TEXAS INSTRUMENTS

TMS320C3x DSP Starter Kit

User's Guide

1996

Digital Signal Processing Products

TEXAS INSTRUMENTS

Printed in U.S.A., March 1996
D413004-9761 revision *

SPRU163

User's Guide

TMS320C3x DSP Starter Kit

1996

TMS320C3x DSP Starter Kit User's Guide

Texas Instruments

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 DSP (digital signal processing) Starter Kit (DSK) and how to use the DSK with these tools:

- The DSK assembler
- The DSK debugger

How to Use This Manual

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

- **Part I: Hands-On Information** is presented first so that you can start using your DSK the same day you receive it.
  - Chapter 1 describes the features and provides an overview of the TMS320C3x DSP Starter Kit.
  - Chapter 2 contains installation instructions for your assembler and debugger. It lists the hardware and software tools you'll need to use the DSK and tells you how to set up its environment.
  - Chapter 3 lists the key features of the assembler and debugger and tells you the steps you need to take to assemble and debug your program.
- **Part II: Functional Description** contains a functional overview of the DSK, which includes the TMS320C3x DSK functional diagram, a description of the DSK hardware components and software operation.
- **Part III: Assembler Description** contains detailed information about using the assembler.
  - Chapter 5 explains how to create DSK assembler source files and invoke the assembler.
  - Chapter 6 discusses the valid directives and gives you an alphabetical reference to these directives.

- **Part IV: Debugger Description** contains detailed information about using the debugger. Chapter 7 explains how to invoke the DSK debugger, and use its function keys, and debugger commands.

- **Part V: Appendices** contains a description of the communications kernel source code, the DSK circuit board dimensions and schematic diagrams, the data sheet of the TLC32040 that provides all specifications of the analog interface circuit, and a glossary.

Notational Conventions

This document uses the following conventions.

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

  Here is a sample program listing:

```
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:\dsk3d testa
```

- 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. Syntax that is entered on a command line is centered in a bounded box. Syntax that is used in a text file is left-justified in an unbounded box. Here is an example of command-line syntax:

dsk3a *filename*

  dsk3a is a command. The command invokes the assembler and has one parameter, *filename*, which is required. When you invoke the assembler, you supply the name of the file that the assembler uses as input.

- In assembler syntax statements, column 1 is reserved for the first character of a label or symbol. If the label or symbol is *optional*, it is usually not shown. If it is a *required* parameter, it is shown starting against the left margin of the shaded box, as in the example below. No instruction, command, directive, or parameter, other than a symbol or label, should begin in column 1.

  symbol **.set** *value*

  The *symbol* is required for the .set directive and must begin in column 1. The *value* is also required.

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

  .entry [*value*]

  The .entry directive has one parameter, which is optional.

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

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

  Note that **.int** does not begin in column 1.

  This syntax shows that .int must have at least one value parameter, but you have the option of supplying additional value parameters, each separated from the previous one by a comma.

Information About Cautions and Warnings

This book may contain warnings.

This is an example of a warning statement.

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

WARNING

Related Documentation From Texas Instruments

The following books describe the TMS320C3x 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.

TMS320C3x User's Guide (literature number SPRU031) describes the 'C3x 32-bit floating-point microprocessor (developed for digital signal processing as well as general applications), its architecture, internal register structure, instruction set, pipeline, specifications, and DMA and serial port operation. Software and hardware applications are included.

TMS320C32 Addendum to the TMS320C3x User's Guide (literature number SPRU132) describes the TMS320C32 floating-point microprocessor (developed for digital signal processing as well as general applications). Discusses its architecture, internal register structure, specifications, and DMA and serial port operation. Hardware applications are also included.

TMS320 Floating-Point DSP Assembly Language Tools User's Guide (literature number SPRU035) 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 'C3x and 'C4x generations of devices.

TMS320 Floating-Point DSP Optimizing C Compiler User's Guide (literature number SPRU034) describes the TMS320 floating-point C compiler. This C compiler accepts ANSI standard C source code and produces TMS320 assembly language source code for the 'C3x and 'C4x generations of devices.

TMS320C3x C Source Debugger User's Guide (literature number SPRU053) tells you how to invoke the 'C3x emulator, evaluation module, and simulator versions of the C source debugger interface. This book discusses various aspects of the debugger interface, including window management, command entry, code execution, data management, and breakpoints. It also includes a tutorial that introduces basic debugger functionality.

TMS320C30 Evaluation Module Technical Reference (literature number SPRU069) describes board-level operation of the TMS320C30 EVM.

TMS320 DSP Designer's Notebook Volume 1 (literature number SPRT125) collection of designer's notebook pages.

If You Need Assistance. . .

If you want to. . .	Do this. . .
Obtain software updates and more example applications	Call the TMS320 BBS: (713) 274-2323 or Anonymous FTP on the Internet ftp.ti.com In directory: /mirrors/tms320bbs
Request more information about Texas Instruments Digital Signal Processing (DSP) products	Write to: Texas Instruments Incorporated DSP Market Communications Manager, MS 737 P.O. Box 1443 Houston, Texas 77251–1443
Order Texas Instruments documentation	Call the TI Literature Response Center: **(800) 477–8924**
Ask questions about product operation or report suspected problems	Contact the DSP hotline: **(713) 274–2320** **FAX: (713) 274–2324** **Email: dsph@msg.ti.com**
Report mistakes in this document or any other TI documentation	Send your comments to: Texas Instruments Incorporated Technical Publications Manager, MS 702 P.O. Box 1443 Houston, Texas 77251–1443

Trademarks

AT is a trademark of International Business Machines Corp.

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

MS-DOS is a registered trademark of Microsoft Corporation.

Contents

Part I: Hands-On Information

1 Introduction 1-1
Describes the key features and provides an overview of the TMS320C3x DSP Starter Kit
1.1 Key Features of the DSK 1-2
1.2 DSK Overview 1-3

2 Installing the DSK Assembler and Debugger 2-1
Lists the hardware and software you'll need to install the DSK assembler and debugger; provides installation instructions for PC systems running DOS
2.1 What You'll Need 2-2
Hardware checklist 2-2
Software checklist 2-3
DSK module connections 2-3
2.2 Step 1: Connecting the DSK to Your PC 2-4
2.3 Step 2: Installing the DSK Software 2-5
2.4 Step 3: Modifying Your CONFIG.SYS File 2-5
2.5 Step 4: Modifying the PATH Statement 2-6
2.6 Step 5: Verifying the Installation 2-7
Installation errors 2-8

3 Overview of a Code Development and Debugging System 3-1
Provides an overview of the assembler and debugger, and describes the overall code development process
3.1 Description of the DSK Assembler 3-2
Key features of the assembler 3-2
3.2 Description of the DSK Debugger 3-2
Key features of the debugger 3-3
3.3 Developing Code for the DSK 3-4
3.4 Getting Started 3-5

Part II: Functional Description

4 Functional Overview 4-1
Describes the DSK hardware and software functionality
4.1 DSK Hardware Interface 4-2
Host hardware interface 4-2
Host communications 4-4
TLC32040 AIC hardware interface 4-6
DSK memory map 4-7
4.2 DSK Communications Kernel 4-8
Data packets 4-8
Commands 4-9
Debugging functions 4-10
4.3 TLC32040 AIC Initialization 4-14
Resetting the AIC 4-14
Initializing the 'C31 timer 4-14
Initializing the 'C31 serial port 4-15
Initializing the AIC 4-16
Primary communications 4-17
Secondary communications 4-18
4.4 Host Software 4-23
Host communications target routines 4-24
Host communications driver routines 4-27
Host communications object routines 4-29

Part III: Assembler Description

5 Using the DSK Assembler 5-1
Tells you how to invoke and use the DSK assembler; describes valid source file formats
5.1 Creating DSK Assembler Source Files 5-2
Using valid labels 5-3
Using the mnemonic field 5-4
Using the operand field 5-5
Commenting your source file 5-7
5.2 Constants 5-8
Binary integers 5-8
Decimal integers 5-8
Hexadecimal integers 5-8
Floating-point constants 5-9
Character constants 5-9
5.3 Character Strings 5-10

5.4 Symbols . . . 5-11
Labels . . . 5-11
Constants . . . 5-11
Predefined symbolic constants . . . 5-11
5.5 Expression Analyzer . . . 5-12
5.6 Assembling Your Program . . . 5-15
5.7 Placing Code Sections in Memory Locations . . . 5-16

6 Assembler Directives . . . 6-1
Tells you how to use assembler directives and describes the available DSK directives
6.1 Using the DSK Assembler Directives . . . 6-2
6.2 Directives That Define Sections . . . 6-5
6.3 Directives That Initialize Constants . . . 6-8
6.4 Directives That Reference Other Files . . . 6-9
6.5 Directives That Enable Conditional Assembly . . . 6-10
6.6 Directives That Align the Section Program Counter . . . 6-11
6.7 Directives That Define Symbols at Assembly Time . . . 6-11
6.8 Miscellaneous Directives . . . 6-12
6.9 Directives Reference . . . 6-13

Part IV: Debugger Description

7 Using the DSK Debugger . . . 7-1
Tells you how to invoke and use the debugger and describes the debugger environment. Discusses valid debugger commands
7.1 Invoking the Debugger . . . 7-2
Displaying a list of available options (? or Help option) . . . 7-2
Selecting the parallel printer port (LPT = 3 or LPT# option) . . . 7-3
Select the parallel printer port at a particular address (PORT option) . . . 7-3
Selecting communication mode (BW option) . . . 7-3
Automatically search for a printer port (TEST option) . . . 7-3
Add extra I/O cycles to each transfer . . . 7-3
7.2 Understanding the Debugger Windows . . . 7-4
DISASSEMBLY window . . . 7-4
CPU REGISTER window . . . 7-5
MEMORY window . . . 7-6
COMMAND window . . . 7-7
7.3 Using the Help Menu . . . 7-8
7.4 Using Software Breakpoints . . . 7-9
Setting a software breakpoint . . . 7-9
Clearing a software breakpoint . . . 7-9
Finding the software breakpoints that are set . . . 7-9
7.5 Debugger Commands . . . 7-10
7.6 Quick Reference Guide . . . 7-13

Part V: Appendices

A Communications Kernel Source Code **A-1**
Contains the source code for the TMS320C3x DSK communications kernel

B DSK Circuit Board Dimensions and Schematic Diagrams **B-1**
Contains the circuit board dimensions and the schematic diagrams for the DSP Starter Kit
Hardware Component Overview B-3

C TLC32040 Data Sheet **C-1**
Contains the complete data sheet for the TLC32040 Analog Interface Circuit

D Glossary **D-1**
Defines acronyms and key terms used in this book

Figures

1–1 TMS320C3x DSK Block Diagram . . . 1-3
2–1 Connecting Your Parallel Printer Port Cable and Transformer Into Your DSK Board . . . 2-4
2–2 DOS Command Setup for the DSK Environment (Sample autoexec.bat file) . . . 2-6
2–3 Basic Debugger Display . . . 2-7
3–1 The Basic Debugger Display . . . 3-3
3–2 DSK Software Development Flow . . . 3-4
4–1 TMS320C3x DSK Functional Circuit Diagram . . . 4-3
4–2 Parallel Port Control Register (0x37A) . . . 4-4
4–3 Parallel Port Status Register (0x379) . . . 4-4
4–4 DSK Memory Map . . . 4-7
4–5 Data-Packet Structure . . . 4-8
4–6 Single-Step Flow Diagram . . . 4-12
4–7 Primary Communication Data Format . . . 4-17
4–8 Secondary Communication Data Format . . . 4-18
4–9 Control Register Bit Fields . . . 4-19
7–1 DISASSEMBLY Window . . . 7-4
7–2 CPU REGISTER Window . . . 7-5
7–3 MEMORY Window . . . 7-6
7–4 COMMAND Window . . . 7-7
7–5 The Monitor Information Screen . . . 7-8
B–1 TMS320C3x DSP Starter Kit (DSK) Circuit Board Dimensions . . . B-2

Tables

4–1	Single-Step Pipeline Flow	4-13
4–2	Primary Communications Mode Selection	4-18
5–1	Indirect Addressing	5-6
5–2	ANSI C Math Library Functions Supported by the DSK Assembler	5-12
5–3	Operators Used in Expressions	5-14
5–4	Summary of Assembler Options	5-15
6–1	Assembler Directives Summary	6-2
7–1	Summary of Debugger Options	7-2
7–2	Editing Command Keys	7-7
7–3	Command-Line Editing	7-10
7–4	Command-Line Buffer Manipulation	7-10
7–5	Running Programs	7-10
7–6	Displaying and Changing Data	7-11
7–7	Managing Breakpoints	7-11
7–8	Loading Programs	7-11
7–9	Performing System Tasks	7-12
7–10	Function Key Shortcuts for Command Window Active	7-13
7–11	Function Key Shortcuts for CPU Window Active	7-13
7–12	Function Key Shortcuts for Memory Window Active	7-14
7–13	Function Key Shortcuts for Disassembler Window Active	7-14

Examples

2–1 Port Selection Display 2-9
3–1 File rand.asm 3-5
4–1 Initialize the Serial Port Global Control Register 4-16
4–2 Setting the TA and TB Registers 4-20
6–1 Sections Directives 6-6

Chapter 1

Introduction

This chapter provides an overview of the TMS320C3x DSP Starter Kit (DSK). The ’C3x DSK is a low-cost, simple, high-performance stand-alone application development board that lets you experiment with and use TMS320C3x DSPs for real-time signal processing. The DSK has a TMS320C31 on board to allow full-speed verification of the TMS320C3x code. The DSK also gives you the freedom to build new boards, create your own software on a host PC, download the software to the DSK, and run the software on the DSK board. The supplied debugger is windows-oriented, simplifying code development and debugging capabilities.

Topic **Page**

1.1	**Key Features of the DSK**	**1-2**
1.2	**DSK Overview**	**1-3**

1.1 Key Features of the DSK

This section details the key features of the TMS320C3x DSP Starter Kit.

- Industry-standard TMS320C31 floating-point DSP
- 40 ns instruction cycle time, 50 MFLOPS, 25 MIPS
- Standard or enhanced parallel printer port interface which connects to a host PC™ and allows the TMS320C31 to communicate with PC programs
- Voice quality analog data acquisition via the TLC32040 analog interface circuit (AIC):
  - 14-bit dynamic range ADC and DAC
  - Variable ADC and DAC sampling rate up to 20 000 samples per second
  - Output reconstruction filter and bypassable, switched-capacitor anti-alias input filter
- Standard RCA plug connectors for analog input and output that provide a direct connection to microphone and speaker
- XDS510 emulator connector
- Expansion connectors, which route all the TMS320C31 pins for use with daughterboards

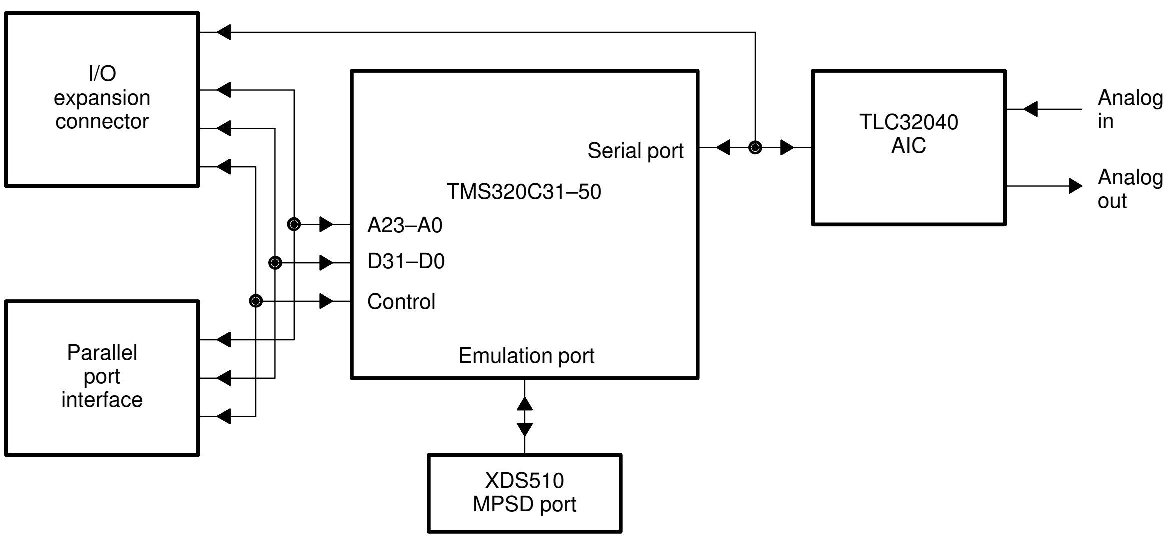
1.2 DSK Overview

Figure 1–1 depicts the block diagram of the TMS320C3x DSK hardware. The basic components are the TMS320C31 DSP, the TLC32040 AIC, expansion connectors, system clock, parallel printer port interface, and tri-color LED. The parallel printer port connects the DSK to a host PC and allows the TMS320C31 to communicate with PC programs.

All of the signals for the ’C3x are routed to expansion connectors. The expansion connectors include four 32-pin headers, an 11-pin jumper block, and a 10-pin XDS510 header.

The TLC32040 AIC interfaces to the TMS320C3x serial port. A jumper block allows removal of this connection to route the serial port to a daughtercard that you supply. Two RCA connectors provide analog input and output on the board.

Figure 1–1. TMS320C3x DSK Block Diagram



See Appendix B, *DSK Circuit Board Dimensions and Schematic Diagrams*, for an explanation of the basic DSK components.

Chapter 2

Installing the DSK Assembler and Debugger

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

Topic **Page**

2.1	What You'll Need	2-2
2.2	Step 1: Connecting the DSK to Your PC	2-4
2.3	Step 2: Installing the DSK Software	2-5
2.4	Step 3: Modifying Your CONFIG.SYS File	2-5
2.5	Step 4: Modifying the PATH Statement	2-6
2.6	Step 5: Verifying the Installation	2-7

2.1 What You'll Need

The following checklists detail items that are shipped with the DSK assembler and debugger and any additional items you'll need to use this tool. The DSK module connections with a parallel printer port are also discussed in this section.

Hardware checklist

☐	**host**	An IBM™ PC/AT™ or 100% compatible PC with a hard disk system and a 1.2 megabyte floppy-disk drive and parallel printer port communication link
☐	**memory**	Minimum of 640K bytes
☐	**display**	Monochrome or color (color recommended)
☐	**power requirements**	A UL Class II power supply with a 2.1-mm power jack connector that provides 7–12 Vdc or 6–9 Vac and at least 400–1500 mA, which is common to most wall-mounted DC transformers. For DC power supplies, the polarity of the 2.1 mm power jack does not matter. However, laboratory type power supplies are not recommended since they can create ground loops and possibly create a short circuit through the DSK full-wave rectifier.

Note:

- ❑ You may want to use the DSK's on-board power supply and regulators for external circuits. If so, you must not overload the circuit. External loads will cause the regulators to operate at a higher temperature. Loads >50 mA are not recommended.
- ❑ If you are using an external power supply, be sure you connect it correctly; the DSK is not warranted after you make modifications to it.

WARNING

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

☐	**board**	DSK circuit board
☐	**cable**	Pass-through parallel printer port cable

☐	**optional hardware**	An EGA- or VGA-compatible graphics display card and monitor.
☐	**miscellaneous materials**	Blank, formatted disks

Software checklist

☐	**operating system**	MS-DOS™ or PC-DOS™ (version 5.0 or later), Windows or OS/2
☐	**files**	*dsk3a.exe* is an executable file for the DSK assembler. *dsk3d.exe* is an executable file needed for running the DSK debugger interface.
☐	**miscellaneous files**	Other files are included in your DSK package, such as sample source files and additional documentation. You can find a brief description of these files in the Readme file included on your disk. Be sure to check the Readme file for the latest information on software changes and DSK operation.

Note:

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

DSK module connections

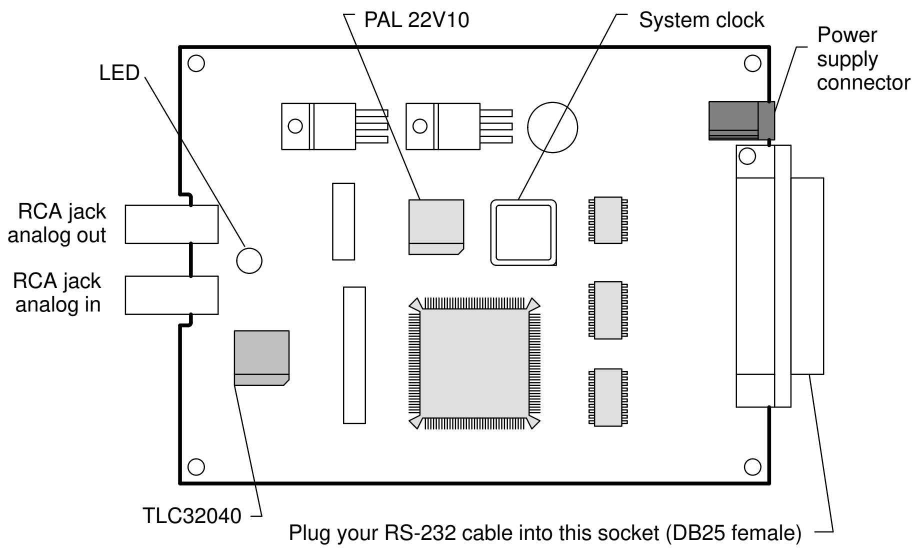
You need a parallel printer port cable to connect your PC to your DSK board. The DSK board is designed with a DB25 parallel printer port connection mounted on the board.

2.2 Step 1: Connecting the DSK to Your PC

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

1) Turn off your PC's power.
2) Connect your parallel printer port cable to the parallel communication port (LPT) on your PC. This port can be identified by its size and pin type, which should be the female matching equivalent to the DSK. (RS232 ports which use DB25 connectors use the opposite pin configuration).
3) Plug the parallel printer port cable into the DSK DB25 connector.
4) Plug 7–12 Vdc or 6–9 Vac power supply into the DSK power supply connector. See Figure 2–1 for details.

Figure 2–1. Connecting Your Parallel Printer Port Cable and Transformer Into Your DSK Board



5) Plug the transformer into a wall socket.
6) Turn on your PC's power.
7) The LED will illuminate either red or green.

Note:

Some manufacturers of plug-in cards may also use DB25 connectors that appear to be of the same type. If this is the case, be sure to check the PC configuration thoroughly before continuing.

2.3 Step 2: Installing the DSK Software

This section explains how to install the debugger software on a hard disk system.

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

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

   ```
   md c:\dsktools
   ```

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

   ```
   copy a:\*.*c:\dsktools\*.*/v
   ```

2.4 Step 3: Modifying Your CONFIG.SYS File

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

```
FILES=20
```

Once you edit your config.sys file and add the line, invoke the file by rebooting the PC (press the reset switch, or turn off the PC's power and turn it on again).

2.5 Step 4: Modifying the PATH Statement

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

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

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

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

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

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

PATH statement →

```
DATE
TIME
ECHO OFF
PATH=c:\dos;c:\dsktools
CLS
```

If you modify the AUTOEXEC.BAT file, be sure to invoke it before invoking the debugger for the first time. To invoke this file, enter:

`autoexec`

2.6 Step 5: Verifying the Installation

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

dsk3d

After entering the dsk3d command, you should see a display similar to the one shown in Figure 2–3.

Figure 2–3. Basic Debugger Display

```
DISASSEMBLY                                        C31 DSP STARTERS KIT
809c03 50700080 start LDIU 00080h,DP               PC  00809c03 SP  008098de
809c04 08349c2c       LDI  @09c2cH,SP              R0  00000000 R1  00000000
809c05 07608000       LDF  0.000000e+00,R0         R2  00000000 R3  00000000
809c06 c610c1c0       LDI  *AR0,R0 || LDI *AR      R4  00000000 R5  00000000
809c07 c610c1c0       LDI  *AR0,R0 || LDI *AR      R6  00000000 R7  00000000
809c08 08600100       LDI  256,R0                  AR0 00000000 AR1 00000000
809c09 09a09c00       LSH  @09c00H,R0              AR2 00000000 AR3 00000000
809c0a 61809c0e       BRD  jump                    AR4 00000000 AR5 00000000
809c0b 07618000       LDF  0.000000e+00,R1         AR6 00000000 AR7 00000000
809c0c 07628000       LDF  0.000000e+00,R2         IR0 00000000 IR1 00000000
809c0d 07630000       LDF  1.000000e+00,R3         ST  00000000 RC  00000000
809c0e 07640000 jump  LDF  1.000000e+00,R4         RS  00000000 RE  00000000
809c0f 087b0003 loop  LDI  3,RC                    DP  00000000 BK  00000000
809c10 64809c1a       RPTB block                   IE  00000000 IF  00000000
809c11 02640001       ADDI 1,R4

COMMAND                       MEMORY
Texas Instruments 1994        809800  00000007 fffffffc 00809802 00809827
                              809804  0080982c 00809839 0080983c 0080983f
                              809808  00809843 00809842 00809868 0080989a
                              80980c  008098a9 10800000 0f350000 0f300000
                              809810  0f200000 0f320000 0f280000 0f290000
load testa                    809814  1a770004 6a050006 628098a9 50700080
```

Note:

When the communications kernel is first loaded, the on-chip timers are initialized causing the LED to cycle through several colors. The sequence is red–yellow–green–yellow–red etc.

If you see a display similar the one shown in Figure 2–3, you have correctly installed your DSK board, assembler, and debugger. If you see the display shown in Example 2–1, then your software or cable may not be installed properly. Go through the installation instructions again and make sure that you have followed each step correctly; then re-enter the command above.

Installation errors

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

- You may have used an incorrect communication port (LPT1 versus LPT2).
- A printer driver or other software may be using the same communication port that you are attempting to use with the DSK. If so, try another communication port for the DSK.
- Your printer port cable and connectors may not be connected snugly.
- Your power transformer may not be plugged in on both ends. If the DSK is receiving power, then the LED will illuminate either red or green.

Some operating systems do not use conventional AT I/O port addresses when mapping port names to addresses. For example, an EISA PC or IBM PS/2 might consider port 0x3BC to be LPT1 instead of LPT3. If this is the case, you should use LPT3 to start the DSK since the DSK works from a physical address instead of the port name LPTx. The last three lines of Example 2–1 show the operating system's lookup table (located at RAM address 0000 0040) that maps physical addresses to port names. This may help you to determine which ports are in use and which name is associated with each port for a particular address. The information in the lookup table in Example 2–1 may not accurate since network and OS software use this table for redirecting printer output.

Example 2–1. Port Selection Display

```
   TESTING TMS320C3x DSK RESET AT PORT 0x378 (LPT1)
   >>>> HPACK (ERROR pin) did not go high during reset
   SELECT: 1) LPT1 0x378   (alternate LPT2)
           2) LPT2 0x278   (alternate LPT3)
           3) LPT3 0x3BC   (alternate LPT1)
           H) Additional online help

           CHECK: TARGET POWER (LED IS RED OR GREEN)
                  PORT SELECTION
                  I/O CONNECTIONS AND CABLES
                  POWER CONSERVATION SOFTWARE (LAPTOPS!)
                  AUTOEXEC.BAT, CONFIG.SYS AND BIOS
                  DAUGHTER CARDS
                  VERY OLD PRINTER PORTS WITHOUT PULLUPS (PRE 1986)
                  IF THE LED IS CYCLING R-Y-G THE KERNEL HAS LOADED
-----------------------------------------------------------------------
  The LPTx name or handle for a port address depends on the operating
system and installed drivers.  The DSK uses standard port conventions so
you might need to use a different port name to get the correct port address.
For reference, the systems LPT cross reference table is given below

       SYSTEM TABLE LOCATED AT   LPT1 @0x378
       RAM ADDRESS 0000:0400     LPT2 @0x278
                                 LPT3 @0x002
```

Chapter 3

Overview of a Code Development and Debugging System

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

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

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

Topic **Page**

Topic		Page
3.1	**Description of the DSK Assembler**	**3-2**
3.2	**Description of the DSK Debugger**	**3-2**
3.3	**Developing Code for the DSK**	**3-4**
3.4	**Getting Started**	**3-5**

3.1 Description of the DSK Assembler

The DSK assembler is a simple and easy to use tool. Only the most significant features of an assembler have been incorporated. However, if you want, you can create and load COFF files by using the TMS320 floating-point DSP assembly language tools that will load and run on the DSK.

Key features of the assembler

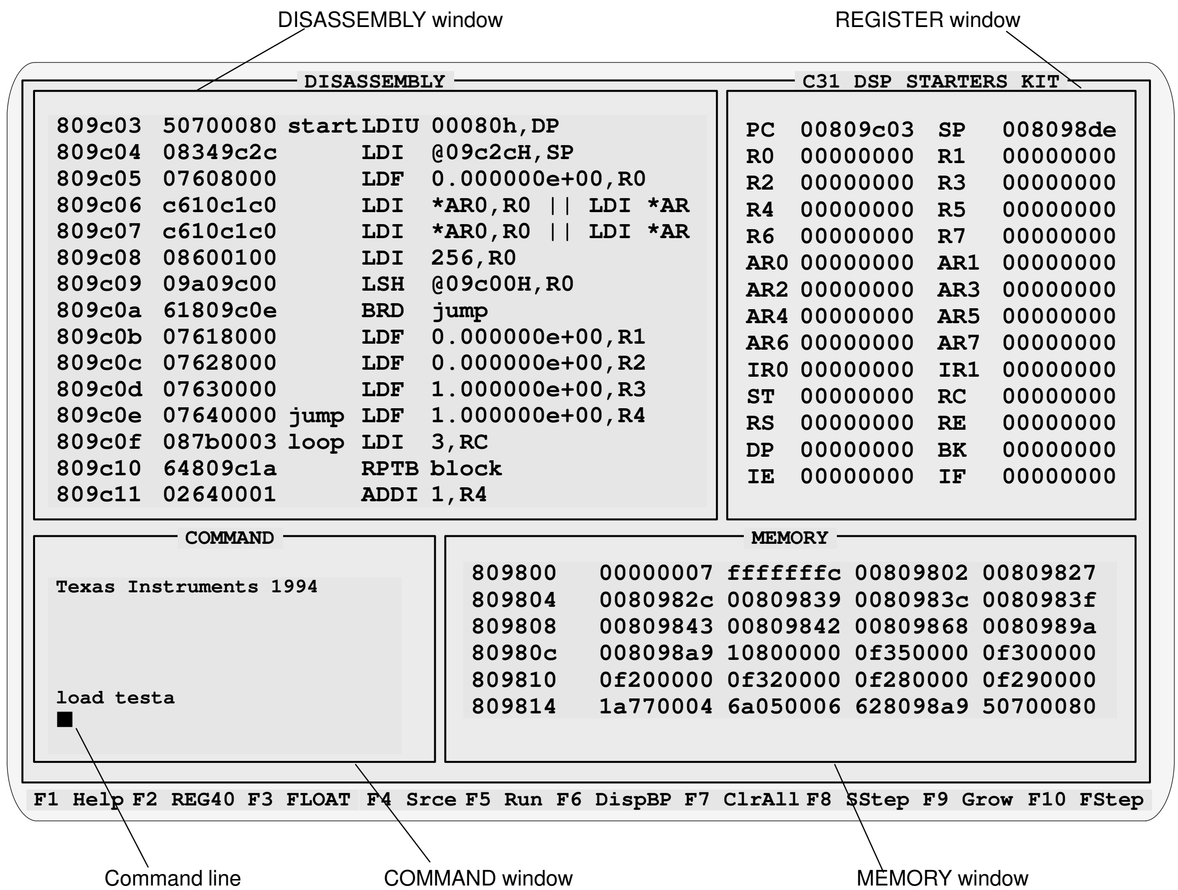
- **Quick.** The DSK assembler differs from many other assemblers in that it does not go through a linker phase to create an output file. Instead, the DSK uses special directives to assemble code at an absolute address during the assembly phase. As a result, you can create small programs quickly and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

3.2 Description of the DSK Debugger

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

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

Figure 3–1. The Basic Debugger Display



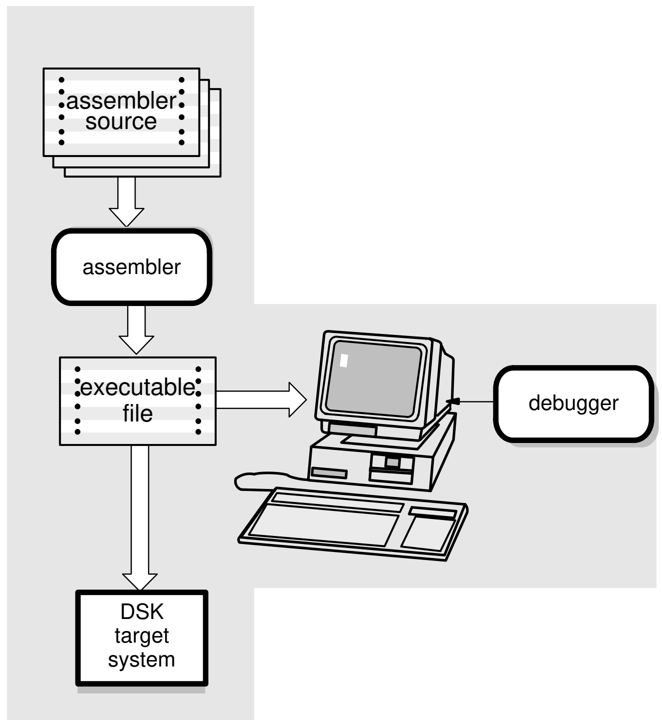
Key features of the debugger

- **Easy-to-use, window-oriented interface.** The DSK debugger separates code, data, and commands into manageable portions.
- **Powerful command set.** Unlike many other debugging systems, this debugger doesn't force you to learn a large, intricate command set. The DSK debugger supports a small but powerful command set.
- **Flexible command entry.** There are two main ways to enter commands. You can enter commands at the command line or use the function keys; choose the method that you like better.

3.3 Developing Code for the DSK

Figure 3–2 illustrates the DSK code development flow.

Figure 3–2. DSK Software Development Flow



The following list describes the tools shown in Figure 3–2.

assembler

The **assembler** translates DSK assembly language source files into machine language object files for the TMS320C3x family of processors. Only the most essential features of an assembler have been incorporated. This is *not* a COFF assembler, although executable object files created by the TI TMS320 floating-point DSP assembly language tools will also load and run on the DSK.

debugger

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

3.4 Getting Started

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

1) Create a source file to use for the walkthrough; call it rand.asm. You do not need to enter the information following a semicolons; such information is comments to help you understand what the program is doing.

Example 3–1. File rand.asm

```
;---------------------------------------------------------------------;
; RAND.ASM                                                            ;
; This example shows nested loops with a call to a random number      ;
; within the inner loop.                                              ;
;                                                                     ;
; NOTE:  This file can be loaded either by using the debugger or a    ;
; bootloader.   This example does not use 0x809800 and 0x809801 since ;
; the bootloader uses these locations for stack space.                ;
;---------------------------------------------------------------------;
        .start ”CODE”,0x809802 ; Start assembling CODE section here
        .sect  ”CODE”          ;
        .entry  SAMPLE         ; Debugger entry point
        ;----------------------
SAMPLE  ldp     @stack         ; Load a data page
        ldi     @stack,SP      ; Load a stack pointer
        ;----------------------
        ldi     0,R0           ; Start with SEED = 0
        ldi     0,R1           ; Inner loop counter
        ldi     0,R2           ; Outer loop counter
        ;----------------------
OUTER   ldi     3,RC           ; Start ’OUTER’ loop
        rptb    INNER          ; Repeat block ’INNER’ (RC+1) times
        call    RAND           ; Call function
        addi    1,R1           ; Count ’INNER’ loops
INNER   addi    1,R2           ; Count ’OUTER’ loops
        b       OUTER          ; Do it again!
;----------------------------------------
; Fast 32 bit random number generator
;----------------------------------------
RANDX:  ldi     @SEED,R0       ; Calculate RAND(SEED)
RAND:   mpyi    @A,R0          ; Calculate RAND(R0)
        addi    @C,R0          ;
        sti     R0,@SEED       ; Result is returned in R0
        rets                   ;
        ;----------------------
A       .word   0107465h       ; Constants needed for RAND
C       .word   0234567h       ;
SEED    .word   0              ;
;------------------------------
stack   .word   $+1            ; Begin stack here
        .end
```

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

   `dsk3a rand` ↵

   This command invokes the TMS320C3x DSK assembler. If the input file extension is .asm (for example, rand.asm), you don't have to specify the extension; the assembler uses .asm as the default. For more information about invoking the assembler, refer to Section 5.6.

   When you enter this command, the assembler creates an executable file called rand.dsk. This file is used for directly loading executable code into the DSK.

   The executable file includes a listing of all errors and warnings that may have occurred during assembly of your program. This listing is helpful because it contains a list of all unresolved symbols and opcodes.

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

   `dsk3d` ↵

4) This command brings up the TMS320C3x DSK debugger on your screen. From here, you can load your rand.dsk sample program by using the LOAD command. For more information on using the debugger, refer to Chapter 7.

Chapter 4

Functional Overview

The TMS320C3x DSK hardware and software work together to create a low-cost development platform that lets you develop real-time signal processing applications. In addition to performing full-speed verification of your TMS320C3x code, the DSK allows you to build new daughterboards to expand your system.

This chapter details the functionality of the hardware and the software.

Topic **Page**

4.1 DSK Hardware Interface **4-2**
4.2 DSK Communications Kernel **4-8**
4.3 TLC32040 AIC Initialization **4-14**
4.4 Host Software **4-23**

4.1 DSK Hardware Interface

The ’C3x DSK starts up by responding to a host reset command and boot-loading a communications kernel or a program that you supply. The communications kernel provides the necessary I/O for interfacing the DSK board and the host system. Host communications occur through the parallel bus of the ’C31, while analog I/O is handled by the TLC32040 analog interface circuit (AIC) and sent to the ’C31’s serial port.

See Appendix A, *Communications Kernel Source Code*, for more information.

Host hardware interface

The host interface connects the ’C31 parallel bus to the host PC parallel printer port. It consists of three devices:

- ❑ A programmable array logic (TICPAL22V10Z)
- ❑ Two high-speed octal bus transceivers with tri-state outputs (74ACT245)

The programmable array logic (PAL) determines when the ’C31 is accessing the host interface by using the $\overline{\text{STROBE}}$ A23, A22, A21, and A20 signals to decode the address of the ’C31.

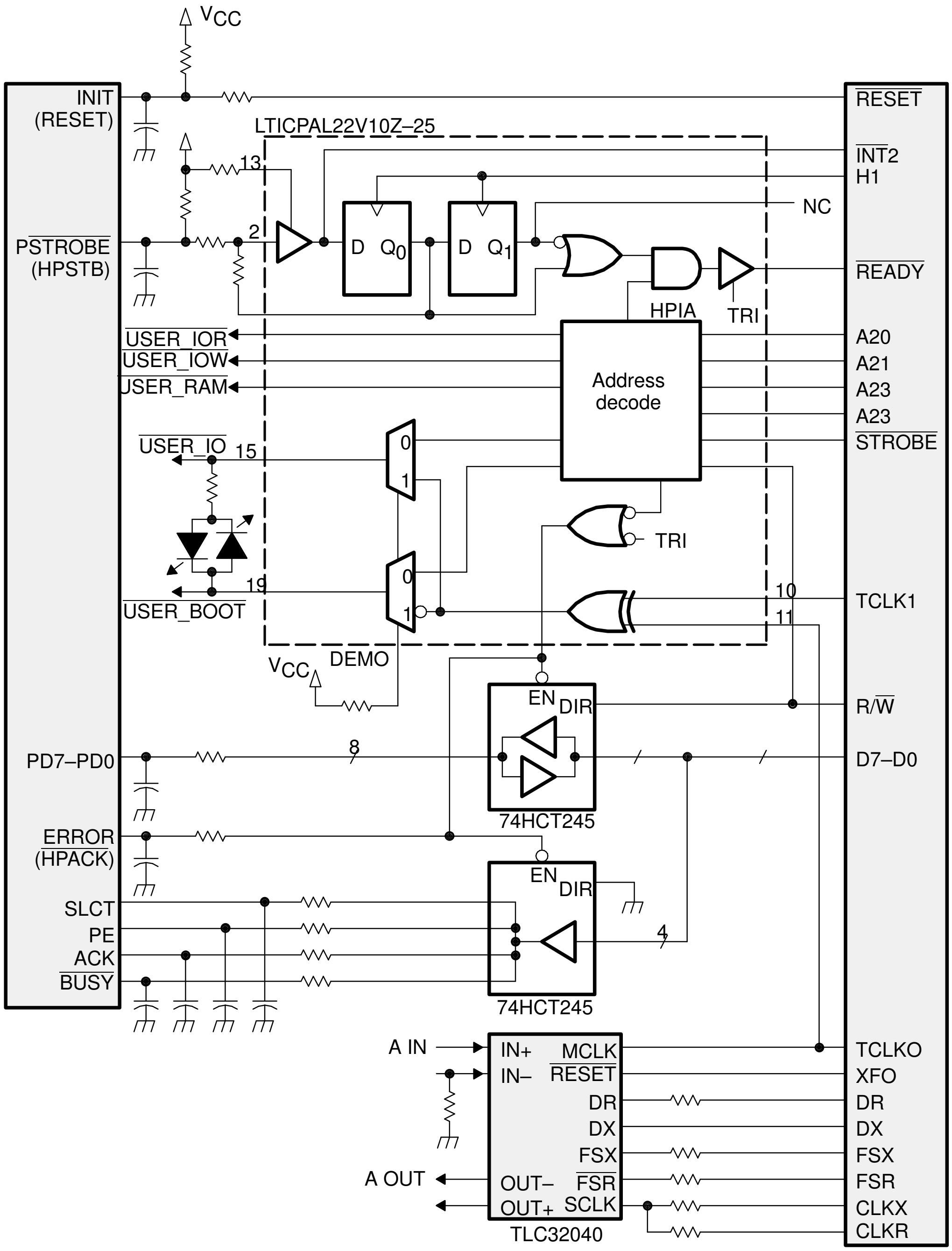
The PAL provides one input (TRI) that disconnects the host interface by tri-stating the PAL $\overline{\text{INT2}}$ and $\overline{\text{READY}}$ signals. The PAL provides five address decode outputs: $\overline{\text{USER\_IOR}}$, $\overline{\text{USER\_IOW}}$, $\overline{\text{USER\_IO}}$, $\overline{\text{USER\_RAM}}$, $\overline{\text{USER\_BOOT}}$; and three outputs: $\overline{\text{READY}}$, $\overline{\text{INT2}}$, and $\overline{\text{EN}}$ signals. When the DEMO signal is pulled high, two of the address decode outputs, $\overline{\text{USER\_IO}}$ and $\overline{\text{USER\_BOOT}}$, drive the tri-color LED.

The bus transceivers buffer data between the PC parallel printer port and the ’C31 parallel bus. The host interface supports two types of transfers:

- ❑ The 8-bit bidirectional mode allows faster transfers on parallel printer ports that support bidirectional transfers.
- ❑ Unidirectional printer ports support an 8-bit transfer from the host to the ’C31 while supporting 4-bit transfers from the ’C31 to the host.

Figure 4–1 shows a high-level circuit diagram of the ’C3x DSK.

Figure 4–1. TMS320C3x DSK Functional Circuit Diagram



Host communications

The host communicates with the ’C31 through the parallel printer port. The PC manipulates the parallel port’s signals by writing to and reading from the host’s parallel port control and status registers. Figure 4–2 and Figure 4–3 show the parallel port control and status register bit fields used by the DSK host software. (The labels within the following figures in parentheses refer to signals of the DSK board as shown in Figure 4–1.)

Figure 4–2. Parallel Port Control Register (0x37A)

7	6	5	4	3	2	1	0
DIR0	X	DIR1	$\overline{\text{INT}}$	$\overline{\text{SLCTIN}}$	$\overline{\text{INIT}}$	$\overline{\text{AUTOFD}}$	$\overline{\text{PSTROBE}}$
					RESET		$\overline{\text{HPSTB}}$

Figure 4–3. Parallel Port Status Register (0x379)

7	6	5	4	3	2	1	0
$\overline{\text{BUSY}}$	ACK	PE	SLCT	ERROR	ACK	X	X
D3	D2	D1	D0	HPACK			

The host initializes the ’C31 by pulsing the $\overline{\text{INIT}}$ signal (writes a 0 followed by a 1 to the $\overline{\text{INIT}}$ bit field of the parallel port control register). This signal resets the ’C31 and activates the bootload mode. The host then downloads your program or the communications kernel to the ’C31. The parallel port is mapped into the ’C31 memory to the address range 0xFFF000–0xFFFFFF, as shown in Figure 4–4, page 4-7.

The host sends data to the ’C31 in the following way:

1) The host writes the byte to be transmitted to the I/O-mapped area of the host’s parallel port data lines (I/O address 0x378 for LPT 1).

2) The host drives the $\overline{\text{HPSTB}}$ signal low and waits for an acknowledgement. The $\overline{\text{HPSTB}}$ signal interrupts the ’C31 by pulsing the $\overline{\text{INT2}}$ signal, indicating that the host is requesting the transfer of a packet. The $\overline{\text{INT2}}$ signal is needed only for the initial packet transfer request and is ignored during the transmission of the packet.

3) The ’C31 starts a one-wait-state read access to location 0xFFF000. The PAL decodes this address as the host interface active ($\overline{\text{HPACK}}$) signal, drives the host’s ERROR signal low, and drives the ’C31’s $\overline{\text{READY}}$ signal high. This prevents the ’C31 from completing its read access. The host uses the ERROR ($\overline{\text{HPACK}}$) signal to acknowledge that the ’C31 is ready to receive the data.

4) The host drives the $\overline{\text{HPSTB}}$ signal high, indicating to the ’C31 that the data is ready. The PAL detects the raising edge of HPSTB and drives the ’C31’s $\overline{\text{READY}}$ signal low, concluding the ’C31 read cycle.

5) This process is repeated until all four bytes are transferred (least significant byte first). At each transfer, the ’C31 pieces the bytes together to form a 32-bit word.

The host receives data in a similar manner:

1) The host waits for the $\overline{\text{HPACK}}$ signal that indicates the ’C31 has understood the host request for a packet transfer.

2) The ’C31 starts a one-wait-state write access to location 0xFFF000. The PAL decodes this address as the $\overline{\text{HPACK}}$ signal, drives the host’s ERROR signal low, and drives the ’C31’s $\overline{\text{READY}}$ signal high. This prevents the ’C31 from completing its write access. The host uses the ERROR signal to acknowledge that the ’C31 is already sending data.

3) When the host receives the HPIA signal, it drives $\overline{\text{PSTROBE}}$ low and the host reads a byte or 4-bit nibble, depending on whether a bidirectional parallel printer is present in the host.

4) The host drives the $\overline{\text{HPSTB}}$ signal high, indicating to the ’C31 that the data was read. The PAL detects the raising edge of $\overline{\text{HPSTB}}$ and drives the ’C31’s $\overline{\text{READY}}$ signal low, concluding the ’C31 write cycle. This completes the ’C31 read cycle.

5) This process is repeated until all four bytes or eight nibbles are transferred (least significant byte first). During each transfer, the host pieces the bytes together to form a 32-bit word.

Note:

During the boot load process, the ’C31 does not read the third and fourth bytes of the first 32-bit word. The boot loader acts as if it is reading from an EPROM and skips these bytes.

TLC32040 AIC hardware interface

The TLC32040 analog interface circuit (AIC) on the DSK provides:

- ❑ A single-channel, input/output, voice-quality analog interface with 14-bit dynamic range ADC and DAC
- ❑ Variable ADC and DAC sampling rate of up to 20 000 samples per second
- ❑ Output reconstruction filter
- ❑ Bypassable, switched-capacitor, anti-aliasing input filter
- ❑ Auxiliary analog input channel, selectable

The DSK connects the TLC32040 AIC to the ’C31 serial port through a header and 200 Ω isolation resistors. The header lets you disconnect the AIC and use the ’C31’s serial port in the daughterboard. Two additional pins from the ’C31 control resetting and clocking signals to the AIC:

- ❑ The ’C31’s TIMER0 pin drives the master input clock to the AIC
- ❑ The ’C31’s XF0 signal resets the AIC

The AIC’s analog input and output are connected to RCA plugs. These signals are line-level compatible (+/–3V peak) and can be connected to audio line-level inputs and outputs.

The output can also be connected directly to a speaker but will not have significant output level as the output drive is limited by the AIC output driver and series isolation register. For best results, use an external amplifier or high impedance speaker, such as a headphone.

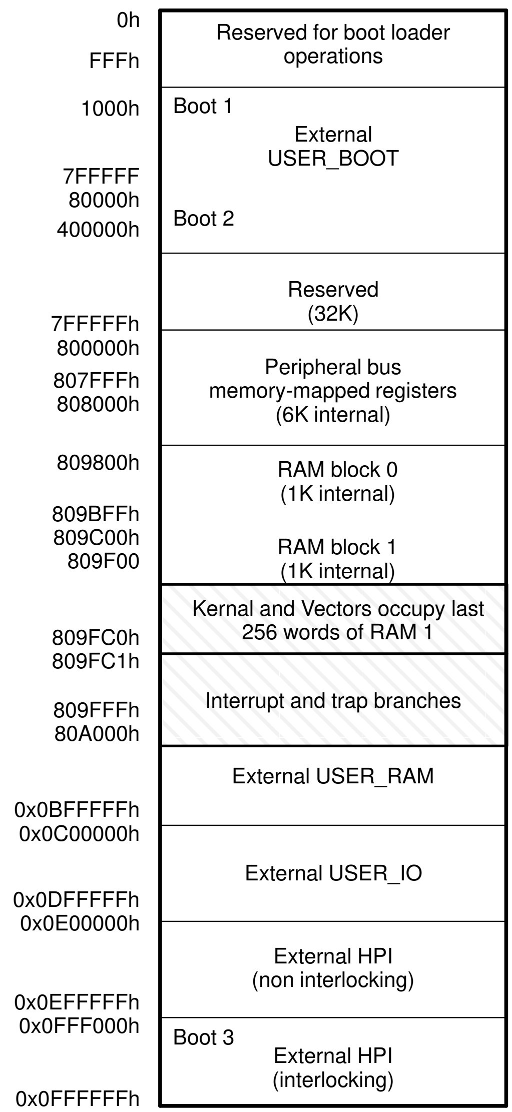
Note:

If the AIC is used with parameters outside the tested range, the AIC performance may be degraded from that specified in the data sheet. See Appendix C, *TLC32040 Data Sheet*, for more information.

DSK memory map

Since host communications occur through the ’C31 parallel bus, the PAL decodes the address of the ’C31 to determine when it is accessing the host interface according to the memory map shown in Figure 4–4.

Figure 4–4. DSK Memory Map



4.2 DSK Communications Kernel

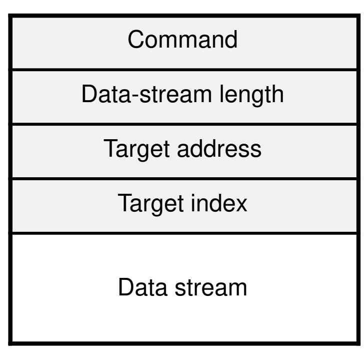
Upon reset, the host downloads a communications kernel to the ’C31 using the boot loader. This communications kernel provides a set of low-level routines that allow the host and the ’C31 to exchange information and perform debugging functions.

Data packets

The host and the ’C31 communicate by exchanging packets of data. Figure 4–5 shows the structure for data packets. The data-packet headers typically consist of four fields: command, data-stream length, target address, and target index. This header is followed by the data stream as shown in Figure 4–5. The header fields are described as follows:

- **Command** directs the handling of the packets. See the *Commands* section for more information.
- **Data-stream length** indicates the length of data in the data stream.
- **Target address** points to the memory location where data is read from or written to.
- **Target index** post-increments the value of the target address after a read or write of a single data item.

Figure 4–5. Data-Packet Structure



Commands

When the ’C31 receives an interrupt from the host ($\overline{INT2}$), the ’C31 saves the current state of the CPU and then receives a packet. Once the ’C31 receives a packet, the communications kernel analyzes the command entry in the header to direct the handling of the packet. The command entry provides the low-level routines necessary to communicate with the host and debug the system. The communications kernel supports these commands:

XWRIT Write a block of data from the host to the DSK. This command takes data-stream-length items from the host and writes them into the ’C31’s memory location pointed to by the target address. The target address is incremented by the target index after each write operation.

XREAD Read a block of data from the DSK to the host. This command reads data-stream-length items from the ’C31’s memory location pointed by the target address and sends them to the host. The target address is incremented by the target index after each read operation.

XCTXT Get the ’C31 context save buffer address.

XRUNF Restore the context of the CPU and execute code until a breakpoint is encountered or a halt command is issued. This command is used for debugging.

XSTEP Restore the context of the CPU, execute a single instruction, and then save the context of the CPU. This command is used for debugging.

XHALT Save the context of the CPU and wait for a new command. This command is used for debugging.

Debugging functions

Several debugging functions are implemented within the communications kernel by building upon the low-level communications commands. The kernel's debugging functions can execute as a background task that is integrated into the system. Debugging does not halt the system, but allows concurrent execution of other tasks. Debugging is fast and efficient and requires only a host interface. However, it does consume some amount of processor memory and bandwidth.

In contrast, scan-based emulation, which is another popular debugging methodology, is extremely helpful since it does not consume system memory and it provides a snapshot in time of the processors in the system. The DSK board has an MPSD header that allows the use of the XDS510 scan-based emulator. However, scan-based emulation is a non real-time emulation that requires the complete system to halt. Due to the low data-transfer rates, it is often inadequate for application data transfers. Also, external interrupts are often masked, and can effectively freeze communications and other interrupt-driven tasks. Halting and restarting the processor causes many breaks in the CPU pipeline, which defeats the purpose of real-time operation.

Since the debugging functions provided in the communications kernel operate as a background task, they never disable the CPU or force a pipeline flush. For example, single-stepping an opcode in scan-based emulation executes the opcode, flushes the pipeline, and freezes the timers and DMA. On the other hand, real-time debugging follows standard interrupt service routine rules for context switching.

Due to the real-time nature of the debugging session, debugging functions save and restore the context of the CPU before and after executing the debugging function. The kernel implements this *context save* as a typical interrupt service routine that saves and restores all CPU registers (28 registers). Peripheral control registers are not preserved, because the communications kernel does not modify them. Note that the extended-precision CPU registers require two memory locations to store the most significant 8 bits and the least significant 32 bits. After saving the context, the CPU enters a spin mode, where it waits for additional commands. During this time, the context area can be downloaded, displayed, or modified, usually under the supervision of a host debugger routine. An XRUNF or XSTEP command indicates to the CPU that it needs to restore the context area to its correct running state and then continue execution. The host accesses the 'C31's context-save area by looking up the pointer to the context through the XCTXT command.

The communications kernel implements breakpoints by replacing the code at the desired location with a TRAPn opcode. When the CPU encounters a TRAP, the context-save routine is invoked, the CPU enters spin mode, writes an acknowledge to the host, and waits for a new command. While in spin mode, the CPU can receive new interrupts.

The communications kernel implements CPU halt (XHALT) in a manner similar to breakpoint halts, but the interrupt source originates from the host, not a TRAP opcode. The main difference is that the registers used by the communications kernel are restored before invoking a full context save and falling into spin mode.

The kernel implements XRUN by restoring the context followed by a standard return from interrupt. The processor is then free to execute code.

The communications kernel implements the opcode XSTEP by using a reserved interrupt in the 'C31: Serial Port 1 transmit interrupt (XINT1). Figure 4–6, on page 4-12, shows the single-step routine flow diagram. The communications kernel:

- Restores the context of the CPU
- Places the program counter into R5
- Clears INT2
- Sets the XINT1 interrupt
- Restores the status register
- Sets a delayed branch on R5

The delayed branch executes the next three instructions:

1) Setting the global interrupt enable
2) Restoring R5
3) Restoring the data page pointer

By coordinating the setting of the XINT1 interrupt and the branch-to-the-user program, the kernel allows only a single instruction to execute before servicing the pending interrupt. When the interrupt is recognized, the kernel saves the CPU context, sends an acknowledge to the host, branches to the spin mode, and waits for a new command.

Figure 4–6. Single-Step Flow Diagram

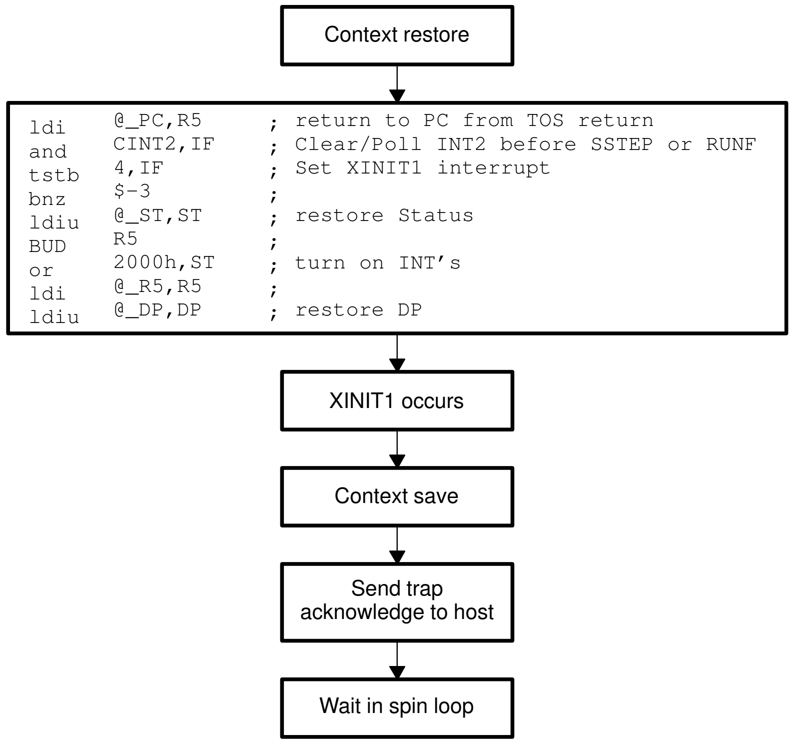


Table 4–1. Single-Step Pipeline Flow

Cycle	Description	Fetch	Decode	Read	Execute
1		BUD R5			
2		or 2000h,ST	BUD R5		
3		ldi @_R5, R5	or 2000h,ST	BUD R5	
4		ldp @_DP,DP	ldi @_R5,R5	or 2000h,ST	BUD R5
5	Set Global Interrupt Enable	USER1	ldp @_DP,DP	ldi @_R5,R5	or 2000h,ST
6	Interrupt recognized	USER2	USER1	ldp @_DP,DP	ldi @_R5,R5
7	Jam interrupt in pipeline (discard USER2 fetch)	– – –	XINT1	USER1	ldp @_DP,DP
8	Execute USER1 instruction	– – –	– – –	XINT1	USER1
9	Clear interrupt flag; clear GIE; store return address on stack; read vector table	– – –	– – –	– – –	XINT1
10	Pipeline begins to fill with interrupt service routine	XSTEP ISR	– – –	– – –	– – –
11	Pipeline continues to fill with ISR	ISR2	XSTEP ISR	– – –	– – –
12	Pipeline continues to fill with ISR	ISR3	ISR2	XSTEP ISR	– – –
13	Execute first instruction of ISR	ISR4	ISR3	ISR2	XSTEP ISR

Table 4–1 describes the pipeline flow that sets the XINT1 interrupt and branches to your code. This table shows that the activities in the pipeline are coordinated so that the code is fetched at the same time global interrupts are enabled. In this way, the interrupt is placed in the pipeline right after fetching the second instruction. This instruction is discarded and the pipeline is filled with the interrupt service routine (ISR).

Another way of interpreting CPU interrupts is to treat them as a special kind of opcode that is inserted into the pipeline. Instructions that are in the pipeline before the interrupt occurs must complete execution.

4.3 TLC32040 AIC Initialization

To use the TLC32040 analog interface circuit (AIC), you must follow a sequence of steps to initialize and set up the 'C31's timer and serial port, and to reset and program the AIC. The following subsections describe this process.

Resetting the AIC

As shown in Figure 4–1, page 4-3, the 'C31's XF0 signal is connected to the $\overline{\text{RESET}}$ signal of the AIC. By toggling the $\overline{\text{RESET}}$ signal, the 'C31 can reset the AIC. This is achieved by executing the following instructions:

```
ldi 2h,IOF              ;  Pull AIC into reset
ldi 6h,IOF              ;  Pull AIC out of reset
```

Initializing the 'C31 timer

As shown in Figure 4–1, page 4-3, the 'C31's timer (TCLKO) signal is connected to the AIC's master clock (MCLK) signal. The MCLK signal drives all the key logic signals of the AIC, such as the shift clock, the switched-capacitor filter clocks, and the A/D and D/A timing signals. The timer pulses the TCLK0 signal whenever the 'C31 timer counter register (memory mapped to 0x0080 8024h) counts up to the timer period register (memory mapped to 0x0080 8028h) value. Then, the timer counter registers reset to zero and repeat. (For a detailed description of the 'C31 timer, refer to the *TMS320C3x User's Guide*). Because of differences between the maximum frequency of the 'C31's timer and the maximum and minimum frequencies of the AIC, the following constraints should be observed:

- ❑ **Minimum Timer Period Register Value.**The 'C31 50 MHz can generate a maximum timer frequency of 12.5 MHz (CLKIN/4), which is above the AIC's master clock frequency maximum of 10 MHz. Therefore, the 'C31's timer counter register's minimum value should be 1, for a master clock frequency of 6.25 MHz (CLKIN/8). If you sample at higher frequencies than those specified for the AIC (greater than 20 kHz), the minimum value of the timer counter register should be 0. However, these higher frequencies are beyond the specifications of the TLC32040 data sheet and resulting performance is not described. See Appendix C, *TLC32040 Data Sheet*, for more information.

- **Maximum Timer Period Register Value**. The AIC's minimum master clock frequency is 75 kHz. Taking into account the 'C31 maximum timer frequency of 12.5 MHz and the AIC's minimum master clock frequency, the 'C31's timer counter register maximum value should be 165 (12.5 MHz / 75 kHz = 166.7). The 'C31's timer counts down to 0, therefore, you need to subtract 1 from this number (166 – 1 = 165). Note that the TLC32040 specification describes a minimum clock frequency since the internal signals of the AIC are stored in capacitors that must be periodically updated.

- **Timer Initialization**. The following 'C31 assembly code shows how to set up the timer with a timer counter of 1 and the timer global control register (TGCR0) set with TCLK0 as the timer pin, start the timer (GO and HLD = 1), start the internal clock source, and start the clock mode:

```
TGCR0   .set    808020h     ; Timer 0 global control register
TCNT0   .set    808024h     ; Timer 0 counter register
TPR0    .set    808028h        ; Timer 0 period register
TIMVAL  .word   3c1h        ; Timer global control register value
        ldp     @TGCR0      ; Set Data Page
        ldi     0h,R4       ; Initialize R4 to zero
        ldi     1h,R0       ; Initialize R0 to 1
        sti     R4,@TGCR0   ; Reset timer0
        sti     R0,@TPR0    ; Store timer0 period
        sti     R4,@TCNT0   ; Reset timer0 counter
        ldi     @TIMVAL,R7  ; Load timer control value
        sti     R7,@TGCR0   ; Start timer 0
```

Initializing the 'C31 serial port

This subsection explains how to initialize the following:

- 'C31 serial port
- 'C31 serial-port control register (memory-mapped to 0x0080 8040h)
- FSX/DX/CLKX control register (memory-mapped to 0x0080 8042h)
- FSR/DR/CLKR control register (memory-mapped to 0x0080 8043h)

For a detailed description of the 'C31 serial port, see the *TMS320C3x User's Guide*.

The ’C31 assembly code in Example 4–1 initializes the serial port global control register (SGCR0) in the following manner:

- By issuing transmit and receive resets
- Enabling receive and transmit interrupts
- Setting 16-bit receive and transmit transfers
- Setting FSX and FSR, CLKX and CLKR active low
- Setting continuous mode
- Setting variable data rate transfers:

Example 4–1. Initialize the Serial Port Global Control Register

```
SGCR0   .set    808040h       ; Serial port 0 global control register  ;
SPCX0   .set    808042h       ; Serial port 0 FSX/DX/CLKX control reg. ;
SPCR0   .set    808043h       ; Serial port 0 FSR/DR/CLKR control reg. ;
SINIT0  .word    0e973300h    ; Enable RINT & 16-bit transfers
SINIT1  .word    111h         ; Configure as serial port pins
          ldp    @SGCR0       ; Set Data Page
          ldi    0h,R4        ; Initialize R4 to zero
          sti    R4,@SGCR0
          ldi    @SINIT1,R7   ; Reset and
          sti    R7,@SPCX0    ;   initialize serial port
          sti    R7,@SPCR0    ;   initialize serial port
          ldi    @SINIT0,R7   ; Reset and
          sti    R7,@SGCR0    ;   initialize serial port
```

Also refer to the example code supplied with the DSK for help on setting up the AIC.

Initializing the AIC

Once the ’C31 supplies MCLK, has its serial port initialized, and resets the AIC, you can initialize the AIC to a specified sample rate. The AIC sampling rate is determined by the values of two registers called A and B in the AIC’s transmit and receive sections. These values are loaded into the respective counter whenever the counter counts down to 0. Tx counter A and B determine the D/A conversion timing, Rx counter A and B determine the A/D conversion timing. For more information, refer to Appendix C, *TLC32040 AIC Data Sheet*. The formula for the conversion frequency is given in Equation 4–1.

Equation 4–1. Conversion Frequency

$$Conversion\_\ frequency = \frac{MCLK}{2 \times A \times B}$$

To ensure that the switched-capacitor lowpass and bandpass filters meet their transfer function characteristics, the frequency of the clock inputs of the switched-capacitor filter must be 288 kHz; otherwise, the upper and lower cut-off frequencies of the low-pass and band-pass are sealed accordingly. The switched capacitor filter frequency is given in Equation 4–2.

Equation 4–2. Switched Capacitor Filter Frequency

$$SCF\_Clock\_\,frequency = \frac{MCLK}{2 \times A}$$

For example, using this equation for an 8-kHz sampling rate with a MCLK of 6.25 MHz, leads to a Tx counter A of 11 [A = MCLK/(2 × SCF)]. Using Equation 4–2, Tx counter B results in 36 [B = MCLK/(2 × A × *Conversion_Frequency*)].

To initialize the AIC's Tx counter A and B registers, you must send a primary communication followed by a secondary communication (explained in the *Primary communications* subsection below, and *Secondary communications* subsection, on page 4-18.) Primary communications load values into the D/A while secondary communications load A/D internal registers, such as the control register, Tx counters A and B, and Rx counters A and B.

Primary communications

Primary communications have a data value in the 14 MSBs (D15–D2) of data and a mode selection in the two LSBs (D1–D0). This format is shown in Figure 4–7.

Figure 4–7. Primary Communication Data Format

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
D/A converter value														mode selection	

The AIC sends the data value to the D/A converter and enables one of the modes shown in Table 4–2 depending on the two LSBs.

Table 4–2. Primary Communications Mode Selection

LSBs	Mode
00	Tx Counter A ← TA, Rx Counter A ← RA Tx Counter B ← TB, Rx Counter B ← RB
01	Tx Counter A← TA + TA’, Rx Counter A ← RA + RA’ Tx Counter B ←TB, Rx Counter B ← RB
10	Tx Counter A ←TA - TA’, Rx Counter A ← RA + RA’ Tx Counter B ←TB, Rx Counter B ← RB
11	Tx Counter A ← TA, Rx Counter A ← RA Tx Counter B ←TB, Rx Counter B ← RB

The second and third modes use TA’ and RA’ registers to advance or slow down the sampling frequency by shortening or lengthening the sample period. This is particularly useful in modem applications. It can also enhance the signal-to-noise performance, perform frequency-tracking functions, and generate nonstandard modem frequencies.

Secondary communications

Secondary communication follows a primary communication that has the two LSBs set to 11. This secondary communication programs the AIC by loading the A, A’, B, or control registers. Figure 4–8 shows the secondary communication data format. The TA, RA, TB, and RB values are unsigned. The TA’ and RA’ values are in signed 2s-complement format. The control register enables and disables auxiliary inputs, bandpass filters, etc.

Figure 4–8. Secondary Communication Data Format

<table>
<tr><th>D15</th><th>D14</th><th>D13</th><th>D12</th><th>D11</th><th>D10</th><th>D9</th><th>D8</th><th>D7</th><th>D6</th><th>D5</th><th>D4</th><th>D3</th><th>D2</th><th>D1</th><th>D0</th></tr>
<tr><td>X</td><td>X</td><td colspan="5">TA register value (unsigned)</td><td>X</td><td>X</td><td colspan="5">RA register value (unsigned)</td><td>0</td><td>0</td></tr>
<tr><td>X</td><td colspan="6">TA’ register value (signed 2s complement)</td><td>X</td><td colspan="6">RA’ register value (signed 2s complement)</td><td>0</td><td>1</td></tr>
<tr><td>X</td><td colspan="6">TB register value (unsigned)</td><td>X</td><td colspan="6">RB register value (unsigned)</td><td>1</td><td>0</td></tr>
<tr><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td>X</td><td colspan="6">Control register</td><td>1</td><td>1</td></tr>
</table>

Figure 4–9 describes the control register bit fields.

Figure 4–9. Control Register Bit Fields

D7 D6	D5	D4	D3	D2
Input Gain	**Transmit/Receive**	**AUX IN Pins**	**Loopback Function**	**Bandpass Filter**
0 0 = 1X for ± 6V analog input	0 = asynchronous	0 = disables	0 = disables	0 = deletes
0 1 = 2X for ± 3V analog input	1 = enables	1 = enables	1 = enables	1 = inserts
1 0 = 4X for ± 1.5V analog input				
1 1 = 1X for ± 6V analog input				

The assembly code in Example 4–2 sets the TA and TB registers of the AIC. This code transmits a 16-bit word to the AIC and then waits until the transmit interrupt is generated by the serial port. Four commands are transmitted starting with a 0, then the TB and RB values, followed by the TA and RA values, and finally the control word. TA and RA values should be the last values transmitted, since they change the AIC sample rate. By transmitting these values last, the sample rate is not changed until the AIC receives the last program word. In this way, very high sample rates can be achieved. Each command transmits three 16-bit words: a primary communication, a secondary communication, and a 0-data word.

Example 4–2. Setting the TA and TB Registers

```
;----------------------------------------------------------------------------
; LOOPAIC.ASM is an example program which shows how to initialize and use
; the TLC32040.  The analog output (DAC output) is either a ramp signal
; (RAMPEN=1) or a loopback of the analog input (RAMPEN=0).
;----------------------------------------------------------------------------
          .start  ”AICTEST”,0x809802 ; Start assembling here
          .sect   ”AICTEST”          ;
;-------------------------------------
; Define constants used by program
;-------------------------------------
RAMPEN    .set   1                   ; Set to 1 to generate ramp at AOUT
T0_ctrl   .set   0x808020            ; TIM0 gl control
T0_count  .set   0x808024            ; TIM0 count
T0_prd    .set   0x808028            ; TIM0 prd
S0_gctrl  .set   0x808040            ; SP 0 global control
S0_xctrl  .set   0x808042            ; SP 0 FSX/DX/CLKX port ctl
S0_rctrl  .set   0x808043            ; SP 0 FSR/DR/CLKR port ctl
S0_xdata  .set   0x808048            ; SP 0 Data transmit
S0_rdata  .set   0x80804C            ; SP 0 Data receive
TA        .set   12                  ; AIC timing register values
TB        .set   15                  ;
RA        .set   12                  ;
RB        .set   15                  ;
GIE       .set   0x2000              ; This bit in ST turns on interrupts
;-------------------------------------
; Define some constant storage data
;-------------------------------------
A_REG        .word  (TA<<9)+(RA<<2)+0 ; A registers
B_REG        .word  (TB<<9)+(RB<<2)+2 ; B registers
C_REG        .word  10000011b         ; control
S0_gctrl_val .word  0x0E970300        ; Serial port control register values
S0_xctrl_val .word  0x00000111        ;
S0_rctrl_val .word  0x00000111        ;
RAMP         .word  0                 ; RAMP count value
ADC_last     .word  0                 ; Last received ADC value
```

Example 4–2. Setting the TA and TB Registers (Continued)

```
;****************************************************
; Begin main code loop here
;****************************************************
main      or    GIE,ST          ; Turn on INTS
          ldi   0x34,IE         ; Enable XINT/RINT/INT2
          b     main            ; Do it again!
;-------------------------------
DAC2      push  ST              ; DAC Interrupt service routine
          push  R3              ;
          .if   RAMPEN          ; If RAMPEN=1 assemble this code
          ldi   @RAMP,R3        ;
          addi  256,R3          ; Add a value to RAMP
          sti   R3,@RAMP        ;
          .else                 ; Else assemble this
          ldi   @ADC_last,R3    ;
          .endif                ;
          andn  3,R3            ;
          sti   R3,@S0_xdata    ; Output the new DAC value
          pop   R3              ;
          pop   ST              ;
          reti                  ;
;-------------------------------
ADC2      push  ST              ;
          push  R3              ;
          ldi   @S0_rdata,R3    ;
          sti   R3,@ADC_last    ;
          pop   R3              ;
          pop   ST              ;
          reti                  ;
;*****************************************************;
; The startup stub is used during initialization only ;
; and can be safely overwritten by the stack or data  ;
;*****************************************************;
          .entry   ST_STUB      ; Debugger starts here
ST_STUB   ldp   T0_ctrl         ; Use kernel data page and stack
          ldi   0,R0            ; Halt TIM0 & TIM1
          sti   R0,@T0_ctrl     ;
          sti   R0,@T0_count    ; Set counts to 0
          ldi   1,R0            ; Set periods to 1
          sti   R0,@T0_prd      ;
          ldi   0x2C1,R0        ; Restart both timers
          sti   R0,@T0_ctrl     ;
          ;---------------------
          ldi   @S0_xctrl_val,R0;
          sti   R0,@S0_xctrl    ; transmit control
          ldi   @S0_rctrl_val,R0;
          sti   R0,@S0_rctrl    ; receive control
          ldi   0,R0            ;
          sti   R0,@S0_xdata    ; DXR data value
          ldi   @S0_gctrl_val,R0; Setup serial port
          sti   R0,@S0_gctrl    ; global control
```

Example 4–2.Setting the TA and TB Registers (Continued)

```
;======================================================;
; This section of code initializes the AIC            ;
;======================================================;
AIC_INIT  LDI   0x10,IE         ; Enable only XINT interrupt
          andn  0x34,IF         ;
          ldi   0,R0            ;
          sti   R0,@S0_xdata    ;
          RPTS  0x040           ;
          LDI   2,IOF           ; XF0=0 resets AIC
          rpts  0x40            ;
          LDI   6,IOF           ; XF0=1 runs AIC
          ;--------------------
          ldi   @C_REG,R0       ; Setup control register
          call  prog_AIC        ;
          ldi   0xfffc  ,R0     ; Program the AIC to be real slow
          call  prog_AIC        ;
          ldi   0xfffc|2,R0     ;
          call  prog_AIC        ;
          ldi   @B_REG,R0       ; Bump up the Fs to final rate
          call  prog_AIC        ; (smallest divisor should be last)
          ldi   @A_REG,R0       ;
;------------------------------
prog_AIC  ldi   @S0_xdata,R1    ; Use original DXR data during 2 ndy
          sti   R1,@S0_xdata    ;
          idle
          ldi   @S0_xdata,R1    ; Use original DXR data during 2 ndy
          or    3,R1            ; Request 2 ndy XMIT
          sti   R1,@S0_xdata    ;
          idle                  ;
          sti   R0,@S0_xdata    ; Send register value
          idle                  ;
          andn  3,R1            ;
          sti   R1,@S0_xdata    ; Leave with original safe value in DXR
          ;--------------------
          ldi   @S0_rdata,R0    ; Fix the receiver underrun by reading
          b     main            ; the DRR before going to the main loop
;***************************************************;
; Install the XINT/RINT ISR handler directly into    ;
; the vector RAM location it will be used for        ;
;***************************************************;
          .start   ”SP0VECTS”,0x809FC5
          .sect    ”SP0VECTS”
          B        DAC2         ; XINT0
          B        ADC2         ; RINT0
```

4.4 Host Software

The DSK software includes three source code files that manipulate the parallel printer port and perform the necessary functions to communicate with the ’C31. The commands in each of the source-code files are summarized in the following subsections. The source files are:

target.cpp includes the low-level routines that manipulate the data transmissions into packets that are recognized by the ’C31 communications kernel.

driver.cpp includes driver-level routines that control the host’s parallel printer port interface.

object.cpp uses the target- and driver-level routines to initialize and download programs to the ’C31.

The following subsections describe the routines contained in each of these files.

The DSK software also includes an assembler and a debugger. These are described in Chapter 5, *Using the DSK Assembler*, and Chapter 7, *Using the DSK Debugger*.

Host communications target routines

The communications kernel resident in the 'C31 assumes that data transfers to and from the host are organized into packets as shown in Figure 4–5 on page 4-8. The target.cpp file includes routines that manipulate data transmissions between the host and the 'C31 into this packet structure. These routines read and write blocks of data from the 'C31 memory, send commands to the 'C31, perform context save and restores, and provide debugging commands, such as run, single-step, and halt.

getmem

Get Memory

Syntax

MSGS getmem (ulong addr, ulong length, ulong *data)

Description

The **getmem** routine reads a block of data from the 'C31 memory.

Arguments

addr	Address of the data to be read
length	Size of memory block to read
data	Pointer to host memory address in which to place data read from the 'C31

Return Value

NO_ERR	Block read completed successfully
RECV_ERR	Failed reception
XMIT_ERR	Failed transmission

putmem

Put Memory

Syntax

MSGS putmem (ulong addr, ulong length, ulong *data)

Description

The **putmem** routine writes a block of data into 'C31 memory.

Arguments

addr	Starting address to write the data to
length	Size of memory block to write
data	Pointer to host memory address to read data from. The data is then placed into 'C31 memory.

Return Value

NO_ERR	Block write completed successfully
XMIT_ERR	Failed transmission

SSTEP_CPU — *Single-Step Command*

Syntax **MSGS SSTEP_CPU (void)**

Description The **SSTEP_CPU** routine single-steps one instruction by restoring the context of the CPU, executing one instruction, and then saving the CPU context. This command places the CPU in command mode.

Arguments None

Return Value

NO_ERR	Command and data completed successfully
XMIT_ERR	Failed transmission
RECV_ERR	Failed reception

RUN_CPU — *Run Command*

Syntax **MSGS RUN_CPU (void)**

Description The **RUN_CPU** routine executes instructions starting at the program counter obtained from the CPU context save area and ending at a breakpoint, if one has been set.

Arguments None

Return Value

NO_ERR	Command and data completed successfully
XMIT_ERR	Failed transmission

HALT_CPU — *Halt Command*

Syntax **MSGS HALT_CPU (void)**

Description The **HALT_CPU** routine halts the execution of instructions. This command places the CPU in command mode and saves the CPU context.

Arguments None

Return Value

NO_ERR	Command completed successfully
RECV_ERR	Failed reception

GET_ DEBUG_CTXT

Return CPU Context Save Address

Syntax **MSGS GET_DEBUG_CTXT (void)**

Description The **GET_DEBUG_CTXT** routine retrieves the ’C31 context save location starting address.

Arguments None

Return Value

NO_ERR	Command completed successfully
RECV_ERR	Failed reception
XMIT_ERR	Failed transmission

Host communications driver routines

To facilitate the data transfer from the host to the 'C31, the DSK software includes several driver-level routines in the file driver.cpp. This file includes routines that manipulate the hardware interface circuitry of the host to reset, send, and receive data through unidirectional and bidirectional parallel printer ports.

DSK_reset — *Reset*

Syntax **MSGS DSK_reset (void)**

Description The **reset** routine resets the DSK by toggling the INIT signal.

Arguments None

Return Value

NO_ERR	Reset sequence completed
RESET_ERR	Reset has failed

input_rdy — *Input Ready*

Syntax **char input_rdy (void)**

Description The **input_rdy** routine indicates that the DSK is ready to receive

Arguments None

Return Value

0	DSK ready to receive data
1	DSK not responding to host command

recv_long_byte — *Receive Long Byte*

Syntax **MSGS recv_long_byte (ulong * rcv_data)**

Description The **recv_long_byte** routine receives a 32-bit value in four 8-bit data transfers (to be used only in bidirectional parallel printer ports).

Arguments **rcv_data** Address of the value to receive

Return Value

NO_ERR	Successful reception
RECV_ERR	Failed reception

recv_long

Receive Long

Syntax

MSGS recv_long (ulong *rcv_data)

Description

The **recv_long** routine receives a 32-bit value in eight 4-bit data transfers (to be used in bidirectional and unidirectional parallel printer ports).

Arguments

rcv_data Address of the value to receive

Return Value

NO_ERR	Successful reception
RECV_ERR	Failed reception

xmit_long

Transmit Long

Syntax

MSGS xmit_long (ulong snd_data)

Description

The **xmit_long** routine transmits a 32-bit value in four 8-bit data transfers (to be used in bidirectional and unidirectional parallel printer ports).

Arguments

snd_data Value to transmit

Return Value

NO_ERR	Successful transmission
XMIT_ERR	Failed transmission

xmit_byte

Transmit Byte

Syntax

MSGS xmit_byte (char snd_data)

Description

The **xmit_byte** routine transmits an 8-bit value in a single data transfer (to be used in bidirectional and unidirectional parallel printer ports)

Arguments

snd_data Value to transmit

Return Value

NO_ERR	Successful transmission
XMIT_ERR	Failed transmission

Host communications object routines

Using the low-level driver routines, the DSK software provides several high-level routines that allow the loading of programs or data from dsk3a files or COFF (Common Object File Format), that move binary data from the host to the DSK, and that initialize the DSK system. These routines assume an active communications kernel resident on the 'C31 to send and receive packets of data. See Appendix A of the *TMS320 Floating-Point Assembly Language Tools User's Guide* for a detailed description of the COFF format.

LF_Cmd — *Load COFF*

Syntax **MSGS LF_Cmd (char *infilename)**

Description The **LF_Cmd** routine transmits a file with DSK formatted output or COFF data into the 'C31's memory.

When invoking the TMS320 optimizing C compiler use the following switches: *–c* (ROM model) and *–e c_int00*.

Arguments **infilename** Pointer to the character string containing the COFF filename

Return Value

NO_ERR	Successful transmission
OPEN_ERR	Cannot open file
ACCESS_ERR	File not found
INV_COFF_MGC	COFF file not created for a TMS320C31
MAX_SECTN	More than 64 sections
BAD_OPTN_HDR	Incorrect optional COFF header
COM_ERR	Communication failure

Init_System — *Initialize System*

Syntax **MSGS Init_System (char *filename)**

Description The **Init_System** routine initializes the DSK by checking for the presence of power on the DSK board, resetting the 'C31, and loading the given boot source file created by hex30.

Arguments Filename pointer to character string containing the boot source filename

Return Value

NO_ERR	Successful initialization
INIT_ERR	Initialization failure

Chapter 5

Using the DSK Assembler

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

Topic **Page**

5.1	Creating DSK Assembler Source Files	5-2
5.2	Constants	5-8
5.3	Character Strings	5-10
5.4	Symbols	5-11
5.5	Expression Analyzer	5-12
5.6	Assembling Your Program	5-15
5.7	Placing Code Sections Memory Locations	5-16

5.1 Creating DSK Assembler Source Files

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

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

The next several lines show examples of source statements:

```
C_REG   .set  ((10100b)<<2)+3  ; Control word

        .text
start   ldi  2h, IOF           ; Pull AIC into reset
        ldi  0h, T4            ; Clear R4
        ldp  SGCR0
        sti  R4, @SGCR0        ; Reset serial port
        ldi  @SINIT1, R7       ; Load initialization value 1 into R7
        sti  R7, @SPCX0        ; Initialize FSX/DX/CLKX control reg.
        sti  R7, @SPCR0        ; Initialize FSR/DR/CLKR control reg.
        ldi  @SINIT0, R7       ; Load initialization value 0 into R7
        sti  R7, @SGCR0        ; Enable RINT and 16-bit transfers
        sti  R4, @DTX0         ; Transmit 0

        sti  R4, @TGCR0        ; Reset timer 0
        ldi  TIMERPER, R7
        sti  R7, @TPR0         ; Store timer 0 period
```

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

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

Follow these guidelines:

- All statements must begin with a label, a blank, an asterisk, or a semicolon.
- Labels are optional; if you use them, they must begin in column 1.
- One or more blanks must separate each field. Note that tab characters are equivalent to blanks.
- Comments are optional. Comments that begin in column 1 can begin with an asterisk or a semicolon (* or ;), but comments that begin in any other column *must* begin with a semicolon.

Using valid labels

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

```
        .start ”.text”,0x809C00
        .entry  start
CTRL    .set    0
IN      .set    1
OUT     .set    2
        .text
WSHIFT          .word   -8
start   ldp     @stack          ; Load data page
        ldi     @stack,SP       ; Initialize the stack
        ldf     0.0,R0
        ldi     0x100,R0
        lsh     @WSHIFT,R0
        BRD     jump
        ldf     0.0,R1
        ldf     0.0,R2
        ldf     1.0,R3
jump    ldf     1.0,R4
        b       start
stack   .word   $ + 1
        .end
```

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

When you use a label, its value is the current value of the section program counter (the label points to the statement it’s associated with). If, for example, you use the .int directive to initialize several words, a label would point to the first word. In the following example, the label Begin has the value 0x00809800.

```
0x00809800 directive  Begin      .int  0Ah,3,7
0x00809800 0x0000000a <int>
0x00809801 0x00000003 <int>
0x00809802 0x00000007 <int>
```

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

```
0x0080981f nocode     XMIT
0x0080981f 0x10760010      or   10h, IE
0x00809820 0x06000000      idle
```

When an opcode or directive references a label, the label is substituted with the address of the label’s location in memory. The only exception to this is the .set directive, which assigns a value to a label. If you don’t use a label, the first character position must contain a blank, a semicolon, or an asterisk.

Using the mnemonic field

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

- Machine-instruction mnemonic (such as ADDI, MPYF)
- Assembler directive (such as .data, .set, .entry)

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

```
;===================================================
          .start  ”AICTEST”,0x809900
          .sect   ”AICTEST”
GIE       .set    0x2000
;=============================================================
A_REG     .word  (TA<<9)+(RA<<2)+0 ; 0x809902
B_REG     .word  (TB<<9)+(RB<<2)+2 ; 0x809903
C_REG     .word  10000011b         ; 0x809904  +/- 1.5 V
;
S0_gctrl_val  .word  0x0E970300
S0_xctrl_val  .word  0x00000111 ;
S0_rctrl_val  .word  0x00000111 ;
;
prog_AIC  push  R1
          push  IE
          ldi   0x10,IE
          andn  0x30,IF
          ldi   @S0_xdata,R1
          sti   R1,@S0_xdata
          idle
          ldi   @S0_xdata,R1
          or    3,R1
          sti   R1,@S0_xdata
          idle
          sti   R0,@S0_xdata
          idle
          andn  3,R1
          sti   R1,@S0_xdata
          pop   IE
          pop   R1
          rets
```

Refer to the *TMS320C3x User’s Guide* for syntax specifications on individual opcodes.

It is necessary to resolve *all* fields in an opcode. If an opcode field (such as the section name in a .sect opcode) is omitted, the assembler generates the error statement, “Invalid, Undefined, or Missing Operand”.

Using the operand field

The operand field is a list of operands that follow the mnemonic field. An operand can be a constant (see Section 5.2, page 5-8), a symbol (see Section 5.4, page 5-11), or a combination of constants and symbols in an expression. You must separate operands with commas.

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

- **No prefix — the operand is a well-defined immediate value**. The assembler expects a well-defined immediate value, such as a register symbol or a constant. For floating-point operations, use an extended register (R0–R7). For integer operations, use any register. For example:

  ```
  Label: ADDI  0x0, R4
  ```

  This instruction adds the integer value 0 to the extended-precision register R4.

- **@ prefix — the operand is direct address**. If you use the @ sign as a prefix, the assembler treats the operand as the contents of a 32-bit address, specified by @addr. The 16 MSBs of the address are specified by the DP register; the 16 LSBs are specified by the instruction word. For example:

  ```
  Label: LDP  0x0080
         ADDI @0x9800, R0
  ```

  The first line of this code sets the DP register to 0x0080. The second line uses the concatenated value of DP and 0x9800 to form an address of 0x0080 9800. The value stored at 0x0080 9800 to is then added the value stored in R0.

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

  ```
  Label: ADDI  *AR3, R0
  ```

  This instruction adds the integer stored in the location pointed to by AR3 to the value stored in R0.

  Table 5–1 lists the various forms that indirect operands may take. The displacement can be specified as a value from 0–255 or as one of the index registers (IR0 or IR1). It is not necessary to specify the displacement if it is 1, because the assembler assumes a default displacement of 1. For example, *++ARn is equivalent to *++ARn(1).

Table 5–1. Indirect Addressing

Operand	Description
*ARn	Indirect with no displacement
*+ARn(*disp*)	Indirect with predisplacement or preindex add
*–ARn(*disp*)	Indirect with predisplacement or preindex subtract
*++ARn(*disp*)	Indirect with predisplacement or preindex add and modification
*––ARn(*disp*)	Indirect with predisplacement or preindex subtract and modification
*ARn++(*disp*)[%] †	Indirect with postdisplacement or postindex add and modification
*ARn––(*disp*)[%] †	Indirect with postdisplacement or postindex subtract and modification
*ARn++(IR0)B	Indirect with postindex (IR0) and bit-reversed modification

† Optional circular modification (specified by %)

For more information on indirect addressing and bit-reversed addressing, refer to the *TMS320C3x User's Guide*.

Commenting your source file

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

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

```
* Memory map register locations
SGR0   .set  0x808040  ; Serial port 0 global control register
SPCX0  .set  0x808042  ; Serial port 0 FSX/DX/CLKX control reg.
SPCR0  .set  0x808043  ; Serial port 0 FSR/DR/CLKR control reg.
DTX0   .set  0x808048  ; Serial port 0 data transmit register
DRX0   .set  0x80804c  ; Serial port 0 data receive register
TGCR0  .set  0x808020  ; Timer 0 global control register
TCNT0  .set  0x808024  ; Timer 0 counter register
TPR0   .set  0x808028  ; Timer 0 period register
```

Another way to comment your source file is to use an asterisk *in column 1* of your code.

If the asterisk is not in column 1, the assembler assumes it is part of your code and may generate an error.

A source statement that contains only a comment is valid.

5.2 Constants

The assembler supports five types of constants:

- Binary integer constants
- Decimal integer constants
- Hexadecimal integer constants
- Floating-point constants
- Character constants

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

Binary integers

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

0101b	Constant equal to 5
10101B	Constant equal to 21
–0101b	Constant equal to –5

Decimal integers

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

1000	Constant equal to $1\ 000_{10}$ or $3E8_{16}$
–32768	Constant equal to $–32\ 768_{10}$ or 8000_{16}
25	Constant equal to 25_{10} or 19_{16}

Hexadecimal integers

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

78H	Constant equal to 120_{10} or 0078_{16}
0x0f	Constant equal to 15_{10} or $000F_{16}$
37ACh	Constant equal to $14\ 252_{10}$ or $37AC_{16}$

Floating-point constants

A floating-point constant is a string of decimal digits, followed by an optional decimal point, fractional portion, and exponent portion. Examples of floating-point numbers include:

1.75e–10	represented internally as 2202 $629A_{16}$
4	represented internally as 0200 0000_{16}
–3.5	represented internally as 01A0 0000_{16}
3.2e5	represented internally as 12E3 $C000_{16}$

A floating-point constant can be preceded with a + or – sign.

Character constants

A character constant is a single character enclosed in *single* quotes. The characters are represented as 8-bit ASCII characters. Examples of valid character constants include:

'ab'	represented internally as 0000 0061_{16}
'C'	represented internally as 0000 0043_{16}

Note the difference between character *constants* and character *strings*. A character constant represents a simple integer value; a string is a list of characters.

5.3 Character Strings

A character string is a string of characters enclosed in *double* quotes. The maximum length of the string varies and is defined for each directive that requires a character string. Examples of valid character strings include:

"sample program"	defines a 14-character string, *sample program*
"temp.asm"	defines an 8-character string, *temp.asm*

Character strings are used for the following:

- Filenames as in `.copy "filename"`
- Section names as in `.sect "section name"`
- Operand of a .string directive

5.4 Symbols

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

Labels

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

Constants

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

```
         .text               ; initialize PC
K        .set   12           ; constant definition K=12
K*2      .set   24           ; constant definition K*2=24
BIN      .set   01010101b    ; BIN = 055h
max_buf  .set   K*2          ; max_buf = K*2 = 24
         LDI    K, R0        ; loads 12
         LDI    -K, R0       ; loads -12
         LDI    K*2, R0      ; loads 24
         LDI    max_buf,R0   ; loads 24
         LDI    !BIN, R0     ; loads 0AAh
```

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	IF	PC	RS
BK	IOF	R0–R7	SP
DP	IR0	RC	ST
IE	IR1	RE	

5.5 Expression Analyzer

The expression analyzer used in the DSK assembler includes ANSI C math library functions that aid in the generation of tables and constants. These functions eliminate the tedious work of calculating tables and constants before including them in the assembly process. The functions are shown in Table 5–2.

Note:

If any of these functions are used, a post-assembly warning is generated to remind you that these functions are not supported by the TMS320 floating-point code generation COFF tools. If you want to use these functions with the COFF toolset, then extract the resulting hexadecimal values from the DSK listing file.

Table 5–2. ANSI C Math Library Functions Supported by the DSK Assembler

Function	Description
long abs(long);	Absolute value
long labs(long);	Absolute value
double fabs(double);	Floating-point absolute
double cos(double);	Cosine
double acos(double);	Arc cosine
double cosh(double);	Hyperbolic cosine
double sin(double);	Sine
double asin(double);	Arc sine
double sinh(double);	Hyperbolic sine
double tan(double);	Tangent
double atan(double);	Arc tangent
double tanh(double);	Hyperbolic tangent
long ceil(long);	Ceiling operator
double floor(double);	Floor operator
double exp(double);	Natural exponent (e) raised to the power of a value
double log(double);	Natural logarithm (ln)

Table 5–2. ANSI C Math Library Functions Supported by the DSK Assembler (Continued)

Function	Description
double log10(double);	Logarithm (based–10)
double pow10(double);	10 raised to the power of a value
double sqrt(double);	Square root
double log2(double);	Logarithm (based–2)
double pow(double,double);	First value raised to the power of the second value
long br(long, long);	Align the first value to the next address located by raising the second value to the power of 2
long circ(long,long);	Align the first value to the next address located by raising the second value to the power of 2

A table of values can be generated using certain assembler directives. To generate a table of values use the **.loop**/**.endloop** directives and the math library functions listed in Table 5–2. For example, to create the twiddle table for an FFT, use the following directives:

```
TWlength  .set   16                               ; Table size is 16
          .brstart ”TwiddleTable”,2*TWlength      ; Align to valid br-address
TWstart:                                          ; create label OUTside loop
          .loop  TWlength                         ; 16 pairs of complex numbers
          .float sin(($-TWStart)*2*pi/TWlength)   ; sin(n*pi/N)
          .float cos(($-TWStart)*2*pi/TWlength)   ; cos(n*pi/N)
          .endloop
```

Table 5–3 shows the operators recognized by the DSK assembler.

Table 5–3. Operators Used in Expressions

Operator	Description	Operator	Description
+	Addition	!=	Not equal
-	Subtraction	=	Equal to
*	Multiplication	==	Equal to
/	Division	&	Logical AND
%	Modulo Division	|	Logical OR
>	Greater than	^	Logical XOR
>=	Greater than or equal to	~	Bitwise negation (1s complement)
<	Less than	!	Logical NOT. If expression = 0 then 1 is returned, else 0 is returned.
<=	Less than or equal to	<<	Shift left
<>	Not equal	>>	Shift right

5.6 Assembling Your Program

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

dsk3a [*filenames*] [*options*]

dsk3a is the command that invokes the assembler.

filenames are one or more assembly language source files. Filenames are not case-sensitive. If you do not specify an extension, the assembler assumes the default extension *.asm*.

options affect the way the assembler processes input files.

You can specify options and filenames in any order on the command line.

Table 5–4 lists the assembler options; the following subsections describe the options.

Table 5–4. Summary of Assembler Options

Option	Description
E*xxx*	Stops assembling after *xxx* error messages occur (5 is the default)
Q	Suppresses the banner and all progress information (quiet)
W*xxx*	Stops assembling after *xxx* warning messages occur

5.7 Placing Code Sections in Memory Locations

The assembly source contains several sections that need to be placed in ‘C31 memory locations. Since the DSK assembler includes several new directives that control the starting address of the sections, a linker is not needed.

In the following code example, an output section named Mysect is placed beginning at address 000x80 9800. The entry (execution start) point is then defined at the label START. Next, a simple code loop that increments R0 is placed into the current section.

```
           .start  ”Mysect”,0x809800      ; Mysect begins at 0x809800
           .sect   ”Mysect”               ; Assemble code into Mysect
           .entry  START                  ; Execution START point
START      LDI     0,R0                   ; Initialize R0=0
LOOP       ADDI    1,R0                   ; Increment R0
           B       LOOP                   ; Do it again
```

To place two sections of code that leave a hole of unused memory, look at the following code. The first section, Mysect, which starts at location 0x0080 9800, is followed by a second section, jumpback, which starts at location 0x0080 9900.

```
           .start  ”Mysect”,0x809800   ; Mysect begins at 0x809800
           .sect   ”Mysect”            ; Assemble code into Mysect
           .entry  START               ; Execution START point
START      LDI     0,R0                ; Initialize R0=0
LOOP       ADDI    1,R0                ; Increment R0
           B       JUMP1
           ;------------------------------
           .start  ”jumpback”,0x809900 ; jumpback begins at 0x809900
           .sect   ”jumpback”          ; Assemble code into jumpback
JUMP1      ADDI    1,R0                ; Increment R0
           B   JUMP2
           ;------------------------------
           .sect   ”Mysect”            ; Add more code to Mysect
JUMP2      ADDI    1,R0                ; Increment R0
           B   LOOP                    ; Finish LOOP
```

To simulate a linker command file such as the one used in the TMS320 code generation tools, you can use a single file to control the starting address of all sections and then use the **.include** directive to append all assembly source files. For example consider the following build file where three source files are appended to each other using a common block statement for several .start directives.

```
;BUILD.ASM
;---------
    .start     ”.text”,0x809800   ; Initialize start address for
                                  ;  each section
    .start     ”.data”,0x809C00   ;
    .start     ”sect1”,0x809900   ;
    .start     ”sect2”,0x809A00   ;
    .include   ”FILE1.ASM”        ; Include source files
    .include   ”FILE2.ASM”        ;
    .include   ”FILE3.ASM”        ;
```

Chapter 6

Assembler Directives

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

- Assemble code and data into specified sections
- Reserve space in memory for uninitialized variables
- Initialize memory
- Assemble conditional blocks

Topic **Page**

6.1	Using the DSK Assembler Directives	6-2
6.2	Directives That Define Sections	6-5
6.3	Directives That Initialize Constants	6-8
6.4	Directives That Reference Other Files	6-9
6.5	Directives That Enable Conditional Assembly	6-10
6.6	Directives That Align the Section Program Counter	6-11
6.7	Directives That Define Symbols at Assembly Time	6-11
6.8	Miscellaneous Directives	6-12
6.9	Directives Reference	6-13

6.1 Using the DSK Assembler Directives

Table 6–1 summarizes the assembler directives. Note that all source statements that contain a directive may have a label and a comment. To improve readability, they are not shown as part of the directive syntax.

Table 6–1. Assembler Directives Summary

(a) Directives that define sections

Mnemonic and Syntax	Description	Page
.data	Assemble source code into data memory	6-18
.sect ”*section name*”	Assemble source code into a named (initialized) section	6-27
.text	Assemble source code into program memory	6-32

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

Mnemonic and Syntax	Description	Page
.byte $value_1$ [,..., $value_n$]	Initialize one or more 8-bit integers	6-16
.fill *size in words*	Reserve *size* words in the current section; note that a label points to the beginning of the reserved space	6-29
.float *expression*	Initialize a 32-bit TMS320C3x floating-point constant	6-21
.float16 *expression*	Initialize a 16-bit TMS320C3x floating-point constant	6-21
.float8 *expression*	Initialize an 8-bit TMS320C3x floating-point constant	6-21
.ieee *expression*	Initialize one or more 32-bit, IEEE single-precision, floating-point constants	6-22
.int $value_1$ [,..., $value_n$]	Initialize one or more 16-bit integers	6-16
.long $value_1$ [**,** ... **,** $value_n$]	Initialize one or more 32-bit integers	6-16
.pfloat16	Initialize 16-bit TMS320C3x floating-point constants into a single word	
.pfloat8	Initialize 8-bit TMS320C3x floating-point constants into a single word	6-21
.q*xx* $value_1$ [,..., $value_n$]	Initialize a 16-bit, signed 2s-complement integer, whose decimal point is displaced *xx* places from the LSB	6-25
.space *size in words*	Reserve *size* words in the current section; note that a label points to the beginning of the reserved space	6-29

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

Mnemonic and Syntax	Description	Page
.string *"$string_1$"* [,..., *"$string_n$"*]	Initialize one or more text strings	6-31
.word *$value_1$* [, ... , *$value_n$*]	Initialize one or more 32-bit integers	6-16

(c) Directives that reference other files

Mnemonic and Syntax	Description	Page
.copy ["]*filename*["]	Include source statements from another file	6-17
.include ["]*filename*["]	Include source statements from another file	6-17

(d) Directives that enable conditional assembly

Mnemonic and Syntax	Description	Page
.else	Optional conditional assembly	6-23
.endif	End conditional assembly	6-23
.if *well-defined expression*	Begin conditional assembly	6-23
.loop [*well-defined expression*]	Begin repeatable assembly of a code block; the loop count is determined by the w*ell-defined expression*.	6-24
.endloop	End .loop code block	6-24

(e) Directives that modify the section program counter (SPC)

Mnemonic and Syntax	Description	Page
.align [*size in bytes*]	Align the SPC on a boundary specified by *size in bytes*, which must be a power of 2; default to byte boundary	6-14
.entry [*address*]	Initialize the starting address of the SPC when loading a file	6-20

(f) Directives that define symbols at assembly time

Mnemonic and Syntax	Description	Page
.set *value*	Equate a value with a local symbol	6-28
.sdef *value*	Equate a value with a local symbol multiple times	6-26

(g) Miscellaneous Directives

Mnemonic and Syntax	Description	Page
.brstart *"section name", n*	Align the named section to the next 2n address boundary.	6-15
.end	Program end	6-19
.start *"section name", address*	Links the named section to start assembling at the location *address*.	6-30

6.2 Directives That Define Sections

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

- The **.data** directive identifies portions of code to be placed in data memory. Data memory usually contains initialized data.
- The **.sect** directive defines an initialized named section and associates subsequent code or data with that section. A section defined with .sect can contain code or data.
- The **.text** directive identifies portions of code in the .text section. The .text section usually contains executable code.

Example 6–1 shows how you can use sections directives to associate code and data with the proper sections. This is an output listing; column 1 shows, the SPC value and column 2 shows the memory contents, if affected by the previous line, or a comment. (Each section has a section program counter (SPC). The .start directive for a section determines that section's initial SPC value. When you resume assembling into a section, its SPC resumes counting as if there had been no intervening code.

After the code in Example 6–1 is assembled, the sections contain:

.text	Bytes with the values 1, 2, 3, 4, 5, and 6
.data	Bytes with the values 9, 10, 11, and 12
mysect	Bytes with the values 21, 22, 23, 24

Note:

The .text and .data directives are short hand representations of .sect statements for that section name.

`.text`	is equivalent to	`.sect ".text"`
.data	is equivalent to	`.sect ".data"`

Example 6–1.Sections Directives

```
0x00809800 directive          .start ”.text”,0x809800
0x00809800 directive          .start ”.data”,0x809900
0x00809800 directive          .start ”mysect”,0x809a00
0x00809800 nocode
0x00809800 nocode     ; Start assembling into .text
0x00809800 nocode
0x00809800 directive          .text
0x00809800 directive          .byte  1,2
0x00909800 0x00000001 <byte>
0x00809801 0x00000002 <byte>
0x00809802 directive          .byte  3,4
0x00809802 0x00000003 <byte>
0x00809803 0x00000004 <byte>
0x00809804 nocode
0x00809804 nocode     ; Start assembling into .data
0x00809804 nocode
0x00809804 directive          .data
0x00809900 directive          .byte 9,10
0x00809900 0x00000009 <byte>
0x00809901 0x0000000a <byte>
0x00809902 directive          .byte 11,12
0x00809902 0x0000000b <byte>
0x00809903 0x0000000c <byte>
0x00809904 nocode
0x00809904 nocode     ; Resume assembling into .text
0x00809904 directive          .text
0x00809804 directive          .byte 5,6
0x00809804 0x00000005 <byte>
0x00809805 0x00000006 <byte>
0x00809806 nocode
0x00809806 nocode     ; Start assembling into mysect
0x00809806 nocode
0x00809806 directive          .sect ”mysect”
0x00809a00 nocode
0x00809a00 directive          .byte 21,22
0x00809a01 0x00000015 <byte>
0x00809a01 0x00000016 <byte>
0x00809a02 directive          .byte 23,24
0x00809a02 0x00000017 <byte>
0x00809a02 0x00000018 <byte>
0x00809a04 nocode
0x00809a04 nocode
0x00809a04 nocode
```

Example 6–1.Sections Directives (Continued)

```
>>>>
>>>> PASS 2  Complete
>>>> Errors: 0  Warnings: 0
>>>>
>>>> ENTRY 0x00809800
>>>>
>>>> Symbol reference table           Type Addressable
>>>> ref    Default-Start 0x00809800  1    1
>>>> ref                  0x00000001  1    2
>>>> ref            .text 0x00809800  1    1
>>>> ref            .data 0x00809900  1    1
>>>> ref           mysect 0x00809a00  1    1
>>>> ref                  0x00000001  1    2
>>>>
>>>>        Output section start       end         length
>>>> sect    Default_Start 0x00809800  0x00809800
0x00000000
>>>> sect            .text 0x00809800  0x00809806
0x00000006
>>>> sect            .data 0x00809900  0x00809904
0x00000004
>>>> sect           mysect 0x00809a00  0x00809a04
0x00000004
>>>>
>>>>
>>>> END DSK
```

6.3 Directives That Initialize Constants

Several directives assemble values for the current section.

- The **.byte** directive places one or more 8-bit values into consecutive words in the current section. A byte in this case uses all 32 bits of the word placing 0s into the upper 24 bits.
- The **.fill** directive reserves a specified number of words in the current section with a value. The assembler advances the SPC and skips the reserved words. When you use a label with .fill, it points to the *first* word of the reserved block.
- The **.float** directive converts an expression value into a 32-bit TMS320C3x floating-point constant. This format has an 8-bit exponent and a 24-bit mantissa.
- The **.float16** directive converts an expression value into a 16-bit TMS320C3x floating-point constant. This format has an 8-bit exponent and an 8-bit mantissa. The format is identical to that used by the .sfloat directive of the TMS320C32. The upper 16 bits are not used and are filled with 0s.
- The **.float8** directive converts an expression value into an 8-bit TMS320C3x floating-point constant. This format has a 4-bit exponent and a 4-bit mantissa. This format can be used for a quick logarithm approximation. The upper 24 bits are not used and are filled with 0s.
- The **.ieee** directive calculates the 32-bit IEEE floating-point representation of a single precision floating-point value.
- The **.int** directive places one or more 16-bit values into consecutive words in the current section. The upper 16 bits are not used and are filled with 0s.
- The **.long** directive places one or more 32-bit values into consecutive bytes in the current section.
- The **.pfloat16** directive converts an expression value into a 16-bit floating-point constant. The values are packed into consecutive fields of memory.
- The **.pfloat8** directive converts an expression value into an 8-bit floating-point constant. The values are packed into consecutive fields of memory.
- The **.q*xx*** directive places one or more 16-bit, signed 2s-complement values into consecutive words in the current section. Note that the decimal point is displaced *xx* places from the LSB (least significant bits.)

- The **.space** directive reserves a specified number of bits in the current section. The assembler advances the SPC and skips the reserved words. When you use a label with .space, it points to the *first* word of the reserved block.
- The **.string** directive places 8-bit characters from one or more character strings into the current section.
- The **.word** directive places one or more 32-bit values into consecutive bytes in the current section.

6.4 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.

6.5 Directives That Enable 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**/**.else**/**.endif** directives tell the assembler to assemble a block of code according to a true or false evaluation of an expression. Note that you cannot nest if statements.

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

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

.loop *well-defined expression*	marks the beginning a repeatable block of code. The optional expression evaluates to the loop count.
.endloop	marks the end of a repeatable block.

6.6 Directives That Align the Section Program Counter

These directives affect the section program counter (SPC).

- The **.align** directive aligns the SPC at a 1-byte to 32K-byte boundary. This ensures that the code following the directive begins on the byte value that you specify. If the SPC is already aligned at the selected boundary, it is not incremented.
- The **.entry** directive identifies the starting address of the section program counter. By default, the current address is used, or, you can specify an optional address.

6.7 Directives That Define Symbols at Assembly Time

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

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

```
bval   .set   0100h
       .byte  bval
       b      bval
```

- The **.sdef** directive equates meaningful symbol names to constant values or strings; the symbol name can be redefined.

6.8 Miscellaneous Directives

These directives enable miscellaneous functions or features:

- The **.brstart** directive aligns the named section to the next 2^n address boundary following the current section.
- The **.end** directive terminates assembly. It should be the last source statement of a program. This directive has the same effect as an end-of-file.
- The **.start**. directive links the named section to start assembling at the location address. This effectively gives the DSK assembler the functionality of a linker.

6.9 Directives Reference

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

Directive	Page	Directive	Page
.align	6-14	.include	6-17
.brstart	6-15	.int	6-16
.byte	6-16	.long	6-16
.copy	6-17	.loop	6-24
.data	6-18	.pfloat16	6-21
.else	6-23	.pfloat8	6-21
.end	6-19	.qxx	6-25
.endif	6-23	.sdef	6-26
.endloop	6-24	.sect	6-27
.entry	6-20	.set	6-28
.fill	6-29	.space	6-29
.float	6-21	.start	6-30
.float8	6-21	.string	6-31
.float16	6-21	.text	6-32
.ieee	6-22	.word	6-16
.if	6-23		

Syntax **.align**

Description The **.align** directive aligns the current section to a 32-word boundary, filling the hole with NOPs. If the hole is greater than 2 words, .align places a branch to the newly-aligned address. This directive is useful for placing critical code blocks on the boundaries that best use the cache resources of the ’C3x architecture.

Example Here is an example of the .align directive.

```
;
; Slightly modified FIR filter example from C3x Users Guide
;-----------------------------------------------------------
          .start ”ISR”,0x809808          ; Create an output section which is
          .sect  ”ISR”                   ; not on a 32-word boundary for demo
          .align                         ;
FIRLENG   .set   64                      ; Size of FIR filter
Critical  ldp    @FIRCOEF                ;
          ldi    @FIRCOEF,AR0            ; AR0=address of h(N-1)
          ldi    @FIRDATA,AR1            ; AR1=address of x(n-(N-1))
          mpyf3  *AR0++(1),*AR1++(1)%,R1 ;
          ldf    0.0,R2                  ;
          ldi    FIRLENG-2,RC            ; Be sure to unroll length by 2
          rptb   FIR                     ; Begin block repeat
          mpyf3  *AR0++(1),*AR1++(1)%,R1 ;
FIR   ||  addf3  R0,R1,R2                ;
          b      $                       ; Done, result is in R2
FIRCOEF   .word  0x809900                ; Address for coefficient storage
FIRDATA   .word  0x809A00                ; Address for input data storage
```

Syntax

.brstart "*section name*", *n*

Description

The **.brstart** directive aligns the *section name* to the next 2^n address boundary immediately following the current section. This directive aligns data buffers in order to use the 'C3x circular and bit-reversed addressing modes. Another method for creating a section whose start is bit-reversed, is to use the br() function within the .start directive's address field.

Example

Here is an example of the .brstart directive.

```
.word     $                ; The present address is
.brstart  "Twiddle", 128   ; Create a new section on a new 128 word boundary
.word     $                ; The new address is
```

Syntax

.byte $value_1$ [,..., $value_n$]
.int $value_1$ [,..., $value_n$]
.long $value_1$ [, ... , $value_n$]
.**word** $value_1$ [, ... , $value_n$]

Description

These directives place one or more values into the current section.

- The **.byte** directive places 8-bit values into consecutive words in the current section. The *value* must be an expression that evaluates to a number within –128 and 127. The upper 24 bits are 0.
- The **.int** directive places 16-bit values into consecutive words in the current section. The *value* must be an expression that evaluates to a number within the range of –32768 and 32767. The upper 16 bits are always 0.
- The **.long** and **.word** directives place 32-bit values into consecutive words in the current section. The *value* is an expression that the assembler evaluates and treats as a 32-bit signed number.

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

Example 1

Here is an example of these directives.

```
.word     'A', 'B', 'C', 1, 0x1234, 0320C31h
.int      111b, 1<<4
.long     0x87654321, 1<<31
.byte     0x20, 'A', 'B', 'C'
.hword    32765,l -32768, -2, 2
```

Syntax

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

Description

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

1) Stops assembling statements in the current source file
2) Assembles the statements in the copied/included file
3) Resumes assembling statements in the main source file, starting with the statement that follows the .copy or .include directive

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

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

Example

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

Source file: (source .asm)

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

First copy file: (byte.asm)

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

Second copy file: (word.asm)

```
                                ; Filename: word.asm
.word    oabcdh, 56             ; Filename: word.asm
                                ; Filename: word.asm
```

Syntax

.data

Description

The **.data** directive tells the assembler to begin assembling source code into data memory. The .data section normally contains tables of data or preinitialized variables.

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

Example

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

```
      .start       “.data”, 0x809900
      .entry       BEGIN
BEGIN ldi          0, R0        ; Initialize R0 and R1
      ldi          1, R1
      .data
value .int   0, 1, 2, 3, 4, 5   ; Integer values
```

Syntax

.end

Description

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

Example

This example shows how the .end directive terminates assembly.

```
ldi   1,R1  ; Assemble this code
mpyi  5,R1  ;
.end        ; Stop assembler
subi  2,R1  ; does not assemble
```

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

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

Example Here is an example of the .entry directive.

```
          .start ”code”,0x809800 ; Create a named section to assemble to
          .sect  ”code”          ;   use the new section
          .entry BEGIN           ; Start program at BEGIN
BEGIN:    ldi   80h,AR0          ; Initialize ARx pointers to RAM0
          lsh   16,AR0           ;
          ldi   AR0,AR1          ;
          ldi   0,R3             ; R3 is used as loop counter
LAB0      ldi   *+AR0(0),R0      ; Both labels resolve to the same address
LAB1: ||  ldi   *+AR1(1),R1      ; Colon ’:’ is recongized as a WS character
          ;----------------------------------;
          ; R0 contains the opcode at BEGIN   ;
          ; R1 contains the opcode at BEGIN+1 ;
          ;----------------------------------;
count:    addi  1,R3             ; Add 1 to count
          b     count            ; Wait in count loop forever
```

Syntax

.float *value*$_1$ [,..., *value*$_n$]
.float16 *value*$_1$ [,..., *value*$_n$]
.float8 *value*$_1$ [,..., *value*$_n$]
.pfloat16 *value*$_1$ [,..., *value*$_n$]
.pfloat8 *value*$_1$ [,..., *value*$_n$]

Description

These directive convert one or more values into TMS320C3x floating-point constants.

- The **.float** directive converts a *value* into a 32-bit TMS320C3x floating-point constant. This format has an 8-bit exponent and a 24-bit mantissa.
- The **.float16** directive converts a *value* into a 16-bit TMS320C3x floating-point constant. This format has an 8-bit exponent and an 8-bit mantissa.
- The **.float8** directive converts a *value* into an 8-bit TMS320C3x floating-point constant. This format has a 4-bit exponent and a 4-bit mantissa. When properly scaled, this format can be used for quick logarithm approximations.
- The **.pfloat16** directive converts a *value* into a 16-bit floating-point constant. The values are packed into consecutive fields of memory.
- The **.pfloat8** directive converts a *value* into an 8-bit floating-point constant. The values are packed into consecutive fields of memory.

The *value* is a required parameter; it is an expression that is evaluated and placed in the constant. The value must be absolute.

Note that the 'C31 expects floating-point numbers to have the 32-bit format.

Example

Here is an example of these directives.

```
PI  .set      3.1415926                    ;.set remembers PI is float
    .float    -10/3, -0.1, 0, 0.1, PI,2*PI ;Some easy to compare values
    .float8   -10/3, -0.1, 0, 0.1, PI,2*PI ;
    .pfloat8  -10/3, -0.1, 0, 0.1, PI,2*PI ;
    .float16  -10/3, -0.1, 0, 0.1, PI,2*PI ;
    .pfloat16 -10/3, -0.1, 0, 0.1, PI,2*PI ;
    .ieee     -10/3, -0.1, 0, 0.1, PI,2*PI ;
```

Syntax

.ieee *expression*

Description

The **.ieee** directive places the IEEE single-precision floating-point representation of a single floating-point constant into three bytes in the current section.

The *expression* is a required parameter; it is an expression that must evaluate to a floating-point constant. Each constant is converted to a floating-point value in IEEE single-precision 32-bit format.

Example

Here is an example of the .ieee directive.

```
.ieee     -10/3, -0.1, 0, 0.1, PI, 2*PI   ;Some values
```

Syntax

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

Description

Three directives provide conditional assembly:

- The **.if** directive marks the beginning of a conditional block. The *expression* is a required parameter.
  - If the expression evaluates to *true* (nonzero), the assembler assembles the code that follows it (up to an .else, or an .endif).
  - If the expression evaluates to *false* (0), the assembler assembles code that follows an .else (if present), or an .endif.
- The **.else** directive identifies a block of code that the assembler assembles when the .if expression is false (0). This directive is optional in the conditional block; if an expression is false and there is no .else statement, the assembler continues with the code that follows the .endif.
- The **.endif** directive terminates a conditional block.

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

Example

Here is an example of conditional assembly:

```
TRUE   .set   1
FALSE  .set   0

       .if TRUE      ;
       nop           ; Assembles 'nop' since TRUE
       .else         ;
       B   $         ; Never assembles
       .endif        ;
```

Syntax

.loop *well-defined expression*

.endloop

Description

Two 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 246.
- The **.endloop** directive terminates a repeatable block of code; it executes when the number of loops performed equals the loop count given by .loop.

Example

This example shows the .loop directive.

```
;==================================================
; Create an FFT Twiddle table
;==================================================
         .start  ”TABLES”,0x809A00
         .sect   ”TABLES”
pi       .set    3.1415926
N        .set    4
         ;-------------------------;
TR       ; REAL twiddles
         ;-------------------------;
         .loop   N/2
         .float    cos(($-TR)*pi/N);
         .endloop
         ;-------------------------;
TI       ; IMAG twiddles           ;
         ;-------------------------;
         .loop   N/2
         .float -1*sin(($-TI)*pi/N)
         .endloop
```

Syntax **.q*xx*** *value*$_1$ [,..., *value*$_n$]

Description The **.q*xx*** directive generates signed, 2s-complement fractional integers and long integers whose decimal point is displaced *xx* places from the LSB.

Example Here's an example of the .qxx directive. The value of *xx* can be either positive or negative.

```
.q0    3.1415926     ; All upper 32 bits are integers
.q1    3.1415926     ; One fractional bit (left shift 1
.q2    3.1415926     ; Two fractional bits (left shift 2)
.q16   3.1415926     ; Upper 16 are whole integers,
                     ;   lower 16 are fractional
```

Syntax

symbol **.sdef** *value*

Description

The **.sdef** directive functions in the same manner as the .set directive; however, .sdef can redefine the symbol name multiple times without generating an error. All instances of .sdef symbols are stripped from the symbol table at the end of pass 1 analysis. When used with the .if directive, .sdef can conditionally assemble included blocks of code. This is useful for turning on and off included library functions.

- The *symbol* must appear in the label field.
- The *value* must be a well-defined expression; that is, all symbols in the expression must be previously defined in the current source module.

Example

This shows how symbols can be assigned with .sdef.

```
VarA   .set   15           ;
VarB   .sdef  0xAAAA       ;
       .word  VarA, VarB   ;
VarB   .sdef  0x5555       ;
       .word  VarA, VarB   ; Note the VarB value change
```

Syntax

.sect "*section name*"

Description

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

The *section name* identifies the section. The section name is significant to 80 characters and must be enclosed in double quotes.

Example

Here's an example of the .sect directive.

```
.start  ”Mysect_1”,0x809800 ; Create two output sections
.start  ”Mysect_2”,0x809880 ; at different addresses

.sect   ”Mysect_1”          ; Begin assembling into Mysect_1
.word   $,1,1,1             ; $ gives present address
.sect   ”Mysect_2”          ; Begin assembling into Mysect_2
.word   $,2,2,2             ;
.sect   ”Mysect_1”          ; Go back to assembling into Mysect_1
.word   $,1,1,1             ;
```

Syntax

symbol **.set** *value*

Description

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

- The *symbol* must appear in the label field.
- The *value* must be a well-defined expression; that is, all symbols in the expression must be previously defined in the current source module.

Example

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

```
TA    .set  1
TB    .set  5
      ldi    *AR0++(TA),R0
      ldi    *AR0++(TB),R0
```

Syntax

.space *size in words*

.fill *size in words*

Description

Two directives reserve space in the current section.

- The **.space** directive reserves *size* number of words in the current section and fills them with 0s. The SPC is incremented to point to the word following the reserved space.
- The **.fill** directive reserves *size* number of words in the current section and fills them with *value*. The value must be an absolute value. The SPC is incremented to point to the word following the reserved space.

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

Example

This example shows how the .space and .fill directives reserve memory.

```
   .space  12        ; Fill 12 locations with the value 0x0
 .  fill   3,0x5555  ; Fill three words with 0x5555

   .start  ”Mysect”,0x809800 ; Initialize start of Mysect
   .sect   ”Mysect”          ;

   .text
   .data
```

Syntax

.start "*section name*", *address*

Description

The **.start** directive links the *section name* to start at location *address*. This directive effectively gives the DSK assembler the same functionality as a linker command file when used only to create runtime executable modules. For the specified section to have a valid starting address, the .start statement for the section must precede the .text, .data, or .sect directive that defines the section name. Note that by using an include file with an imbedded .if/.sdef/.endif, the .start directive can effectively be used in place of the linker.

Example

Here is an example of the .start directive.

```
       .entry START
       .start ”MAIN”,0x809800 ; Create an output sections
       .sect  ”MAIN”          ; Begin assembling into MAIN
LOOP:  addi   1,R0            ; Top of loop
       addi   1,R1
START: ldi    0,R0            ; Initialize R0,R1
       ldi    0,R1
       b LOOP                 ; Go to top of loop
```

Syntax

.string "*string*$_1$" [,..., "*string*$_n$"]

Description

The **.string** directive places one or more 8-bit character strings into consecutive bytes of the current section.

The character string must be enclosed in double quotes. Each character in a string represents a separate value.

The .string directive places the 8-bit values into memory in a packed form in the order they are encountered. If a word is not filled, the remaining bits are filled with 0s.

Example

This example shows several 8-bit values placed into consecutive bytes in memory. The label Str_3 has the value 0h, which is the location of the first initialized byte.

```
Str_3: .string   "ABCD"
       .string   51h, 52h, 53h, 54h
       .string   "Hoston"
       .string   36+12
```

Syntax **.text**

Description The **.text** directive tells the assembler to begin assembling into the .text section. The .text section usually contains executable code. The section program counter (SPC) is set to 0, if nothing has been assembled into the .text section. If code has already been assembled into the .text section, the SPC is restored to its previous value in the section.

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

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

```
      .start       ".text", 0x809800
      .entry       START
START ldi          0, R0        ; Initialize R0 and R1
      ldi          1, R1
      .text
value .int   0, 1, 2, 3, 4, 5   ; integer values
```

Chapter 7

Using the DSK Debugger

This chapter tells you how to invoke the DSK debugger and use its function keys and commands.

Topic **Page**

Topic		Page
7.1	Invoking the Debugger	7-2
7.2	Understanding the Debugger Windows	7-4
7.3	Using the Help Menu	7-8
7.4	Using Software Breakpoints	7-9
7.5	Debugger Commands	7-10
7.6	Quick Reference Guide	7-13

7.1 Invoking the Debugger

Here's the command for invoking the debugger:

dsk3d [*options*]

dsk3d is the command that invokes the debugger.

options supply the debugger with additional information.

Table 7–1 lists the debugger options; the following subsections describe the options.

Table 7–1. Summary of Debugger Options

Option	Brief Description
? or HELP	Displays a listing of the available options
AUTO	Automatically detects if the parallel port supports 8- or 4-bit mode
BW=4, Nibble	Selects communication through the parallel port in standard 4-bit unidirectional mode
BW=8, Byte	Selects communication base through the parallel port in 8-bit bidirectional mode
LPTx, LPT=x	Selects a parallel printer port (LPT1 is default)
PORT=0x378	Selects any port address
RESET	Resets (cold boots) the DSK
TEST	Searches automatically through LPT1, LPT2, and LPT3 for the presence of a DSK
T=xx	Adds extra *xx* I/O bus cycles to each transfer for long or noisy cables
WIN=1	Enables Windows Time Slice management
WIN=0	Disables Windows Time Slice management and enables STI/CLI

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

You can display the contents of Table 7–1 on your screen by using the ? or Help option. For example, enter:

```
dsk3d ?
```

Selecting the parallel printer port (LPT = 3 or LPT# option)

The LPT option selects a parallel printer port from the host to communicate with the DSK.

- LPT1 or LPT = 0x378
- LPT2 or LPT = 0x278
- LPT3 or LPT = 0x3BC

Note:

Some EISA machines and IBM PS/2s use a different naming convention for the LPTx.

AT Convention	EISA and PS/2
LPT1	LPT2
LPT2	LPT3
LPT3	LPT1

Select the parallel printer port at a particular address (PORT option)

The port option selects the parallel printer port at the given address. For example:

```
port = 0x378
```

selects the host's parallel port mapped to the address 0x378.

Selecting communication mode (BW option)

Use the bw option to select 8-bit bidirectional or 4-bit unidirectional communication between DSK and host's parallel printer port.

Automatically search for a printer port (TEST option)

Use the test option to systematically search for a parallel port that has a DSK connected. The search commences at LPT1 and ends.

Add extra I/O cycles to each transfer

Use the *t* option to add extra I/O cycles to each transfer if you have very long parallel port cables that have noisy signals.

7.2 Understanding the Debugger Windows

DISASSEMBLY window

The DISASSEMBLY window shows the reverse assembly of memory contents. As shown in Figure 7–1, this window displays several lines of code. Each line shows the instruction address, instruction opcode, label, and instruction mnemonic. The highlighted line corresponds to the next instruction to be executed.

Figure 7–1. DISASSEMBLY Window

Instruction address   Instruction opcode   Label   Instruction mnemonic

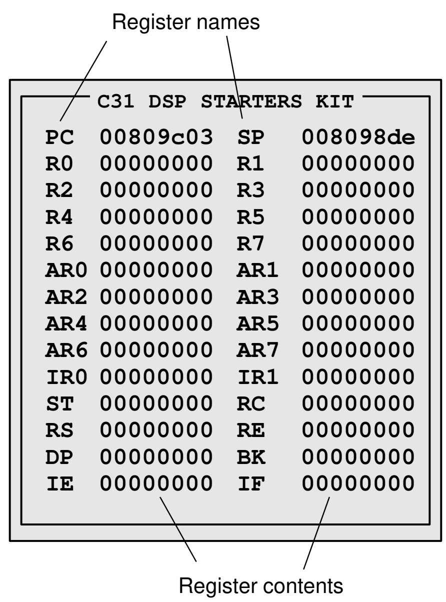
```
                      DISASSEMBLY
809c03  50700080 start LDIU 00080h,DP
809c04  08349c2c       LDI  @09c2cH,SP
809c05  07608000       LDF  0.000000e+00,R0
809c06  c610c1c0       LDI  *AR0,R0 || LDI *AR
809c07  c610c1c0       LDI  *AR0,R0 || LDI *AR
809c08  08600100       LDI  256,R0
809c09  09a09c00       LSH  @09c00H,R0
809c0a  61809c0e       BRD  jump
809c0b  07618000       LDF  0.000000e+00,R1
809c0c  07628000       LDF  0.000000e+00,R2
809c0d  07630000       LDF  1.000000e+00,R3
809c0e  07640000 jump  LDF  1.000000e+00,R4
809c0f  087b0003 loop  LDI  3,RC
809c10  64809c1a       RPTB block
809c11  02640001       ADDI 1,R4
```

To select the DISASSEMBLY window, press ALT D. While in the DISASSEMBLY window, you can use the cursor to select a line and then use a function key to set or clear a breakpoint. Refer to Table 7–10 for more information about function keys.

CPU REGISTER window

The CPU REGISTER window displays the content of all CPU registers as shown in Figure 7–2. The register's contents are normally displayed in hexadecimal format. You can press F3 to display the extended-precision registers in floating-point decimal format. You can press F2 to display the extended-precision registers in 40-bit hexadecimal format.

Figure 7–2. CPU REGISTER Window



To modify the contents of a register, activate the CPU REGISTER window by pressing ALT C. You can type over the highlighted data and press ENTER to accept the changes when you are satisfied with them. Use the following keys to select the data you want to edit:

→ ↑ ↓ ← PAGE UP PAGE DOWN TAB

MEMORY window

The MEMORY window shows the contents of a range of memory as shown in Figure 7–3. The MEMORY window has two parts:

- **Addresses**. The first column of numbers identifies the addresses of the first column of display data. No matter how many columns of data you display, only one address column is displayed. Each address in this column identifies the address of the data immediately to its right.
- **Data**. The remaining columns display values at the listed addresses.

For example, the MEMORY window below has four columns of data, so each new address is incremented by 4. Although the window shows four columns of data, there is still only one column of addresses; address 0x0080 9800 contains 0x0000 0007, address 0x0080 9801 contains 0xFFFF FFFC, address 0x0080 9804 (the first value in the second row) contains 0x0080 982C, address 0x0080 9805 contains 0x0080 9839, etc.

Figure 7–3. MEMORY Window

Address column   Data columns

```
                         MEMORY
 809800   00000007 fffffffc 00809802 00809827
 809804   0080982c 00809839 0080983c 0080983f
 809808   00809843 00809842 00809868 0080989a
 80980c   008098a9 10800000 0f350000 0f300000
 809810   0f200000 0f320000 0f280000 0f290000
 809814   1a770004 6a050006 628098a9 50700080
```

To modify the contents of the MEMORY window, press ALT M to activate the window and then type over the data. To select a cell, you can use the following keys:

→ ↑ ↓ ← PAGE UP PAGE DOWN TAB

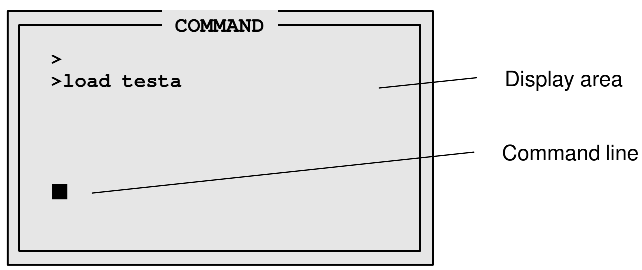
COMMAND window

The COMMAND window provides an area for entering commands, echoing commands, and displaying command output errors and messages. The COMMAND window has two parts:

- **Command line**. This area is where you enter commands. When you want to enter a command, just type – no matter which window is active.
- **Display area**. This area echoes the commands that you enter, shows any output from your commands, and displays debugger error messages.

Figure 7–4 shows the window command line and display area.

Figure 7–4. COMMAND Window



You can use the ↑ and ↓ keys to select a previously entered command from the buffer (a > is used to indicate the buffer). The editing command keys are shown in Table 7–2.

Table 7–2. Editing Command Keys

To do this	Use this command
Move through the command	← →
Toggle the insert and type over mode	INS
Delete the character at the cursor	DEL
Move to the beginning of the line	HOME
Move to the end of the line	END
Clear the command	ESC
Select a command from the buffer	↑ ↓

7.3 Using the Help Menu

You can press the F1 or H key to bring up the Help Window Display shown in Figure 7–5. Choose from the menu selections listed below to find additional information.

Figure 7–5. The Monitor Information Screen

```
KEYBOARD COMMANDS
F1       Help Screen
F2       40-bit hex display
F3       FLOAT display
F4       Source/DASM debug toggle
F5       Run
F6       Display breakpoints
F7       Clear all breakpoints
F8       Singlestep
F9       Toggle DASM window size
F10      Step over function

ALT+D    Selects Disassembly Window
ALT+M    Selects Memory Window



Move Up/Dn/Pup/Pdn — H-Xtra help — S-save help to file
```

To move through the help window, you can use:

- PGUP to move ahead a page
- PGDN to move back a page
- HOME to return to the first page of the help menu
- END to go to the last page of the help menu
- S to save help text to a file
- ESC to exit the help menu and return to the debugger
- H to enter a second help level. The second help level is more hardware-oriented and deals less with debugger-specific commands.

7.4 Using Software Breakpoints

This section describes how to set and clear software breakpoints and how to obtain a listing of all the breakpoints that are set.

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

Setting a software breakpoint

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

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

You can set a software breakpoint by entering the SB command.

sb *addr* If you know the address where you'd like to set a software breakpoint, you can use the SB command. This command is useful because it doesn't require you to search through code to find the desired line. When you enter the SB command, you enter an absolute address (*addr*). (Once you have entered the address, you are asked to choose the line number you want the breakpoint set on.) Note that you cannot set multiple breakpoints at the same statement.

Clearing a software breakpoint

cb *addr* If you'd like to clear a breakpoint, you can use the CB command. You can use the CB command to clear a specific address by entering an absolute address (*addr*) after the command. You can clear all breakpoints by entering the CB command without an address.

Finding the software breakpoints that are set

db Sometimes, you may need to know where software breakpoints are set. The DB command provides an easy way to get a complete listing of all the software breakpoints that are currently set in your program.

7.5 Debugger Commands

The following tables provide a summary of the debugger function keys and commands.

Table 7–3. Command-Line Editing

To do this	Use this command
Move the cursor to the beginning of the command line	HOME
Move the cursor to the end of the command line	END
Delete the character to the left of the cursor	DEL
Delete the character to the right of the cursor	SHIFT END
Move the cursor to the left	←
Move the cursor to the right	→

Table 7–4. Command-Line Buffer Manipulation

To do this	Use this command
Recall the last command typed	PAGE UP or ↑
Recall the first command in the command-line buffer	PAGE DOWN or ↓
Reexecute the last command typed	TAB

Table 7–5. Running Programs

To do this	Use this command
Step through the instructions one at a time (single-step)	SS
Execute *n* instructions	XN *n*
Single-step through the instructions until you reach address *addr*	XG *addr*
Execute the program until a breakpoint is encountered	RUN
Execute the program and ignore breakpoints (run-free)	RUNF

Table 7–6. Displaying and Changing Data

To do this	Use this command
Display the contents of memory starting at address *addr* in the MEMORY window	MEM *addr*
Modify memory at address *addr*	MM *addr*
Fill *leng* locations of memory starting at address *addr* with value *val.* If *val* is expressed in a floating-point format (with a decimal point), it will be converted into a TMS320 floating-point format.	MM *addr leng val*
Display assembly language code starting at address *addr* in the DISASSEMBLY window	DASM *addr*
Display extended-precision registers in 40-bit hexadecimal format in the register window	REG40
Display extended-precision registers in floating-point decimal format in the register window	FLOAT
Modify *reg* register in the CPU REGISTER window with the value from *expression*. For example PC = 0x809800 R0 = 1.34	*reg = expression*

Table 7–7. Managing Breakpoints

To do this	Use this command
Set a breakpoint at address *addr*	SB *addr*
Clear a breakpoint at address *addr*	CB *addr*
Clear all the breakpoints	CB
Display a list of all the breakpoints that are set	DB

Table 7–8. Loading Programs

To do this	Use this command
Load an object file	LOAD *filename*
Load symbols	SLOAD *filename*
Load binary only	BLOAD *filename*
Clear symbols	SCLEAR

Table 7–9. Performing System Tasks

To do this	Use this command
Reset the DSK	RESET
Quit or exit the debugger	QUIT or EXIT
Enter the DOS shell and optionally execute the expression. Enter EXIT to return to debugger	DOS (*expression to Run*)
Enter the DOS shell and execute the editor to edit *filename*. (If no filename is given, the name of the presently loaded file is used).	EDIT *filename*
Enter the DOS shell and execute the DSK assembler to assemble *file*	dsk3a *filename.asm*

7.6 Quick Reference Guide

The following tables provide a quick-reference guide of the function key definitions.

Table 7–10. Function Key Shortcuts for Command Window Active

Function Key	Description
F1	Displays a list of commands
F2	Displays extended-precision registers in 40-bit hexadecimal format
F3	Displays extended-precision registers in floating-point decimal format
F4	Toggles between displaying the source file and the memory disassembly.
F5	Executes your program to the next breakpoint
F6	Displays all breakpoints
F7	Clears all breakpoints
F8	Single-steps your program
F9	Toggles the DISASSEMBLY window size
F10	Single-steps your program and steps past calls
ALT D	Selects the DISASSEMBLY window
ALT M	Selects the MEMORY window
ALT C	Selects the CPU REGISTER window
ESC	Exits the active window

Table 7–11. Function Key Shortcuts for CPU Window Active

Function Key	Description
F1	This help screen
ESC	Exit CPU window
HOME	Move to top
END	Move to bottom
↑ or ↓	Move cell vertical
TAB	Move cell horizontal

Table 7–12. Function Key Shortcuts for Memory Window Active

Function Key	Description
F1	This help screen
F9	Toggle window size
ESC	Exit memory window
HOME	Move to top
END	Move to bottom
PAGE UP or PAGE DOWN	Move by page up/down
↑ or ↓	Move cell vertical
TAB	Move cell horizontal

Table 7–13. Function Key Shortcuts for Disassembler Window Active

Function Key	Description
F1	Help screen
F2	Set breakpoint at cursor
F3	Clear breakpoint at cursor
F4	Run to cursor
F5	Run
F6	Display breakpoints
F7	Clears all breakpoints
F8	Single-steps your program
F9	Grow window
F10	Step over
SHIFT F9	Selects the DISASSEMBLY window
ESC or ENTER	Escape

Appendix A

Communications Kernel Source Code

This appendix contains the source code for the TMS320C3x DSK communications kernel.

```
;-------------------------------------------------------------------;
; TMS320C3x DSK COMMUNICATIONS AND DEBUG MONITOR KERNAL             ;
; Texas Instruments Incorporated                                    ;
; (C) 1995,1996                                                     ;
;-------------------------------------------------------------------;
           .start ”vectors”,0x809FC1
           .start ”kernel” ,vectors–0xAB ; Use size report from DSK3A
           .start ”sstack” ,0x809F00     ; output to pack to end of RAM
           .entry START
;===================================================================;
; COMMUNICATION MONITOR START                                       ;
;                                                                   ;
; STACK SPACE                                                       ;
; -----------                                                       ;
;   A section of unoccupied free memory of STACKSIZE size words just  ;
; below the kernel is used on startup for initialization and stack  ;
; space.  If more (or less) stack space is required, a new stack    ;
; pointer value can be initialized within the users applications code ;
; to any location, or by re–assembling this code with a new STACKSIZE ;
;                                                                   ;
; When initialization is complete, the startup stub can be safely   ;
; overwritten since it is no longer needed.  In this case the startup ;
; stub is placed after the stack.  Another ’safe’ location would be  ;
; a section of memory which is used for I/O or uninitialized data.  ;
;                                                                   ;
; This section of code also initializes the timers which are used by  ;
; the PAL to create the PWM signal which drives the LED.  The rate   ;
; at which the LED changes color is F0–F1 where F0 and F0 are the two ;
; timer output frequencies. (See the Users Guide                    ;
;===================================================================;
        .sect   ”sstack”
stack:  .word   stack–1          ; start of kernel stack
MMRBASE .word   0x00808000       ;
PRD0    .word   0x0000A000       ;
PRD1    .word   0x0000A060       ;
TSTART  .word   0x000003C3       ;
START   ldp     @START           ; Set up stack and other params
        ldi     @stack,SP        ;
        ;=======================
        ldi     @MMRBASE,AR0     ; Init timers for slow PWM modulation
        ldi     3,R0             ; HALT timers
        sti     R0,*+AR0(0x20)   ;
        sti     R0,*+AR0(0x30)   ;
        sti     R0,*+AR0(0x24)   ; Init count registers
        sti     R0,*+AR0(0x34)   ;
        ldi     @PRD0,R0         ; Init periods
        sti     R0,*+AR0(0x28)   ;
        ldi     @PRD1,R0         ;
        sti     R0,*+AR0(0x38)   ;
        ldi     @TSTART,R0       ; Start timers
        sti     R0,*+AR0(0x20)   ;
        sti     R0,*+AR0(0x30)   ;
        b       spin0            ;
```

```
;=========================================================================;
; DEBUGGER COMMANDS                                                       ;
;   The debugger commands are assembled into the lowest available kernel ;
;   memory.  If an application were to overgow this section the debugger ;
;   functions would be corrupted, but the application would continue to  ;
;   run so long as the debugger functions were not used.                  ;
;=========================================================================;
; XSTEP/XRUNF                                                             ;
;                                                                         ;
;   These functions restore the CPU registers from the context save area ;
; before returning to the code pointed to by the program counter value.  ;
; The only difference is that XSTEP purposely sets the interrupt flag    ;
; used for single stepping before returning to the users code.           ;
;                                                                         ;
; SINGLE STEPPING                                                         ;
;   The tail end of this function is written such that a pending         ;
; interrupt will not be serviced until one opcode has been fetched from  ;
; the return address and executed (there may be other dummy fetches).    ;
; This 'pending' interrupt then causes the processor to return back to   ;
; the context save routine, effectively singlestepping the CPU.          ;
;                                                                         ;
;=========================================================================;
S0_xdata  .set 0x808048         ;  SP 0 Data transmit
S0_rdata  .set 0x80804C         ;  SP 0 Data receive
               .sect "kernel"
        .word    0x00320C31     ; Prepend a few easily recognizable markers
        .word    0x00320C31     ;
XSTEP   or      0x40,IF         ; set  XINT1 (safe INT for C31/C32 debug!)
XRUNF   or      0xC4,IE         ; set EXINT1 (safe INT for C31/C32 debug!)
        ;----------------------
        ldi     @CPUCTXT,AR0    ; Use parallel opcodes for squeeze
        ldi     AR0,AR1         ;
        addi    1,AR1           ;
        ldi     2,IR0           ;
        ;----------------------
        ldi     @S0_rdata,R0    ; Clear under/overrun conditions before exit
        ldi     0,R0            ; 0 ensures low bits during SP recovery
        sti     R0,@S0_xdata    ; XSR resends - should all be zero
        ldf     *AR0++(IR0),R0  ; load floats (exponents)
    ||  ldf     *AR1++(IR0),R1  ;
        ldf     *AR0++(IR0),R2  ;
    ||  ldf     *AR1++(IR0),R3  ;
        ldf     *AR0++(IR0),R4  ;
    ||  ldf     *AR1++(IR0),R5  ;
        ldf     *AR0++(IR0),R6  ;
    ||  ldf     *AR1++(IR0),R7  ;
        ;- - - - - - - - - - - -
        ldi     *AR0++(IR0),R0  ; load longs (mantissa)
    ||  ldi     *AR1++(IR0),R1  ;
        ldi     *AR0++(IR0),R2  ;
    ||  ldi     *AR1++(IR0),R3  ;
        ldi     *AR0++(IR0),R4  ;
    ||  ldi     *AR1++(IR0),R5  ;
        ldi     *AR0++(IR0),R6  ;
    ||  ldi     *AR1++(IR0),R7  ;
```

```
        ;- - - - - - - - - - - - -
        ldi      @_AR0,AR0       ; load ARx
        ldi      @_AR1,AR1       ;
        ldi      @_AR2,AR2       ;
        ldi      @_AR3,AR3       ;
        ldi      @_AR4,AR4       ;
        ldi      @_AR5,AR5       ;
        ldi      @_AR6,AR6       ;
        ldi      @_AR7,AR7       ;
        ldi      @_IR0,IR0       ;
        ldi      @_IR1,IR1       ;
;       or       @_IF,IF         ; CPU interrupt flags
        ldi      @_IOF,IOF       ; IO flags
        ldi      @_RS,RS         ; Repeat start
        ldi      @_RE,RE         ; Repeat end
        ldi      @_RC,RC         ; Repeat counter
        ldi      @_BK,BK         ; Block size
        ldi      @_SP,SP         ; get user SP

        ;- - - - - - - - - - - - -
        ldi      @_PC,R5         ; return to PC from TOS return
        andn     0x4,IF          ; Clear/Poll INT2 before SSTEP or RUNF
        tstb     4,IF            ;
        bnz      $-3             ;
        ldiu     @_ST,ST         ; restore Status
        or       @_IE,IE
        BUD      R5              ;
        or       2000h,ST        ; turn on INT's
        ldiu     @_R5,R5         ;
        ldiu     @_DP,DP         ; restore DP
;======================================================================;
; XHALT                                                                ;
;   When called this function restores the temporary use registers used ;
; for quick returns from the XWRITE/XREAD before falling into a full   ;
; context save, followed by waiting for a new command.                 ;
;======================================================================;
XHALT   pop     AR1              ; restore original registers before save
        pop     AR0              ;
        pop     IR1              ;
        pop     R0               ;
        pop     DP               ;
        pop     ST               ; User PC now at TOS
```

```
;==========================================================================;
; SSTEP                                                                    ;
;   This section of code is executed after the pending interrupt, which  ;
; was set in XSTEP, has feteched the ISR vector and begun execution.     ;
; This code performs a full CPU context save before going to the spin    ;
; loop to await further commands.                                          ;
;==========================================================================;
SSTEP   push    DP              ; temp storage of user DP
        ldp     @_ST            ; DP for kernal
        sti     ST,@_ST         ; store ST
        sti     IR0,@_IR0       ; IR0 used as temp, later for indexed store
        pop     IR0             ; save user DP
        sti     IR0,@_DP        ;
        pop     IR0             ; save user PC
        sti     IR0,@_PC        ;
        sti     SP,@_SP         ; save user SP
        sti     BK,@_BK         ; Block size
        sti     IE,@_IE         ; Internal int enable
        sti     IF,@_IF         ; CPU interrupt flags
        sti     IOF,@_IOF       ; IO flags
        sti     RS,@_RS         ; Repeat start
        sti     RE,@_RE         ; Repeat end
        sti     RC,@_RC         ; Repeat counter
      ; sti     IR0,@_IR0       ; Keep everything  <- IR0 Saved previously
        sti     IR1,@_IR1       ;
        ;- - - - - - - - - - - - -
        sti     AR0,@_AR0       ; Use parallel opcodes for squeeze
        sti     AR1,@_AR1       ;
        ldi     @CPUCTXT,AR0    ;
        ldi     AR0,AR1         ;
        addi    1,AR1           ;
        ldi     2,IR0           ;
        ;- - - - - - - - - - - - -
        stf     R0,*AR0++(IR0)  ; Store floats
    ||  stf     R1,*AR1++(IR0)  ;
        stf     R2,*AR0++(IR0)  ;
    ||  stf     R3,*AR1++(IR0)  ;
        stf     R4,*AR0++(IR0)  ;
    ||  stf     R5,*AR1++(IR0)  ;
        stf     R6,*AR0++(IR0)  ;
    ||  stf     R7,*AR1++(IR0)  ;
        ;- - - - - - - - - - - - -
        sti     R0,*AR0++(IR0)  ; Store longs
    ||  sti     R1,*AR1++(IR0)  ;
        sti     R2,*AR0++(IR0)  ;
    ||  sti     R3,*AR1++(IR0)  ;
        sti     R4,*AR0++(IR0)  ;
    ||  sti     R5,*AR1++(IR0)  ;
        sti     R6,*AR0++(IR0)  ;
    ||  sti     R7,*AR1++(IR0)  ;
```

```
        ;- - - - - - - - - - - - -
        sti     AR2,@_AR2       ; AR0 & AR1 Already saved
        sti     AR3,@_AR3       ;
        sti     AR4,@_AR4       ;
        sti     AR5,@_AR5       ;
        sti     AR6,@_AR6       ;
        sti     AR7,@_AR7       ;
;-------------------------------
        ldi     @_PC,AR1        ; Send ACK (value of PC at time of halt)
        ldi     *-AR1(1),R0     ; to the host processor to indicate that
TRAP_AK call    W_HOST          ; a halt/spin condition has been entered
        ldi     07000h,AR0      ;
        ldi     *AR0,AR0        ;
     ;  b       spin0           ; <- Branch is removed (spin0 is inline)
;--------------------------------------------------------------------;
; The spin0 code loop is used by the kernel as a known program loop  ;
; when a process is halted.  While in the spin loop, commands can be ;
; processed.  This code loop is primarily used while debugging or    ;
; during startup as a known useable code loop.                       ;
;--------------------------------------------------------------------;
spin0   or      4,IE            ; Enable DSK31 HPI interrupt
        and     4,IE            ; Shut down all interrupts except host
    ;   ldi     0,R0            ; Pump the DXR with 0 to prevent underflow
    ;   sti     R0,@S0_xdata    ;
        idle                    ; IDLE saves power but prevent S.P. refresh
        b       spin0           ;
;-------------------------------
S0xdata .word 0
GIE     .set  0x2000
```

```
;======================================================================;
; REGISTER CONTEXT STORAGE                                             ;
;  This block of memory holds the register values when a process is   ;
;  stopped.  Essentially the registers displayed in the debugger are  ;
;  the contents of this memory block.                                  ;
;======================================================================;
context                          ;
_F0             .word   0        ; R0
_F1             .word   0        ; R1
_F2             .word   0        ; R2
_F3             .word   0        ; R3
_F4             .word   0        ; R4
_F5             .word   0        ; R5
_F6             .word   0        ; R6
_F7             .word   0        ; R7
_R0             .word   0        ; F0
_R1             .word   0        ; F1
_R2             .word   0        ; F2
_R3             .word   0        ; F3
_R4             .word   0        ; F4
_R5             .word   0        ; F5
_R6             .word   0        ; F6
_R7             .word   0        ; F7
_AR0            .word   0        ; AR0
_AR1            .word   0        ; AR1
_AR2            .word   0        ; AR2
_AR3            .word   0        ; AR3
_AR4            .word   0        ; AR4
_AR5            .word   0        ; AR5
_AR6            .word   0        ; AR6
_AR7            .word   0        ; AR7
_DP             .word   0        ; Data page
_IR0            .word   0        ; Index register 0
_IR1            .word   0        ; Index register 1
_BK             .word   0        ; Block size
_SP             .word   stack-1  ; Stack pointer (initial DSK3D value)
_ST             .word   0        ; Status
_IE             .word   0        ; Internal int enable
_IF             .word   0        ; CPU interrupt flags
_IOF            .word   0        ; I/O flags
_RS             .word   0        ; Repeat start
_RE             .word   0        ; Repeat end
_RC             .word   0        ; Repeat counter
_PC             .word   0        ; program counter
CPUCTXT         .word   context  ;
```

```
;*******************************************************************;
; KERNEL COMMANDS                                                   ;
; ---------------                                                   ;
;   These commands are the primary functions required by the kernel ;
; to perform host based communications.  They have been packed into ;
; the avalable memory in such a way as to minimize the kernels size. ;
; The non-debugger functions have also been placed after the debugger;
; commands making it easier to simply allow the application to      ;
; 'overwrite' the debugger commands.                                ;
;*******************************************************************;
;===================================================================;
; INTx is the starting point for all host generated commands.       ;
; A host generated command is received when INT2 goes active (driven ;
; low) indicating HPSTB has gone low and that the host would like to ;
; transfer a piece of data or command.                              ;
;===================================================================;
INTx  ;  maxspeed
        push    ST              ; Push ISR variables
        push    DP              ;
        push    R0              ; NOTE: A HALT command pops these
        push    IR1             ;       values followed by a full save
        push    AR0             ;
        push    AR1             ;
        ldp     @JUMP           ; Get address of command from JUMP table

        ldi     @S0_xdata,R0    ; Put a zero in the DXR making startup
        sti     R0,@S0xdata     ; from a stalled port safe for the AIC
        ldi     0,R0            ; which cannot accept 'garbage' which
        sti     R0,@S0_xdata    ; would reprogram it.

        tstb    4,IF            ; Get here by driving INT2 low
        bz      SR2             ; Make sure INT2 is active
        call    R_HOST          ; R0==command
        ldi     R0,AR1          ;
        addi    @JUMP,AR1       ;
        ldi     *AR1,AR1        ;
        b       AR1             ; execute command
;******************************************************************
; COMN is used by both the XWRIT and XREAD functions to receive the
; block transfer length, address and address increment value.
;******************************************************************
COMN    call    R_HOST          ;
        ldi     R0,AR1          ; data packet length
        call    R_HOST          ;
        ldi     R0,AR0          ; source address
        call    R_HOST          ;
        ldi     R0,IR1          ; source index
        subi    1,AR1           ;
        rets                    ;
```

```
;======================================================================;
; The XCTXT command returns the address of the context save area to  ;
; the host.  Subsequently, the host can use this address to 'get'    ;
; and put the CPU registers to modify the exectution of the processor;
;----------------------------------------------------------------------;
XCTXT   ldi     @CPUCTXT,R0     ; Transmit location of context to CPU
        call    W_HOST          ;
    ;   b       SR2             ;
;======================================================================;
; SR2 is the short 'common' return sequence used by most commands.   ;
; when executed, the return will send the CPU back to the users code ;
;======================================================================;
SR2     ldi     07F00h,AR0      ; Dummy non-HPI read releases READY
        ldi     *AR0,AR0        ;
        pop     AR1             ; restore ISR variables
        pop     AR0             ;
        pop     IR1             ;
        pop     R0              ;
        pop     DP              ;
      ; andn    0x4,IF          ;
      ; or      4,IE            ;
        pop     ST              ;
        reti                    ; return to original code
;======================================================================;
; TMS320C31 SECONDARY VECTOR TABLE                                     ;
; --------------------------------                                     ;
;   When the TMS320C31 receives an interrupt it first fetches an       ;
; address from the primary vector table (located in the bootloader    ;
; ROM).  This 32 bit value is then used as an address where the       ;
; new execution begins.                                                ;
;                                                                      ;
;   Since it is impossible to relocate the vector table, or modify    ;
; the contents of the bootloader ROM, a 'secondary' or 'branch'       ;
; vector table is used to direct execution to the correct routines.  ;
; In this case the C31's primary vector table has been filled with    ;
; interrupt routine addresses which point to the upper memory of      ;
; internal RAM beginning at 0x809FC0.  Since these locations are      ;
; were execution actually begins, and can be modified, a branch       ;
; opcode can be used to direct execution to the desired location.    ;
;======================================================================;
        .sect   "vectors"
INT0    b       $               ; 0x809FC1  0x001
INT1    b       $               ; 0x809FC2  0x002
INT2    b       INTx            ; 0x809FC3  0x004 <- HPI
INT3    b       $               ; 0x809FC4  0x008
XINT0   b       $               ; 0x809FC5  0x010
RINT0   b       $               ; 0x809FC6  0x020
XINT1   b       SSTEP           ; 0x809FC7  0x040 <- SSTEP
RINT1   b       SSTEP; TRAPFIX  ; 0x809FC8  0x080 <- ETRAP 0x74000008
TINT0   b       $               ; 0x809FC9  0x100
TINT1   b       $               ; 0x809FCA  0x200
DINT    b       $               ; 0x809FCB  0x400
```

```
;=====================================================================;
; HOST HPI communications routines packed into himem                  ;
;                                                                     ;
;  NOTE: These routines can be called from a high level langauge     ;
;  compiler using the C31s TRAP commands, by directly linking their  ;
;  resolved addresses or by using the jump table.                     ;
;=====================================================================;
;   W_HOST performs an interlocked Host Port write of the contents   ;
; of R0 to the host using the HPSTB/HPACK protocol.  When called the ;
; host PC should be waiting for this function to send data.          ;
;=====================================================================;
W_HOST  push    AR0              ; Used for HPI address
        push    AR1              ; Used for loop counter
        push    ST               ; Keep flags
        push    DP               ; Might not be on same page
        ldp     WSCOUNT          ;
        ldi     0xF000,AR0       ; HPI address sign extends to 0xFFF000
        ldi     @WSCOUNT,AR1     ;
WH      sti     R0,*AR0++(16)    ; Store lsbs to HPI
        lsh     @WSHIFT,R0       ; shift to next lsbs
        db      AR1,WH           ; loop until done
        pop     DP               ;
        b       COMNHST          ;
;=====================================================================;
;   R_HOST performs an interlocked Host Port read from the printer   ;
; port interface and places the result into R0.                      ;
;=====================================================================;
R_HOST  push    AR0              ; HPI Address
        push    AR1              ; loop counter
        push    ST               ;
        push    R1               ; temp register
        ldi     0xF000,AR0       ; HPI address sign extends to 0xFFF000
        ldi     3,AR1            ; bytes-1 to receive
RH      lsh     -8,R0            ; shift result right one byte
        ldi     *AR0++,R1        ; Load byte
        lsh     24,R1            ; shift to upper byte
        or      R1,R0            ; or w/result
        db      AR1,RH           ; loop until done
        pop     R1               ; restore
   ;;;; b       COMNHST          ; <- Branch can be saved
COMNHST pop     ST               ; Next 4 opcodes common to W_HOST/R_HOST
        pop     AR1              ;
        pop     AR0              ;
        rets                     ;
;=====================================================================;
;   XWRIT is a host port command designed to transfer a block of     ;
; data from the host to the C31's memory.                            ;
;=====================================================================;
XWRIT   call    COMN             ;
XW1     call    R_HOST           ;
        sti     R0,*AR0++(IR1)   ;
        db      AR1,XW1          ;
        b       SR2              ;
```

```
;======================================================================;
;   XREAD is a host port command designed to transfer a block of      ;
; data from C31 memory to the host.                                    ;
;======================================================================;
XREAD   call    COMN            ;
XR1     ldi     *AR0++(IR1),R0  ;
        call    W_HOST          ;
        db      AR1,XR1         ;
        b       SR2             ;
;======================================================================;
;   There are a few leftover traps that can be used by appliactions  ;
; The number of TRAPs coincides with the amount of available unused  ;
; memory before the JUMP table is encountered and was adjusted by    ;
; hand by looking at the assembler listing                           ;
;======================================================================;
TRAP00  b       $               ;          Leftover TRAPs which can be
TRAP01  b       $               ;          used by applictions
;======================================================================;
; A JUMP table can also be used to access the DSK3 routines from      ;
; other applications that require host communications.  In this case ;
; the contents of the loaction specified can be used in a register   ;
; call or branch.                                                     ;
;======================================================================;
;        .start ”JMPTBL”,0x809FF4
;        .sect  ”JMPTBL”
JUMP    .word   JUMP            ;0x809FF4 Jump table base address
        .word   XWRIT       ;1  ;0x809FF5 for DSK3 routines
        .word   XREAD       ;2  ;0x809FF6
        .word   XCTXT       ;3  ;0x809FF7
        .word   XRUNF       ;4  ;0x809FF8
        .word   XSTEP       ;5  ;0x809FF9
        .word   XHALT       ;6  ;0x809FFA
        .word   W_HOST      ;7  ;0x809FFB
        .word   R_HOST      ;8  ;0x809FFC
        .word   spin0       ;10 ;0x809FFD  Use for spare command
;======================================================================;
; The last two locations of internal memory hold the two parameters  ;
; which define the printer ports bus return width.  Depending on the ;
; values, either 8 bit bi-directional or 4 bit nibble returns can    ;
; be implimented.  These values control the loop count and shift     ;
; value needed to place the correct bits on the proper return buffer ;
; inputs.                                                             ;
;                                                                     ;
; DO NOT OVERWITE THESE VALUES unless you are performing buswidth    ;
; verification or setup.  For more details, see the communications   ;
; initialization routines within the host side code.                 ;
;======================================================================;
WSCOUNT .word   7               ;0x809FFE These locations hold the W_HOST
WSHIFT  .word   -4              ;0x809FFF buswidth parameters (Nibble/Byte)

        .end
```

Appendix B

DSK Circuit Board Dimensions and Schematic Diagrams

This appendix contains the circuit board dimensions and the schematic diagrams for the TMS320C3x DSP Starter Kit.

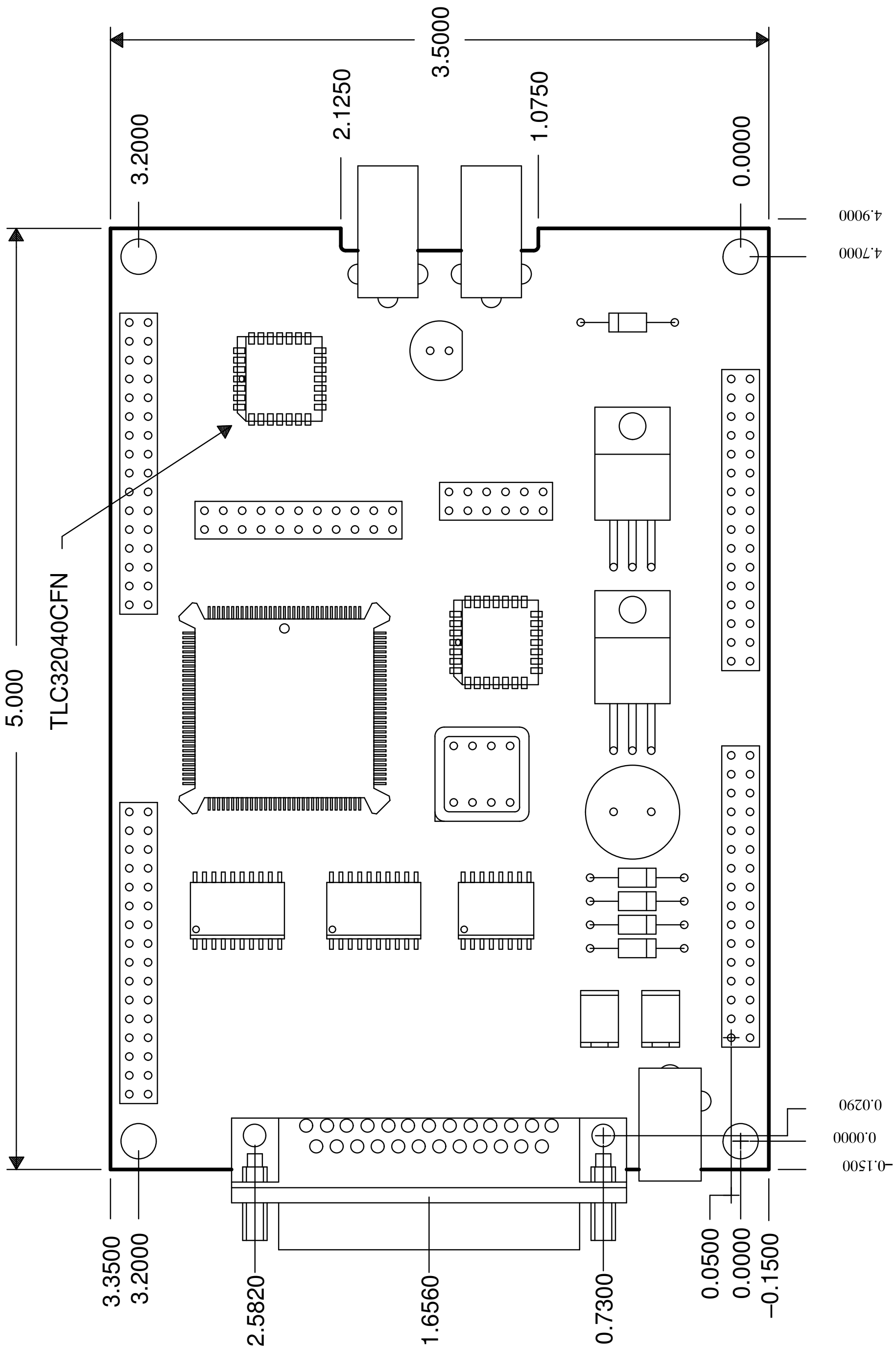


Figure B–1. TMS320C3x DSP Starter Kit (DSK) Circuit Board Dimensions

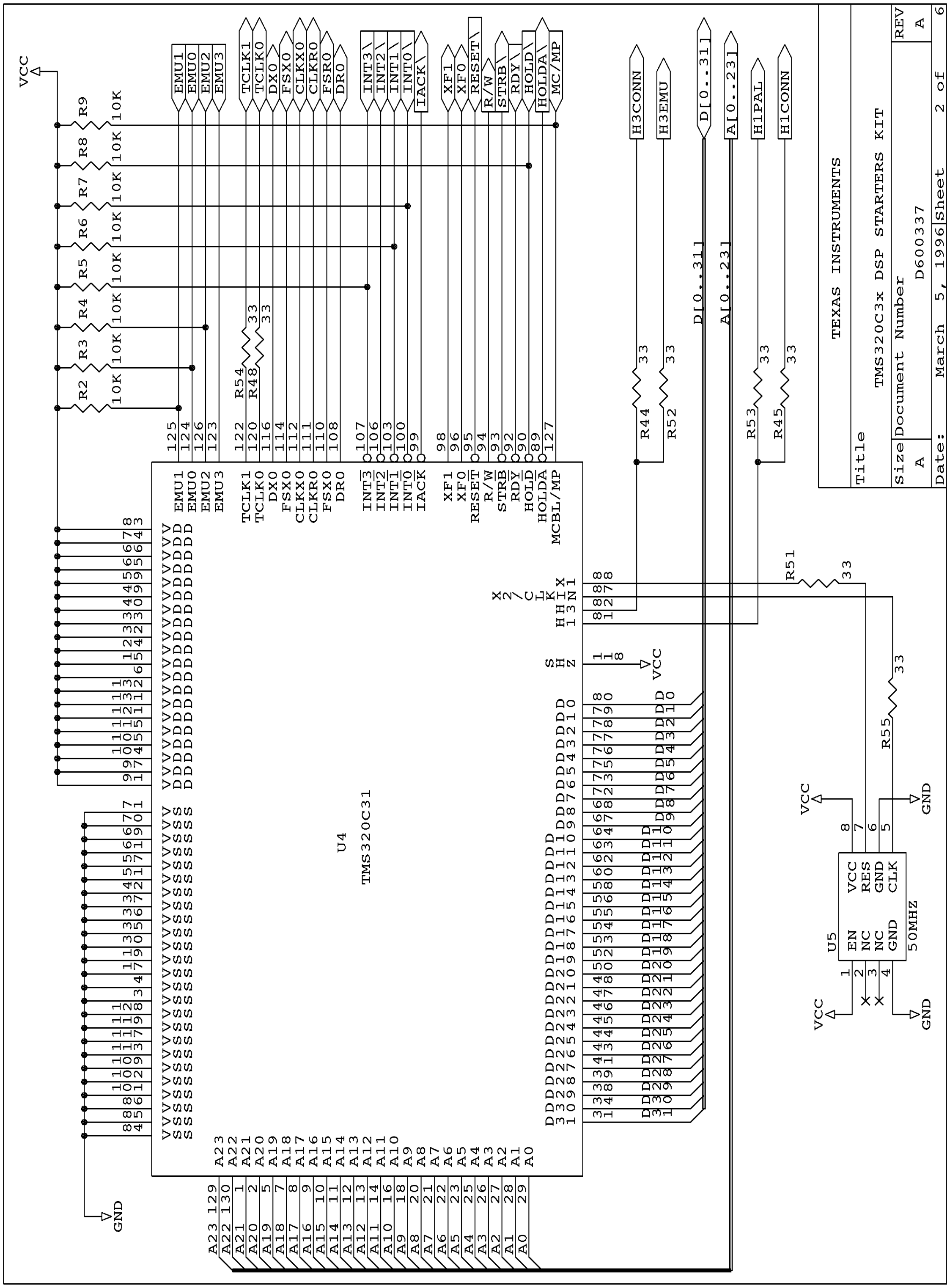
Hardware Component Overview

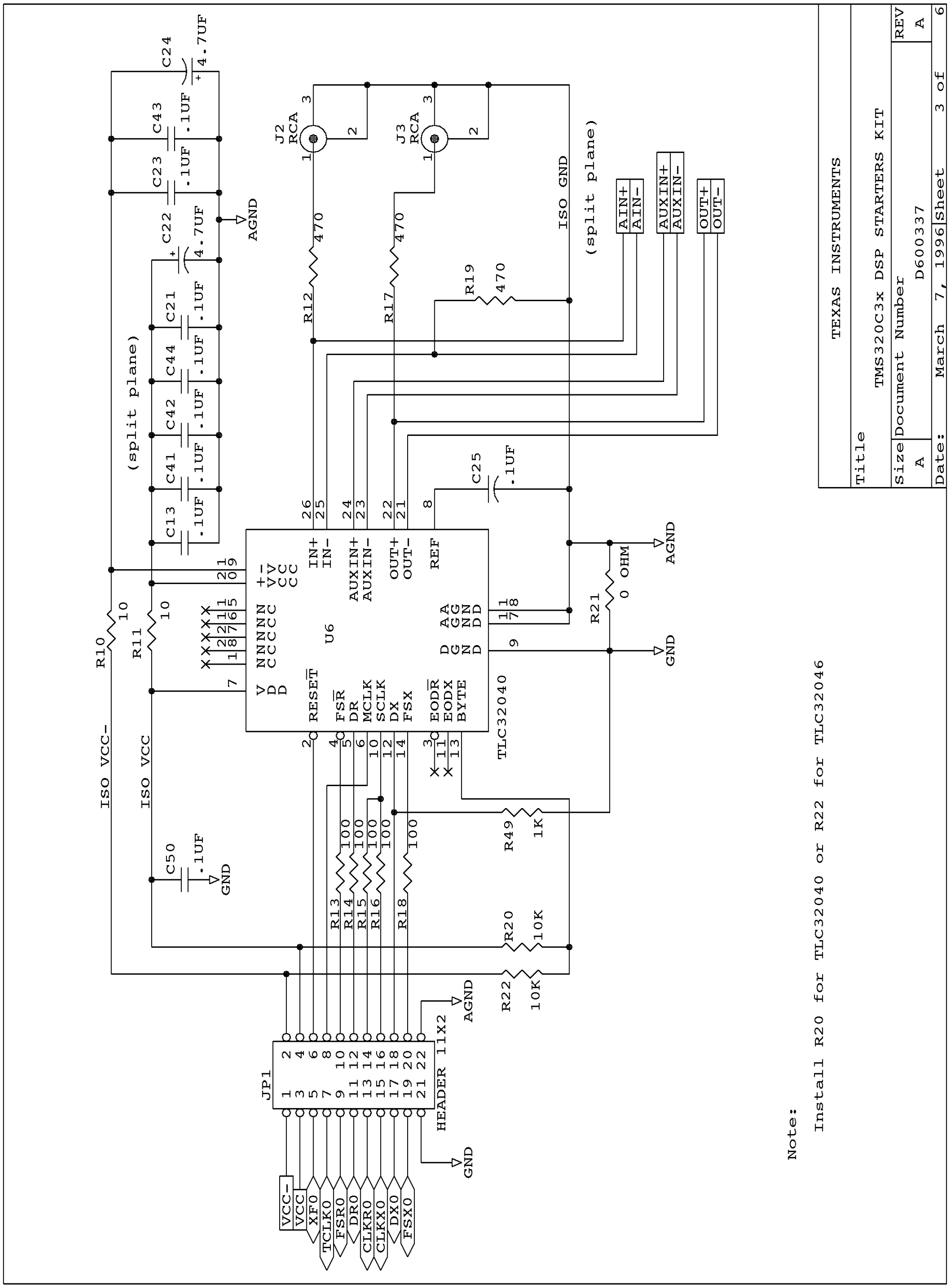
This section describes the basic functions of the DSK components:

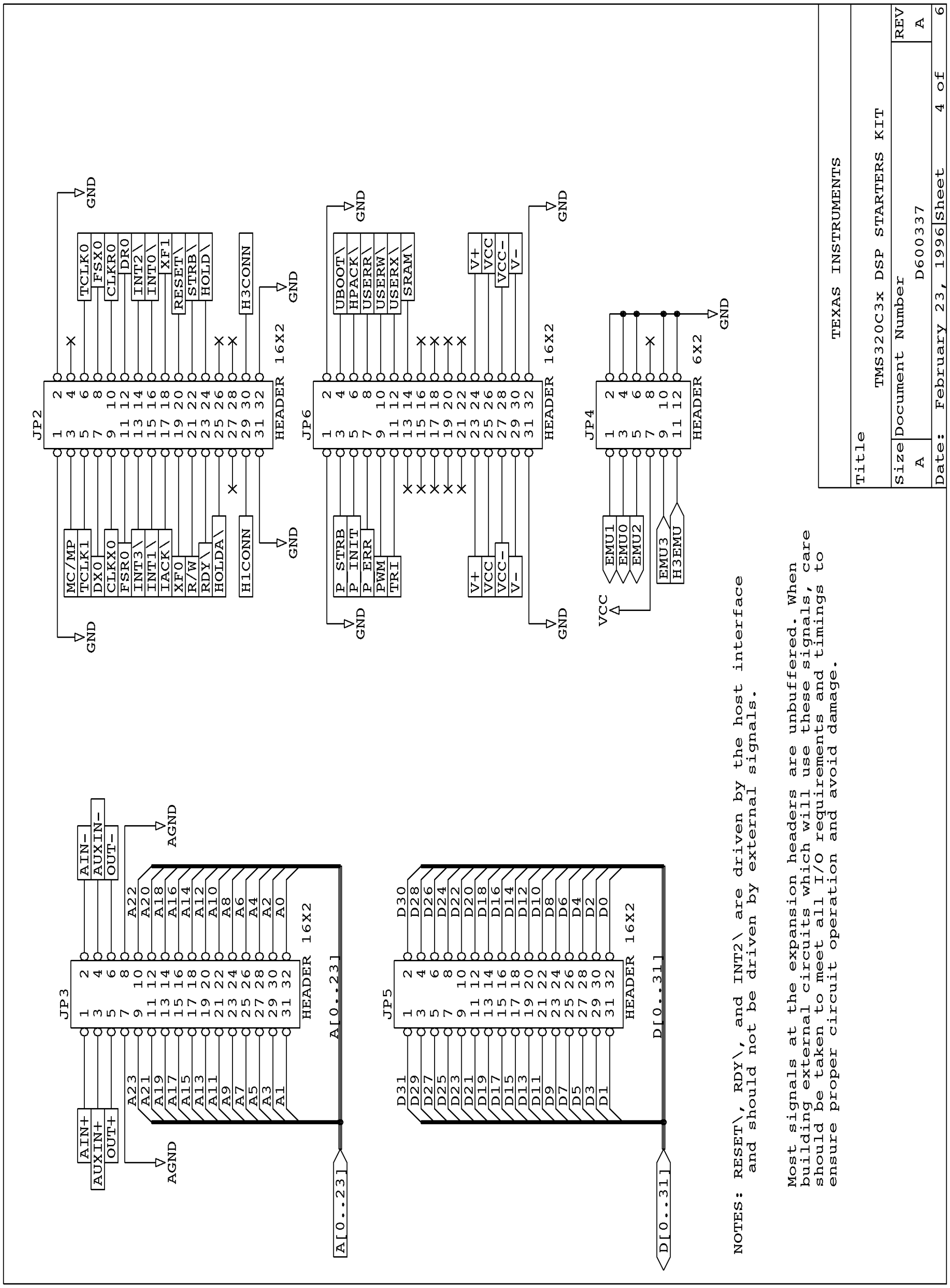
- **Expansion Connectors** — The four 32-pin headers allow you to develop add-on cards that can directly interface to all of the 'C31 signals.
- **Jumper block header** — An 11-pin jumper block connects the 'C31 serial port to the TLC32040 AIC. Removal of the jumpers disconnects the AIC from the 'C31 serial port, so that a daughtercard can use the serial port signals.
- **Host Interface Logic** — The host interface logic consists of a programmable array logic (PAL) 22V10Z and two high-speed octal bus transceivers with tri-state outputs (74ACT245). These devices interface the 'C31 with the host parallel printer port. This interface logic supports 8-bit bidirectional or 4-bit unidirectional data modes of the PC host.
- **Oscillator** — The on-board 50Mhz oscillator drives the 'C31 clock input. The 'C31 internal clock value is divided by 1 (same frequency).
- **Parallel Printer Port Connector** — The DB25 25-pin connector connects directly to the host parallel printer port.
- **RCA Jacks** — The RCA jacks supply analog input or output and are routed to the I/O pins of the AIC.
- **Resettable Fuses** — The polyswitch resettable fuses interrupt the flow of excessive current. The fuses reset after they cool down and the faulty condition is corrected. The fuses require no manual resetting or replacement.
- **TLC32040 AIC** — The analog interface circuit provides the 'C31 access to the analog world. The AIC samples analog data and converts it into a digital stream for 'C31 analysis. The 'C31 operates on this digital data and returns the "transformed" digital data to the AIC for conversion into an analog signal.
- **TMS320C31** — The main processor is a 32-bit, floating-point digital signal processor. You develop application code and load it to the on-chip memory of the 'C31. This code can be executed, single-stepped, and viewed in the debugger.

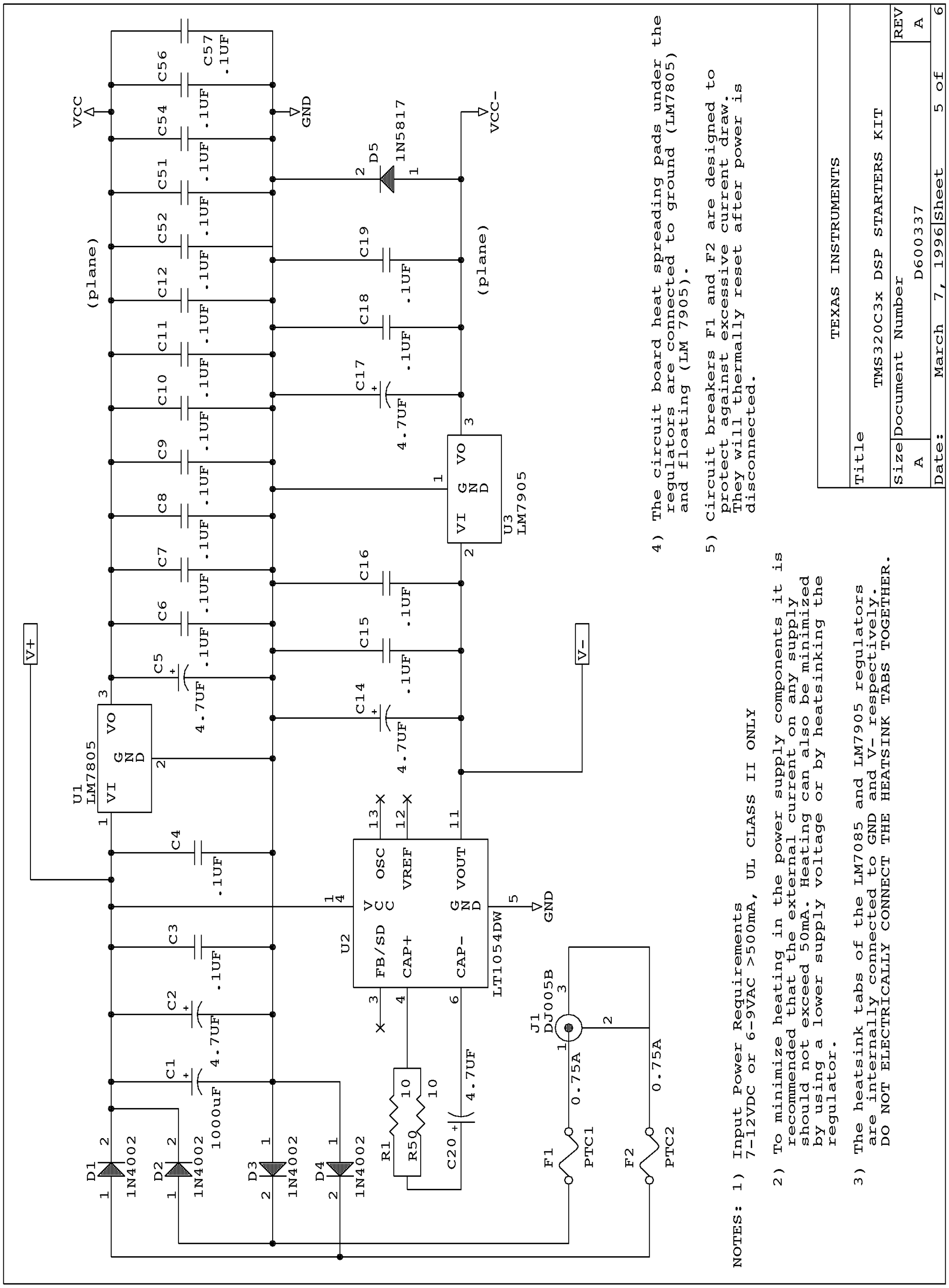
- **Voltage Regulators** - The DSK uses a 7–12 Vdc or 6–9 Vac wall mount power supply. The 7–12 Vdc supply voltage is full-wave rectified and then regulated up to 5 volts by the LM7805. It is also converted to –5 volts by the capacitive switching circuit LT1054, and then regulated by the LM7905. The 6–9 Vac supply is full-wave rectified and then regulated by the LM7805 and LM7905 to +5V and –5V, respectively. The +5V and –5V supplies are used to power all of the DSK on-board circuitry. The TLC32040 AIC requires a negative poser supply of –5 volts.

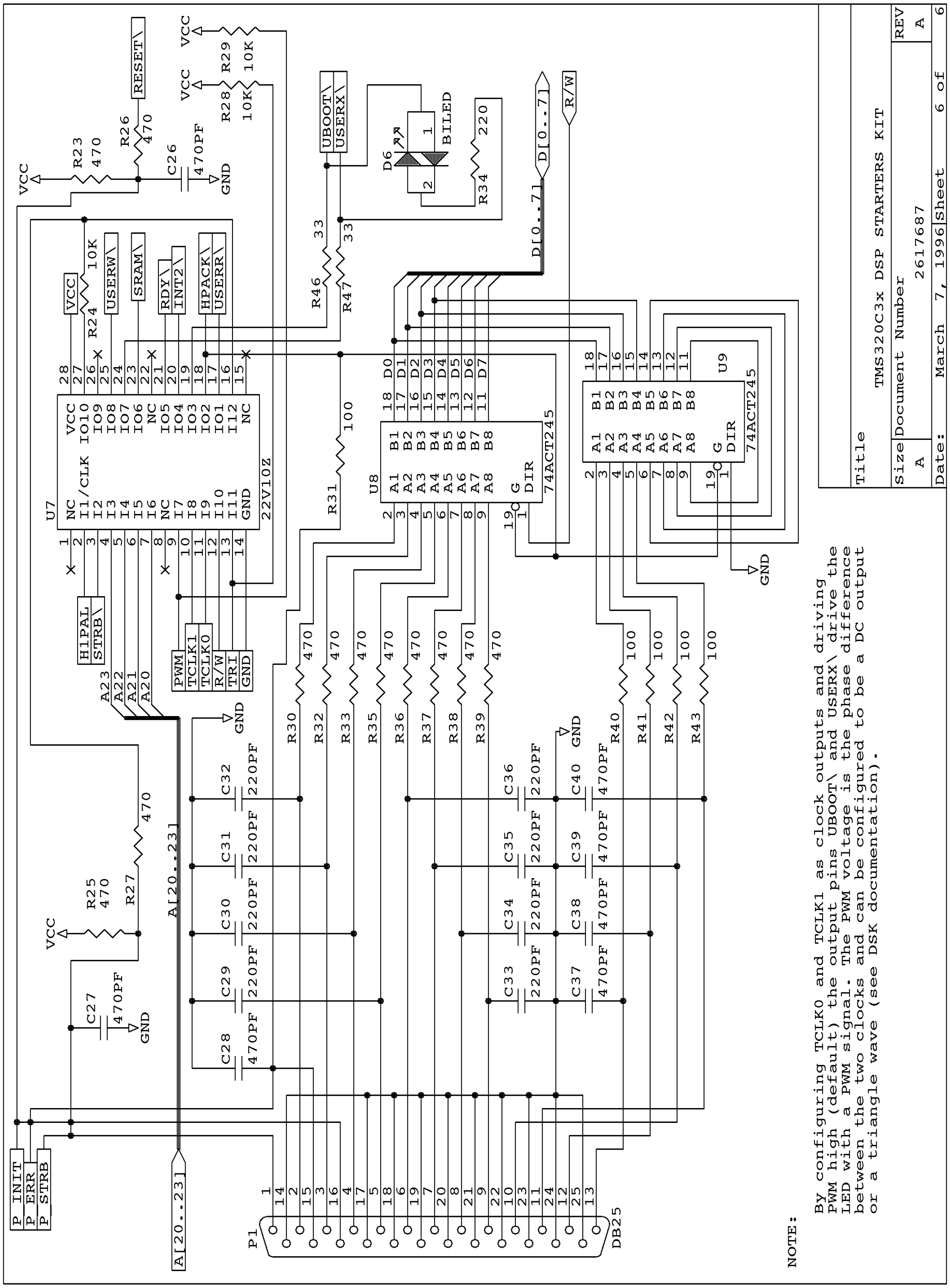
- **XDS Emulator Port** — An 11-pin header that connects the XDS510 emulator to the 'C31. The emulator allows you to upgrade to the full-featured XDS debugger to debug your application code while using the DSK as the XDS target board.


TEXAS INSTRUMENTS
Title
TMS320C3x DSP STARTERS KIT
Size A
Document Number D600337
REV A
Date: March 5, 1996
Sheet 2 of 6
U4
TMS320C31
U5
50MHZ
VCC
GND
D[0..31]
A[0..23]
H3CONN
H3EMU
H1PAL
H1CONN


TEXAS INSTRUMENTS
Title
TMS320C3x DSP STARTERS KIT
Size
A
Document Number
D600337
REV
A
Date: March 7, 1996 Sheet 3 of 6
Note:
Install R20 for TLC32040 or R22 for TLC32046
U6
TLC32040
IN+
IN-
AUXIN+
AUXIN-
OUT+
OUT-
REF
RESET
FSR
DR
MCLK
SCLK
DX
FSX
EODR
EODX
BYTE
J2
RCA
J3
RCA
ISO GND
(split plane)
AIN+
AIN-
AUXIN+
AUXIN-
OUT+
OUT-
C24
4.7UF
C43
.1UF
C23
.1UF
C22
4.7UF
C21
.1UF
C44
.1UF
C42
.1UF
C41
.1UF
C13
.1UF
C25
.1UF
C50
.1UF
R10
10
R11
10
R12
470
R17
470
R19
470
R21
0 OHM
R49
1K
R13
R14
R15
R16
R18
100
R20
10K
R22
10K
ISO VCC-
ISO VCC
AGND
GND
JP1
HEADER 11X2
VCC-
VCC
XF0
TCLK0
FSR0
DR0
CLKR0
CLKX0
DX0
FSX0


JP2
HEADER 16X2
JP6
HEADER 16X2
JP4
HEADER 6X2
JP3
HEADER 16X2
JP5
HEADER 16X2
A[0..23]
D[0..31]
GND
AGND
VCC
NOTES: RESET\, RDY\, and INT2\ are driven by the host interface
and should not be driven by external signals.
Most signals at the expansion headers are unbuffered. When
building external circuits which will use these signals, care
should be taken to meet all I/O requirements and timings to
ensure proper circuit operation and avoid damage.
TEXAS INSTRUMENTS
Title
TMS320C3x DSP STARTERS KIT
Size
A
Document Number
D600337
REV
A
Date: February 23, 1996
Sheet 4 of 6


VCC
GND
VCC-
V+
V-
(plane)
U1 LM7805
U2 LT1054DW
U3 LM7905
D1 1N4002
D2 1N4002
D3 1N4002
D4 1N4002
D5 1N5817
J1 DJ005B
F1 PTC1 0.75A
F2 PTC2 0.75A
R1 10
R50 10
C1 1000uF
C20 4.7UF
NOTES: 1) Input Power Requirements
7-12VDC or 6-9VAC >500mA, UL CLASS II ONLY
2) To minimize heating in the power supply components it is recommended that the external current on any supply should not exceed 50mA. Heating can also be minimized by using a lower supply voltage or by heatsinking the regulator.
3) The heatsink tabs of the LM7085 and LM7905 regulators are internally connected to GND and V- respectively. DO NOT ELECTRICALLY CONNECT THE HEATSINK TABS TOGETHER.
4) The circuit board heat spreading pads under the regulators are connected to ground (LM7805) and floating (LM 7905).
5) Circuit breakers F1 and F2 are designed to protect against excessive current draw. They will thermally reset after power is disconnected.
TEXAS INSTRUMENTS
Title
TMS320C3x DSP STARTERS KIT
Size A
Document Number D600337
REV A
Date: March 7, 1996
Sheet 5 of 6


Title
TMS320C3x DSP STARTERS KIT
Size A
Document Number 2617687
REV A
Date: March 7, 1996
Sheet 6 of 6
NOTE:
By configuring TCLK0 and TCLK1 as clock outputs and driving PWM high (default) the output pins UBOOT\ and USERX\ drive the LED with a PWM signal. The PWM voltage is the phase difference between the two clocks and can be configured to be a DC output or a triangle wave (see DSK documentation).

Host Interface Control Design Notes

```
TITLE       HOST INTERFACE CONTROL
DWG. NAME   TMS320C3X DSK
ASSY #      D600335-0001
PAL #       U7
COMPANY     TEXAS INSTRUMENTS INCORPORATED
ENGR        KEITH LARSON
DATE        3/7/96
;
;  DESIGN NOTES:
;
;  The power consumption of the TMS320C31 DSK was considerably lowered by
;  the use of a CMOS TIBPAL22V10Z.  When clocked at 25MHz (H1 rate) the
;  TIBPAL22V10Z typicaly consumes 40mA (80mA max) as compared to 200mA for
;  bipolar PAL devices.  If lower consumption is needed the TMS320C31 can be
;  programmed to use the LOPOWER or IDLE2 when full speed execution is not
;  required.  LOPOWER essentially runs the DSP at 1/16 of full speed and
;  IDLE2 shuts the the clock completely off.  This results in 1/16 and
;  practically zero power for these modes respectively for both the PAL
;  and the DSP.  However due to the 25nS propogation delay through the
;  TIBPAL22V10Z a wait state is required for host and peripheral decodes.
;
;  Memory access times for the /SRAM decoded output are as follows
;
;   TIBPAL22V10Z (CMOS) at 50MHz, H1 = 40ns:
;
;          t-access = H1 * (1 + WS) - Tpal - ( Td(H1L-A) - Tsu(D)R )
;          t-access = H1 * (1 + WS) - 25ns - 19ns
;
;          wait states   ==>   0   1   2   3   4 ...
;          t-access read ==>  -4  36  76 116 156 ...
;
;  IDLE2 wakeup is initiated by asserting the INT2 pin low.  Since the
;  clock is stopped during IDLE2, gating with synchronized signals cannot
;  be used.  A buffer is used with INT2 to avoid differences in the logic
;  thresholds of the PAL22V10 and the C31 and to improve the rise and fall
;  time of that signal.
;
;  TRI-COLOR LED (POWER AND PWM)
;  -----------------------------
;  If a logic high is applied to PWM (default state), the outputs /UBOOT
;  and /USERX become an XOR and /XOR of T0 and T1.  The XOR gate in this
;  case is being used to detect the phase angle between T0 and T1.  Therefor
;  if T0 and T1 are configured as outputs, such as when the debugger is
;  started, the color can be controled by adjusting the timers.
;
;
;  USING THE PWM AS A DAC:
;  -----------------------
;  If the output is filtered to a DC level by a low pass filter the
;  DC level can be controlled by setting the two timers to identical
;  freqencies seperated by a constant phase angle (delay).  Since both the
;  XOR and /XOR are provided a differential signal is also available.
;
```

```
;  If T0=T1 the output is a DC level proportional to the phase difference
;
;  T0 ------______------______------______------______------______------____
;  T1 ----______------______------______------______------______------______
;  XOR____--____--____--____--____--____--____--____--____--____--____--____
;
;  USING THE PWM AS A TRIANGLE WAVE GENERATOR
;  ------------------------------------------
;  If T0 and T1 are set to different frequencies a PWM modulated triangle
;  wave at a frequency of F0_t0 - F1_t1 is produced.  Since the two XOR
;  outputs are compliments a bridged output is created.  This then allows the
;  current in the LED to reverse resulting in an ;  alternating color sequence
;  of R-Y-G-Y-R-Y-G-Y...
;
;  If T0!=T1 the output is a continuous triangle wave
;
;  T0 ------______------______------______------______------______------____
;  T1 -----_____-----_____-----_____-----_____-----_____-----_____-----_____
;  XOR_____-____--___---__----_----------_----__---___--____-__________-____
;
;  If an H bridge drive circuit is used with these signals an AC motor can
;  be driven with an DSP controlled frequency.  By using an external PAL to
;  provide additional references signals and phase detectors a 3-phase PWM
;  driver can be easily constructed.   In this case the external PAL would
;  contain a counter whose output is decoded to provide one of the reference
;  frequencies in three phases seperated by 2*pi/3 radians.  By then using
;  one of the DSP timers for the other reference a variable frequency
;  3 phase output can be constructed.
;
;  NOTE: The amplitude of the PWM triangle wave cannot be controlled from
;        the timers alone.  Either the DSP would have to continuously
;        calculate the ouptuts as a DC reference or an external circuit
;        would have to chop the output.
;---------------------------------------------------------------------------
;
;             STRB           Q0
;           A23  | H1     VCC|  Q1
;             |  |  |     |  |  |
;        /----+--+--+--+--+--+--+---+
;        |    4  3  2  1  28 27 26  |
;        |                          |
;    A22-|5                       25|-UW
;    A21-|6                       24|-UX
;    A20-|7                       23|-SRAM
;        |8                       22|
;   DEMO-|9                       21|-RDY
;   TCK1-|10                      20|-INT2
;   TCK0-|11                      19|-UBOOT
;        |                          |
;        |   12 13 14 15 16 17 18   |
;        +----+--+--+--+--+--+--+---+
;             |  |  |     |  |  |
;           R/W  |GND   HPS  |HPA
;              TRI          UR
;
```

```
NC CLK  STRB  A23  A22   A21  A20   NC   DEMO  T1    T0 RW TRI GND
NC HPIS USERR HPIA UBOOT INT2 READY NC SRAM USERX USERW Q1 Q0  VCC
global
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
EQUATIONS                                                  ;
READY.TRST = TRI                                           ;
INT2.TRST  = TRI                                           ;
INT2   = HPIS                                              ;
HPIA   = /(A23 * A22 * A21 *       /STRB) + /TRI           ; 245 enable and HPIA
Q0    := INT2                                              ; 1st tap
Q1    := Q0                                                ; 2nd tap for pulse gen
READY  = /(Q0*/Q1) * (A23*A22*A21*A20*/STRB)             ;
;
;                 A23  A22  A21 A20  /STRB
;
SRAM  =        /( A23*/A22          */STRB)
USERR =        /( A23* A22*/A21     */STRB*  RW)
USERW =        /( A23* A22*/A21     */STRB* /RW)
USERX =(/DEMO* /( A23* A22*/A21     */STRB))    +(DEMO* ((T0*/T1)+(/T0*T1)))
UBOOT =(/DEMO* /(/A23*/A22*/A21*/A20*/STRB))    +(DEMO*/((T0*/T1)+(/T0*T1)))
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; The decoded address ranges are as follows
;   NOTE: By using A23 as an enable, it is possible to use external
;         zero wait state RAM.  Essentialy by ignoring decoded outputs
;- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
; USER_BOOT           000000   0FFFFF   EPROM boot or uP mode operation
;                     100000   7FFFFF   No decode
; SRAM              0x800000 0xBFFFFF   1ws decoded external memory
; USER_R            0xC00000 0xDFFFFF > Read access
; USER_W            0xC00000 0xDFFFFF > Write access
; USER_X            0xC00000 0xDFFFFF > Read or Write access
; HPI(asynch)       0xE00000 0xEFFFFF   DSP access to bus w/o host lock
; HPI(host locked)  0xF00000 0xFFFFFF   Must pulse HPIS to advance DSP state
;
SIMULATION
TRACE_ON CLK HPIS STRB HPIA READY INT2 T0 T1 DEMO RW USERX UBOOT SRAM USERR USERW
;
; Simulate access outside decoded range
;
setf        /HPIS  STRB  A23  A22  A21  A20 TRI DEMO T0 T1 RW
clockf  CLK
setf           HPIS  STRB  A23  A22  A21  A20
clockf  CLK
clockf  CLK
setf          /HPIS  STRB  A23  A22  A21  A20
clockf  CLK
clockf  CLK
setf           HPIS  STRB  A23  A22  A21  A20
clockf  CLK
clockf  CLK
clockf  CLK
;
; Simulate access inside decoded range
;
setf           HPIS /STRB  A23  A22  A21  A20
```

```
clockf  CLK
clockf  CLK
clockf  CLK
clockf  CLK
setf           /HPIS /STRB  A23  A22  A21  A20
clockf  CLK
clockf  CLK
clockf  CLK
clockf  CLK
setf            HPIS /STRB  A23  A22  A21  A20
clockf  CLK
setf            HPIS  STRB  A23  A22  A21  A20
clockf  CLK
clockf  CLK
clockf  CLK
clockf  CLK
;
; Simulate access outside decoded range
;
setf            HPIS /STRB /A23 /A22 /A21 /A20
clockf  CLK
clockf  CLK
clockf  CLK
clockf  CLK
setf           /HPIS /STRB /A23 /A22 /A21 /A20
clockf  CLK
clockf  CLK
clockf  CLK
clockf  CLK
setf            HPIS /STRB /A23 /A22 /A21 /A20
clockf  CLK
setf            HPIS  STRB /A23 /A22 /A21 /A20
clockf  CLK
clockf  CLK
clockf  CLK
clockf  CLK
;
; Second access occurs with little delay
;
setf            HPIS /STRB  A23  A22  A21  A20
clockf  CLK
setf           /HPIS /STRB  A23  A22  A21  A20
clockf  CLK
clockf  CLK
clockf  CLK
clockf  CLK
setf            HPIS /STRB  A23  A22  A21  A20
clockf  CLK
setf            HPIS  STRB  A23  A22  A21  A20
clockf  CLK
clockf  CLK
clockf  CLK
;
; Second access occurs with little delay
;
```

```
setf           HPIS /STRB  A23  A22  A21  A20 /TRI
clockf  CLK
setf          /HPIS /STRB  A23  A22  A21  A20
clockf  CLK
clockf  CLK
clockf  CLK
clockf  CLK
setf           HPIS /STRB  A23  A22  A21  A20
clockf  CLK
setf           HPIS  STRB  A23  A22  A21  A20
clockf  CLK
clockf  CLK
clockf  CLK
setf   /T0  T1
setf    T0 /T1
setf    T0 /T1
setf   /T0  T1
setf   /T0  T1
setf    T0 /T1
setf    T0 /T1
setf   /T0 /T1
setf   /T0 /T1
setf    T0  T1
setf    T0  T1
setf   /T0 /T1
setf   /T0 /T1
setf   /T0  T1
setf   /T0 /T1
setf    T0 /T1
setf    T0 /T1
setf    T0  T1
setf    T0  T1
setf    T0 /T1
setf    T0 /T1
setf   /T0 /T1
setf   /T0 /T1
setf   /T0  T1
setf   /T0  T1
setf    T0  T1
setf    T0  T1
setf   /A23 /A22 /A21 /A20 /DEMO /STRB RW
setf   /A23 /A22 /A21 /A20 /DEMO /STRB/RW
setf   /A23 /A22 /A21 /A20
setf   /A23 /A22 /A21 /A20 /DEMO /STRB RW
setf   /A23 /A22 /A21  A20
setf   /A23 /A22  A21 /A20
setf   /A23 /A22  A21  A20
setf   /A23  A22 /A21 /A20
setf   /A23  A22 /A21  A20
setf   /A23  A22  A21 /A20
setf   /A23  A22  A21  A20
setf    A23 /A22 /A21 /A20
setf    A23 /A22 /A21  A20
setf    A23 /A22  A21 /A20
setf    A23 /A22  A21  A20
```

Appendix C

TLC32040 Data Sheet

Appendix C contains the TLC32040 data sheet. This data sheet provides all specifications of the analog interface circuit used by the ’C3x DSK.

Appendix D

Glossary

A

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

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

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

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

B

batch file: A file that contains DOS commands for the PC to execute.

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

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

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

C

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

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

command-line cursor: A block-shaped cursor that identifies the current character position on the command line.

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

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

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

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

D

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

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

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

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

DSP: Digital signal processing.

E

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

entry point: The starting execution point in target memory.

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

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

F

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

G

global: A kind of symbol that is either: 1) defined in the current module and accessed in another or 2) accessed in the current module but defined in another.

I

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

L

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

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

LSB: Least significant bit.

LSByte: Least significant byte.

M

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

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

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

MSB: Most significant bit.

MSByte: Most significant byte.

N

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

O

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

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

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

P

PC: Personal computer or program counter, depending on the context and how it's used. In this book, 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.

parallel port: The parallel printer port interface is primarily used for connecting printers to the computer system, although the parallel port can also be used for other peripherals. In this case, the 'C3x DSK is connected to the parallel printer port.

R

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

S

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

serial port: The serial port that the DSK uses for communicating with the analog interface circuit (AIC). The port address is selected, based on which communcation port the AIC is attached to.

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

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

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

V

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

W

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

word: A 32-bit addressable location in target memory.

This template is for the "See" and "See also" references in your index. Since these entries do not have a page number associated with them, it's extremely difficult to locate one if you need to modify or delete it and you don't remember which chapter it's in. By using this template, you can alphabetize your entries according to the first letter of the first level entry.

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Index

? debugger option, 7-2

; in assembly language source, 5-7

$ symbol for SPC, 5-11

@ operand prefix, 5-5

* in assembly language source, 5-7

* operand prefix, 5-5

A

absolute address
  definition D-1

adding a software breakpoint 7-9

AIC
  hardware interface 4-6

AIC Initialization 4-14
  AIC reset 4-14
  'C31 timer initializing 4-14 to 4-15
    example code 4-15
  initializing AIC 4-16 to 4-17
  primary communications 4-17 to 4-18
  secondary communications 4-18 to 4-22
    control register bit fields 4-19
    data format 4-18
  serial port initializing 4-15 to 4-16

.align directive 6-11, 6-14

assembler 3-4, 5-15
  –l option 3-5
  constants 5-8
  definition D-1
  description of 3-2
  key features 3-2
  options 5-15
  source
    listings 5-2
  source statement format 5-2
  symbols 5-11

assembler directives
  aligning the section program counter 6-11
  alphabetical reference 6-13 to 6-32
  conditional assembly 6-10
  defining assembly-time symbols 6-11
  defining sections 6-5 to 6-7
  enabling conditional assembly
    .endloop 6-24
    .loop 6-24
  initializing constants 6-8 to 6-9
  miscellaneous 6-12
  referencing other files 6-9
  summary table 6-2

assembling your program 5-15

assignment statement
  definition D-1

autoexec.bat file
  definition D-1

B

BA command 7-9

batch files
  definition D-1

BD command 7-9

binary integers 5-8

BL command 7-9

block
  definition D-1

block diagram of TMS320C3x DSK 1-3

board requirements 2-2

breakpoints. *See* software breakpoints

breakpoints (hardware)
  definition D-1

breakpoints (software)
  definition D-1

.brstart directive 6-12, 6-15

byte
  definition D-1

.byte directive 6-8, 6-16

C

c or com debugger option 7-3
'C31 serial port
  initializing 4-15 to 4-16
'C31 timer
  initializing 4-14 to 4-15
    example code 4-15
  maximum timer period register value 4-15
  minimum timer period register value 4-14
cable requirements 2-2
character
  constants 5-9
circuit diagram 4-3
clearing software breakpoints 7-9
code-display windows
  definition D-2
COFF
  definition D-2
command line
  definition D-2
comment
  definition D-2
comments 5-7 to 5-18
  in assembly language source code 5-7
communications kernel 4-8 to 4-13
  commands 4-9
  data packets 4-8
    structure 4-8
  debugging functions 4-10 to 4-13
    flow diagram 4-12
    pipeline flow 4-13
  source code A-1
conditional assembly 6-10
conditional block
  definition D-1
config.sys file 2-5
connecting the DSK 2-4
constant
  definition D-2
constants 5-8, 5-11
  assembly-time 5-8
  binary integers 5-8
  character 5-9
  decimal integers 5-8
constants (continued)
  hexadecimal integers 5-8
  symbols as 5-8
.copy directive 6-9, 6-17
cursors
  command-line cursor
  definition D-2

D

D_DIR environment variable
  definition D-2
.data directive 6-5, 6-18
data packets 4-8
  structure 4-8
debugger
  definition D-2
  description of 3-2 to 3-3
  display
    basic 3-3
  key features 3-3
  options 7-2
    ? 7-2
    c or com 7-3
    h 7-2
debugging functions
  communications kernel 4-10 to 4-13
  single-step flow diagram 4-12
  single-step pipeline flow 4-13
decimal integer constants 5-8
developing code 3-4
directives
  assembler
    binary integers 5-8
    character constants 5-9
    hexadecimal integers 5-8
  definition D-2
disassembly
  definition D-2
display directory
  function key method 7-13
display requirements 2-2
driver.cpp 4-23
DSK
  schematics B-5
DSK assembler
  using 5-1 to 5-18
DSK host software 4-23

DSK overview 1-3
dsk3a.exe command 2-3
dsk3d.exe command 2-3
dska command 3-5, 5-15
dskd command 3-5, 7-2
DSP
  defined D-2

E

EGA
  definition D-3
.else directive 6-10, 6-23
.end directive 6-12, 6-19
.endif directive 6-10, 6-23
.endloop directive 6-10, 6-24
.entry directive 6-11, 6-20
entry point
  definition D-3
execute program to breakpoint
  function key method 7-13
external symbol
  definition D-3

F

file header
  definition D-3
.fill directive 6-8, 6-29
.float directive 6-8, 6-21
.float16 directive 6-8, 6-21
.float8 directive 6-8, 6-21
functional overview 4-1

G

GET DEBUG_CTXT D-26
getmem D-24
getting started 3-5
global symbol, definition D-3

H

h debugger option 7-2
HALT_CPU D-25
hardware
  checklist 2-2
hardware component overview B-3
hardware interface 4-2
  AIC 4-6
  host 4-2 to 4-3
  host communications 4-4 to 4-5
  memory map 4-7
  TLC32040 4-6
hardware overview 4-1
hardware requirements
  optional 2-3
hexadecimal integers 5-8
host requirements 2-2
host software 4-23

I

.ieee directive 6-8, 6-22
.if directive 6-10, 6-23
.include directive 6-9, 6-17
Init_System D-29
input section
  definition D-3
input_rdy D-27
installing the DSK software 2-1 to 2-10
  instructions 2-5
  possible errors 2-8
.int directive 6-8, 6-16
introduction 1-1
invoking
  assembler 5-15

K

key features of the DSK 1-2

L

–l option 3-5
label
  definition D-3
labels 5-3 to 5-18
  case sensitivity 5-3
  in assembly language source 5-2
  syntax 5-2
LF_Cmd D-29

listing file
  definition D-3
listing software breakpoints 7-9
.long directive 6-8, 6-16
.loop directive 6-10, 6-24
LSB
  defined D-3
LSByte
  defined D-3

M

member
  definition D-4
memory map 4-7
  definition D-4
memory requirements 2-2
miscellaneous files 2-3
mnemonic
  definition D-4
mnemonic field 5-4
  syntax 5-2
MSB 5-3
  definition D-4
MSb
  definition D-4

N

named section
  definition D-4

O

object file
  definition D-4
object.cpp 4-23
opcodes
  defining 5-4 to 5-18
operand
  definition D-4
operands 5-5
  label 5-11
  prefixes 5-5
operating system 2-3
options
  assembler 5-15
  debugger 7-2
  definition D-4
overview
  DSK system 1-3

P

PATH statement 2-6
PC
  definition D-4
.pfloat16 directive 6-21
.pfloat8 directive 6-8, 6-21
power requirements 2-2
predefined symbols 5-11
primary communications 4-17 to 4-18
print screen
  function key method 7-13
program
  assembling 5-15
  entry point
    definition D-3
putmem D-24

Q

.qxx directive 6-8, 6-25

R

raw data
  definition D-5
recv_long D-28
recv_long_byte D-27
required files 2-3
reset D-27
RUN_CPU D-25

S

schematics B-5
.sdef directive 6-11, 6-26
secondary communications 4-18 to 4-22
  control register bit fields 4-19
  data format 4-18

.sect directive 6-5, 6-27
section
  definition D-5
section program counter. *See* SPC
serial port
  definition D-5
  identifying 7-3
.set directive 6-11, 6-28
single-step
  definition D-5
singlestep
  function key method 7-13, 7-14
single-step flow diagram 4-12
single-step pipeline flow 4-13
software breakpoints 7-9
  BA command 7-9
  BD command 7-9
  BL command 7-9
  clearing 7-9
  listing 7-9
  setting 7-9
software checklist 2-3
source
  listings 5-2
  statement
    format 5-2
      comment field 5-7
      label field 5-3
      mnemonic field 5-4
      operand field 5-5
    number (source listing) 5-2
source file
  definition D-5
source files 5-2 to 5-7
  commenting 5-7 to 5-18
  labeling 5-3 to 5-18
  opcodes 5-4 to 5-18
.space directive 6-9, 6-29
SPC
  assigning a label to 5-3
  value
    associated with labels 5-3
SSTEP_CPU D-25
.start directive 6-12, 6-30
.string directive 6-9, 6-31
symbol
  definition D-5
symbolic constants 5-11
symbols 5-11
  predefined 5-11

T

target.cpp 4-23
.text directive 6-5, 6-32
timer period register value
  maximum 4-15
  minimum 4-14
TLC32040
  hardware interface 4-6
TLC32040 AIC initialization 4-14 to 4-22
  See also AIC initialization

V

VGA
  definition D-5

W

windows
  definition D-5
word
  definition D-5
.word directive 6-9, F-16

X

xmit_long D-28