. When parentheses do not determine the order of expression evaluation, the highest precedence operation is evaluated first. Shifts (<< >>) have higher precedence than (+ –).

8 + 4 / 2 = 10 (4 / 2 is evaluated first)

Left-to-right evaluation

When parentheses and precedence groups do not determine the order of expression evaluation, the expressions are evaluated as happens in the C language.

8 / 4 * 2 = 4 , but 8 / (4 * 2) = 1

5.9.1 Operators

Table 5–1 lists the operators that can be used in expressions.

Table 5–1. Operators Used in Expressions (Precedence)

Symbols	Operators	Evaluation
+ – ~	Unary plus, minus, 1s complement	Right to left
* / %	Multiplication, division, modulo	Left to right
<< >>	Left shift, right shift	Left to right
+ –	Addition, subtraction	Left to right
< <= > >=	Less than, LT or equal, greater than, GT or equal	Left to right
!=, =[=]	Not equal to, equal to	Left to right
&	Bitwise AND	Left to right
^	Bitwise exclusive OR	Left to right
|	Bitwise OR	Left to right

Note: Unary +, –, and * have higher precedence than the binary forms.

5.9.2 Expression Overflow and Underflow

The assembler checks for overflow and underflow conditions when arithmetic operations are performed at assembly time. It issues a Value Truncated warning whenever an overflow or underflow occurs. The assembler *does not* check for overflow or underflow in multiplication.

5.9.3 Well-Defined Expressions

Some assembler directives require well-defined expressions as operands. Well-defined expressions contain only symbols or assembly-time constants that are defined before they are encountered in the expression. The assembler is absolute therefore, all defined expressions are absolute.

This is an example of a well-defined expression:

```
1000h+X
```

where x was previously defined.

5.9.4 Conditional Expressions

The assembler supports relational operators that can be used in any expression; they are especially useful for conditional assembly. Relational operators include the following:

=	Equal to	**= =**	Equal to
! =	Not equal to		
<	Less than	**< =**	Less than or equal to
>	Greater than	**> =**	Greater than or equal to

Conditional expressions evaluate to 1 if true and 0 if false; they can be used only on operands of equivalent types.

5.10 Source Listings

A source listing shows source statements and the object code they produce. To obtain a listing file, invoke the assembler with the –l (lowercase L) option.

Two banner lines, a blank line, and a title line are at the top of each source listing page. Any title supplied by a .title directive is printed on the title line; a page number is printed to the right of the title. If you don't use the .title directive, the name of the source file is printed. The assembler inserts a blank line below the title line.

A source statement produces at least one word of object code. The assembler lists the SPC value and object code on a separate line for each word of object code produced. Each additional line is listed immediately following the source statement line. Each line in the source file produces a line in the listing file that shows a source statement number, an SPC value, the object code assembled, and the source statement.

Field 1: source statement number

Line Number

The source statement number is a decimal number. The assembler numbers source lines as it encounters them in the source file. Some statements increment the line counter but are not listed; for example, .title statements and statements following a .nolist are not listed. The difference between two consecutive source line numbers indicates the number of intervening statements in the source file that are not listed.

Include File Letter

The assembler may precede a line number with a letter; the letter indicates that the line is assembled from an included file.

Nesting Level Number

The assembler may precede a line number with a number; the number indicates the nesting level of loop blocks.

Field 2: section program counter value (absolute target address)

This field contains the section program counter (SPC) value, which is hexadecimal. All sections (.text, .data, .bss, and named sections) maintain separate SPCs. Some directives do not affect the SPC and leave this field blank. Each section's SPC value can be initialized using section-address initializers (that is, .setsect "section name").

Field 3: object code

This field contains the hexadecimal representation of the object code. All machine instructions and directives use this field to list object code.

Field 4: source statement field

This field contains the characters of the source statement as they were scanned by the assembler. The assembler accepts a maximum line length of 200 characters. Spacing in this field is determined by the spacing in the source statement.

Example 5–1 shows an assembler listing with each of the four fields identified.

Example 5–1. Assembler Listing

```
       1
       2
       4                    .setsect ”vecs”, 080h
       5                    .setsect ”.text”, 0500h
       6                    .setsect ”.bss”, 0600h
       7              ***********************************************************
       8              *              Reserve space for a variable               *
       9              ***********************************************************
      10 000600                .bss    reserve, 1
      11              ***********************************************************
      12              *               Reset and Interrupt Vectors
      13              ***********************************************************
      14 000080                .sect   ”vecs”
      15 000080 F073  RESET:   goto    main
         000081 0104
      16 000082 F073  NMI:     goto    NOPS
         000083 0100
      17              ***********************************************************
      18              *                        Main Program
      19              ***********************************************************
      20                       .copy   ”4nops.inc”
A      1 000500                .text
A      2 000500 F495  NOPS:    nop           ; NOPS function begins here
A      3 000501 F495           nop
A      4 000502 F495           nop
A      5 000503 F495           nop
A      6
      21 000504 771D  main     PMST = #00a0h
         000505 00A0
      22                       : : :
                               : : :
\_____/ \__/ \__/  \_____________________________________________/
Field 1  Field 2 Field 3                  Field 4
```

5.11 DSKplus Assembler Directives

Assembler directives supply data to the program and control the assembly process. Assembler directives enable 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
- ❏ Initializing the addresses for specified sections
- ❏ Assemble conditional blocks
- ❏ Examine symbolic debugging information

Appendix C, *Assembler Directives Reference*, contains the individual directives' descriptions in alphabetical order.

5.11.1 Directives Summary

Table 5–2 summarizes the assembler directives.

Note:

Any source statement that contains a directive may also contain a label and a comment. Labels begin in the first column (and they are the only item allowed to appear in the first column) and all comments are preceded by a semicolon or an asterisk. To improve readability, labels and comments are not shown as part of the directive syntax.

Table 5–2. DSKplus Assembler Directives Summary

(a) Directives that define sections

Mnemonic and Syntax	Description	Page
.bss *symbol*, *size in words* [, *alignment*]	Reserve *size* words in the .bss (uninitialized data) section.	C-3
.data	Assemble into the .data (initialized data) section.	C-9
.sect "*section name"*	Assemble into a named (initialized) section.	C-26
.setsect "*section name*", *address* [, *page*]	Initialize the section's absolute address.	C-29
.text	Assemble into the .text (executable code) section.	C-33
symbol **.usect** "*section name*", *size in words* [, *alignment*]	Reserve *size* words in a named (uninitialized) section.	C-35

Table 5–2. DSKplus Assembler Directives Summary (Continued)

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

Mnemonic and Syntax	Description	Page
.bes *size in bits*	Reserve *size* bits in the current section; a label points to the last addressable word in the reserved space.	C-31
.byte $value_1$[, ... , $value_n$]	Initialize one or more successive bytes in the current section.	C-5
.field *value* [, *size in bits*]	Initialize a variable-length field.	C-13
.float *value* [, ... , $value_n$]	Initialize one or more 32-bit, IEEE single-precision, floating-point constants.	C-16
.int $value_1$ [, ... , $value_n$]	Initialize one or more 16-bit integers.	C-19
.long $value_1$[, ... , $value_n$]	Initialize one or more 32-bit integers.	C-23
.space *size in bits*	Reserve *size* bits in the current section; a label points to the beginning of the reserved space.	C-31
.string "$string_1$**"** [, ... , "$string_n$"]	Initialize one or more text strings.	C-32
.pstring "$string_1$**"** [, ... , "$string_n$"]	Initialize one or more text strings (packed).	C-32
.xfloat $value_1$ [, ...,$value_n$]	Initialize one or more 32-bit integers, IEEE single-precision, floating-point constants, but does not align the result on the long word boundary.	C-16
.xlong $value_1$ [, ...,$value_n$]	Initialize one or more 32-bit integers, but do not align on long word boundary.	C-23
.word $value_1$ [, ... , $value_n$]	Initialize one or more 16-bit integers.	C-19

(c) Directives that format the output listing

Mnemonic and Syntax	Description	Page
.length *page length*	Set the page length of the source listing.	C-20
.list	Restart the source listing.	C-21
.nolist	Stop the source listing.	C-21
.page	Eject a page in the source listing.	C-25
.title "*string*"	Print a title in the listing page heading.	C-34
.width *page width*	Set the page width of the source listing.	C-20

Table 5–2. DSKplus Assembler Directives Summary (Continued)

(d) Directives that reference other files

Mnemonic and Syntax	Description	Page
.copy ["]*filename*["]	Include source statements from another file.	C-6
.include ["]*filename*["]	Include source statements from another file.	C-6

(e) Directives that control conditional assembly

Mnemonic and Syntax	Description	Page
.break [*well-defined expression*]	End .loop assembly if condition is true. The .break construct is optional.	C-24
.else *well-defined expression*	Assemble code block if the .if condition is false. The .else construct is optional.	C-17
.elseif *well-defined expression*	Assemble code block if the .if condition is false and the .elseif condition is true. The .elseif construct is optional.	C-17
.end	End assembly file.	C-10
.endif	End .if code block.	C-17
.endloop	End .loop code block.	C-24
.if *well-defined expression*	Assemble code block if the condition is true.	C-17
.loop [*well-defined expression*]	Begin repeatable assembly of a code block.	C-24

(f) Directives that define symbols at assembly time

Mnemonic and Syntax	Description	Page
.set/.equ	Equate a value with a symbol.	C-27
.eval *well-defined expression*, *substitution symbol*	Perform arithmetic on numeric substitution symbols.	C-11

(g) Directives that align the section program counter (SPC)

Mnemonic and Syntax	Description	Page
.align [*size in words*]	Align the SPC on a word boundary specified by the parameter, which must be a power of 2, or default to page boundary.	C-2

5.11.2 Directives That Define Sections

Six directives associate portions of an assembly language program with the appropriate sections and initialize each section's address:

- **.bss** reserves space in the .bss section for uninitialized variables.
- **.data** identifies portions of code in the .data section. The .data section usually contains initialized data.
- **.sect** defines initialized named sections and associates subsequent code or data with that section. A section defined with .sect can contain code or data.
- **.setsect** initializes the section program counter (SPC) to a value that corresponds to the absolute address of the section.
- **.text** identifies portions of code in the .text section. The .text section usually contains executable code.
- **.usect** reserves space in an uninitialized named section. The .usect directive is similar to the .bss directive, but it allows you to reserve space separately from the .bss section.

Example 5–2 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 program counter, or SPC.) When code is first placed in a section, its SPC equals 0. When you resume assembling into a section after other code is assembled, the section's SPC resumes counting as if there had been no intervening code.

The directives in Example 5–2 perform the following tasks:

.text	initializes words with the values 1, 2, 3, 4, 5, 6, 7, and 8.
.data	initializes words with the values 9, 10, 11, 12, 13, 14, 15, and 16.
var_defs	initializes words with the values 17 and 18.
.bss	reserves 21 words.
xy	reserves 20 words.

The .bss and .usect directives do not end the current section or begin new sections; they reserve the specified amount of space, and then the assembler resumes assembling code or data into the current section.

Example 5–2. Using Sections Directives

```
 1
 2
 4                    .setsect ”Var_defs”, 080h
 5                    .setsect ”.text”, 0500h
 6                    .setsect ”.data”, 0600h
 7                    .setsect ”.bss”, 0550h
 8                    .setsect ”XY”, 0750h
 9              ************************************************************
10              *          Start assembling into .text section at 0500h
11              ************************************************************
12 000500                 .text
13 000500 0001            .word   1,2
   000501 0002
14 000502 0003            .word   3,4
   000503 0004
15
16              ************************************************************
17              *          Start assembling into .data section at 0600h
18              ************************************************************
19 000600                 .data
20 000600 0009            .word   9,10
   000601 000A
21 000602 000B            .word   11,12
   000603 000C
22
23              ************************************************************
24              *            Start assembling into a named, initialized
25              *                         section, Var_defs at 080h
26              ************************************************************
27 000080                 .sect ”Var_defs”
28 000080 0011            .word   17,18
   000081 0012
29
30              ************************************************************
31              *     Resume assembling into .data section at current .data
32              ************************************************************
33 000604                 .data
34 000604 000D            .word   13,14
   000605 000E
35 000550                 .bss    sym,19       ;Assemble into .bss section
36 000606 000F            .word   15,16        ;return to .data section
   000607 0010
37
38              ************************************************************
39              *     Resume assembling into .text section at current .text
40              ************************************************************
41 000504                 .text
42 000504 0005            .word   5,6
   000505 0006
43 000750       usym      .usect  ”XY”, 20        ;rsv space unitialized name
44 000563                 .bss    more,2       ;rsv 2 more locations in .b
45 000506 0007            .word   7,8
   000507 0008
```

5.11.3 Directives That Initialize Constants

Several directives assemble values for the current section:

- The **.bes** and **.space** directives reserve a specified number of bits in the current section. The assembler fills these reserved bits with 0s.

  You can reserve a specified number of words by multiplying the number of bits by 16.

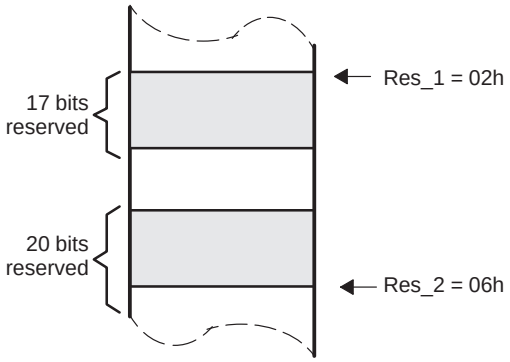
  - When you use a label with .space, it points to the *first* word that contains reserved bits.
  - When you use a label with .bes, it points to the *last* word that contains reserved bits.

  Figure 5–2 shows the .space and .bes directives. Assume the following code has been assembled for this example:

```
1
2            ** .space and .bes directives
3
4 0000 0100           .word           100h, 200h
  0001 0200
5 0002       Res_1:   .space          17
6 0004 000f           .word           15
7 0006       Res_2:   .bes            20
8 0007 00ba           .byte           0BAh
```

  Res_1 points to the first word in the space reserved by .space. Res_2 points to the last word in the space reserved by .bes.

Figure 5–2. Using the .space and .bes Directives



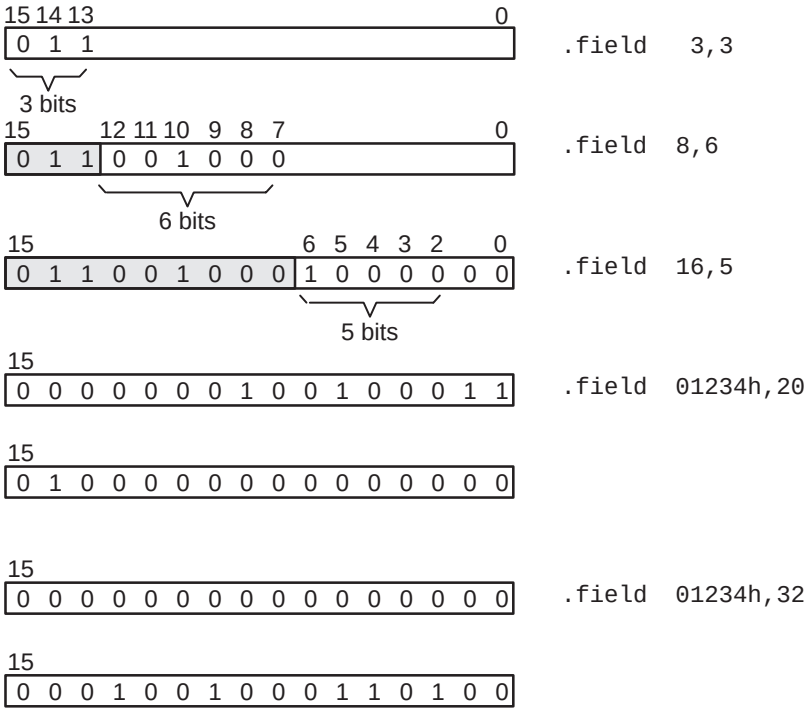
- **.byte** places one or more 8-bit values into consecutive words of the current section. This directive is similar to .word, except that the width of each value is restricted to eight bits.

- The **.field** directive places a single value into a specified number of bits in the current word. With .field, you can pack multiple fields into a single word; the assembler does not increment the SPC until a word is filled.

  Figure 5–3 shows how fields are packed into a word. For this example, assume the following code has been assembled; the SPC doesn't change for the first three fields (the fields are packed into the same word):

```
 4 0000 6000          .field          3, 3
 5 0000 6400          .field          8, 6
 6 0000 6440          .field          16, 5
 7 0001 0123          .field          01234h,20
   0002 4000
 8 0003 0000          .field          01234h,32
   0004 1234
```

Figure 5–3. Using the .field Directive

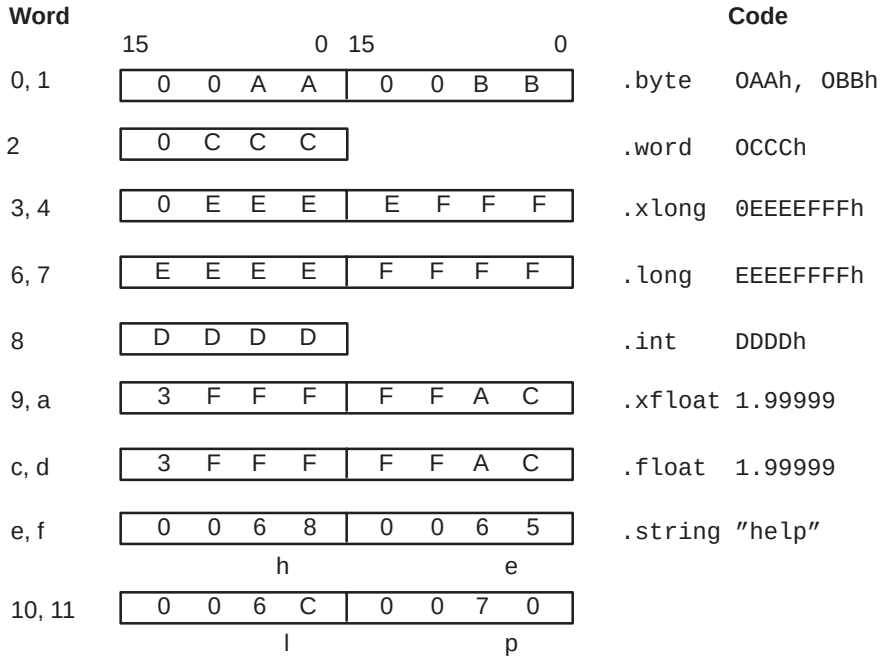


- **.float** and **.xfloat** calculate the single-precision (32-bit) IEEE floating-point representation of a single floating-point value and store it in two consecutive words in the current section. The most significant word is stored first. The .float directive automatically aligns to the long word boundary, and .xfloat does not.
- **.int** and **.word** place one or more 16-bit values into consecutive words in the current section.
- **.long** and **.xlong** place 32-bit values into consecutive 2-word blocks in the current section. The most significant word is stored first. The .long directive automatically aligns to a long word boundary, and the .xlong directive does not.
- **.string** and **.pstring** place 8-bit characters from one or more character strings into the current section. The .string directive is similar to .byte, placing an 8-bit character in each consecutive word of the current section. The .pstring also has a width of eight bits but packs two characters into a word. For .pstring, the last word in a string is padded with null characters (0) if necessary.

Figure 5–4 compares the .byte, .int, .long, .xlong, .float, .xfloat, .word, and .string directives. For this example, assume that the following code has been assembled:

```
1 0000 00aa          .byte           0AAh, 0BBh
  0001 00bb
2 0002 0ccc          .word           0CCCh
3 0003 0eee          .xlong          0EEEEFFFh
  0004 efff
4 0006 eeee          .long           0EEEEFFFFh
  0007 ffff
5 0008 dddd          .int            0DDDDh
6 0009 3fff          .xfloat         1.99999
  000a ffac
7 000c 3fff          .float          1.99999
  000d ffac
8 000e 0068          .string         ”help”
  000f 0065
  0010 006c
  0011 0070
```

Figure 5–4. Using Initialization Directives



5.11.4 Directives That Align the Section Program Counter

The **.align** directive aligns the SPC at a 1-word to 128-word boundary. This ensures that the code following the directive begins on an x-word or page boundary. If the SPC is already aligned at the selected boundary, it is not incremented. Operands for the .align directive must equal a power of 2 between 2^0 and 2^{16} (although alignments beyond 2^7 are not meaningful). For example:

Operand of	1	aligns SPC to word boundary
	2	aligns SPC to long word/even boundary
	128	aligns SPC to page boundary

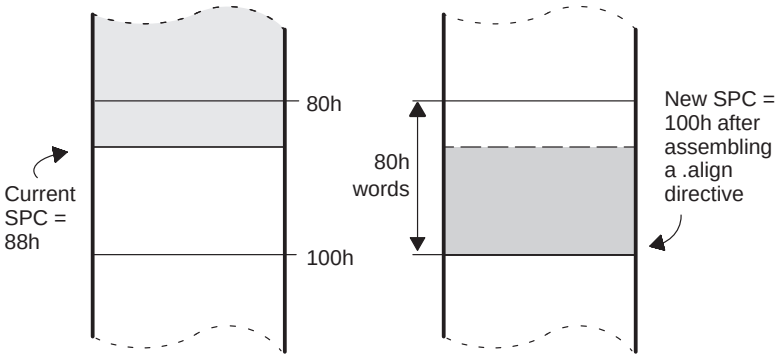
The .align directive with no operands defaults to 128, that is, to a page boundary.

Figure 5–5 demonstrates the .align directive. Assume that the following code has been assembled:

```
1 0000 4000          .field          2, 3
2 0000 4160          .field          11, 8
3                    .align          2
4 0002 0045          .string         "Errorcnt"
  0003 0072
  0004 0072
  0005 006f
  0006 0072
  0007 0063
  0008 006e
  0009 0074
5                    .align
6 0100 0004          .byte           4
```

Figure 5–5. Using the .align Directive

(a) Result of .align without an argument



5.11.5 Directives That Format the Output Listing

The following directives format the listing file:

- The **.length** directive controls the page length of the listing file. You can use this directive to adjust listings for various output devices.
- The **.list** and **.nolist** directives turn the output listing on and off. You can use the .nolist directive to stop the assembler from printing selected source statements in the listing file. Use the .list directive to turn the listing on again.
- The **.page** directive causes a page eject in the output listing.
- The **.title** directive supplies a title that the assembler prints at the top of each page.
- The **.width** directive controls the page width of the listing file. You can use this directive to adjust listings for various output devices.

5.11.6 Directives That Reference Other Files

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 a copied file are printed in the listing file; the statements read from an included file are *not* printed in the listing file.

5.11.7 Directives That Control Conditional Assembly

Conditional assembly directives enable you to instruct the assembler to assemble certain sections of code according to a true or false evaluation of an expression. Two sets of directives allow you to assemble conditional blocks of code:

- The **.if/.elseif/.else/.endif** directives tell the assembler to conditionally assemble a block of code according to the evaluation of an expression.

.if *expression*	marks the beginning of a conditional block and assembles code if the .if condition is true.
.elseif *expression*	marks a block of code to be assembled if the .if condition is false and .elseif is true.
.else	marks a block of code to be assembled if the .if condition is false.
.endif	marks the end of a conditional block and terminates the block.

- The **.loop/.break/.endloop** directives tell the assembler to assemble a block of code repeatedly according to the evaluation of an expression.

.loop *expression*	marks the beginning a repeatable block of code.
.break *expression*	tells the assembler to continue to assemble repeatedly when the .break expression is false, and to go to the code immediately after .endloop when the expression is true.
.endloop	marks the end of a repeatable block.

The assembler supports several relational operators that are useful for conditional expressions. For more information about relational operators, see subsection 5.9.4, page 5-17.

5.11.8 Directives That Assign Assembly-Time Symbols

Assembly-time symbol directives equate meaningful symbol names to constant values or strings.

- The **.eval** directive evaluates an expression, translates the results into a character, and assigns the character string to a substitution symbol. This directive is most useful for manipulating counters:

```
.eval     1 , x
.loop
.byte     x*10h
.break    x = 4
.eval     x+1, x
.endloop
```

- The **.set** and **.equ** directives set a constant value to a symbol. The symbol is stored in the symbol table and cannot be redefined. For example:

```
bval   .set   0100h
       .byte  bval, bval*2, bval+12
       goto   #bval
```

  The .set and .equ directives produce no object code. The two directives are identical and can be used interchangeably.

5.11.9 Directives That Terminate Assembly

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.

Chapter 6

Hardware

This chapter describes the DSKplus development hardware, including parallel port registers, signal definitions, ports, and modes. The section also covers the functionality of the on-board PAL® device, 'AC01 operation and register definitions, and the XDS510 emulator port.

Topic **Page**

6.1 Power and Cables

The DSKplus development hardware is powered by a universal 5-V dc @ 3.3-A wall-mount power supply. Depending on what function is being performed, the board's total power consumption can vary, so it is recommended that the external loading not exceed 1.0 A when using the the power and ground connections on the JP headers. If the total board power requirement exceeds 1.0 A, unplug the board's power supply immediately and check connections. The power supply included with the kit is regulated and filtered and feeds directly into the DSKplus board. If for some reason the power supply needs to be replaced, you can replace it with a power supply of similar specifications. Use only 5-V dc power supplies with a plug of 2.1-mm inner diameter $\times$ 5.5-mm outer diameter.

Be careful! The DSKplus board is not regulated; therefore, power requirements exceeding 1.0 A could result in board damage.

The cable is a straight DB25M-to-DB25F connection. Please use the cable included in the kit. If for some reason the cable needs to be replaced, it is recommended that the cable be high quality and not exceed 6 feet in length. Below is a pin description of the typical DB25 connector and its connection to DSKplus.

Table 6–1. DB25 Connector Pin Connections

Pin	Name	Connection	Description
1	STROBE	HR/$\overline{W}$	Controls HR/$\overline{W}$ of the host port interface (HPI) and the direction of the bidirectional buffer, and resets the NBL state machine (STROBE = 0)
2	D0	HD0	Bit 0 (LSB) data line; controls MODE and TRIST in latch mode
3	D1	HD1	Bit 1 data line; controls LS mode and RESET in latch mode
4	D2	HD2	Bit 2 data line
5	D3	HD3	Bit 3 data line
6	D4	HD4	Bit 4 data line
7	D5	HD5	Bit 5 data line
8	D6	HD6	Bit 6 data line
9	D7	HD7	Bit 7 (MSB) data line

Table 6–1. DB25 Connector Pin Connections (Continued)

Pin	Name	Connection	Description
10	ACK	HD2 or HD6 (NBL)	Bits 2 and 6 when reading from status register in nibble mode
11	$\overline{\text{BUSY}}$	HD3 or HD7 (NBL)	Bits 3 and 7 when reading from status register in nibble mode
12	PE	HD1 or HD5 (NBL)	Bits 1 and 5 when reading from status register in nibble mode
13	SLCT	HD0 or HD4 (NBL)	Bits 0 and 4 when reading from status register in nibble mode
14	AUTOFD	HCNTL0	Controls HCNTL0 of the HPI
15	ERROR	HINT	DSP-to-host interrupt signal
16	INIT	BYTE	Controls HBIL of the ’C54x HPI and is low for first transfer, high for second, low for third, and high for fourth (nibble mode). There are only two transfers when operating in byte mode.
17	SLCTIN	HCNTL1	Controls HCNTL1 of the HPI and which latch is selected in latch mode
18-25	GND	ground	

6.2 DSKplus Communications Protocol

The DSKplus development board communicates through your PC's parallel port. A PC manufactured earlier than 1993 probably has a 4-bit unidirectional port, which provides an 8-bit write function but only a 4-bit read function, and its data lines are not bidirectional—return data is read through four status register bits. A PC manufactured in 1993 or later probably has an 8-bit bidirectional port, which allows 8-bit directional data transfers for both reads and writes, and the status register bits can be used for other information.

Today's high-end computers often use enhanced parallel ports and extended capabilities ports, called EPP and ECP, respectively. These are much more complex; they provide a multiplexed data and address scheme across the eight data lines and they use a completely different protocol for communications. However, these ports support standard compatibility by default, which is 8-bit bidirectional.

The DSKplus support the following parallel port modes:

- ❑ 4-bit unidirectional (called nibble mode)
- ❑ 8-bit bidirectional (called byte mode)
- ❑ EPP standard compatibility (also 8-bit bidirectional or byte mode)
- ❑ EPC standard compatibility (also 8-bit bidirectional or byte mode)

The DSKplus does not support the EPP and EPC extended capabilities.

The parallel port's registers are located in the PC's data memory. You must use a 100% IBM/AT-compatible computer to eliminate any parallel port specification discrepancies. Compatible PCs use the following port addresses:

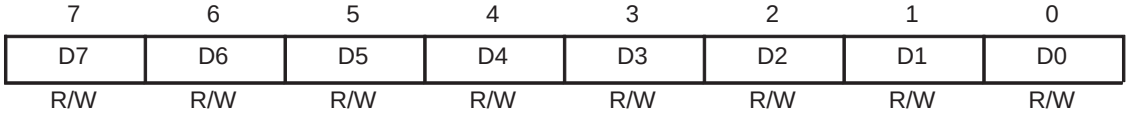
Port	Data Register	Status Register	Control Register
1	0x3BC	0x3BD	0x3BE
2	0x378	0x379	0x37A
3	0x278	0x279	0x27A

Don't get confused by the port number and the LPT designator. The LPTx can change due to other peripherals connected to your PC. When running custom applications, always take note of available ports by their addresses.

6.2.1 The PC's Data Register

The data registers of of the PC's parallel port are connected to the DSP HPI data lines (HD0–HD7) through a bidirectional buffer. The data register is always used for writing to the DSP HPI. However, when data is being read; the data register is needed only if the PC's parallel port is bidirectional (8-bit), otherwise, data is read from the status register. The data register of the PC's parallel port is shown in Figure 6–1:

Figure 6–1. Data Register



6.2.2 The PC's Status Register

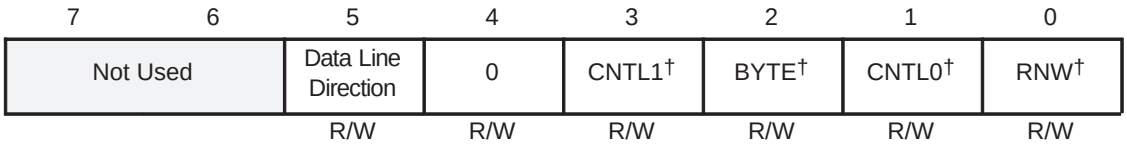
The status register has five input lines that directly indicate the levels of four signals connected to the PC external hardware, used for bits of data information only when in nibble (4-bit) mode. The fifth line is used as a DSP-to-host interrupt (HINT) line. The status register's configuration is shown in Figure 6–2:

Figure 6–2. Status Register

6.2.3 The PC's Control Register

The control register has four bit-addressable output signals available and a direction bit that controls the direction of the data lines. When transferring data to the DSP, load the data register first and then the control register. The control register is shown in Figure 6–3:

Figure 6–3. Control Register



† Indicates that this bit controls a hardware signal

Bit 5 This bit controls the direction of the parallel port data lines. Writing to this bit in 4-bit unidirectional ports has no effect.

Bit 4 This bit enables the PC interrupt when ACK in the status register is set. This bit is always loaded with zero.

Bit 3 Controls the hardware to set/reset the CNTL1 signal

Bit 2 Controls the hardware reset (inverted BYTE signal)

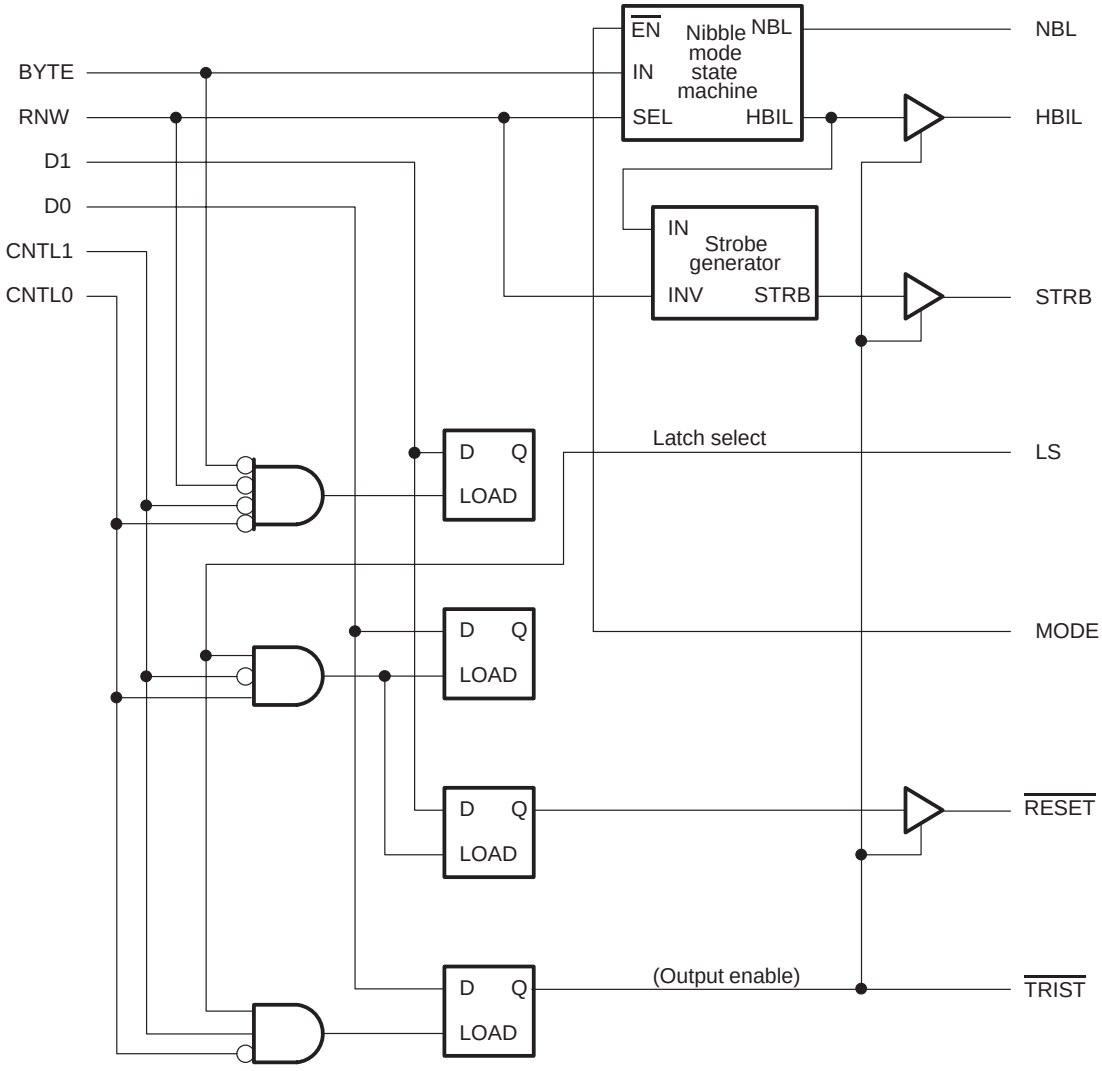
Bit 1 Controls the hardware to set/reset the CNTL0 signal

Bit 0 Controls the hardware to set/reset the RNW signal. When you are setting bit 0, be sure to set bit 5 appropriately.

6.3 Using a PAL® Device

The DSKplus board includes a socketed 22V10 programmable-array-logic (PAL®) device. The PAL® has been factory programmed to interface the PC's parallel port to the 'C54x HPI with additional capabilities to control the 'C54x reset line ($\overline{RS}$), a nibble state machine, a 3-state controller, and strobe generator.

Figure 6–4. PAL® Device's Internal Logic Diagram



Upon power up of the DSKplus board, the PAL® device is reset and all D-latches are set to 0, driving TRIST low, MODE low, and RESET low. TRIST is the 3-state controller that places the output pins in the high-impedance state when TRIST is low. Therefore, at power up, the PAL® device is in the high-impedance state and STROBE, HBIL, and RESET output values have no effect. Loading the PAL® device TRIST latch with a 1 drives RESET, STROBE, and HBIL. Once out of the high-impedance state, the DSP is placed in reset and the DSKplus board operates in nibble mode.

When MODE = 1, the board operates in 8-bit bidirectional mode.

When MODE = 0, the board operates in 4-bit unidirectional mode (nibble mode).

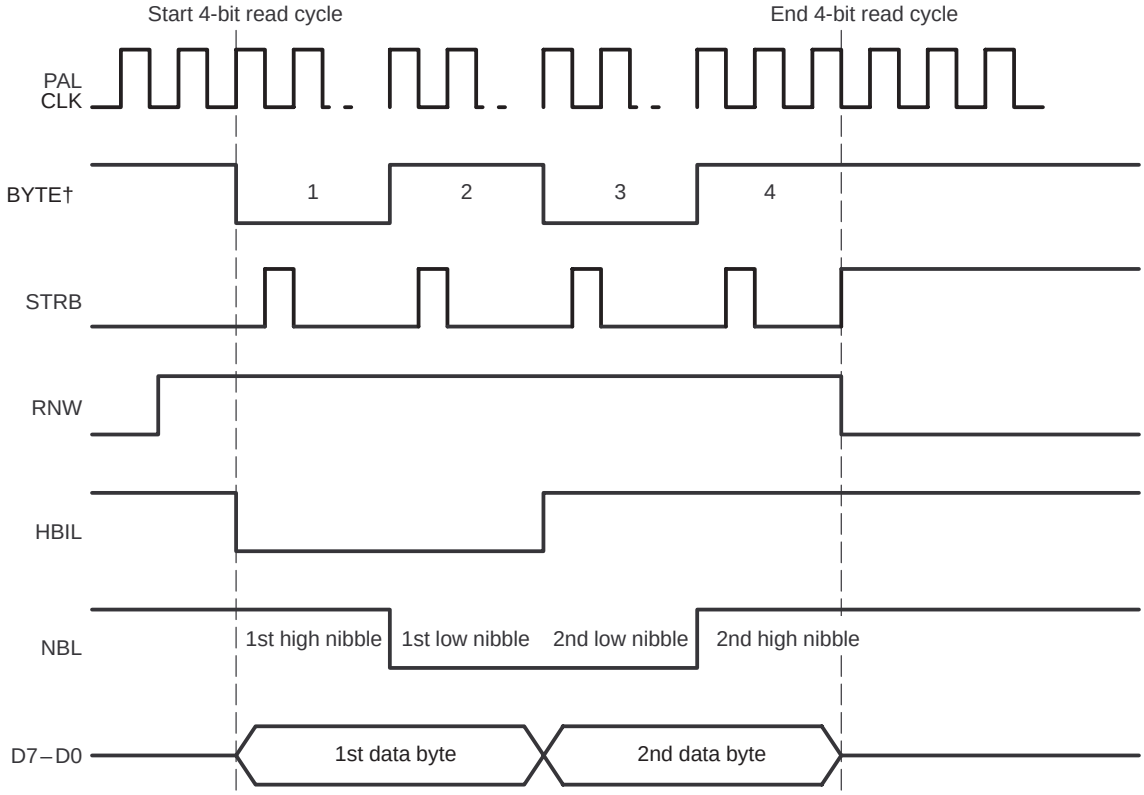
The host PC reads its data register twice to complete the 16-bit read when MODE = 1. The host must read the status register four times to complete a 16-bit read when MODE = 0.

During a read or write access to the HPI, both bytes (high and low) must be accessed to complete the HPI read/write cycle. Always perform a complete 16-bit read/write. See the *TMS320C54x DSP Reference Set, Volume 1: CPU and Peripherals,* for more information about the HPI.

6.3.1 Strobe Generator

Figure 6–4 illustrates the interaction between the PC parallel port and the PAL® device. During normal operation, the strobe line (STRB) generates a 1-cycle pulse, which is delayed by one clock cycle after the BYTE signal transitions to a new level. The STRB is connected to the ’C54x HDS2 line. The ’C54x drives or reads the data on the data lines on the falling edge of HDS2. The ’C54x also latches the levels of the CNTL1, CNTL0, HR/$\overline{W}$, and BYTE to determine whether the current transaction is a read or a write, which HPI mode to use and, which of the two bytes is being transferred.

Figure 6–5. Functional Diagram for a 4-Bit Read Cycle



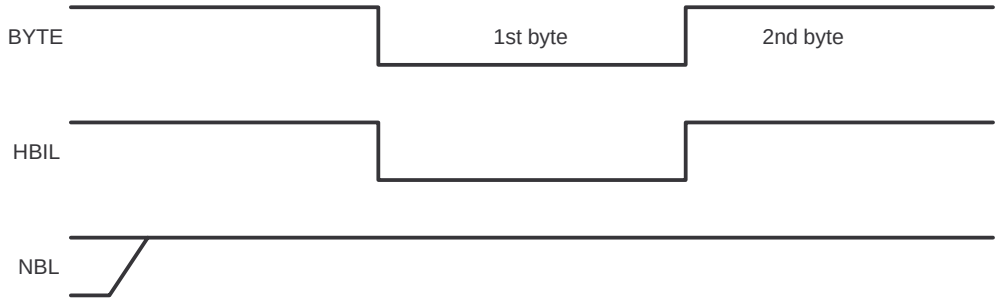
† BYTE cycle time corresponds to the speed at which the host PC can write to the BYTE location in the PC’s port control register.

6.3.2 Nibble Mode State Machine

Figure 6–5 is a functional diagram for the nibble mode. The nibble mode state machine controls the NBL signal when performing 4-bit reads only. The NBL signal controls which nibble of the read appears in the PC status register. Therefore, each time the PAL® device strobes the HPI, two 4-bit reads must be performed before strobing the HPI for the second byte. Since the PC parallel port always supports 8-bit writes and 4-bit reads as a minimum, the nibble state machine is active only during 4-bit reads (reads when MODE = 0). The state machine can be reset by setting the RNW line low while operating in nibble mode. The state machine is turned off when MODE = 1.

Figure 6–6 is the disabled state machine functional diagram. When the state machine is disabled (MODE = 1) or when it is not selected (RNW is low), HBIL mimics the BYTE signal.

Figure 6–6. Functional Diagram for a Write or 8-Bit Read Cycle



6.3.3 Latch/Select (LS) Mode

The latch mode is used to set the D-latches for the RESET, MODE, and TRIST signals. Latches in the PAL® device can be accessed only when the LS latch is set. LS is actually a latch accessed by writing 0x2 to the HPI while CNTL1 and CNTL0 are both low. This corresponds to writing 0x0202 to the HPIC register because of the required 16-bit data write to the HPIC. Bit 1 of the HPIC register has no effect on the DSP itself; instead, this bit sets/resets the LS latch. Writing a 0x0202 to the HPIC sets the LS latch high. As a result, changes to the CNTL1 and CNTL0 signals results in loading another latch with data. Analyzing the logic diagram in Figure 6–4 shows that changes in the CNTL1 and CNTL0 states will have the following effect:

CNTL (1,0)	Effect
00	Neutral state
01	Level of D0 propagates to the MODE output.
	Level of D1 propagates to the reset output. (MODE and reset must be loaded simultaneously.)
10	Level of D1 propagates to TRIST output
11	Unused

The levels of the data lines propagate to the respective output pins in real time. To load the data into the latch, load the host data register first, then enter the correct CNTL1 and CNTL0 levels and return to the neutral state to latch in the data. You must return to the neutral state after loading the data to the latch to allow the loading of the data register before performing another load. Also, returning to the neutral state avoids transient states of the CNTL1 and CNTL0 lines. Transient states occur when attempting to change the two signal levels at the same time. For example, if the current states of CNTL1 and CNTL0 are high and low (equivalent to 1 and 0) and a load of the PC control register changes the states to low and high (01), the bits may not change exactly at the same time, generating erroneous states 00 or 11. Because both of these states are neutral in the factory-programmed PAL® device, there is no concern. However, if the state 11 is used in a customized PAL® device, transient states may have adverse effects. Always return to the neutral state to be safe.

To disable the LS mode, reset the LS latch to 0 by writing a 0x0000 to the HPIC. At this point, the data latches are loaded and DSP/PC communications operate normally. The HPIC BOB bit = 0; therefore, the byte ordering is least significant byte first. There are PC-based C functions that perform this task automatically.

Refer to the host interface library C54XHIL on your DSKplus diskette for more information about using the PAL® device. This library includes all of the communication software needed to run application code on your PC. Remember that when you are running host PC applications, usually you will be unable to use the debugger.

6.4 PAL® Device Modifications

The information contained in this section is for experienced users ONLY, applying it improperly may result in board damage. Modifications to the DSKplus board are not supported by Texas Instrument and void all warranties. The debugger and other included software will not work correctly following board modifications!

The PAL® device can be modified by reprogramming a 28-pin PLCC 22V10 device and placing it into the socket. All pins of the PAL® device are accessible though the JP headers. When modifying the PAL® device, keep in mind the pin connections listed:

- **Pin 1.** No connection
- **Pin 2.** PAL® device CLK input
- **Pin 3.** Connected to the PC's D0 data line; cannot be disconnected from the HPI data lines
- **Pin 4.** Connected to the PC's D1 data line; cannot be disconnected from the HPI data lines
- **Pin 5.** Connected to the PC's CNTL1 line; can be disconnected from the host by removing the series resistor, but cannot be disconnected from the HPI's HCNTL1 signal
- **Pin 6.** Connected to the PC's BYTE line; can be disconnected from the host by removing the series resistor, but cannot be disconnected from the HPI's HBIL signal
- **Pin 7.** Connected to the PC's CNTL0 line; can be disconnected from the host by removing the series resistor, but cannot be disconnected from the HPI's HCNTL0 signal
- **Pin 8.** No connection
- **Pin 9.** Connected to the host PC's RNW line; can be disconnected from the host by removing the series resistor, but cannot be disconnected from the HPI's HR/$\overline{W}$ signal
- **Pin 10.** Used for an asynchronous PAL® device reset; forces PAL® device into 3-state mode by pulling this pin low (the pin is high for normal operation)
- **Pin 11.** General-purpose; input only
- **Pin 12.** General-purpose; input only

- **Pin 13.** General-purpose; input only
- **Pin 14.** Ground
- **Pin 15.** No connection
- **Pin 16.** General purpose, input only
- **Pin 17.** 3-state controller; general-purpose I/O
- **Pin 18.** Strobe polarity; general-purpose I/O
- **Pin 19.** Used to create a strobe signal. The strobe signal is delayed by one PAL® device CLK cycle and has a one PAL® device CLK cycle duration, required to satisfy the $t_{su(HAD)}$ and $t_{w(HDSl)}$ HPI timings. There is sufficient guardband to accomodate an increase in the PAL® clock rate.
- **Pin 20.** Connected to the output enable (OE) of the bidirectional buffer (74245). This pin is used to set the outputs of the bidirectional buffer to the high-impedance state, for lower power consumption, or to allow an external device control of the DSP HPI data lines.
- **Pin 21.** Latch select mode output; general purpose I/O
- **Pin 22.** No connection
- **Pin 23.** Connected to the DSP $\overline{\text{HDS2}}$ pin and cannot be disconnected. If using an external strobe via the JP4 (pin 25) header, be sure pin 23 of the PAL® device is in the 3-state mode.
- **Pin 24.** Connected to the DSP reset pin ($\overline{\text{RS}}$) and cannot be disconnected. If you are using an external reset line via the JP4 (pin 2) header, be sure pin 24 of the PAL® device is in 3-state mode. The reset LED (D2) is connected to this pin.
- **Pin 25.** Connected to the DSP HBIL pin and cannot be disconnected. If you are using an external HBIL line via the JP1 (pin 31) header, be sure pin 25 of the PAL® device is in 3-state mode.
- **Pin 26.** NBL; selects which nibble is used by the 74257 multiplexer; can be used as a general-purpose pin if the host PC has an 8-bit bidirectional port
- **Pin 27.** Mode latch; can be used as a general-purpose pin if the host PC has an 8-bit bidirectional port. This pin is also pulled high to disable the 74257 multiplexer. When pin 27 is high, additional external status/data, etc., lines can be connected to be available in the host PC's status register.
- **Pin 28.** V_{CC} isolated through inductor (L1)

6.5 Connecting Boards to Headers

External boards, sometimes called daughter boards, can be connected to the DSKplus board via the six JP headers. The universal power supply included with the kit has two independent 5-V power supplies capable of driving a total of 3.3 A. Daughter boards may include a 5 DIN connector to separate these power supplies, or you may choose to use the included DIN-to-5.5-mm adapter. Connectors can be soldered to the JP header holes. Many electronic suppliers can support 12 $\times$ 3 configurations.

6.6 Connecting the XDS510 Emulator Port

This section explains how to add debugging capabilities to your DSKplus board by using Texas Instruments XDS510 emulator.

The XDS510 emulator port is the JP2 header on the DSKplus board. To connect the XDS510 pod and cable, you must first solder a 7×2 header to the header on the board. The emulator connector is keyed at pin 6, so you must use wire cutters to cut off pin 6 of the soldered JP2 header.

Once the XDS510 header has been installed, turn off the power to the DSKplus board, connect the XDS510 emulator to the board, and turn the power back on. You do not need to disconnect the parallel port from the DSKplus board.

You can use the host printer port to cycle the reset line and to generate HPI interrupts, etc. Additionally, by using the XDS510 emulator, debugging is less intrusive and much more powerful. The XDS510 emulator debugs the ’C54x DSP via the JTAG emulator port allowing you to debug applications that involve the host port interface.

If you are controlling the reset line externally, be sure the PAL® device is in 3-state mode (TRISTATE is low). The PAL® device can be placed in the high-impedance state mode by loading the TRIST latch with 0 (if TRISTATE is low, RESET is in the high-impedance state).

Chapter 7

Initialization Routines

The chapter describes how to initialize each of the devices on the DSKplus board and the PC's parallel port.

Initialization of system elements must occur in a specific order:

1) Parallel port
2) PAL® device
3) HPI
4) DSP serial port
5) 'AC01 (analog interface device)

The first three elements form part of the communications link and the remaining two elements are DSP peripherals.

Topic **Page**

7.1 Communication Link (CommLink) Initialization

The CommLink initialization routine performs all initialization functions required for the host PC to communicate with the DSP. It also implements many of the C functions included in the host interface library C54XHIL. The CommLink initialization resides on the PC host.

7.1.1 Parallel Port and PAL® Device Initialization

The first function of a host application is to find the PC parallel port that connects to the DSKplus board. To do this, using the host interface library, call the C function locate port(). Other functions include:

```
void init_port(int);
set_latch(1,0);
mode();
```

The first C function performed is init_port(), which initializes the PC port to a known state. The set_latch() function sets the appropriate latch in the PAL® device. In this case, the PAL® device is brought out of the high-impedance state (1) and the DSP is placed in reset (0). By default the port operates in nibble mode. Call the function mode( ) to set the port to the 8-bit mode if you have an 8-bit port.

7.1.2 Host Port Interface Initialization

For the DSP's host port interface to operate correctly, call the function hpi_init(), which performs a write to the HPI control register to configure the byte ordering of the communication and to clear any pending interrupts. The second operation hpi_init() performs is HPI address register initialization. The routine initializes the HPIA to point to the beginning of the HPI RAM block (1000h).

After the init_port(), set_latch(1,0), mode(), and hpi_init() functions execute, the DSKplus board is ready to bootload.

7.2 Serial Port and TLC320AC01 Initialization

The second type of initialization in an application is DSP peripheral initialization. The on-board ’AC01 analog interface is an external device that connects to the DSP’s TDM serial port. To use the analog interface, you must initialize the TDM serial port control register (TSPC) and the ’AC01 registers.

Code for performing the initialization of the serial port and the ’AC01 is included in the PERIPHS directory of the DSKplus software.

Note:

If you wish to use the DSP’s on-chip peripherals in your own applications, your code must perform the appropriate peripheral initializations.

Perform the peripheral initialization by loading and running the following DSP code:

```
XF   = 0        /* Force the AC01 to reset state             */
TSPC = #0008h   /* Store 8h to the TDM serial port cotrl reg */
TSPC = #00C8h   /* Store C8h to the TDM serial port ctrl reg */
CALL  AC01INIT /* Call the AC01 init routine                 */
```

The first store to the TSPC stops the serial port from operating by resetting the $\overline{\text{XRST}}$ and $\overline{\text{RRST}}$ bits each to 0. The second store to the TSPC configures the serial port to receive the CLKX and CLKR clock signals externally (MCM = 0) and configures FSR and FSX to receive the frame sync pulses externally (TXM = 0). As a result of the second store to the TSPC, the serial port begins operating, dependent upon external clock and frame sync pulses.

Note:

The ’AC01 is configured to generate the CLKX, CLR, FSX, and FSR signals. Do not program the DSP serial port to generate these signals. Always store #00C8h to the TSPC as the second store.

The CALL statement calls the function AC01INIT to initialize the ’AC01 registers. By default, the ’AC01 begins operating with a sampling rate of 15.4 kHz. By programming the internal ’AC01 registers, sampling rates can be changed quickly and easily. See the *TLC320AC01C Single-Supply Analog Interface Circuit Data Manual* for more information about programming the ’AC01.

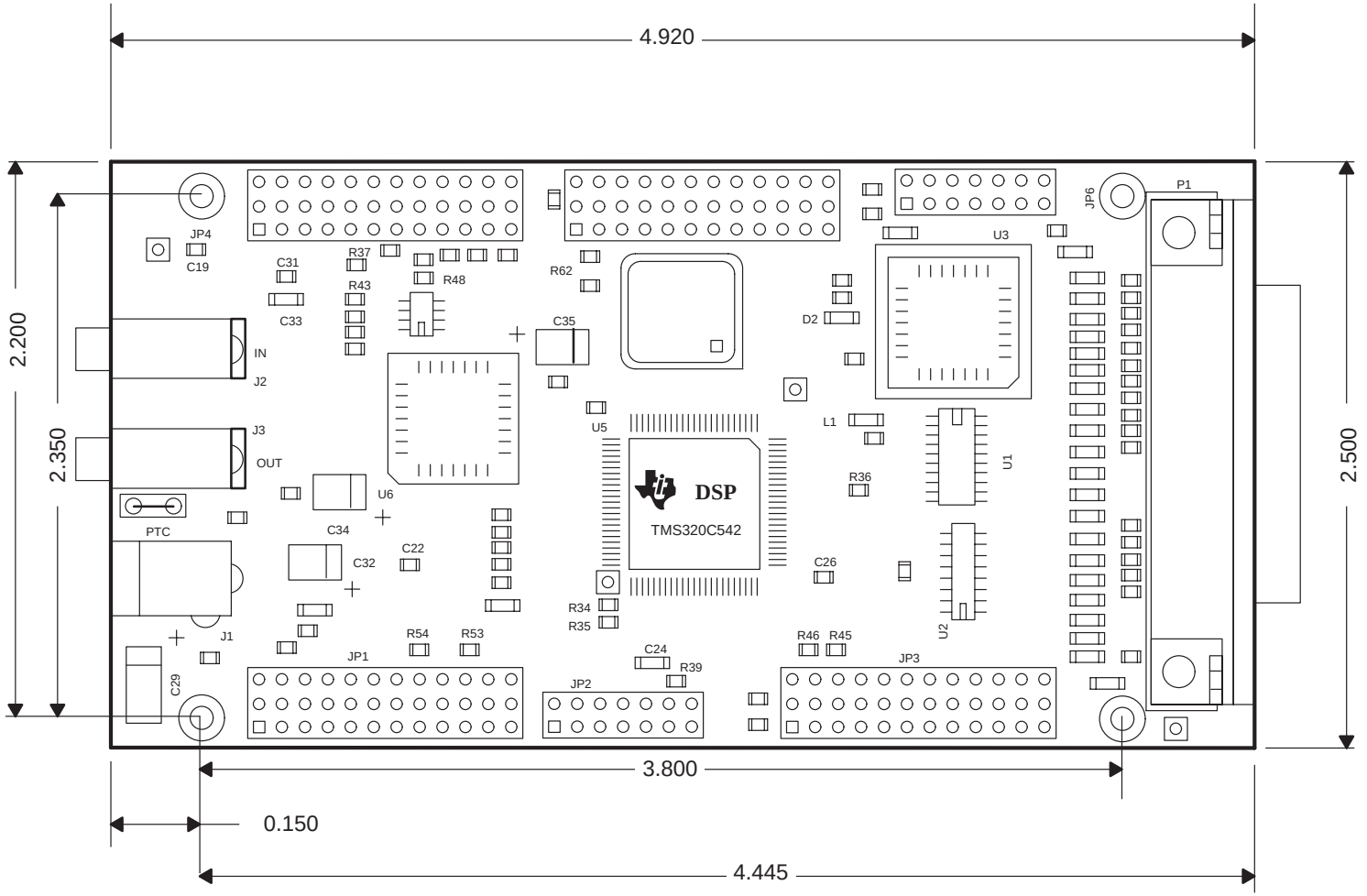
The AC01INIT routine is located in the PERIPHS directory of the DSKplus software. Normally, this routine is included as one of the first operation in the DSP's application source code. The PERIPHS directory also includes initialization routines for all of the peripherals of the TMS320C542. Your DSP software does not need to initialize peripherals that it does not use.

Appendix A

DSKplus Circuit Board Dimensions and Schematic Diagram

This appendix contains the circuit board dimensions shown in Figure A–1 and the schematic diagram shown in Figure A–2 for the TMS320C54x DSKplus.

Figure A–1. TMS320C54x DSKplus Circuit Board Dimensions



Note: Dimensions are in inches.

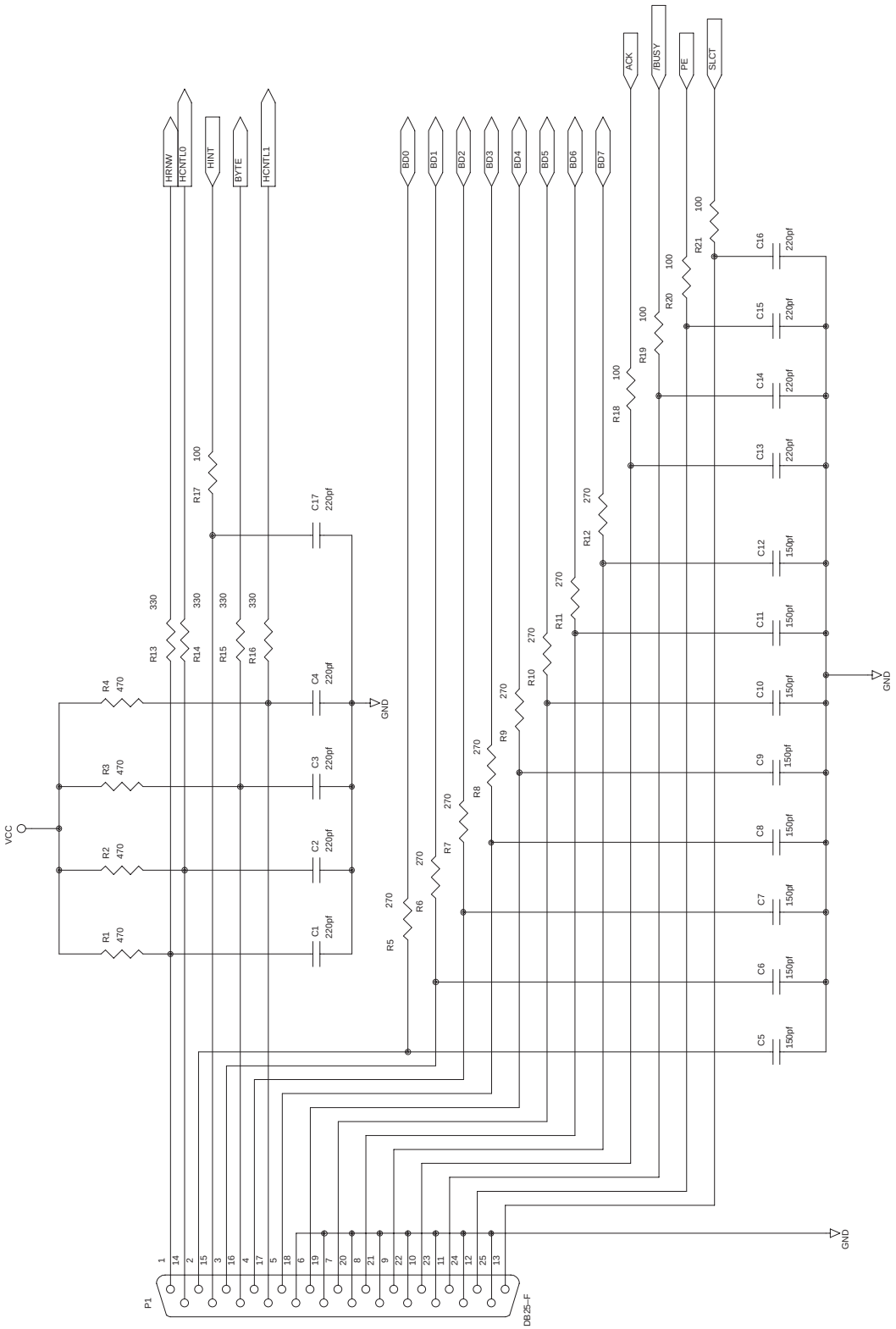


Figure A–2. Schematic Diagram of DSKplus Circuit Board

Figure A–2. Schematic Diagram of DSKplus Circuit Board (Continued)

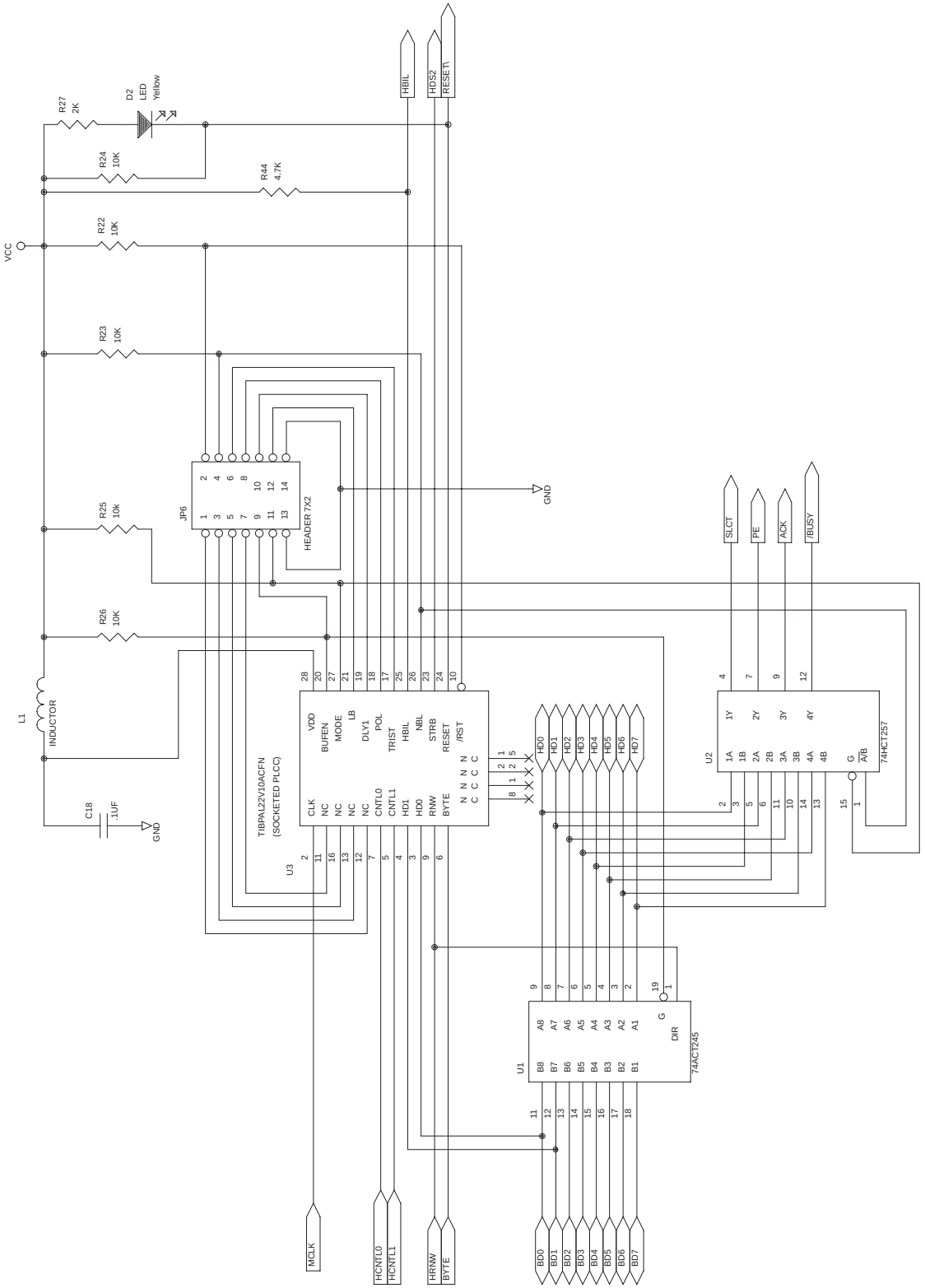


Figure A–2. Schematic Diagram of DSKplus Circuit Board (Continued)

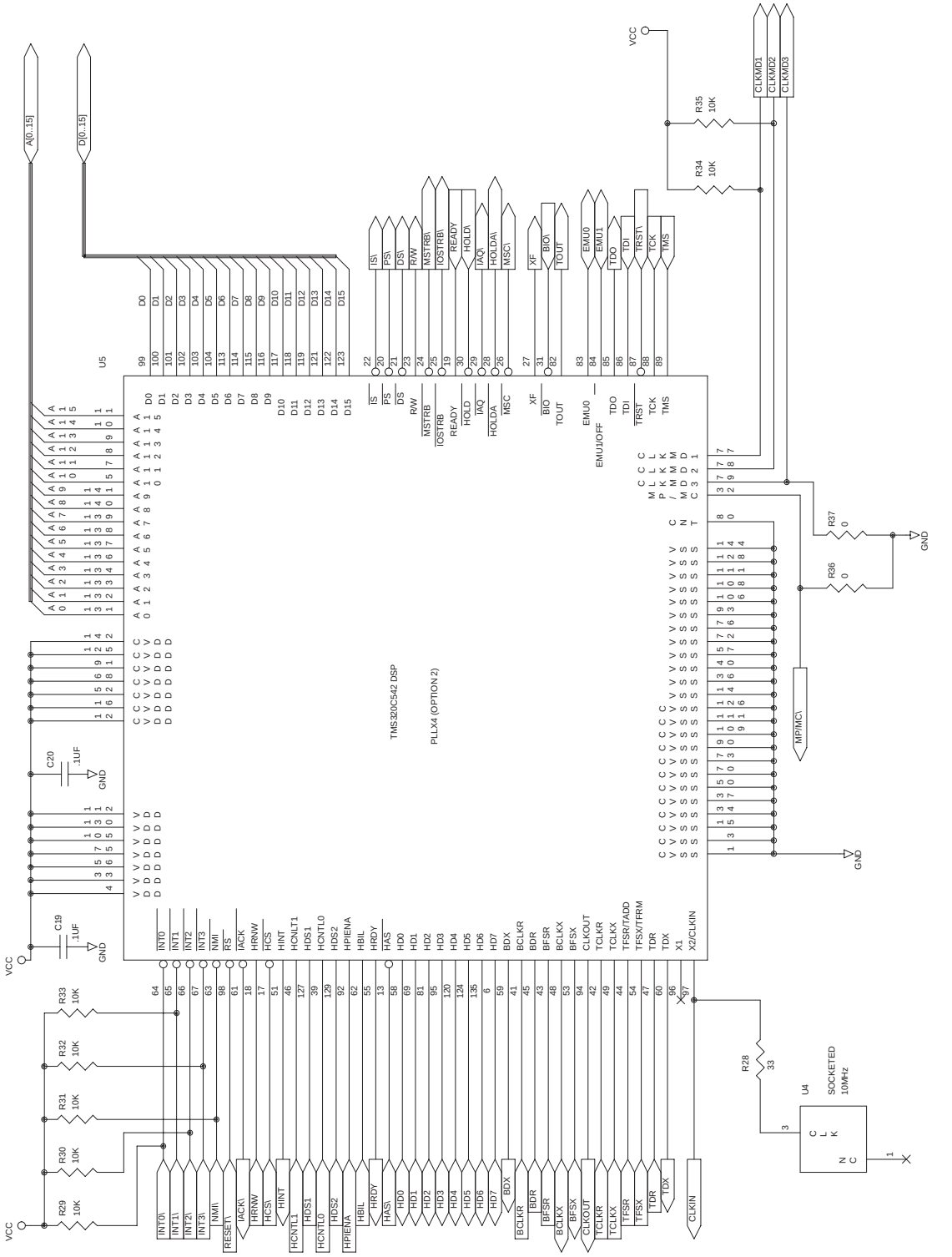


Figure A–2. Schematic Diagram of DSKplus Circuit Board (Continued)

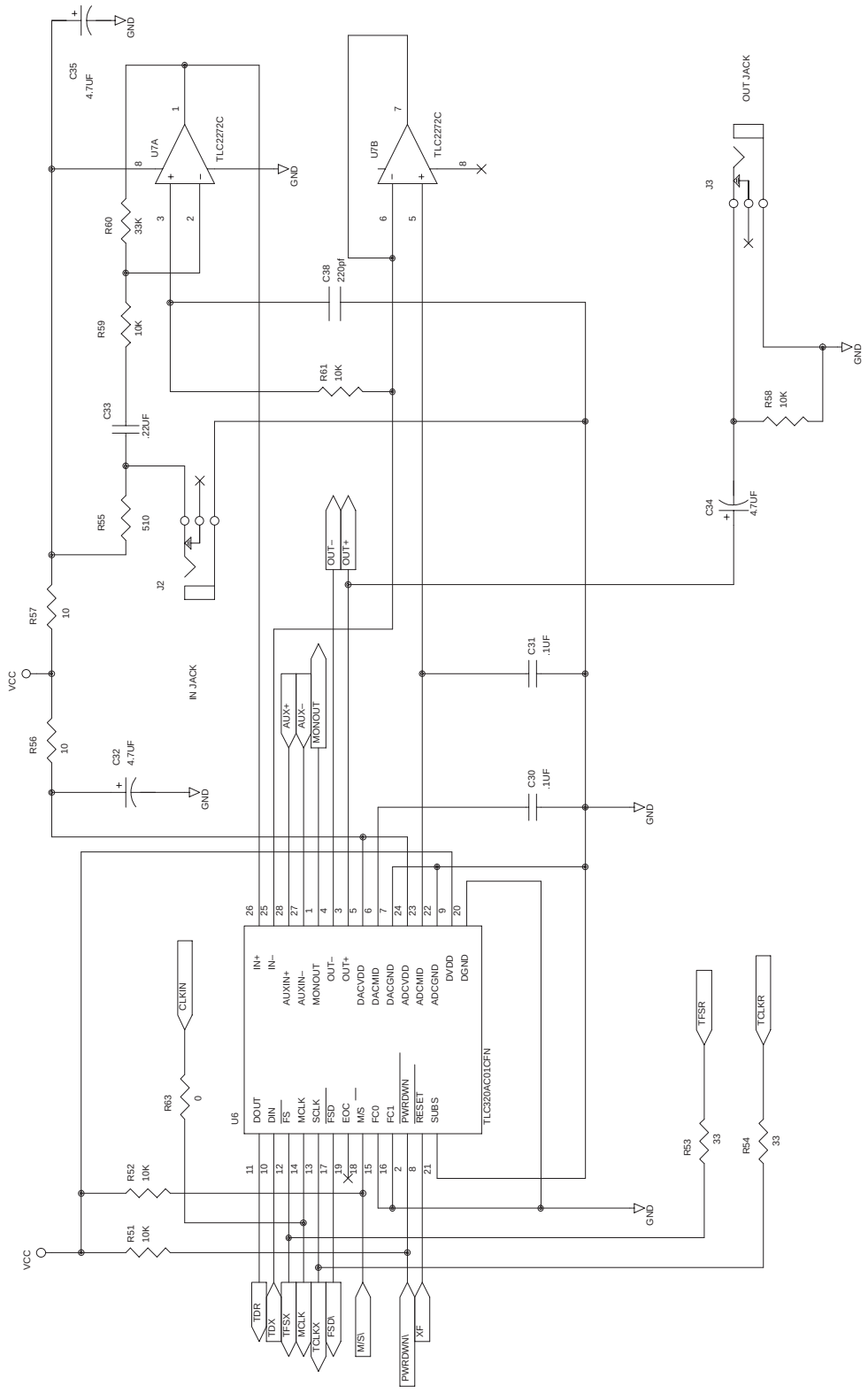


Figure A–2. Schematic Diagram of DSKplus Circuit Board (Continued)

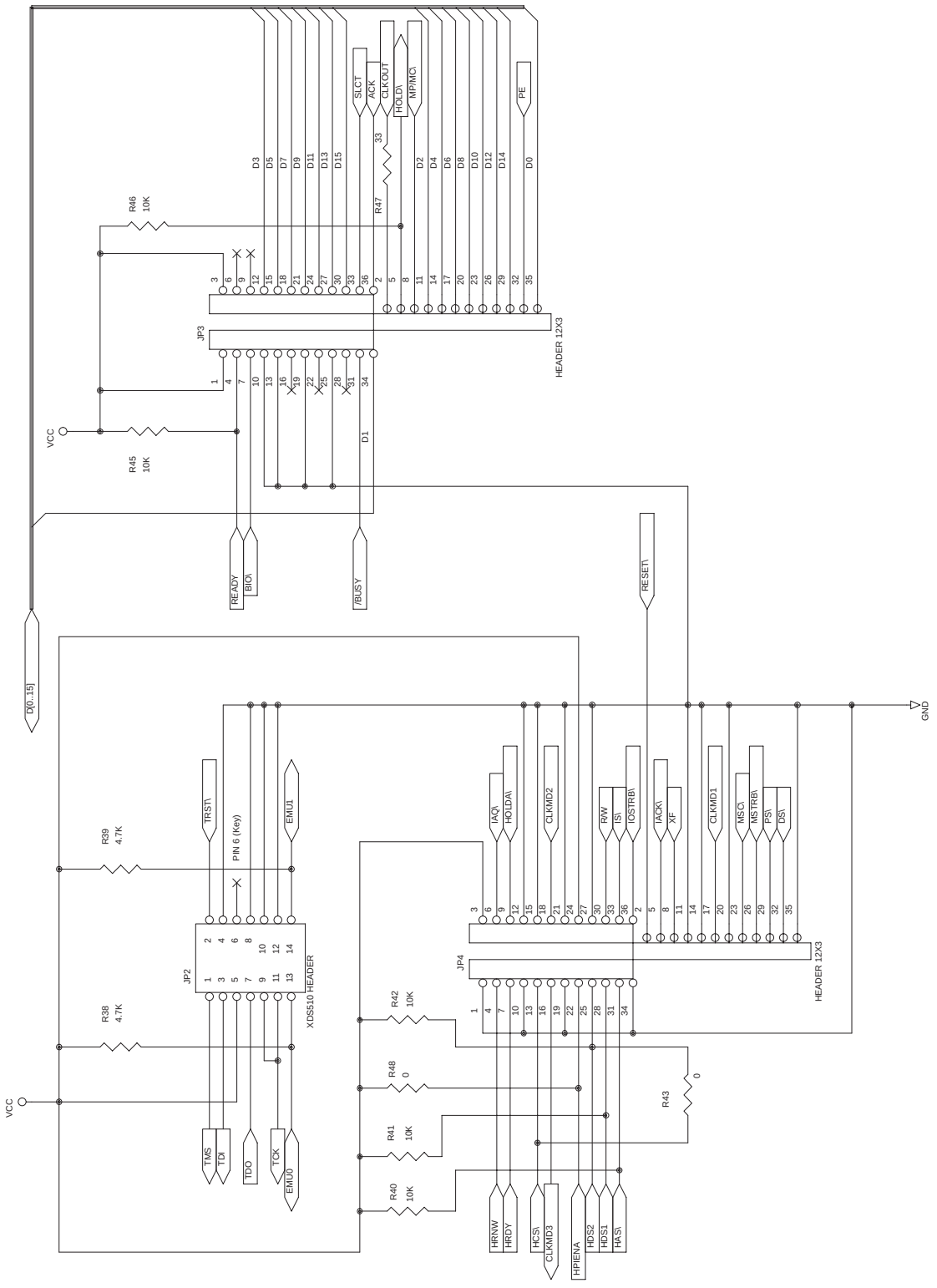
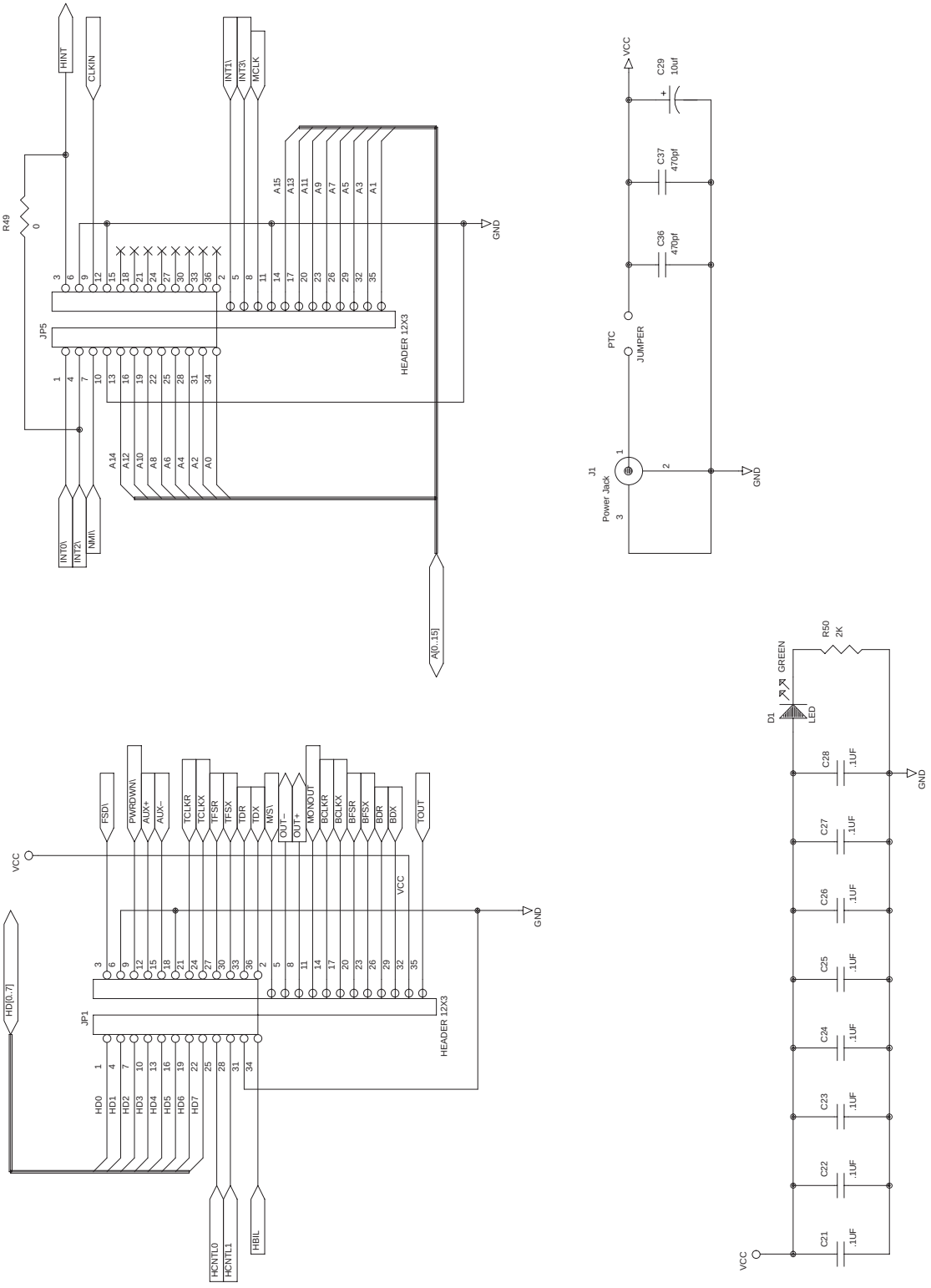


Figure A–2. Schematic Diagram of DSKplus Circuit Board (Continued)



Appendix B

PAL Equations

Included here are the PAL® equations and associated test vectors for the factory default PAL® device with a brief functional description for each equation:

LS Loaded only when performing a HPIC write with data line HD1=1 and BYTE=0. This sets the latch select bit and enables loading of the RESET, MODE, and TRIST latches.

RESET Controls the DSP reset line. This latch is reset and set with !HD1 (inverted HD1 level). It is loaded only when LS=1 and CNTL(0,1)=1 0.

TRIST Controls the 3-state controller. If TRIST=0 then RESET, STROBE, and HBIL have no effect on the DSP. It is loadable only when LS=1 and CNTL(0,1)=0 1.

MODE Determines how the HPI will interface to the PC. If MODE=0, the PC parallel port functions in 4-bit mode. If MODE=1, the parallel port functions in 8-bit mode. When MODE=1, the 74257 mutiplexer is disabled, since 8-bit bidirectional data can be read from the data register and the multiplexer is not needed. MODE can be loaded only when LS=1 and CNTL(0,1)=1 0.

BUFEN These 4-bit port data lines cannot be turned off or reversed. This logic is used to disable the bidirectional buffer when performing a read in 4-bit mode. By doing so, the parallel port and bidirectional buffer will never drive into each other. The series resistors can tolerate this but will cause data lines to be very noisy in some cases. BUFEN is also disabled when TRIST=0 and MODE=1.

NBL Selects which four bits of the byte are received into the parallel ports status register. Two BYTE cycles must be performed to receive a byte and four BYTE cycles must be performed to receive a 16-bit word. NBL is only active during a 4-bit read. Writes are always eight bits.

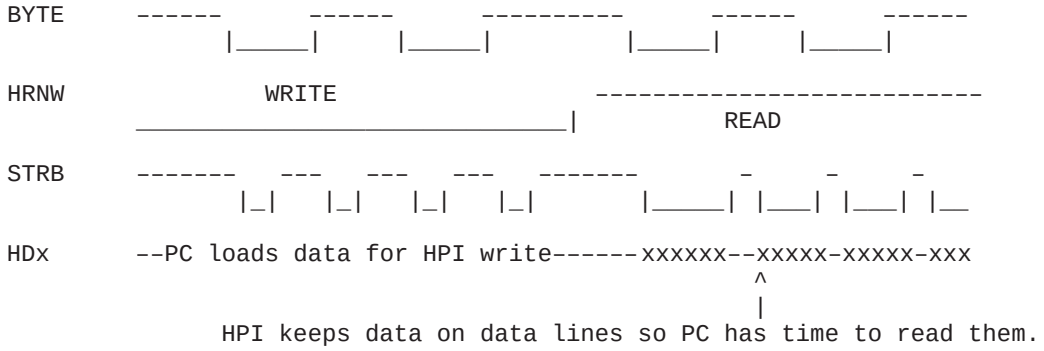
HBIL Tied to the DSP HPI to select which byte is to be transferred to/from the HPI. HBIL mimics BYTE during all writes and 8-bit reads. HBIL changes level every two BYTE cycles during a 4-bit read, since two 4-bit reads from the status register must be performed before reading the next byte from the HPI.

DLY1 Creates the 1-cycle delay in the synchronous delay line. The 1-cycle (100-ns) delay is used to conform to the HPI setup timings for the HBIL, CNTL(0,1),and HR/$\overline{W}$ signals.

STRB This latch is the second stage of the synchronous delay line. This creates a 1-cycle (100-ns) strobe signal. During the HPI read, the HPI is strobed on the falling edge of STRB (connected to HDS2). The data remains on the data lines only while STRB=0. During an HPI write the HPI data is read in on the rising edge of STRB. The polarity of this signal is controlled by the POL latch.

POL Reverses the polarity of the STRB signal to keep the data on the data lines during the BYTE cycle (HPI READS only). This allows the relatively slow PC to read the data from the data register and keeps the data on the data lines for the duration of the BYTE cycle.

Example B–1. PAL® Equation Routine



Example B–1. PAL® Equation Routine (Continued)

```
module C54xDSKp
title 'TMS320C54x DSKplus HPI/PC Interface Logic
Texas Instruments 17 Jul 1996'

        C54xDSKp  device  'P22V10C' ;


"Inputs
        Clk     pin 2  ; "clock input
        HD0     pin 3  ; "HPI data line 0
        HD1     pin 4  ; "HPI data line 1
        CNTL1   pin 5  ; "HPI HCNTL1 input
        BYTE    pin 6  ; "Byte indicator
        CNTL0   pin 7  ; "HPI HCNTL0 input
        RNW     pin 9  ; "HPI RW indicator
        RST     pin 10 ; "PAL® async reset

"Outputs
        POL     pin 18  ; "Strobe Polarity controller
        TRIST   pin 17  ; "tri–state controller
        DLY1    pin 19  ; "delay state machine bit
        BUFEN   pin 20; ; "Bi–directional buffer enabler
        LS      pin 21  ; "Latch select mode
        STRB    pin 23  ; "Strobe output
        RESET   pin 24  ; "DSP reset pin
        HBIL    pin 25  ; "Synced BYTE, HPI byte indicator
        NBL     pin 26  ; "Nibble selector (4-bit mode)
        MODE    pin 27  ; "Mode latch output

        RD,WR,H,L = 1,0,1,0;
```

Example B–1. PAL® Equation Routine (Continued)

```
equations

        [POL,TRIST,LS,NBL,HBIL,DLY1,STRB,RESET,MODE].clk = Clk;
        [STRB, RESET, HBIL].oe = TRIST;
        [POL,NBL,HBIL,TRIST,LS,RESET,MODE].ar= !RST;


        LS       := HD1 & (!BYTE & !RNW & !CNTL0 & !CNTL1)
                    # LS & !(!BYTE & !RNW & !CNTL0 & !CNTL1);

        RESET    := !HD1 & (LS & CNTL0 & !CNTL1)
                    # RESET & !(LS & CNTL0 & !CNTL1);

        TRIST    := HD0 & (LS & !CNTL0 & CNTL1)
                    # TRIST & !(LS & !CNTL0 & CNTL1);

        MODE     := HD0 & (LS & CNTL0 & !CNTL1)
                    # MODE & !(LS & CNTL0 & !CNTL1);

        BUFEN     = (!MODE & RNW) & TRIST
                    # (MODE & !TRIST);

        NBL      := ((BYTE & HBIL # NBL & !BYTE) & !MODE
                    # MODE # !RNW) & TRIST
                    # BYTE & !RNW & !TRIST;

        HBIL     := ((!NBL & !BYTE) # (HBIL & BYTE)) & (!MODE # RNW)
                    # (BYTE & (MODE # !RNW));

        DLY1     := HBIL;

        STRB     := (!(DLY1 $ HBIL) & !POL)
                    # ((DLY1 $ HBIL) & POL);

        POL      := STRB & HBIL & POL
                    # (RNW & !(STRB & HBIL));


trace ([CNTL0,CNTL1,BYTE,RNW] ->[BUFEN,MODE,TRIST,NBL,HBIL,DLY1,STRB]);
```

Example B–1. PAL® Equation Routine (Continued)

```
test_vectors    ”latch test vectors
([Clk, RST,LS,BYTE,RNW,CNTL0,CNTL1,HD0,HD1] –> [NBL,BUFEN,LS,RESET,TRIST,MODE]);
 [.C., 1, .X., 1  , 1 ,  .X., .X. ,.X.,.X.] –> [ 0,0,0 ,.Z., 0 , 0 ];
 [.C., 1,  1 , 0  , 0 ,   0 ,  0  ,.X., 1 ] –> [ 0,0,1 ,.Z., 0 , 0 ];”LB=1
 [.C., 1,  1 , 1  ,.X.,   0 ,  0  ,.X., 1 ] –> [ 1,0,1 ,.Z., 0 , 0 ];”HOLD
 [.C., 1,  1 , 1  ,.X.,   1 ,  0  ,.X., 1 ] –> [ 1,0,1 ,.Z., 0 , 0 ];”RESET=0
 [.C., 1,  1 , 1  ,.X.,   1 ,  0  ,.X., 0 ] –> [ 1,0,1 ,.Z., 0 , 0 ];”RESET=1
 [.C., 1,  1 , 1  ,.X.,   0 ,  0  , 1 ,.X.] –> [ 1,0,1 ,.Z., 0 , 0 ];”HOLD
 [.C., 1,  1 , 1  ,.X.,   0 ,  1  , 1 ,.X.] –> [ 1,0,1 , 1 , 1 , 0 ];”TRIST=1
 [.C., 1,  1 , 1  ,.X.,   0 ,  0  , 1 , 0 ] –> [ 1,0,1 , 1 , 1 , 0 ];”HOLD
 [.C., 1,  1 , 1  ,.X.,   1 ,  0  , 1 , 0 ] –> [ 1,0,1 , 1 , 1 , 1 ];”MODE=1
 [.C., 1,  1 , 1  ,.X.,   1 ,  0  , 0 , 0 ] –> [ 1,0,1 , 1 , 1 , 0 ];”MODE=0
 [.C., 1,  1 , 1  ,.X.,   0 ,  0  ,.X.,.X.] –> [ 1,0,1 , 1 , 1 , 0 ];”HOLD
; ” TURN OFF LS VIA WRITE CYCLE
 [.C., 1,  0 , 0  , 0 ,   0 ,  0  ,.X., 0 ] –> [ 1,0,0 , 1 , 1 , 0 ];”HOLD
 [.C., 1,  0 , 0  , 0 ,   0 ,  0  ,.X., 0 ] –> [ 1,0,0 , 1 , 1 , 0 ];”HOLD
 [.C., 1,  0 , 0  , 0 ,   0 ,  0  ,.X., 0 ] –> [ 1,0,0 , 1 , 1 , 0 ];”HOLD
 [.C., 1,  0 , 0  , 0 ,   0 ,  0  ,.X., 0 ] –> [ 1,0,0 , 1 , 1 , 0 ];”HOLD
 [.C., 1,  0 , 1  , 0 ,   0 ,  0  ,.X., 0 ] –> [ 1,0,0 , 1 , 1 , 0 ];”HOLD
 [.C., 1,  0 , 1  , 0 ,   0 ,  0  ,.X., 0 ] –> [ 1,0,0 , 1 , 1 , 0 ];”HOLD
 [.C., 1,  0 , 1  , 0 ,   0 ,  0  ,.X., 0 ] –> [ 1,0,0 , 1 , 1 , 0 ];”HOLD
 [.C., 1,  0 , 1  , 0 ,   0 ,  0  ,.X., 0 ] –> [ 1,0,0 , 1 , 1 , 0 ];”HOLD
```

Example B–1. PAL® Equation Routine (Continued)

```
test_vectors "8-bit WRITE -> 4-bit READ OPERATION -> 8-bit WRITE
([Clk, RST, BYTE, RNW] -> [BUFEN, NBL, HBIL, DLY1, STRB]);
 [.C.,  1 ,  1  ,  1 ] -> [ 1   ,  1 ,  1  ,  1  ,  1  ];
 [.C.,  1 ,  1  ,  0 ] -> [ 0   ,  1 ,  1  ,  1  ,  1  ];
 [.C.,  1 ,  0  ,  0 ] -> [ 0   ,  1 ,  0  ,  1  ,  1  ];
 [.C.,  1 ,  0  ,  0 ] -> [ 0   ,  1 ,  0  ,  0  ,  0  ];
 [.C.,  1 ,  0  ,  0 ] -> [ 0   ,  1 ,  0  ,  0  ,  1  ];
 [.C.,  1 ,  0  ,  0 ] -> [ 0   ,  1 ,  0  ,  0  ,  1  ];
 [.C.,  1 ,  1  ,  0 ] -> [ 0   ,  1 ,  1  ,  0  ,  1  ];
 [.C.,  1 ,  1  ,  0 ] -> [ 0   ,  1 ,  1  ,  1  ,  0  ];
 [.C.,  1 ,  1  ,  0 ] -> [ 0   ,  1 ,  1  ,  1  ,  1  ]; "MAKE W->R TRANS
 [.C.,  1 ,  1  ,  1 ] -> [ 1   ,  1 ,  1  ,  1  ,  1  ];
 [.C.,  1 ,  1  ,  1 ] -> [ 1   ,  1 ,  1  ,  1  ,  1  ];
 [.C.,  1 ,  1  ,  1 ] -> [ 1   ,  1 ,  1  ,  1  ,  1  ];
 [.C.,  1 ,  0  ,  1 ] -> [ 1   ,  1 ,  0  ,  1  ,  1  ]; "R IN EFFECT
 [.C.,  1 ,  0  ,  1 ] -> [ 1   ,  1 ,  0  ,  0  ,  0  ];
 [.C.,  1 ,  0  ,  1 ] -> [ 1   ,  1 ,  0  ,  0  ,  0  ];
 [.C.,  1 ,  0  ,  1 ] -> [ 1   ,  1 ,  0  ,  0  ,  0  ];
 [.C.,  1 ,  1  ,  1 ] -> [ 1   ,  0 ,  0  ,  0  ,  0  ];
 [.C.,  1 ,  1  ,  1 ] -> [ 1   ,  0 ,  0  ,  0  ,  0  ];
 [.C.,  1 ,  1  ,  1 ] -> [ 1   ,  0 ,  0  ,  0  ,  0  ];
 [.C.,  1 ,  1  ,  1 ] -> [ 1   ,  0 ,  0  ,  0  ,  0  ];
 [.C.,  1 ,  0  ,  1 ] -> [ 1   ,  0 ,  1  ,  0  ,  0  ];
 [.C.,  1 ,  0  ,  1 ] -> [ 1   ,  0 ,  1  ,  1  ,  1  ];
 [.C.,  1 ,  0  ,  1 ] -> [ 1   ,  0 ,  1  ,  1  ,  0  ];
 [.C.,  1 ,  0  ,  1 ] -> [ 1   ,  0 ,  1  ,  1  ,  0  ];
 [.C.,  1 ,  1  ,  1 ] -> [ 1   ,  1 ,  1  ,  1  ,  0  ];
 [.C.,  1 ,  1  ,  1 ] -> [ 1   ,  1 ,  1  ,  1  ,  0  ];
 [.C.,  1 ,  1  ,  0 ] -> [ 0   ,  1 ,  1  ,  1  ,  0  ]; "MAKE R->W TRANS
 [.C.,  1 ,  1  ,  0 ] -> [ 0   ,  1 ,  1  ,  1  ,  1  ]; "W IN EFFECT
 [.C.,  1 ,  0  ,  0 ] -> [ 0   ,  1 ,  0  ,  1  ,  1  ];
 [.C.,  1 ,  0  ,  0 ] -> [ 0   ,  1 ,  0  ,  0  ,  0  ];
 [.C.,  1 ,  0  ,  0 ] -> [ 0   ,  1 ,  0  ,  0  ,  1  ];
 [.C.,  1 ,  0  ,  0 ] -> [ 0   ,  1 ,  0  ,  0  ,  1  ];
 [.C.,  1 ,  1  ,  0 ] -> [ 0   ,  1 ,  1  ,  0  ,  1  ];
 [.C.,  1 ,  1  ,  0 ] -> [ 0   ,  1 ,  1  ,  1  ,  0  ];
 [.C.,  1 ,  1  ,  0 ] -> [ 0   ,  1 ,  1  ,  1  ,  1  ];

end
```

Appendix C

Assembler Directives Reference

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

This appendix is a detailed reference for all of the DSkplus assembler directives. Each directive is described individually including syntax and examples,, and the directives are presented in alphabetical order. Generally, the directives are organized, one directive per page; however, related directives (such as .if/.else/.endif) are presented together on one page. Following is an alphabetical table of contents for the directives reference:

Syntax

.align [*size in words*]

Description

The **.align** directive aligns the section program counter (SPC) on the next boundary, depending on the *size in words* parameter. The *size* may be any power of 2, although only certain values are useful for alignment. An operand of 128 aligns the SPC on the next page boundary, and this is the default if no *size* is given. The assembler assembles words containing null values (0) up to the next x-word boundary.

Operand of	1	aligns SPC to word boundary
	2	aligns SPC to long word/even boundary
	128	aligns SPC to page boundary

The assembler aligns the SPC on an x-word boundary *within* the current section.

Example

This example shows several types of alignment, including .align 2, .align 4, and a default .align.

```
1  0000  0004  .byte    4
2              .align   2
3  0002  0045  .string  "Errorcnt"
   0003  0072
   0004  0072
   0005  006F
   0006  0072
   0007  0063
   0008  006E
   0009  0074
4              .align
5  0080  6000  .field   3,3
6  0080  6A00  .field   5,4
7              .align   2
8  0082  6000  .field   3,3
9              .align   8
10 0088  5000  .field   5,4
11             .align
12 0100  0004  .byte    4
```

Syntax

.bss *symbol, size in words* [*, alignment*]

Description

The **.bss** directive reserves space for variables in the .bss section. This directive is usually used to allocate variables in RAM.

symbol	points to the first location reserved by this invocation of the .usect directive. The symbol corresponds to the name of the variable for which you're reserving space.
size in words	is an expression that defines the number of words that are reserved in section *name.*
alignment	is an optional parameter. This flag causes the assembler to allocate size on long word boundaries.

The assembler follows one rule when it allocates space in the .bss section:

Whenever a hole is left in memory, the .bss directive attempts to fill it. When a .bss directive is assembled, the assembler searches its list of holes left by previous .bss directives and tries to allocate the current block into one of the holes.

Section directives for initialized sections (.text, .data, and .sect) end the current section and begin assembling into another section. The .bss directive, however, does not affect the current section. The assembler assembles the .bss directive and then resumes assembling code into the current section.

Example

In this example, the .bss directive is used to allocate space for two variables, TEMP and ARRAY. The symbol TEMP points to 4 words of uninitialized space (at .bss SPC = 550h). The symbol ARRAY points to 100 words of uninitialized space (at .bss SPC = 554h); this space must be allocated contiguously within a page. Note that symbols declared with the .bss directive can be referenced in the same manner as other symbols.

```
1                       .setsect ".text", 0500h
2                       .setsect ".bss", 0550h
3                  *******************************************************
4                  *          Start Assembling into .text section        *
5                  *******************************************************
6   000500                .text
7   000500 E800           A = #0
8
9                  *******************************************************
10                 *           Allocate 4 words in .bss for TEMP         *
11                 *******************************************************
12  000550         Var_1: .bss   TEMP,4
13
14                 *******************************************************
15                 *               Still in .text section                *
16                 *******************************************************
17  000501 F000           A = A + #56h
    000502 0056
18  000503 F066           A = T * #73h
    000504 0073
19
20                 *******************************************************
21                 *  Allocate 100 words in .bss for the symbol named ARRAY *
22                 *******************************************************
23  000554                .bss   ARRAY,100
24
25                 *******************************************************
26                 *         Assemble more code into .text section       *
27                 *******************************************************
28  000505 8050           @Var_1 = A
29                        .end
```

Syntax

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

Description

The **.byte** directive places one or more bytes into consecutive words of the current section. Each byte is placed in a word by itself; the eight LSBs are filled with 0s. A *value* can be either of the following:

- An expression that the assembler evaluates and treats as an 8-bit signed number
- A character string enclosed in double quotes. Each character in a string represents a separate value.

Values are not packed or sign extended; each byte occupies the eight least significant bits of a full 16-bit word. The assembler truncates values greater than eight bits. You can use up to 100 value parameters, but the total line length cannot exceed 200 characters.

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

Example

In this example 8-bit values—10, –1, abc, and a—are placed into consecutive words in memory. The label STRX has the value 100h, which is the location of the first initialized word.

```
1  0000                .space   100h * 16
2  0100  000a  STRX    .byte    10, –1, ”abc”, ’a’
   0101  00ff
   0102  0061
   0103  0062
   0104  0063
   0105  0061
```

Syntax

.copy [**"**]*filename*[**"**]
.include [**"**]*filename*[**"**]

Description

The **.copy** and **.include** directives tell the assembler to read source statements from a different file. The statements that are assembled from a copy file are printed in the assembly listing. The statements that are assembled from an included file are *not* printed in the assembly listing, regardless of the number of .list/.nolist directives assembled. 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. It may be enclosed in double quotes and must follow operating system conventions. You can specify a full pathname (for example, c:\dsp\file1.asm). If you do not specify a full pathname, the assembler searches for the file in:

1) The directory that contains the current source file.
2) Any directories named with the –i assembler option.
3) Any directories specified by the environment variable A_DIR.

The .copy and .include directives can be nested within a file being copied or included. The assembler limits nesting to ten levels; the host operating system may set additional restrictions. The assembler precedes the line numbers of copied files with a letter code to identify the level of copying. An A indicates the first copied file, B indicates a second copied file, etc..

Example 1

In this example, the .copy directive is used to read and assemble source statements from other files; then the assembler resumes assembling into the current file.

The original file, copy.asm, contains a .copy statement copying the file byte.asm. When copy.asm assembles, the assembler copies byte.asm into its place in the listing (note listing below). The copy file byte.asm contains a .copy statement for a second file, word.asm.

When it encounters the .copy statement for word.asm, the assembler switches to word.asm to continue copying and assembling. Then the assembler returns to its place in byte.asm to continue copying and assembling. After completing assembly of byte.asm, the assembler returns to copy.asm to assemble its remaining statement.

copy.asm (Source file)	**byte.asm (First copy file)**	**word.asm (Second copy file)**
`.space 29`	`** In byte.asm`	`** In word.asm`
`.copy "byte.asm"`	`.byte 32,1+ 'A'`	`.word 0ABCDh, 56q`
	`.copy "word.asm"`	
`**Back in original file`	`** Back in byte.asm`	
`.pstring "done"`	`.byte 67h + 3q`	

Listing file:

```
      1  0000          .space 29
      2                .copy "byte.asm"
A     1                ** In byte.asm
A     2  0002  0020    .byte 32,1+ 'A'
         0003  0042
A     3                .copy "word.asm"
B     1                * In word.asm
B     2  0004  ABC     .word 0ABCDh, 56q
         0005  002
A     4                ** Back in byte.asm
A     5  0006  006     .byte 67h + 3q
      3
      4                ** Back in original file
      5  0007  646F    .pstring "done"
         0008  6E65
```

Example 2 In this example, the .include directive is used to read and assemble source statements from other files; then the assembler resumes assembling into the current file. The mechanism is similar to the .copy directive, except that statements are not printed in the listing file.

include.asm (Source file)

```
   .space  29
   .include "byte2.asm"

**Back in original file
   .string "done"
```

byte2.asm (First include file)

```
** In byte2.asm
    .byte 32,1+ 'A'
    .include "word2.asm"
** Back in byte2.asm
    .byte  67h + 3q
```

word2.asm (Second include file)

```
** In word2.asm
    .word 0ABCDh, 56q
```

Listing file:

```
1  0000          .space 29
2                .include "byte2.asm"
3
4                ** Back in original file
5  0007  0064    .string "done"
   0008  006F
   0009  006E
   000a  0065
```

Syntax **.data**

Description The **.data** directive tells the assembler to begin assembling source code into the .data section; .data becomes the current section. The .data section is normally used to contain tables of data or preinitialized variables.

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 use a section control directive.

Example In this example, code is assembled into the .data and .text sections.

```
 1                    ********************************************
 2                    **          Reserve space in .data.       **
 3                    ********************************************
 4  0000   .                      .data
 5  0000                          .space      0CCh
 6
 7                    ********************************************
 8                    **           Assemble into .text.         **
 9                    ********************************************
 10 0000                          .text
                                                ; constant into .data.
 11 0000                  INDEX   .set        0
 12 0000 e800                     A = #INDEX
 13
 14                   ********************************************
 15                   **          Assemble into .data.          **
 16                   ********************************************
 17 000d                  Table:  .data
 18 000d ffff                     .word     –1    ; Assemble 16–bit
 19
 20 000e 00ff                     .byte     0FFh  ; Assemble 8–bit
 21                                             ; constant into .data.
 22
 23                   ********************************************
 24                   **          Assemble into .text.          **
 25                   ********************************************
 26 0001                          .text
 27 0001 000d                     A = A + @Table
 28
 29                   ********************************************
 30                   **  Resume assembling into the .data section **
 31                   **  at address 0Fh.                       **
 32                   ********************************************
 33 000f                          .data
```

Syntax

```
.end
```

Description

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

This directive has the same effect as an end-of-file character. You can use .end when you're debugging and would like to stop assembling at a specific point in your code.

Example

This example shows how the .end directive terminates assembly. If any source statements follow the .end directive, the assembler ignores them.

Source File:

```
START: .space      300
TEMP   .set        15
       .bss        LOC1, 48h
       A = |A|
       A = A + #TEMP
       @LOC1 = A
       .end
       .byte      4
       .word    CCCh
```

Listing file:

```
1  0000          START:    .space   300
2  000F          TEMP      .set     15
3  0000                    .bss     LOC1, 48h
4  0013  F485              A = |A|
5  0014  F000              A = A + #TEMP
   0015  000F
6  0016  8000              @LOC1 = A
7                          .end
```

Syntax

.eval *well-defined expression, substitution symbol*

Description

The **.eval** directive performs arithmetic on substitution symbols, which are stored in the substitution symbol table. This directive evaluates the expression and assigns the *string value* of the result to the substitution symbol. The .eval directive is especially useful as a counter in .loop/.endloop blocks.

well-defined expression	is an alphanumeric expression consisting of legal values that have been previously defined.
substitution symbol	is a required parameter that must be a valid symbol name. The substitution symbol may be 32 characters long and must begin with a letter. Remaining characters of the symbol can be a combination of alphanumeric characters, underscores, and dollar signs.

Example This example shows how .eval can be used.

```
   1                    .sslist;show expanded sub. symbols
   2                 *
   3                 *  .eval example
   4                 *
   5
   6  0000 f000     A += #100
      0001 0064
   7  0002 6d90     *AR0+
   8  0003 6d90     *AR0+
   9
   10                .asg      0, x
   11                .loop     5
   12                .eval     x+1, x
   13                .word     x
   14                .endloop
1                    .eval     x+1, x
#                    .eval     0+1, x
1     0004 0001      .word     x
#                    .word        1
1                    .eval     x+1, x
#                    .eval     1+1, x
1     0005 0002      .word     x
#                    .word     2
1                    .eval     x+1, x
#                    .eval     2+1, x
1     0006 0003      .word     x
#                    .word     3
1                    .eval     x+1, x
#                    .eval     3+1, x
1     0007 0004      .word     x
#                    .word     4
1                    .eval     x+1, x
#                    .eval     4+1, x
1     0008 0005      .word     x
#                    .word     5
```

Syntax

.field *value* [, *size in bits*]

Description

The **.field** directive can initialize multiple-bit fields within a single word of memory. This directive has two operands:

value is a required parameter; it is an expression that is evaluated and placed in the field.

size is an optional parameter; it specifies a number from 1 to 32, which is the number of bits in the field. If you do not specify a size, the assembler assumes that the size is 16 bits. If you specify a size of 16 or more, the field will start on a word boundary. If you specify a value that cannot fit into *size* bits, the assembler truncates the value and issues an error message. For example, .field 3,1 causes the assembler to truncate the value 3 to 1; the assembler also prints the message:

```
***warning – value truncated.
```

Successive .field directives pack values into the specified number of bits starting at the current word. Fields are packed starting at the most significant part of the word, moving toward the least significant part as more fields are added. If the assembler encounters a field size that does not fit into the current word, it writes out the word, increments the SPC, and begins packing fields into the next word. You can use the .align directive with an operand of 1 to force the next .field directive to begin packing into a new word.

If you use a label, it points to the word that contains the specified field.

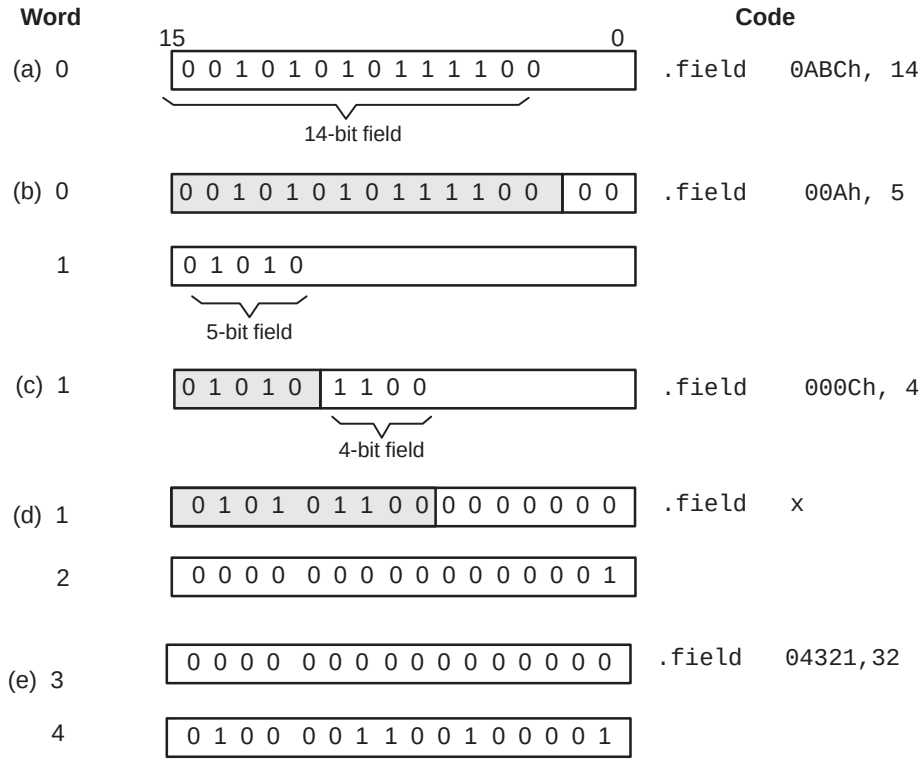
Example

This example shows how fields are packed into a word. Notice that the SPC does not change until a word is filled and the next word is begun. For more examples of the .field directive, see page 5-26.

```
1                    ************************************
2                    **    Initialize a 14-bit field.  **
3                    ************************************
4   0000 2AF0           .field   0ABCh, 14
5
6                    ************************************
7                    **    Initialize a 5-bit field    **
8                    **         in a new word.         **
9                    ************************************
10  0001 5000        L_F:  .field   0Ah, 5
11
12                   ***********************************
13                   **    Initialize a 4-bit field    **
14                   **        in the same word.       **
15                   ************************************
16  0001 5600 x      :     .field   0Ch, 4
17
18                   ************************************
19                   **    16-bit relocatable field    **
20                   **        in the next word.       **
21                   ************************************
22  0002 0001           .field   x
23
24                   ************************************
25                   **   Initialize a 32-bit field.   **
26                   ************************************
27  0003 0000              .field   04321h, 32
    0004 4321
```

Figure C–1 shows how the directives in this example affect memory.

Figure C–1. The .field Directive



Syntax

.float *value*$_1$ [, ... , *value*$_n$]
.xfloat *value*$_1$ [, ... , *value*$_n$]

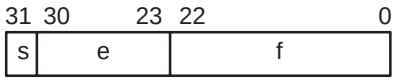
Description

The **.float** and **.xfloat** directives place the floating-point representation of one or more floating-point constants into the current section. The *value* must be a floating-point constant or a symbol that has been equated to a floating-point constant. Each constant is converted to a floating-point value in IEEE single-precision 32-bit format. Floating point constants are aligned on the long-word boundaries unless the .xfloat directive is used. The .xfloat directive performs the same function as the .float directive but does not align the result on the long word boundary.

The 32-bit value consists of three fields:

Field	Meaning
s	A 1-bit sign field
e	An 8-bit biased exponent
f	A 23-bit fraction

The value is stored most significant word first, least significant word second, in the following format:



Example

This example shows the .float directive.

```
1 0000 E904  .float  –1.0e25
  0001 5951
2 0002 4040         .float  3
  0003 0000
3 0004 42F6         .float  123
  0005 0000
```

Syntax

```
.if   well-defined expression
.elseif   well-defined expression
.else
.endif
```

Description

Four directives provide conditional assembly:

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

- If the expression evaluates to *true* (nonzero), the assembler assembles the code that follows the expression (up to a .elseif, .else, or .endif).
- If the expression evaluates to *false* (0), the assembler assembles code that follows a .elseif (if present), .else (if present), or .endif (if no .elseif or .else is present).

The **.elseif** directive identifies a block of code to be assembled when the .if expression is false (0) and the .elseif expression is true (nonzero). When the .elseif expression is false, the assembler continues to the next .elseif (if present), .else (if present) or .endif (if no .elseif or .else is present). The .elseif directive is optional in the conditional blocks, and more than one .elseif can be used. If an expression is false and there is no .elseif statement, the assembler continues with the code that follows a .else (if present) or a .endif.

The **.else** directive identifies a block of code that the assembler assembles when the .if expression and all .elseif expressions are 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.

The .elseif and .else directives can be used in the same conditional assembly block and the .elseif directive can be used more than once within a conditional assembly block.

For information about relational operators, see subsection 5.9.4, page 5-17.

Example This example shows conditional assembly.

```
1                       SYM1        .set   1
2                       SYM2        .set   2
3                       SYM3        .set   3
4                       SYM4        .set   4
5
6                       If_4:       .if        SYM4 = SYM2 * SYM2
7   0000 0004                       .byte      SYM4              ; Equal values
8                                   .else
9                                   .byte      SYM2 * SYM2       ; Unequal values
10                                  .endif
11
12                      If_5:       .if        SYM1 <= 10
13  0001 000a                       .byte      10                ; Less than / equal
14                                  .else
15                                  .byte      SYM1              ; Greater than
16                                  .endif
17
18                      If_6:       .if        SYM3 * SYM2 != SYM4 + SYM2
19                                  .byte      SYM3 * SYM2       ; Unequal value
20                                  .else
21  0002 0008                       .byte      SYM4 + SYM4       ; Equal values
22                                  .endif
23
24                      If_7:       .if        SYM1 = 2
25                                  .byte      SYM1
26                                  .elseif    SYM2 + SYM3 = 5
27  0003 0005                       .byte      SYM2 + SYM3
28                                  .endif
```

Syntax

.int *value*$_1$ [, ... , *value*$_n$]
.**word** *value*$_1$ [, ... , *value*$_n$]

Description

The **.int** and **.word** directives are equivalent; they place one or more values into consecutive 16-bit fields in the current section.

You can use as many values as fit on a single line. If you use a label, it points to the first word that is initialized.

Example 3

In this example, the .int directive is used to initialize words.

```
1  0000                .space 73h
2  0000                .bss   PAGE, 128
3  0080                .bss   SYMPTR, 3
4  0008 E856     INST:  A = #56h
5  0009 000A           .int   10, SYMPTR, –1, 35 + ’a’, INST
   000a 0080
   000b FFFF
   000c 0084
   000d 0008
```

Example 4

In this example, the .word directive is used to initialize words. The symbol WordX points to the first word that is reserved.

```
1  0000 0C80     WORDX: .word  3200, 1 + ’AB’, –0AFh, ’X’
   0001 4143
   0002 FF51
   0003 0058
```

Syntax

.length *page length*
.width *page width*

Description

The **.length** directive sets the page length of the output listing file. It affects the current and following pages. You can reset the page length with another .length directive.

- Default length: 60 lines
- Minimum length: 1 line
- Maximum length: 32,767 lines

The **.width** directive sets the page width of the output listing file. It affects the next line assembled and the lines following; you can reset the page width with another .width directive.

- Default width: 80 characters
- Minimum width: 80 characters
- Maximum width: 200 characters

The width refers to a full line in a listing file; the line counter value, SPC value, and object code are counted as part of the width of a line. Comments and other portions of a source statement that extend beyond the page width are truncated in the listing.

The assembler does not list the .width and .length directives.

Example

In this example, the page length and width are changed.

```
*************************************************
**       Page length = 65 lines.               **
**       Page width  = 85 characters.          **
*************************************************
         .length  65
         .width   85

*************************************************
**       Page length = 55 lines.               **
**       Page width  = 100 characters.         **
*************************************************
         .length  55
         .width   100
```

Syntax

```
.list
.nolist
```

Description

Two directives enable you to control the printing of the source listing:

The **.list** directive allows the printing of the source listing.

The **.nolist** directive suppresses the source listing output until a .list directive is encountered. The .nolist directive can be used to reduce assembly time and the source listing size.

The assembler does not print the .list or .nolist directives or the source statements that appear after a .nolist directive. However, it continues to increment the line counter. You can nest the .list/.nolist directives; each .nolist needs a matching .list to restore the listing.

By default, the source listing is printed to the listing file; the assembler acts as if the .list directive had been specified.

Note:

If you don't request a listing file when you invoke the assembler, the assembler ignores the .list directive.

Example

This example shows how the .copy directive inserts source statements from another file. The first time this directive is encountered, the assembler lists the copied source lines in the listing file. The second time this directive is encountered, the assembler does not list the copied source lines, because a .nolist directive was assembled. The .nolist, the second .copy, and the .list directives do not appear in the listing file. Also, the line counter is incremented, even when source statements are not listed.

Source file:

```
   .copy ″copy2.asm″
* Back in original file
   NOP
   .nolist
   .copy ″copy2.asm″
   .list
* Back in original file
   .string  ″Done″
```

Listing file:

```
       1                        .copy   ″copy2.asm″
 A     1                 * In copy2.asm (copy file)
 A     2 0000 0020              .word 32, 1 + ’A’
         0001 0042
       2                 * Back in original file
       3 0002 F495              NOP
       7                 * Back in original file
       8 0005 0044              .string ″Done″
         0006 006F
         0007 006E
         0008 0065
```

Syntax

.long *value*$_1$ [, ... , *value*$_n$]
.xlong *value*$_1$ [, ... , *value*$_n$]

Description

The **.long** and **.xlong** directives place one or more 32-bit values into consecutive words in the current section. The most significant word is stored first. The .long directive aligns the result on the long word boundary, while the .xlong directive does not.

You can use up to 100 values, but they must fit on a single source statement line. If you use a label, it points to the first word that is initialized.

Example

This example shows how the .long and .xlong directives initialize double words.

```
1 0000 0000  DAT1: .long  0ABCDh, 'A' + 100h, 'g', 'o'
  0001 ABCD
  0002 0000
  0003 0141
  0004 0000
  0005 0067
  0006 0000
  0007 006F
2 0008 0000       .xlong  DAT1, 0AABBCCDDh
  0009 0000
  000a AABB
  000b CCDD
3 000c       DAT2:
```

Syntax

.loop [*well-defined expression*]
.break [*well-defined expression*]
.endloop

Description

Three directives enable you to repeatedly assemble a block of code:

The **.loop** directive begins a repeatable block of code. The optional expression evaluates to the loop count (the number of loops to be performed). If there is no expression, the loop count defaults to 1024, unless the assembler first encounters a .break directive with an expression that is true (nonzero) or omitted.

The **.break** directive is optional, along with its expression. When the expression is false (0), the loop continues. When the expression is true (nonzero), or omitted, the assembler breaks the loop and assembles the code after the .endloop directive.

The **.endloop** directive terminates a repeatable block of code; it executes when the .break directive is true (nonzero) or when number of loops performed equals the loop count given by .loop

Example

This example illustrates how these directives can be used with the .eval directive.

```
       1                    .eval       0,x
       2              COEF  .loop
       3                    .word       x*100
       4                    .eval       x+1, x
       5                    .break      x = 6
       6                    .endloop
1        0000 0000          .word       0*100
1                           .eval       0+1, x
1                           .break      1 = 6
1        0001 0064          .word       1*100
1                           .eval       1+1, x
1                           .break      2 = 6
1        0002 00C8          .word       2*100
1                           .eval       2+1, x
1                           .break      3 = 6
1        0003 012C          .word       3*100
1                           .eval       3+1, x
1                           .break      4 = 6
1        0004 0190          .word       4*100
1                           .eval       4+1, x
1                           .break      5 = 6
1        0005 01F4          .word       5*100
1                           .eval       5+1, x
1                           .break      6 = 6
```

Syntax

```
.page
```

Description

The **.page** directive produces a page eject in the listing file. The .page directive is not printed in the source listing, but the assembler increments the line counter when it encounters it. Using the .page directive to divide the source listing into logical divisions improves program readability.

Example

This example shows how the page directive causes the assembler to begin a new page of the source listing.

Source file:

```
        .title   "**** Page Directive Example ****"
;         .
;         .
;         .
        .page
```

Listing file:

```
TMS320C54x DSKplus Assembler       Version x.xx     Sun Apr 23 13:06:08 1995
 Copyright (c) 1996    Texas Instruments Incorporated
**** Page Directive Example ****                                    PAGE    1

       2            ;        .
       3            ;        .
       4            ;        .
TMS320C54x DSKplus Assembler       Version x.xx     Sun Apr 23 13:06:08 1995
 Copyright (c) 1996    Texas Instruments Incorporated
**** Page Directive Example ****                                    PAGE    2
```

Syntax

.sect ”*section name*”

Description

The **.sect** directive defines a named section that can be used like the default .text and .data sections. The .sect directive begins assembling source code into the named section.

The *section name* identifies a section that the assembler assembles code into. The name is significant to eight characters and must be enclosed in double quotes.

Example

This example defines two special-purpose sections, Sym_Defs and Vars, and assembles code into them.

```
 1                *******************************************
 2                **   Begin assembling into .text section.   **
 3                *******************************************
 4 0000                  .text
 5 0000 E878             A = #78h        ; Assembled into .text
 6 0001 F000             A = A + #36h    ; Assembled into .text
   0002 0036
 7                *******************************************
 8                **  Begin assembling into Sym_Defs section. **
 9                *******************************************
10 0000                  .sect   ”Sym_Defs”
11 0000 3D4C             .float  0.05    ; Assembled into Sym_Defs
   0001 CCCD
12 0002 00AA  X:         .word   0AAh    ; Assembled into Sym_Defs
13 0003 F000             A = A + #X      ; Assembled into Sym_Defs
   0004 0002
14               *******************************************
15               **   Begin assembling into Vars section.    **
16               *******************************************
17 0000                  .sect   ”Vars”
18      0010     WORD_LEN        .set    16
19      0020     DWORD_LEN       .set    WORD_LEN * 2
20      0008     BYTE_LEN        .set    WORD_LEN / 2
21               *******************************************
22               **   Resume assembling into .text section.  **
23               *******************************************
24 0003                  .text
25 0003 F000             A = A + #42h    ; Assembled into .text
   0004 0042
26 0005 0003             .byte   3, 4    ; Assembled into .text
   0006 0004
27               *******************************************
28               **   Resume assembling into Vars section.   **
29               *******************************************
30 0000                  .sect   ”Vars”
31 0000 000D             .field  13, WORD_LEN
32 0001 0A00             .field  0Ah, BYTE_LEN
33 0002 0000             .field  10q, DWORD_LEN
   0003 0008
34
```

Syntax

symbol **.set** *value*
symbol **.equ** *value*

Description

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

symbol points to the first location reserved by this invocation of the .usect directive. The symbol corresponds to the name of the variable for which you're reserving space.

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

Undefined external symbols and symbols that are defined later in the module cannot be used in the expression. If the expression is relocatable, the symbol to which it is assigned is also relocatable.

The value of the expression appears in the object field of the listing. This value is not part of the actual object code and is not written to the output file.

Symbols defined with .set can be made externally visible with the .def or .global directive. In this way, you can define global absolute constants.

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

```
 1                  ********************************************
 2                  **   Equate symbol AUX_R1 to register AR1   **
 3                  **   and use it instead of the register.    **
 4                  ********************************************
 5       0011  AUX_R1  .set    AR1
 6  0000 7711          MMR(AUX_R1) = #56h
    0001 0056
 7
 8                  ********************************************
 9                  **   Set symbol index to an integer expr.   **
10                  **    and use it as an immediate operand.   **
11                  ********************************************
12       0035  INDEX   .equ    100/2 +3
13  0002 F000          A = A + #INDEX
    0003 0035
14
15                  ********************************************
16                  ** Set symbol SYMTAB to a relocatable expr. **
17                  **   and use it as a relocatable operand.   **
18                  ********************************************
19  0004 000A  LABEL   .word   10
20       0005  SYMTAB  .set    LABEL + 1
21
22                  ********************************************
23                  **   Set symbol NSYMS equal to the symbol   **
24                  **   INDEX and use it as you would INDEX.   **
25                  ********************************************
26       0035  NSYMS   .set    INDEX
27  0005 0035          .word   NSYMS
```

Syntax

.setsect **"***section name***"***, address [,page]*

Description

The **.setsect** directive initializes the absolute address of the named section. The .setsect directive must be specified before the declaration of a section (.bss, .text, etc.) or the section program counter (SPC) begins to assemble code into the section's default address (0).

section name can be either a section directive such as .text or .bss. If *section name* is an initialized or uninitialized named section, such as .sect or .usect, the actual name is used to reference the section (see the following example).

address specifies the beginning address of the section specified as the section name. The assembler maintains each section counter of the assembly program.

page specifies where the section resides in memory. Page 0 is program memory and page 1 is data. You can force the program memory to be loaded to data space by specifying page = 1 in the *page* field. The same applies for data memory being loaded to program space.

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

```
 1                       .setsect ".text", 0500h
 2                       .setsect ".bss", 0550h
 3                       .setsect "Vectors", 080h
 4                  ************************************************************
 5                  *            Start Assembling into .text section           *
 6                  ************************************************************
 7 000500                  .text
 8 000500 E800             A = #0
 9
10                  ************************************************************
11                  *             Allocate 4 words in .bss for TEMP            *
12                  ************************************************************
13 000550           Var_1:  .bss    TEMP,4
14
15                  ************************************************************
16                  *                  Still in .text section                  *
17                  ************************************************************
18 000501 F000             A = A + #56h
   000502 0056
19 000503 F066             A = T * #73h
   000504 0073
20
21                  ************************************************************
22                  *              Assemble into the Vectors section           *
23                  ************************************************************
24 000080                  .sect  "Vectors"
25 000080 F4EB  RESET      return_enable
26 000081                  .space (3*16)
27 000084 F4EB  NMI:       return_enable
28
29                  ************************************************************
30                  *             Assemble more code into .text section        *
31                  ************************************************************
32 000505                  .text
33 000505 8050             @Var_1 = A
34                        .end
```

Syntax

.space *size in bits*
.bes *size in bits*

Description

The **.space** and **.bes** directives reserve *size* number of bits in the current section and fill them with 0s.

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

Example

This example shows how memory is reserved with the .space and .bes directives.

```
 1                  ******************************************
 2                  **  Begin assembling into .text section.   **
 3                  ******************************************
 4 0000                     .text
 5
 6                  ******************************************
 7                  **  Reserve 0F0 bits (15 words) in the     **
 8                  **           .text section.                **
 9                  ******************************************
10 0000                     .space  0F0h
11 000f 0100               .word   100h, 200h
   0010 0200
12                  ******************************************
13                  **  Begin assembling into .data section.   **
14                  ******************************************
15 0000                     .data
16 0000 0049               .string "In .data"
   0001 006E
   0002 0020
   0003 002E
   0004 0064
   0005 0061
   0006 0074
   0007 0061
17                  ******************************************
18                  **  Reserve 100 bits in the .data section; **
19                  **   RES_1 points to the first word that   **
20                  **         contains reserved bits.         **
21                  ******************************************
22 0008             RES_1:  .space  100
23 000f 000F               .word   15
24 0010 0008               .word   RES_1
25
26                  ******************************************
27                  **  Reserve 20 bits in the .data section;  **
28                  **  RES_2 points to the last word that     **
29                  **         contains reserved bits.         **
30                  ******************************************
31 0012             RES_2:  .bes    20
32 0013 0036               .word   36h
33 0014 0012               .word   RES_2
```

Syntax

.string ”*string*$_1$” [, ... , ”*string*$_n$”]
.pstring ”*string*$_1$” [, ... , ”*string*$_n$”]

Description

The **.string** and **.pstring** directives place 8-bit characters from a character string into the current section. With the .string directive, each 8 bit character has its own 16-bit word, but with the .pstring directive, the data is packed so that each word contains two 8-bit bytes. Each *string* is either:

- An expression that the assembler evaluates and treats as a 16-bit signed number, or
- A character string enclosed in double quotes. Each character in a string represents a separate byte.

With .pstring, values are packed into words starting with the most significant byte of the word. Any unused space is padded with null bytes.

The assembler truncates any values that are greater than eight bits. You may have up to 100 operands, but they must fit on a single source statement line.

If you use a label, it points to the location of the first word that is initialized.

Example

This example shows 8-bit values placed into words in the current section.

```
1 0000 0041  Str_Ptr:  .string  ”ABCD”
  0001 0042
  0002 0043
  0003 0044
2 0004 0041            .string  41h, 42h, 43h, 44h
  0005 0042
  0006 0043
  0007 0044
3 0008 4175            .pstring ”Austin”, ”Houston”
  0009 7374
  000a 696E
  000b 486F
  000c 7573
  000d 746F
  000e 6E00
4 000f 0030            .string  36 + 12
```

Syntax

```
.text
```

Description

The **.text** directive tells the assembler to begin assembling into the .text section, which usually contains executable code. The section program counter is set to the address specified by the preceding *.setsect* directive if nothing has yet been assembled into the .text section. If code has already been assembled into the .text section, the section program counter is restored to its previous value in the section.

.text is the default section. Therefore, at the beginning of an assembly, the assembler assembles code into the .text section unless you specify a different sections directives (.data or .sect).

Example

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

```
 1                ****************************************
 2                ** Begin assembling into .data section.**
 3                ****************************************
 4 0000                   .data
 5 0000 000a              .byte   0Ah, 0Bh
   0001 000b
 6
 7                ****************************************
 8                ** Begin assembling into .text section. **
 9                ****************************************
10 0000                   .text
11 0000 0041     START:  .string "A","B","C"
   0001 0042
   0002 0043
12 0003 0058     END:    .string "X","Y","Z"
   0004 0059
   0005 005a
13
14 0006 0000             A = A + @START
15 0007 0003             A = A + @END
16
17               ****************************************
18               **  Resume assembling into .data section.**
19               ****************************************
20 0002                  .data
21 0002 000c             .byte   0Ch, 0Dh
   0003 000d
22
23               ****************************************
24               **  Resume assembling into .text section.**
25               ****************************************
26 0008                  .text
27 0008 0051             .string "Quit"
   0009 0075
   000a 0069
   000b 0074
```

Syntax

.title ”*string*”

Description

The **.title** directive supplies a title that is printed in the heading on each listing page. The source statement itself is not printed, but the line counter is incremented.

The *string* is a quote-enclosed title of up to 65 characters. If you supply more than 65 characters, the assembler truncates the string and issues a warning.

The assembler prints the title on the page that follows the directive, and on subsequent pages until another .title directive is processed. If you want a title on the first page, the first source statement must contain a .title directive.

Example

In this example, one title is printed on the first page and a different title on succeeding pages.

Source file:

```
          .title  ”**** Fast Fourier Transforms ****”
;         .
;         .
;         .
          .title  ”**** Floating–Point Routines ****”
         .page
```

Listing file:

```
TMS320C54x DSKplus Assembler       Version x.xx     Sun Apr 23 16:25:49 1995
 Copyright (c) 1996    Texas Instruments Incorporated
**** Fast Fourier Transforms ****                                    PAGE    1

       2            ;         .
       3            ;         .
       4            ;         .
TMS320C54x DSKplus Assembler       Version x.xx     Sun Apr 23 16:25:49 1995
 Copyright (c) 1996    Texas Instruments Incorporated
**** Floating–Point Routines ****                                    PAGE    2
```

Syntax

symbol **.usect "***section name***"***, size in words* [, *alignment flag*]

Description

The **.usect** directive reserves space for variables in an uninitialized, named section. This directive is similar to the .bss directive; both simply reserve space for data and have no contents. However, .usect defines additional sections that can be placed anywhere in memory, independently of the .bss section.

symbol	points to the first location reserved by this invocation of the .usect directive. The symbol corresponds to the name of the variable for which you are reserving space.
section name	must be enclosed in double quotes; only the first eight characters are significant. This parameter names the uninitialized section.
size in words	is an expression that defines the number of words that are reserved in section *name.*
alignment flag	is an optional parameter. This flag causes the assembler to allocate size on long word boundaries.

Other sections directives (.text, .data, and .sect) end the current section and tell the assembler to begin assembling into another section. The .usect and the .bss directives, however, do not affect the current section. The assembler assembles the .usect and the .bss directives and then resumes assembling into the current section.

Variables that can be located contiguously in memory can be defined in the same specified section; to do so, repeat the .usect directive with the same section name.

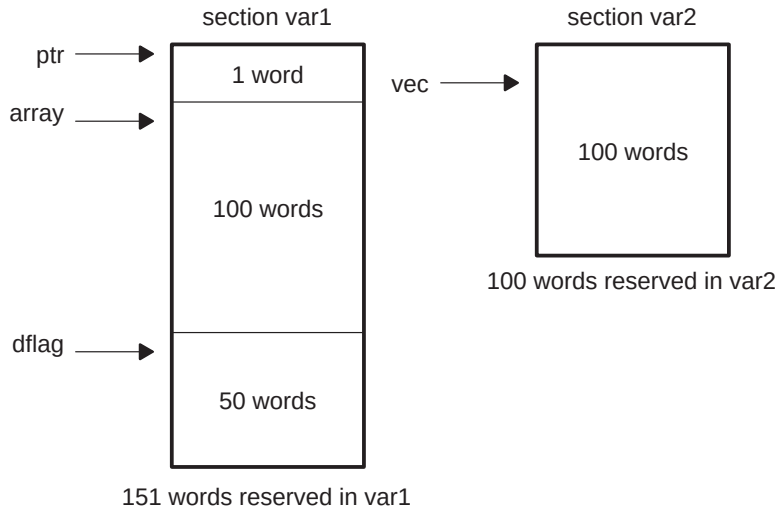
Example

This example uses the .usect directive to define two uninitialized, named sections, var1 and var2. The symbol ptr points to the first word reserved in the var1 section. The symbol array points to the first word in a block of 100 words reserved in var1, and dflag points to the first word in a block of 50 words in var1. The symbol vec points to the first word reserved in the var2 section.

Figure C–2, page C-37, shows how this example reserves space in two uninitialized sections, var1 and var2.

```
 1                ****************************************
 2                **     Assemble into .text section.     **
 3                ****************************************
 4 0000                   .text
 5 0000 E803              A = A + #03h
 6
 7                ****************************************
 8                **       Reserve 1 word in var1.        **
 9                ****************************************
10 0000           ptr     .usect  "var1", 1
11
12                ****************************************
13                **      Reserve 100 words in var1.      **
14                ****************************************
15 0001           array   .usect  "var1", 100
16
17 0001 F000              A = A + #037h     ; Still in .text
   0002 0037
18
19                ****************************************
20                **      Reserve 50 words in var1.       **
21                ****************************************
22 0065           dflag   .usect  "var1", 50
23
24 0003 F000              A = A + #dflag    ; Still in .text
   0004 0065
25
26                ****************************************
27                **      Reserve 100 words in var2.      **
28                ****************************************
29 0000           vec     .usect  "var2", 100
30
31 0005 F000              A = A + #vec      ; Still in .text
   0006 0000
```

Figure C–2. Using the .usect Directive



Appendix D

Assembler Error Messages

The assembler issues two types of error messages:

- Fatal
- Nonfatal

When the assembler completes its second pass, it reports any errors that it encountered during the assembly. It also prints these errors in the listing file (if one is created); an error is printed following the source line that incurred it.

This appendix lists the three types of assembler error messages in alphabetical order according to the error message number. Most errors are fatal errors; if an error is not fatal, this is noted in the assembler listing file. Each error message consists of its class number and text showing the specific error that was detected. Each class number group has a *Description* of the problem and an *Action* that suggests possible remedies.

E0000: Comma required to separate arguments
Left parenthesis expected
Matching right parenthesis is missing
Missing right quote of string constant
Syntax Error

Description These are errors about general syntax. The required syntax is not present.

Action Correct the source per the error message text.

E0001: Section *sym* is not defined
Section *sym* is not an initialized section

Description These are errors about invalid symbol names. A symbol is invalid for the context in which it is used.

Action Correct the source per the error message text.

E0002: Invalid directive specification
Invalid mnemonic specification

Description These errors are about invalid mnemonics. The instruction or directive specified was not recognized.

Action Check the directive or instruction used.

E0003: Cluttered character operand encountered
Cluttered string constant operand encountered
Cluttered identifier operand encountered
Condition must be EQ, LT, GT, or NEQ
Condition must be srcLT, LEQ, GT, or GEQ
Illegal condition, operand, or combination
Illegal indirect memaddr specification
Invalid binary constant specified
Incorrect bit symbol for specified status register
Invalid constant specification
Invalid decimal constant specified
Invalid float constant specified
Invalid hex constant specified
Invalid immediate expression or shift value
Invalid octal constant specified
Invalid operand Shift value out of range

Description These are errors about invalid operands. The instruction, parameter, or other operand specified was not recognized.

Action Correct the source per the error message text.

E0004: **Absolute, well-defined integer value expected**
Accumulator specified in second half of parallel instruction may not be the same as the first
Data size must be equal to pointer size
Expecting accumulator A or B
Expecting ASM or shift value
Expecting dual memory addressing
Identifier operand expected
Illegal character argument specified
Illegal floating-point expression
Illegal string constant operand specified
Invalid identifier, *sym*, specified
Not expecting direct operand *op*
Not expecting indirect operand *op*
Not expecting immediate value operand *op*
Operand must be auxiliary register or SP
Operand must be auxiliary register
Offset Addressing modes not legal for MMRs
Pointer too big for this data size
String constant or substitution symbol expected
Substitution symbol operand expected

Description These errors are about illegal operands. The instruction, parameter or other operand specified was not legal for this syntax.

Action Correct the source per the error message text.

E0005: **Missing field value operand**
Missing operand(s)

Description These are errors about missing operands; a required operand is not supplied.

Action Correct the source so that all required operands are declared.

E0006: .break must occur within a loop
Conditional assembly mismatch
Matching .endloop missing
No matching .if specified
No matching .endif specified
No matching .endloop specified
No matching .loop specified
Unmatched .endloop directive
Unmatched .if directive

Description These are errors about unmatched conditional assembly directives. A directive was encountered that requires a matching directive but the assembler could not find the matching directive.

Action Correct the source per the error message text.

E0007: Conditional nesting is too deep
Loop count out of range

Description These are errors about conditional assembly loops. Conditional block nesting cannot exceed 32 levels.

Action Correct the .if/.elseif/.else/.endif or .loop/.break/.endloop source.

E0009: Cannot apply bitwise NOT to floats
Unary operator must be applied to a constant

Description These are errors about an illegally used operator. The operator specified was not legal for the given operands.

Action Correct the source per the error message text so that all required operands are declared.

E0100: Label missing
.setsym requires a label

Description These are errors about required labels. The given directive requires a label, but none is specified.

Action Correct the source by specifying the required label.

E0101: Labels are not allowed with this directive

Description The error is about an invalid label. The given directive does not permit a label, but one is specified.

Description Remove the invalid label.

E0200: Binary operator can't be applied
Division by zero is illegal
Expression must be absolute integer value
Offset expression must be integer value
Operation cannot be performed on given operands
Unary operator can't be applied
Well-defined expression required

Description These are errors about general expressions. An illegal operand combination was used, or an arithmetic type is required but not present.

Action Correct the source per the error message text.

E0201: Absolute operands required for FP operations!
Cannot apply bitwise NOT to floats
Floating-point divide by zero
Floating-point overflow
Floating-point underflow
Floating-point expression required
Ilegal floating-point expression
Invalid floating-point operation

Description These are errors about floating-point expressions. A floating-point expression was used where an integer expression is required, an integer expression was used where a floating-point expression is required, or a floating-point value is invalid.

Action Correct the source per the error message text.

E0300: Cannot redefine this section name
Symbol can't be defined in terms of itself
Symbol expected in label field
Symbol, *sym*, has already been defined
Symbol, *sym*, is not defined in this source file

Description These are errors about general symbols. An attempt was made to redefine a symbol or to define a symbol illegally.

Action Correct the source per the error message text.

E0301: Cannot redefine local substitution symbol
Substitution Stack Overflow
Substitution symbol not found

Description These are errors about general substitution symbols. An attempt was made to redefine a symbol or to define a symbol illegally.

Action Correct the source per the error message text. Make sure that the operand of a substitution symbol is defined with a .asg or .eval directive.

E0400: Symbol table entry is not balanced

Description A symbolic debugging directive does not have a complementing directive (i.e., a .block without a .endblock).

Action Check the source for mismatched conditional assembly directives.

E0802: Expecting parallel instruction
Incorrect instruction used in parallel
Illegal form of LD used in parallel

Description These are errors about illegal used parallel instructions.

Action Correct the source per the error message text.

E0900: Can't include a file inside a loop
Invalid load-time label

Description These are errors about illegally used directives. Specific directives are not permitted where they were encountered because they will cause a corruption of the object file.

Action Correct the source per the error message text.

E1000: Include/Copy file not found or opened

Description The specified filename cannot be found.

Action Check spelling, pathname, environment variables, etc.

E1300: Copy limit has been reached

Description These errors are about general assembler limits that have been exceeded. The nesting of .copy/.include files in limited to 10 levels.

Action Check the source to determine how limits have been exceeded.

W0000: No operands expected. Operands ignored
Trailing operands ignored
***+ARn addressing is for write-only**

Description These are warnings about operands. The assembler encountered operands that it did not expect.

Action Check the source to determine what caused the problem and whether you need to correct the source.

W0001: Field value truncated to *value*
Field width truncated to *size in bits*
Line too long, will be truncated
Power of 2 required, *next larger power of 2* assumed
Section Name is limited to 8 characters
String is too long – will be truncated
Value truncated
Value truncated to byte size
Value out of range

Description These are warnings about truncated values. The expression given was too large to fit within the instruction opcode or the required number of bits.

Action Check the source to make sure the result will be acceptable, or change the source if an error has occurred.

W0002: Address expression will wrap around
Expression will overflow, value truncated

Description These are warnings about arithmetic expressions. The assembler has done a calculation that produces the indicated result, which may or may not be acceptable.

Action Verify the result is acceptable, or change the source if an error has occurred.

Appendix E

Glossary

A

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

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

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

assembler: A software program that creates a machine-language program from a source file that contains assembly language instructions, and 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.

B

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

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

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

boot: The process of loading a program into program memory.

boot loader: A built-in segment of code that transfers code from an external source to program memory at power-up.

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

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

C

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

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

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

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

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

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

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

constant: A fixed or invariable value or data item.

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

D

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

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

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

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

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

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

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

E

EGA: *enhanced graphics array.* An industry-standard video card.

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

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

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

F

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

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

G

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

H

host port interface (HPI): An 8-bit parallel interface that the CPU uses to communicate with a host processor.

HPIC: *host port interface control register*. 16-bit register that controls the operation of the host port interface (HPI).

HPIA: *host port interface address register*. 16-bit pointer to HPI memory.

I

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

IMR: *interrupt mask register*. A 16-bit memory-mapped register used to enable or disable external and internal interrupts. A 1 written to any IMR bit position enables the corresponding interrupt (when INTM=0).

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

interrupt: A condition caused either by an event external to the CPU or by a previously executed instruction that forces the current program to be suspended and causes the processor to execute an interrupt service routine corresponding to the interrupt.

L

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

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

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

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

M

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

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

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

MP/$\overline{\text{MC}}$ bit: A bit in the processor mode status register PMST that indicates whether the processor is operating in microprocessor or microcomputer mode. See also *microcomputer mode; microprocessor mode.*

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

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

N

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

O

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

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

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

P

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

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

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

R

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

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

S

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

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

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

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

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

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

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

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

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

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

T

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

U

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

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

V

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

W

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

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

IMPORTANT NOTICE

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

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

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

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

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

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

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