USER'S GUIDE

ZSP™ SDK Software Development Kit

April 2002

Revision 4.0

LSI LOGIC®

This document contains proprietary information of LSI Logic Corporation. The information contained herein is not to be used by or disclosed to third parties without the express written permission of an officer of LSI Logic Corporation.

DB15-000126-06, Third Edition (April 2002)
This document describes Rev. 4.0 of LSI Logic Corporation's ZSP™ SDK Software Development Kit and will remain the official reference source for all revisions/releases of this product until rescinded by an update.

LSI Logic Corporation reserves the right to make changes to any products herein at any time without notice. LSI Logic does not assume any responsibility or liability arising out of the application or use of any product described herein, except as expressly agreed to in writing by LSI Logic; nor does the purchase or use of a product from LSI Logic convey a license under any patent rights, copyrights, trademark rights, or any other of the intellectual property rights of LSI Logic or third parties.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

TRADEMARK ACKNOWLEDGMENT
The LSI Logic logo design and ZSP are trademarks or registered trademarks of LSI Logic Corporation. Microsoft, Microsoft Access, MS-DOS, Windows, and Windows NT are registered trademarks of Microsoft Corporation. UNIX is a registered trademark of X/Open Company, Ltd. Solaris is a trademark of Sun Microsystems, Inc. All other brand and product names may be trademarks of their respective companies.

For a current list of our distributors, sales offices, and design resource centers, view our web page located at

http://www.lsilogic.com/contacts/na_salesoffices.html

Preface

This book is the primary reference and user's guide for the ZSP™ SDK Software Development Kit. The SDK supports digital signal processors based on the ZSP400 core (for example, the LSI402ZX and LSI403Z) and the next generation ZSP G2 architecture.

Audience

This document assumes that you have some familiarity with the C language, and with the ZSP architecture and assembly language. Those who will benefit from this book are

- Engineers and managers who are evaluating the ZSP processor for possible use in a system
- Engineers who are designing products based on the ZSP architecture and wish to perform cost and performance analysis
- Engineers who are developing software for systems based on the ZSP architecture

Organization

This document has the following chapters and appendices:

- Chapter 1, **Introduction**, introduces the ZSP SDK software development kit.
- Chapter 2, **Installation**, describes how to install the SDK.
- Chapter 3, **C Cross Compiler**, describes the SDK C compiler.
- Chapter 4, **Assembler**, describes the assembler in the SDK tool set.
- Chapter 5, **Linker**, describes the linker in the SDK tool set.

- Chapter 6, **Utilities**, describes miscellaneous utilities in the SDK tool set.
- Chapter 7, **ZISIM Simulator**, describes the SDK functional-accurate simulator.
- Chapter 8, **ZSIM Simulator**, describes the SDK cycle-accurate simulator.
- Chapter 9, **Debugger**, describes the SDK debugger.
- Chapter 10, **ZSP Integrated Development Environment (ZSP IDE)**, describes the SDK Project Manager provided by LSI Logic with Windows 95/98/NT versions of the SDK.
- Chapter 11, **ZSP IDE Debugger**, describes the GUI Debugger provided by LSI Logic with Windows 95/98/NT versions of the SDK.
- Appendix A, **Example Programs**, provides a sample program for use with the SDK.
- Appendix B, **ZSP400 Control Registers**, lists the ZSP400 control registers.
- Appendix C, **ZSPG2 Control Registers**, lists the ZSPG2 control registers.
- Appendix D, **L-Intrinsic Functions**, describes the L-Intrinsic functions supported by the SDK compiler.
- Appendix E, **Signal Processing Library**, describes the `libalg.a` library.

Related Publications

LSI402ZX Digital Signal Processor User's Guide, LSI Logic Corporation, order number R14021. Provides detailed information on the LSI402ZX Digital Signal Processor.

LSI403Z Digital Signal Processor User's Guide, LSI Logic Corporation, order number R14025. Provides detailed information of the LSI403Z digital Signal Processor.

ZSP400 Digital Signal Processor Architecture Technical Manual, LSI Logic Corporation, order number I14036. Provides detailed information on the registers and instruction set defined by the ZSP architecture and implemented in the LSI4xx family of processors.

Using and Porting GNU CC, by Richard M. Stallman, Free Software Foundation, June 1996. Provides detailed information on how to use GCC, which is the foundation of SDCC.

Using AS: The GNU Assembler, by Dean Elsner, et. al., Free Software Foundation, January 1994. Provides detailed information on how to use AS, which is the foundation of SDAS.

Using LD: The GNU Linker, by Steve Chamberlain, Free Software Foundation, January 1994. Provides detailed information on how to use LD, which is the foundation of SDLD.

Debugging with GDB: The GNU Source Level Debugger, by Richard Stallman, *et*. *al*., Free Software Foundation, January 1994. Provides detailed information on how to use GDB, which is the foundation of SDBUG.

EB402 Evaluation Board Getting Started, LSI Logic Corporation, order number DBO6-000264-01, September, 2000. Provides information on using the EB402 Evaluation Board.

EB402 Evaluation Board User's Guide, LSI Logic Corporation, order number DB15-000143-00, September, 2000. Provides detailed information on how to use the EB402 Evaluation Board.

PCMCIA-1149.1 Windows 95/NT Software Development Kit User's Guide, Corelis, Inc. Provides detailed information on using the JTAG interface.

Man pages for `ar`, `nm`, `objdump`, `string`, `size`, `objcopy`, `strip` and `ranlib` from the Free Software Foundation, available from the FTP site `prep.ai.mit.edu`.

Conventions Used in This Manual

The first time a word or phrase is defined in this manual, it may be *italicized.*

Hexadecimal numbers are indicated by the prefix “0x”, for example, 0x32CF. Binary numbers are indicated by the prefix “0b”, for example, 0b0011.0010.1100.1111.

The term ‘DOS’, unless otherwise noted, includes the MS-DOS operating system and its Windows 3.1, Windows 95, Windows 98, and Windows NT supersets.

The term ‘PC’, unless otherwise noted, includes the 386-, the 486-, and the Pentium-based IBM-PC or compatible host computers.

Additional notational conventions used throughout this manual are listed below.

Notation	**Example**	**Meaning and Use**
courier typeface	`.nwk` file	Names of commands, files, directories, and code are shown in courier typeface
bold typeface	**fd1sp**	In a command line, command keywords are shown in bold, nonitalic courier typeface. Enter them exactly as shown, including case.
italics	*module*	In command lines and syntax descriptions, italics indicate user-defined variables of a type defined by the italicized noun. Italicized text must be replaced with appropriate user-specified items.
italic underscore	*full_pathname*	When an underscore appears in an italicized string, enter a user-supplied item of the type called for, without spaces.
brackets	[*version*] [*filename* | *register*]	In command formats, you may, but need not, enter an item enclosed within brackets. When vertical bars are used within brackets, you may select one (but not more than one) of the items separated by bars. Do not enter the brackets or bar.

braces	{ *directory* }
	{ *filename* | *register* }
		In command formats, you must select one (but not more than one) item enclosed within braces. Do not enter the braces. When vertical bars are used within braces, you may select one (but not more than one) of the items separated by braces. Do not enter the braces or bar.

ellipses	*option*...	In command formats, elements preceding ellipses may be repeated any number of times. Do not enter the ellipses. In menu items, if an ellipsis appears after an item, clicking that item brings up a dialog box.

Notation	Example	Meaning and Use
vertical dots	. . .	Vertical dots indicate that a portion of a program or listing has been omitted from the text.
semicolon, and other punctuation	;	Use as shown in the text.

Contents

Appendix E
Signal Processing Library

Figures

Tables

Chapter 1
Introduction

The ZSP Software Development Kit (ZSP SDK) from LSI Logic supports all aspects of software development for systems incorporating devices based on the ZSP400 and ZSPG2 architectures. The ZSP SDK includes an optimizing C cross compiler, assembler, and linker, both a functional-accurate simulator and a cycle-accurate simulator, and a source- and assembly-level debugger.

The ZSP SDK is available for Windows 95, Windows 98, Windows NT, and Solaris platforms. For the Windows platforms only, the software development tools can be used in the context of the SDK Integrated Development Environment (IDE), which includes a project manager and a GUI debugger. The GUI debugger provides a graphical environment for developing and debugging your code, with interactive viewing and control of project source files and run-time data.

1.1 Overview of the SDK Tools

The ZSP SDK tools are all placed under one directory which is referred to with the environment variable SDSP_HOME. The sdspI subdirectory contains all tools related to the ZSP400 architecture. The zspg2 subdirectory contains all tools related to the ZSPG2 architecture. There are no dependencies between the two directories. Users who only need tools for the ZSP400 can delete the zspg2 subdirectory. Likewise, users who only need tools for the ZSPG2 can delete the sdspI subdirectory. The two subdirectories closely mirror one another. Both have the following subdirectories: bin, lib, include, misc, tmp. The bin directories contain the command-line tools. The lib directories contain the C libraries. The include directories contain the C header files. The misc directories contain auxilary files. The tmp directories are used by the tools for temporary space. The GNU based tools for the ZSP400 all have a “sd” prefix. The analgous tools for ZSPG2 all have a “zd” prefix. In addition the assembly optimizer, sdopt/zdopt, also follows this prefix convention. The simulators do not follow this convention. The ZSP400 simulators are: zsim400 and zisim400. The ZSPG2 simulators are: zsimg2 and zisimg2.

The ZSP SDK also supports users who want to take assembly and C code written for the ZSP400 architecture and run it without modification on the ZSPG2 architecture. The compiler zdxcc compiles for a ZSPG2 target but uses the calling convention and pointer sizes designed for the ZSP400. The zspg2 directory also contains a subdirectory libg1g2, which contains C libraries for zdxcc. There is also a debugger, zdxbug, for debugging code developed with zdxcc.

The ZSP SDK tools are based on the GNU tools from the Free Software Foundation, Inc. The GNU project has well-proven software development tools that have been successfully ported to many different target machines and platforms. Documentation for the GNU project tools can be obtained from the web site www.gnu.org and the FTP site `prep.ai.mit.edu`. To gain access to the FTP site, log in as ‘anonymous’ and type your e-mail address as the password. The descriptions of the tools in this document, for the most part, include only the differences from their GNU counterparts (refer to Table 1.1).

Table 1.1 SDK Tools and GNU Counterparts

Tool	GNU Equivalent	Function
sdcc zdcc zdxcc	gcc	Compiles
sdas zdas	as	Assembles
sdld zdld	ld	Links
sdbug400 zdbug zdxbug	gdb	Debugs

The GNU tools have been enhanced so as to take advantage of the many high-performance features of the ZSP LSI402ZX and LSI403Z devices and ZSP400-based ASICs, such as single-cycle multiply-accumulate instructions, fast context switching, circular buffer support, single-cycle compare/select, and zero-overhead loops.

The SDK provides utilities for manipulating the files that are generated by the tools during project creation. These SDK-specific utilities, described in Table 1.2, replace their GNU counterparts.

Table 1.2 SDK Utilities and GNU Counterparts

Utility	GNU Equivalent	Function
`sdar` `zdar`	`ar`	Creates, modifies, and extracts files from an archive.
`sdnm` `zdnm`	`nm`	Lists symbols from object files.
`sdobjdump` `zdobjdump`	`objdump`	Displays information from object files.
`sdranlib` `zdranlib`	`ranlib`	Generates an index for an archive.
`sdstrings` `zdstrings`	`strings`	Prints the printable characters in the files.
`sdsize` `zdsize`	`size`	Lists section sizes and total size of object file.
`sdstrip` `zdstrip`	`strip`	Discards symbols from object files.
`sdobjcopy` `zdobjcopy`	`objcopy`	Copies and translates object files.

The SDK Tools also includes the following non-GNU-based tools:

- The compiler's optimizer, sdopt/zdopt/zdxopt, performs additional optimizations to those performed by SDCC/ZDCC/ZDXCC.
- Both the functional-accurate and cycle-accurate simulators are provided in a standalone form that support a simple command-line interface and that can be provided in a dynamic linkable format that can be used in conjuction with the debugger.
- For Windows platforms only, the GUI tools include an IDE and a GUI Debugger.

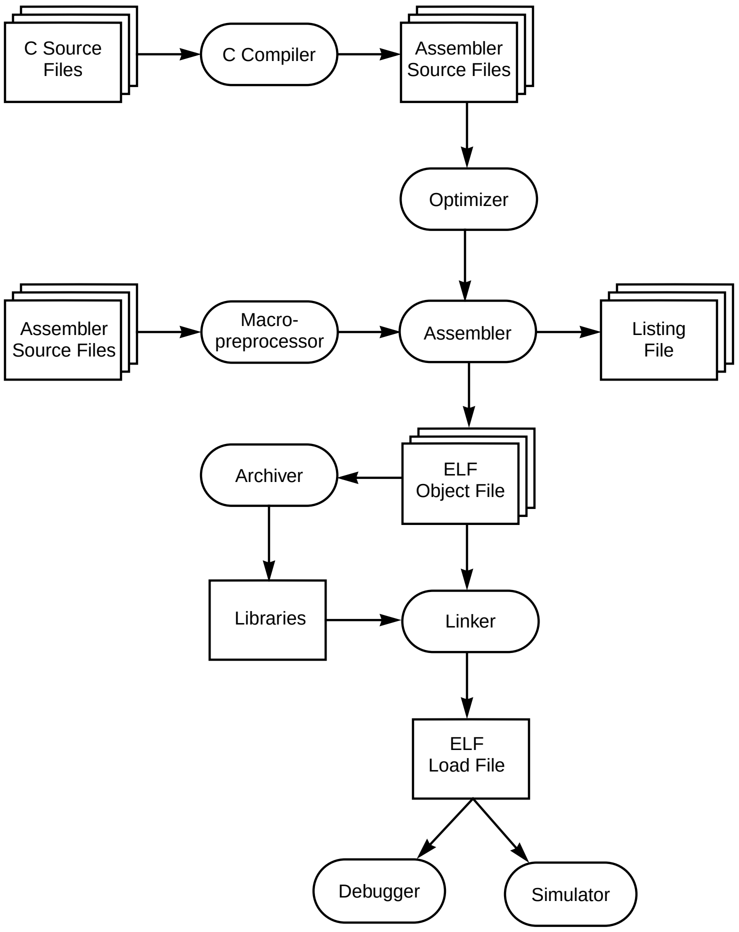
For Solaris platforms, there are freely-available GUI front ends that do not have all the capabilities of the GUI supplied by LSI Logic for Windows platforms, but that can be configured for use with all the LSI Logic ZSP SDK Tools.

1.2 Overview of Software Development Using the SDK Tools

An overview of the software development process utilizing the SDK tools is shown in Figure 1.1. As shown in the figure, the compiler accepts C source files (`.c`) and produces a relocatable assembly language source module (`.s`). The assembly source file is input to the assembler, which translates the module into a relocatable object file in the Executable and Linkable Format (ELF) file format (`.obj` (Windows) or `.o` (UNIX)). ELF files are then linked with other ELF files (for example, library files) to produce a single executable ELF load file (`a.out`). The load file can be loaded into host memory for symbolic simulation/debugging, or it can be downloaded to a ZSP architecture-based target system for real-time debugging.

On Windows 95/98/NT and Solaris platforms, the tools can be accessed using the standard GNU command-line interface, as described in Chapter 3, "C Cross Compiler" through Chapter 9, "Debugger." On Windows 95/98/NT platforms, the tools can also be accessed using the the ZSP Integrated Development Environment (ZSP IDE), (Chapter 10, "ZSP Integrated Development Environment (ZSP IDE)"), and the ZSP IDE Debugger (Chapter 11, "ZSP IDE Debugger").

Figure 1.1 Overview of Software Development



Chapter 2
Installation

This chapter describes how to install the ZSP Software Development Kit.

The SDK is available for Windows 95, Windows 98, Windows NT, and Solaris. The media used to provide the tools is a CD-ROM.

2.1 Contents of the CD-ROM

The CD-ROM has the following five top-level directories:

- doc

  Contains the complete documentation for the SDK tools and the GNU tools.
- `examples`

  Contains example code for the SDK tools.
- `solaris`

  Contains initialization code and tools for installing the SDK on the Solaris platform.
- `windows`

  Contains the initialization code and tools for installing the SDK on Windows 95/98/NT platforms, and examples that can be added to an ZSPIDE project.

2.2 Installation on Windows Systems

The minimum system requirements for the SDK tools are

- a Pentium Pro-based PC

- 64 Mbytes of RAM
- 48 Mbytes of Disk Space

On Windows NT systems, you must have administrator privileges to install the ZSP SDK Tools.

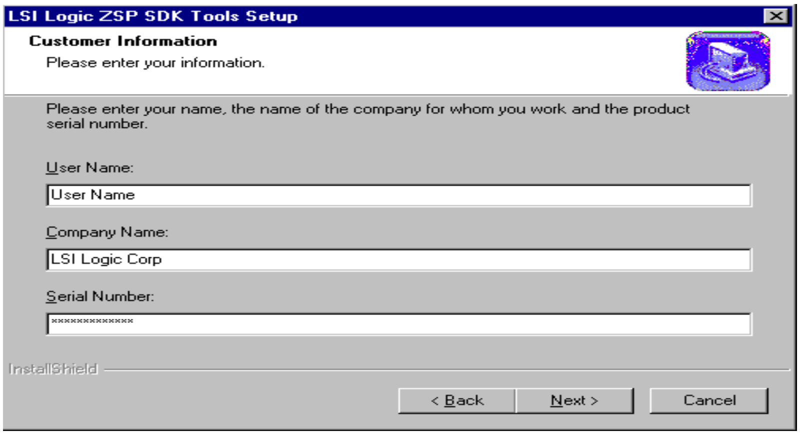
2.2.1 Installing SDK Tools

Before you install the SDK tools, make sure you have uninstalled any older version. Refer to Section 2.3, "Uninstalling the SDK Tools on Windows Systems."

Step 1. Insert the CD-ROM in the CD drive, click Add/Remove Program on the Control Panel, then click Install and select `D:\windows\Setup.exe`.

Step 2. Follow the Setup Instructions.

Step 3. A dialog box will be displayed for entering the serial Number.


LSI Logic ZSP SDK Tools Setup

Customer Information

Please enter your information.

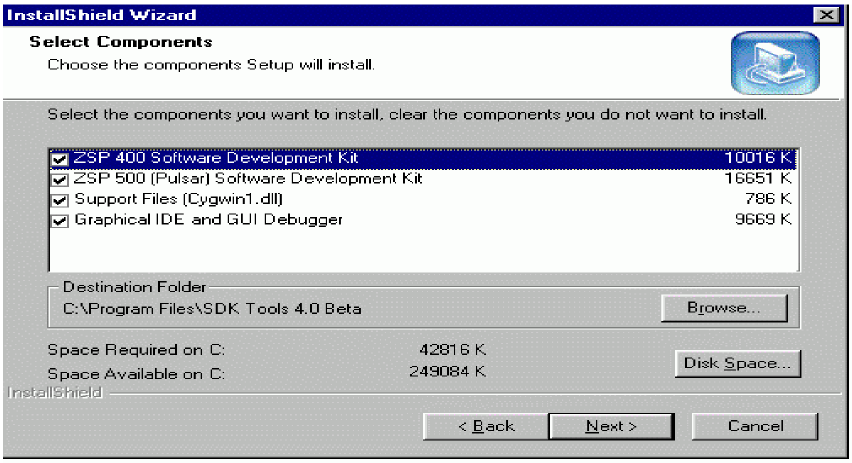
Please enter your name, the name of the company for whom you work and the product serial number.

User Name: User Name

Company Name: LSI Logic Corp

Serial Number: **************

Step 4. Type the serial Number, and then click on the Next button. The dialog box shown below will be displayed.


InstallShield Wizard

Select Components

Choose the components Setup will install.

Select the components you want to install, clear the components you do not want to install.

✓ ZSP 400 Software Development Kit	10016 K
✓ ZSP 500 (Pulsar) Software Development Kit	16651 K
✓ Support Files (Cygwin1.dll)	786 K
✓ Graphical IDE and GUI Debugger	9669 K

Destination Folder
C:\Program Files\SDK Tools 4.0 Beta

Space Required on C: 42816 K
Space Available on C: 249084 K

Step 5. The default directory is `C:\Program Files\SDK Tools<Version Number>\`. You can change the default directory by clicking on the Browse button, specifying a directory name, and then clicking OK.

The Setup program installs the SDK files in the selected directory. When the setup is complete, a dialog box is displayed confirming successful setup.

By default, the Setup program installs the files listed below in *`C:\Installation_Directory`*\sdspI\bin, where *`Installation_Directory`* is the directory specified in Step 5

Filename	**Function**
`elfread.exe`	Produces a simple dump of entire contents of an object file
`libzisim400.dll`[1]	Dynamic link library used in sdbug400 for target zisim
`libzsim400.dll`[1]	Dynamic link library used in sdbug400 for target zsim
`libzperiph.dll`[1]	Dynamic link library used in sdbug400 for target zsim
(Sheet 1 of 2)	

Filename	**Function**
sdar.exe	Archive utility
sdas.exe	Assembler
sdbug400.exe	Source-level debugger for ZSP400-based Devices
sdcc.exe	Driver
sdcc1.exe[1]	Compiler
sdcpp.exe[1]	Preprocessor
sdld.exe	Linker
sdnm.exe	Symbol listing utility
sdobjcopy.exe	Object file copying utility
sdobjdump.exe	Object dump utility
sdopt.exe[1]	Optimizer
sdranlib.exe	Ranlib utility
sdsize.exe	Size utility
sdstrings.exe	String print utility
sdstrip.exe	Symbol discarding utility
zisim400.exe	Functional-accurate simulator for ZSP400-based devices
zsim400.exe	Cycle-accurate simulator for ZSP400-based devices
(Sheet 2 of 2)	

1. These files are not intended to be used on the command line, but instead are always called by other functions.

The libraries listed below are installed by default in the directory `C:Installation_Directory\sdspI\lib`.

Filename	Function
`crt0.obj`	Startup file
`libc.a`	C library
`libg.a`	C library with debug information
`liblongc.a`	C library with long calls.

The header files listed below are installed by default in the directory `C:\Installation_Directory\sdspI\include`.

Filename	Function
`cbuf.h`	Circular buffer
`ctype.h`	Standard header file
`creg.h`	Control registers
`dsp.h`	L-Intrinsics
`libsdsp.h`	SDSP-specific printing
`limits.h`	Standard header file
`N_Intrinsic.h`	N-Intrinsics
`q15.h`	Support file
`setjmp.h`	Standard header file
`simios.h`	Standard header file
`stdarg.h`	Standard header file
`stddef.h`	Standard header file
`stdio.h`	Standard header file
`stdlib.h`	Standard header file
`string.h`	Standard header file

By default, the Setup program installs the files listed below in *C:\Installation_Directory*\zspg2\bin, where *Installation_Directory* is the directory specified in Step 5

Filename	**Function**
elfread.exe	Produces a simple dump of entire contents of an object file
libzisimg2.dll[1]	Dynamic link library used in zdbug and zdxbug for target zisim
libzidlmssg2.dll[1]	Dynamic link library used in zdbug and zdxbug for target zisim
zdar.exe	Archive utility
zdas.exe	Assembler
zdbug.exe	Source-level debugger for ZSP500-based Devices
zdxbug.exe	Source-level debugger for ZSP500-based devices running code designed for ZSP400
zdcc.exe	Compiler
zdxcc.exe	Cross (“X”) compiler for ZSP400 to ZSP500
zdcc1.exe[1]	Compiler Driver for zdcc
zdxcc1.exe[1]	Compiler Driver for zdxcc
zdcpp.exe[1]	Preprocessor
zdxcpp.exe[1]	Preprocessor for zdxcc
zdld.exe	Linker
zdnm.exe	Symbol listing utility
zdobjcopy.exe	Object file copying utility
zdobjdump.exe	Object dump utility
zdopt.exe[1]	Optimizer
zdxopt.exe[1]	Optimizer for ZSP400 to ZSP500 code.
zdranlib.exe	Ranlib utility
(Sheet 1 of 2)	

Filename	Function
`zdsize.exe`	Size utility
`zdstrings.exe`	String print utility
`zdstrip.exe`	Symbol discarding utility
`zisimg2.exe`	Functional-accurate simulator for ZSP400-based devices
(Sheet 2 of 2)	

The libraries listed below are installed by default in the directory *`C:Installation_Directory`*\zspg2\\`lib`.

Filename	Function
`crt0.obj`	Startup file
`libc.a`	C library
`libg.a`	C library with debug information

The libraries listed below are installed by default in the directory *`C:Installation_Directory`*\zspg2\\`lib`.g1g2

Filename	Function
`crt0.obj`	Startup file
`libc.a`	C library
`libg.a`	C library with debug information
`libalg.a`	Basic signal processing functionality

The header files listed below are installed by default in the directory *C:\Installation_Directory*\zspg2\include.

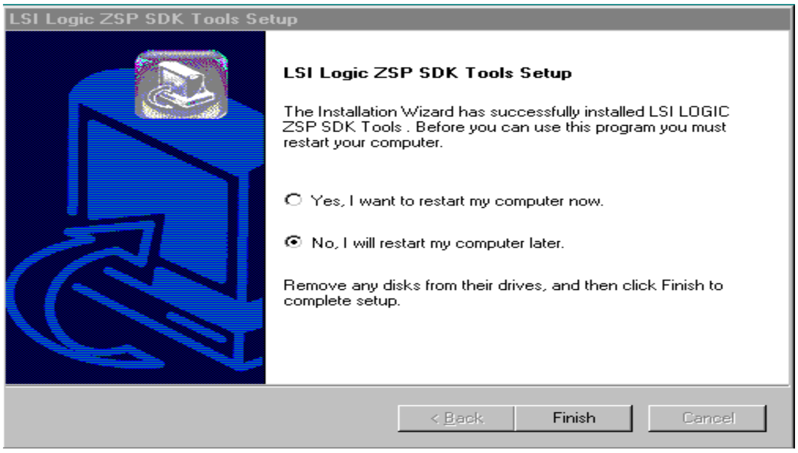
Filename	**Function**
cbuf.h	Circular buffer
ctype.h	Standard header file
creg.h	Control registers
dsp.h	L-Intrinsics
libsdsp.h	SDSP-specific printing
limits.h	Standard header file
N_Intrinsic.h	N-Intrinsics
q15.h	Support file
setjmp.h	Standard header file
simios.h	Standard header file
stdarg.h	Standard header file
stddef.h	Standard header file
stdio.h	Standard header file
stdlib.h	Standard header file
string.h	Standard header file

The files listed below are installed by default in the directory *C:Installation_Directory*\ide\bin

Filename	**Function**
en.rc	IDE menu resource
guidebug.exe	GUI debugger frontend.
zspide.exe	IDE for the ZSP family of processors.

2.2.2 Restarting Windows

The Setup program installs system files and updates some shared files that are required for running the SDK tools. The system prompts you to reboot after you have installed the SDK tools.



LSI Logic ZSP SDK Tools Setup

LSI Logic ZSP SDK Tools Setup

The Installation Wizard has successfully installed LSI LOGIC ZSP SDK Tools . Before you can use this program you must restart your computer.

○ Yes, I want to restart my computer now.

● No, I will restart my computer later.

Remove any disks from their drives, and then click Finish to complete setup.

Click Finish to exit from the Setup program.The system will be restarted according to the option selected in the above dialog box.

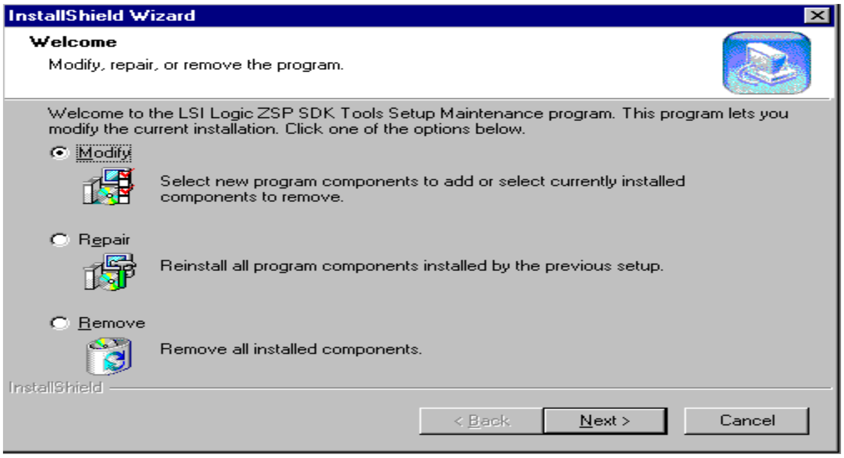
2.3 Uninstalling the SDK Tools on Windows Systems

Perform the following steps to uninstall the SDK tools:

Step 1. Open the Control Panel window. (The Control Panel is accessed by clicking on the Start menu, then selecting Settings, then selecting Control Panel.

Step 2. In the Control Panel window, double-click on Add/Remove Program.

Step 3. Then select the LSI LOGIC SDK tools and click on Add/Remove. In the dialog box shown below, click on Remove to uninstall the tools.


InstallShield Wizard

Welcome

Modify, repair, or remove the program.

Welcome to the LSI Logic ZSP SDK Tools Setup Maintenance program. This program lets you modify the current installation. Click one of the options below.

[✓ Modify]

Select new program components to add or select currently installed components to remove.

Repair

Reinstall all program components installed by the previous setup.

Remove

Remove all installed components.

2.4 Installation on Solaris Systems

The ZSP SDK may be hosted on the Solaris 2.6 operating system and later versions.

Step 1. If your Solaris system has `vold`, it will automatically mount the CD-ROM after it has been inserted. To access the CD-ROM, change the directory to `/cdrom/SDK`.

Step 2. If `vold` is not running, mount the CD-ROM and enter the following command:

```
mount /dev/sr0 /mnt/cdrom
```

Step 3. Use one of the following commands to invoke the installation script under `/cdrom/SDK/solaris` or `/mnt/cdrom/solaris`:

```
/cdrom/SDK/solaris/sdsp_install
```

or

```
/mnt/cdrom/solaris/sdsp_install
```

Step 4. Follow the directions given in the script.

Step 5. Specify an installation directory for the SDK tools. Assign the installation path to the `SDSP_HOME` environment variable, followed by a forward slash (`/`).

For example, if you install the tools in `MyInstallDirectory`, assign the directory to the `SDSP_HOME` variable:

```
SDSP_HOME = MyInstallDirectory/
```

Step 6. Export the `SDSP_HOME` variable.

Step 7. If you want to be able to invoke the SDK tools from any directory, add the installation directory to the path.

Step 8. In order to use the sdbug400 debugger, the environment variable LD_LIBRARY_PATH must be included in the installation path. Use the following command:

```
setenv LD_LIBRARY_PATH
${LD_LIBRARY_PATH}:$SDSP_HOME/sdspI/bin
```

The Setup program installs the SDK files.

The following files containing the tools are installed in the directory `$SDSP_HOME/sdspI/bin`.

Filename	**Function**
`elfread`	Produces a simple dump of entire contents of an object file
`sdar`	Archive utility
`sdas`	Assembler
`sdbug400`	Source-level Debugger for ZSP400
`sdcc`	Driver
`sdcc1`	Compiler
`sdcpp`	Preprocessor
`sdld`	Linker
`sdnm`	Symbol listing utility
`sdobjcopy`	Object file copying utility

Filename	**Function**
`sdobjdump`	Object dump utility
`sdopt`	Optimizer
`sdranlib`	Random library (ranlib) utility
`sdsize`	Size utility
`sdstrings`	String print utility
`sdstrip`	Symbol discarding utility
`zisim400`	Functional-accurate simulator for ZSP400-based Devices
`zsim400`	Cycle-accurate simulator for ZSP400-based Devices

The libraries listed below are installed in the directory `$SDSP_HOME/sdspI/lib`.

Filename	**Function**
`crt0.o`	Startup file
`libc.a`	C library
`libg.a`	C library with debug information

The header files listed below are installed in the directory `$SDSP_HOME/sdspI/include`.

Filename	**Function**
`cbuf.h`	Circular buffer
`ctype.h`	Standard header file
`dsp.h`	L-Intrinsics
`libsdsp.h`	SDSP-specific printing
`limits.h`	Standard header file

Filename	Function
`N_Intrinsic.h`	N-Intrinsics
`q15.h`	Support file
`setjmp.h`	Standard header file
`simios.h`	Standard header file
`stdarg.h`	Standard header file
`stddef.h`	Standard header file
`stdio.h`	Standard header file
`stdlib.h`	Standard header file
`string.h`	Standard header file

For both the Windows and Solaris setups, there are additional files and directories installed by the Setup program that are required for running the tools. For this reason, do not delete or modify any of the files or directories in the installation directory.

The ZSP SDK tools use the `tmp` directory, which is created during setup, to store temporary files.

The `misc` directory contains a file called `mapfile`. This file contains information about the scan chain of the target processor that is used for hardware-assisted debug with the JTAG port (on Windows platforms only). The correct map file is required for hardware-assisted debugging. The map file supplied with the ZSP SDK tools corresponds to the LSI402ZX rev2. If you are using a different ZSP400-based part, you must replace the map file installed during setup with the proper map file for your device.

Chapter 3
C Cross Compiler

This chapter describes the SDK C Cross Compiler and the compilation process.

The SDK C Cross Compilers; SDCC, ZDCC, and ZDXCC; are based on the GNU C compiler (GCC) from the Free Software Foundation. SDCC is the compiler for the ZSP400 architecture. ZDCC is the compiler for the ZSPG2 architecture. ZDXCC is targeted for the ZSPG2 architecture, but it uses the same calling convention and pointer size as SDCC. This allows C/assembly programs written for the ZSP400 architecture to be easily ported to the ZSPG2 architecture. CC will be used to refer to all three compilers. GCC is described in *Using and Porting GNU CC,* by Richard M. Stallman, Free Software Foundation, June 1996. The description of CC in this chapter, for the most part, includes only the differences from GCC.

The compiler is invoked from the shell using the following command:

```
cc [options] sourcefile
```

The command-line options and source file name extension determine the type of compilation. In the simplest case, with no options and a `.c` source file, the compiler will produce an executable, `a.out`.

3.1 Compiler Options

The CC compiler supports all GCC compiler options in addition to the SDK-specific options described in Table 3.1.

Table 3.1 Compiler Options

Option	Description	Availability
`-mlong_call`	The compiler generates code for a call instruction using a register operand. Use this option to resolve the limitation of the call immediate range.	SDCC ZDCC ZDXCC[1]
`-mno_sdopt`	The compiler disables the assembly optimizer, `sdopt/zdopt/zdxopt`. By default, the optimizer is automatically invoked at optimization levels -01, -02 and -03.	SDCC ZDCC ZDXCC
`-mlong_cond_branch`[2]	The compiler generates code for a conditional branch by using a register operand.	SDCC ZDXCC
`-mlong_uncond_branch`[3]	The compiler generates code for an unconditional branch by using a register operand.	SDCC ZDXCC
`-minfer_mac`	Enable the compiler to generate mac and macn instructions without using intrinsics. Use this option only if the code generated will be run with the sat, q15, sre and mre bits of %fmode turned off.	SDCC ZDXCC
`-mlarge_data`	Use large data model.	ZDCC

1. Since the range of a call immediate on ZSPG2 is 16-bits versus 13-bits on ZSP400, the -mlong_call option will be less frequently needed for ZDXCC and ZDCC.
2. This option is preserved for backward compatibility with previous versions of the SDK, but it is no longer needed, since the compiler will automatically use register based branches when needed. This option will be removed in a future version of the SDK.
3. (same as 2.)

The `-mlong_call` option is frequently necessary with SDCC because call-immediates on the ZSP400 architecture have a 13-bit range, and its use is therefore recommended for applications that are larger than the range of a call-immediate. The ZSPG2 architecture has a larger call immediate range (16-bits), so this option is not as critical for it. Better performance and code size can be obtained by selectively using the `-mlong_call` option, but this may require repetitive trial and error to determine which files can safely be converted to use call-immediates. One important exception is a file which does not call a function external to the file; in this case, the necessity of `-mlong_call` does not depend on other files or on the link order—this kind of file will either always require `-mlong_call` or it will never require it. Note that with SDCC, the use of `-mlong_call` does not guarantee that all calls will be long calls; that is, the assembly optimizer, `sdopt`, will convert long calls to call immediates if it can determine that such a conversion is safe. The assembly optimizer can be disabled by specifying the `-mno_sdopt` option; otherwise, it is automatically invoked when optimization is selected. Note that to debug optimized code, the `-mno_sdopt` option should be used, because the assembly optimizers perform optimizations that make debugging extremely difficult.

`sdopt` takes in assembly generated by the compiler proper and optimizes it to produce improved assembly. The optimizations that `sdopt` performs include simplification of constant generation, conversion of loops to use loop registers, dead code elimination, loop invariant code motion, conversion of long calls to call-immediate, instruction scheduling, and improved register utilization.

`zdopt` takes in assembly generated by ZDCC and optimizes it to produce improved assembly. The optimizations that `zdopt` performs include dead code elimination, loop invariant code motion, instruction scheduling, and improved register utilization.

ZDCC supports two models of memory. The default small memory model requires that data and bss sections be placed in the first 64K words of data memory. The large data model has no requirements on the size or placement of the data and bss sections. The large data model is selected with the "`-mlarge_data`" option. For both models, the pointer size is 32-bits. Both models allow stack and heap space to use all addressable memory. Code generated with the small data model will be more compact and have better performance than code generated with the large data model. The small data model allows a shorter instruction sequence to be used to access memory in the data or bss sections.

Some of the key options that control the compiler's output are shown in Table 3.2.

Table 3.2 Output Options

`-c`	Compile or assemble source files but do not link. Output file is named by replacing the suffix of the source file with '.o'.
`-o` *`file`*	Place output in *file*. This option is applicable whether the output is preprocessed C, assembly, an object file, or an executable.
`-E`	Stop after preprocessing. Output is sent to standard output.
`-S`	Stop after compilation. Do not assemble. Output file is named by replacing the '.c' suffix with '.s'.
`-save-temps`	Store the intermediate preprocessed C, assembly, and object files permanently. The names used for these intermediate files will be based on the name of the input file: compiling `foo.c` with `-save-temps` will produce `foo.i`, `foo.s`, and `foo.o`.
`-g`	Generate debugging information for use by the debugger.

The optimization levels supported by GCC are described in Table 3.3.

Table 3.3 Optimization Options

Option	**Description**
`-O0`	No optimization is performed. All variables are placed on the stack.
`-O1`	Only those optimizations that allow the debugger to behave as expected are performed.
`-O2`	Only those optimizations that do not greatly increase code size are performed. These optimizations include dead-code elimination, constant propagation, common subexpression elimination, and loop invariant code motion.
`-O3`	All optimizations performed at level -O2 are performed, as well as function inlining and loop unrolling.

3.2 Compiler Conventions

This section describes the software conventions defined by the SDK assembler and compiler. You must follow these conventions when writing assembly-language routines that will be called by your C program.

3.2.1 Data Type Conventions

The compiler's representation of C data types is summarized in Table 3.4. The `q15` data type can be printed by the `fprintf` and `printf` functions. The `%q` format specifier will print a 16-bit value in fixed-point notation. For example, the call:

```
printf("%q\n",0x6000);
```

will print:

```
0.75000
```

Table 3.4 Compiler's Representation of C Data Types

C Data Type	Representation
`char`	16 bits
`unsigned char`	16 bits
`int`	16 bits
`short int`	16 bits
`unsigned short int`	16 bits
`q15`	16 bits
`enum`	16 bits
`pointer`	16 bits with SDCC/ZDXCC 32 bits with ZDCC
`long`	32 bits
`unsigned long`	32 bits
`accum_a`	32 bits
`accum_b`	32 bits

Use the accum_a and accum_b data types to select a specific register for variable storage: variables declared as type accum_a or accum_b are placed in registers r1r0 and r3r2 respectively with SDCC/ZDXCC. They are placed in r13r12 and r15r14 respectively with ZDCC. This change was necessary with ZDCC because registers r0-r3 are clobbered by the ZSPG2 calling convention. The accum_a and accum_b data types can be used to declare local variables; global accumulators are not supported. From the compiler's point of view, accum_a and accum_b are 32-bit variables that must be stored in a specified register. On the ZSP400, the accum_a and accum_b data types are placed in r1r0 and r3r2, respectively, to allow the use of accumulator-specific operations. Although the compiler treats accum_a and accum_b as 32-bit variables, the accumulator instructions (for example, `mac.a`, `mac2.a`, `macn.a` ...) operate on a 40-bit accumulator. The high-order 8 bits for each accumulator are in the %guard register. If 40-bit accumulators are needed, the high-order bits can be accessed through inline assembly instructions that read or modify the %guard register. In ZSPG2, since every GPR pair supports accumulator operations, other accumulators can be used by declaring them with:

```
register long acc_c asm("rX");
```

In fact, `accum_a` and `accum_b` declarations are equivalent to:

```
    register long x asm ("rX");
```

where "X" is the appropriate register.

It should be remembered that only accumulators r12-r15 have their guard bits preserved across calls.

3.2.2 Register Usage

3.2.2.1 SDCC/ZDXCC Register Usage

Register usage SDCC/ZDXCC is summarized below. Registers r0 through r15 are general-purpose registers, and registers beginning with '%' are control registers.

- Registers used by the compiler: r0–r15, %fmode, %smode, %amode, %hwflag, %loop0, %loop1, %loop2, %loop3, %rpc, %pc, %cb0_beg, %cb0_end, %cb1_beg, %cb1_end, %guard.
- Stack pointer: r12
- Parameter registers: r4-r6
- Callee preserved registers: r0-r3, r7-r12, r14, r15, %guard
- There are no caller saved registers.

- Return registers: r4 for 16-bit return values, and r5r4 for 32-bit return values.

The mode registers are never modified by SDCC/ZDXCC except through inline assembly. The circular buffer registers are never accessed or modified except through predefined macros in the header file `cbuf.h`. The file `cbuf.h` also has predefined macros to set and clear the cb0 and cb1 bits in %smode.

3.2.2.2 ZDCC Register Usage

Register usage by ZDCC is summarized below. Registers r0-r15 are general-purpose registers, a0-a7 are address registers, n0-n7 are index registers, g0-g7 are guard registers and registers beginning with ‘%’ are control registers.

- Registers used by the compiler: r0–r15, a0-a7, n0-n7, g0-g7, %fmode, %smode, %amode, %hwflag, %loop0-%loop3, %rpc, %pc, %cb0_beg-%cb3_beg, %cb0_end-%cb3_end.
- Stack pointer: a7
- Parameter registers: r2-r7, a0, a1, a6
- Callee preserved registers: r8-r15, g6, g7, a2-a5, a7, n4-n7, %loop2, %loop3
- Scratch registers: r0, r1, g0-g5, n0-n3, %loop0, %loop1
- Return registers: a0 for pointer values, r4 for 16-bit return values, and r5r4 for 32-bit non-pointer values.

The mode registers are never modified by ZDCC except through inline assembly. The circular buffer registers are never accessed or modified except through predefined macros in the header file `cbuf.h`. The file `cbuf.h` also has predefined macros to set and clear the cb0-cb3 bits in

%amode. Table 3.5 shows the mode bits that may affect the behavior of compiler-generated code.

Table 3.5 Effect of Mode Bits on Compiler-Generated Code

Mode Register	**Mode Register Bit**	**Affects Non-intrinsic Code**		**Required Entry Value (before C function call)**		**May be Modified Within Function**	
		SDCC ZDXCC	**ZDCC**	**SDCC ZDXCC**	**ZDCC**	**SDCC ZDXCC**	**ZDCC**
%amode	ld	yes		0		no	
	st	yes		0		no	
	cbX	n/a	yes	n/a	0	n/a	yes
%fmode	sat[1]	yes	no	0	x	yes	yes
	q15[2]	no		x		yes	
	sre[3]	yes		x		yes	
	mre[4]	no		x		yes	
%smode	shd[5]	yes	n/a	x	n/a	no	n/a
	lis	yes	yes	0	x	no	no
	sis	yes	yes	0	x	no	no
	cbX[6]	yes	yes	0	0	yes	no
	dir[7]	yes		x		no	
	ddr[8]	yes		x		no	

1. Wtih SDCC/ZDXCC, the sat bit of %fmode can affect nonintrinsic code because of the add and subtract instructions. Nonintrinsic code expects unsaturated arithmetic. If you require saturated arithmetic for some intrinsics, it is safest to enable saturation over as small a region of code as possible, because the sat bit also affects adds and subtracts that must not be saturated (e.g. address arithmetic, stack pointer manipulation, counters, etc.). If you use the -minfer_mac option, the compiler also generates mac and macn instructions, which are also affected by the sat bit.

2. With SDCC/ZDXCC, the q15 mode bit affects nonintrinsic code if the -minfer_mac option is used.
3. The sre bit of %fmode affects nonintrinsic code because of the shra and shra.e instructions. Only perform right shifts of signed variables when the sre bit is cleared.

4. With SDCC/ZDXCC, the mre mode bit affects nonintrinsic code if the -minfer_mac option is used.
5. This bit is ZSP400 specific and selects/unselects the use of shadow registers. Compiled code operates correctly with either shadow registers or nonshadow registers.
6. For ZSPG2, these bits affect the behavior of r14 and r15. They exist for compatibility with ZSP400. They should never be set in code compiled with the ZDCC. When using SDCC/ZDXCC, to prevent these bits from affecting nonintrinsic code, clear these bits when the portion of code requiring circular buffers is exited.
7. This bit controls whether instructions are fetched from internal or external memory. Compiled code operates correctly when it resides in internal or external memory, though normally it resides in internal memory.
8. This bit controls whether data is fetched from internal or external memory. Compiled code operates correctly when data resides in internal or external memory, though normally data resides in internal memory. Note that data includes the stack, and that compiled code does not operate correctly if global data resides in one memory and the stack resides in another memory.

3.2.3 Conventions Used for Passing Parameters

SDCC/ZDXCC's conventions for passing parameters are described below.

- The first three (16-bit) word parameters (scalar type) are passed in registers r4–r6.
- All other parameters are passed on the stack.
- A 16-bit value is returned in r4; a 32-bit value is returned in r5r4.
- A structure is returned using a hidden pointer, which is passed by the caller in r4.
- A structure is passed using two arguments. The first argument is a pointer to the structure, and the second argument is the structure to be passed. The pointer to the structure is a 16-bit value and can be passed in a register if it is one of the first three word-sized arguments. The second argument, the structure, is passed on the stack. For structures with a size of one or two words, the pointer argument is eliminated.

ZDCC's conventions for passing parameters are described below.

Parameters are examined from first to last

- A pointer value will be passed in the first unused register in the following list: a0, a1, a6, r5r4, r7r6, r3r2.
- A 32-bit non-pointer value will be passed in the first unused register in the following list: r5r4, r7r6, r3r2, a0, a1, a6.

- A 16-bit value will be passed in the first unused register in the following list: r4, r5, r6, r7, r2, r3.
- All other parameters are passed on the stack.
- A pointer value is returned in a0. A non-pointer 32-bit value is returned in r5r4. A 16-bit value is returned in r4.
- A structure is returned using a hidden pointer, which is passed by the caller in a0.

Note that registers that were skipped so that a 32-bit parameter could be passed can be used later when passing a 16-bit parameter. For example, a function with prototype:

```
void f(int, long, int)
```

will expect its' arguments to be in: r4, r6r7, and r5 respectively.

3.2.4 Run Time Stack

The C run time stack grows towards lower addresses in memory. The stack pointer (r12 with SDCC/ZDXCC, a7 with ZDCC) decrements when items are pushed on the stack. The initial memory location of the stack is specified in the initialization file `crt0.o`.

Table 3.6 shows the layout of a function's stack frame.

Table 3.6 Stack Frame Layout

Callee saved registers
%rpc
Local variables and temporaries
Outgoing arguments (The stack allocates enough space to accommodate any call by the function.)

Table 3.7 shows the two example stack frames for the functions `foo` and `bar`, after `foo` calls `bar`.

Table 3.7 Stack Frame Example

high address	Callee saved registers of foo	foo's stack frame
	locals/temps of foo	
	max args of all functions called by foo	
	callee saved registers of bar	bar's stack frame
low address	locals/temps of bar	
	max args of all functions called by bar	

Note that within the body of a function, the stack pointer points to the beginning of the next stack frame. When a function is called, the compiler places arguments into registers, if possible, and puts the remaining arguments in the outgoing arguments of the caller's stack frame. The compiler places any required arguments on the stack from lower to higher addresses. Thus the first argument placed on the stack is the one closest to the callee's stack frame. The function call is made after all the arguments have been properly placed.

3.2.5 Example Code for Function Prologue and Epilogue

3.2.5.1 SDCC/ZDXCC

The following is a sample prologue that saves `r0-r3`, `r7-r9`, and `%rpc` and reserves 30 words of space on the stack. Note that with optimization, this code will be reordered with non-prologue code for better scheduling by sdopt.

```
stdu    r0, r12, -2
stdu    r2, r12, -2
stu     r7, r12, -1
stdu    r8, r12, -2
mov     r13, %rpc
stu     r13, r12, -1
mov     r13, 30
sub     r12, r13
```

The appropriate epilogue code for the above prologue is shown below. ZSP interrupt routines expect the stack pointer to point to a writable location. This requirement prevents the use of the stack pointer to directly restore the saved registers. Instead, the stack pointer is copied to r6, and r6 is used to restore the saved registers. After all the registers are restored, r6 is copied back to the stack pointer.

```
mov     r6, r12
mov     r13, 31
add     r6, r13
ldu     r13, r6, 1
mov     %rpc, r13
lddu    r8, r6, 2
ldu     r7, r6, 1
lddu    r2, r6, 2
lddu    r0, r6, 2
add     r6, -1
mov     r12, r6
ret
```

3.2.5.2 ZDCC

The following is a sample epilogue that saves `r8`, `r9`, a2, and `%rpc` and reserves 20 words of space on the stack. Note that with optimization, this code will be reordered with non-prologue code for better scheduling

```
pushd  r8, a7
mov.e  r8, %rpc
pushd  r8, a7
pushd  a2, a7
add    a7, -20
```

The appropriate epilogue code for the above prologue is shown below.

```
add    a7, 20
popd   a2, a7
popd   r8, a7
```

```
           mov.e   %rpc, r8
           popd    r8, a7
           ret
```

3.2.6 Parameter Passing Examples

3.2.6.1 SDCC/ZDXCC

In the example below, function `foo` calls function bar, passing two `long` (32-bit) arguments from r1r0 and r3r2. The first argument is placed in the stack at r12 + 1, and the second argument is placed at r12 + 3.

Function `bar` has a frame size of 16 and accesses the passed arguments in function `foo`'s outgoing argument stack space.

```
   mov            r13, 1         !! The first argument location on the stack
   add            r13, r12
   stdu           r0, r13, 2     !! Store r0 at r12+1 and r1 at r12+2.
   mov            r13, 3
   add            r13, r12       !! Compute r12+3 and store in r13.
   stdu           r2, r13, 2     !! Store r2 in r12+3 and r3 in r12+4.
```

The function `bar` retrieves arguments from `foo`'s stack space by loading the values from `foo`'s outgoing argument space. The first word of `foo`'s outgoing arguments is located at r12+(bar's stack space)+1, or r12+(16)+1.

```
   mov            r13, 17
   ldx            r0, r12
   mov            r13, 18
   ldx            r1, r12
   mov            r13, 19
   ldx            r2, r12
   mov            r13, 20
   ldx            r3, r12
```

3.2.6.2 ZDCC

The following C code:

```
void callee(int i1, long l1, int i2, long l2, long l3, long *p1, long l4, long l5,
long l6) {
   global = l5+l6;
}

void caller() {
  long a=7;

  callee(1,2,3,4,5,&a,7,8,9);
}
```

The arguments will be passed in the following locations:

```
    i1 - r4
    l1 - r7r6
    i2 - r5
    l2 - r3r2
    l3 - a0
    p1 - a1
    l4 - a6
    l5 - stack+1
    l6 - stack+3
```

The above code will produce the following calling code sequence:

```
     mov     a1, 8
     std      a1, a7, 1     !l5, fifth 32-bit non-pointer parameter passed on stack
     mov     a0, 7
     mov     a1, 9
     std      a0, a7, 5
     std      a1, a7, 3     !l6, sixth 32-bit non-pointer parameter passed on stack
     mov      a6, a0        !l4, fourth 32-bit non-pointer parameter passed in a6
     mov      r4, 0x1       !i1, first 16-bit parameter passed in r4
     mov     r6, 2          !l1, first 32-bit non-pointer parameter passed in r7r6
     mov     r7, 0
     mov      r5, 0x3       !i2, second 16-bit parameter passed in r5
     mov     r2, 4          !l2, second 32-bit non-pointer parameter passed in r3r2
     mov     r3, 0
     mov     a0, 5          !l3, third 32-bit non-pointer parameter passed in a0
     mov      a1, a7        !p1, first pointer parameter passed in a1
     add      a1, 5
     call     _callee
```

The function `callee` retrieves `l5` and `l6` from `caller`'s stack space by loading the values from `caller`'s outgoing argument space. The first word of `caller`'s outgoing arguments is located at a7+(callee's stack space)+1, or a7+(0)+1.

```
     mov      a0, a7
     add      a0, 1
     ldd      r4, a0
     ldd      r6, a7, 3
```

3.3 Run Time Environment

The compiler run time environment is initialized in the startup file `crt0.o` on Solaris platforms or `crt0.obj` on Windows platforms. By default, the startup file is automatically linked by the compiler. The initialization file will fill the bss section with zeroes.

The run-time memory map contains three main sections: text, data, and bss, in that order. The heap grows from lower addresses to higher addresses and starts after the bss section. The stack grows from higher to lower addresses, and starts at the address specified by the predefined variable `__stack_start – 4`, which has a default value of 0xF7FB for SDCC/ZDXCC and 0xFFEFFF for ZDCC. This can be modified as shown below.

```
    sdcc -Wl,-defsym,__stack_start=0xd000 test.c
```

3.4 C Run Time Library Functions

The `libc.a` libraries supplied with the C compiler contain the run time library functions. These functions are equivalent to those found in other C programming environments, having the same names and parameter lists. Thus existing programs that use these functions may be recompiled without any changes. The compiler provides a debugging form of the library, `libg.a`, which allows you to debug standard library functions.

The library functions are grouped into the following categories:

- String functions (`string.h`)

  `bcmp`, `bcopy`, `bzero`, `index`, `memchr`, `memcmp`, `memcpy`, `memmove`, `memset`, `rindex`, `strcat`, `strchr`, `strcmp`, `strcpy`, `strcspn`, `strlen`, `strncat`, `strncmp`, `strncpy`, `strpbrk`, `strrchr`, `strspn`, `strstr`, `strtok`.

- I/O functions (`stdio.h`)

  `fopen, fclose, fwrite, fread, fgetc, fputc, fprintf, printf, sprintf, vfprintf, vprintf, vsprintf, getc, putc, getchar, putchar`

The `fprintf` and `printf` functions have been extended to allow printing of the q15 data type. A “%q” format specifier will print a 16-bit value in fixed-point notation.

- The filehandles `stdin`, `stdout`, and `stderr` are available for use with `fwrite`, `fread`, `fgetc`, `fputc`, and `fprintf`.

- Memory allocation functions (`stdlib.h`)

  `malloc, free, mbtowc`

- Interprocedural control flow functions (`setjmp.h`)

  `setjmp, longjmp`

In the case of I/O functions, the SDK performs file I/O by sending a message to the program running on the host (sdbug400, zsim400, zisim400, zdbug, zdxbug, zsimg2 or zisimg2). These messages will cause the host program to perform the requested file I/O operation. All host programs and all zdbug targets support file I/O.

3.5 N-Intrinsics

SDCC N-Intrinsics provide support for DSP instructions. N-Intrinsics are implemented as macros in the header file `N_Intrinsic.h`. The name of an N-Intrinsic begins with an `N_` , followed by a suffix that indicates the operation’s data type: _s for `int`, `_l` for `long`, and _h for high-order `int` of a `long`.

To use N-intrinsics, add the following line in each of your C files:

```
#include <N_Intrinsic.h>
```

N-intrinsics are implemented by the compiler using the assembly instructions shown in Table 3.8. The older L-intrinsics are still supported and are described in Appendix D, “L-Intrinsic Functions.”

Table 3.8 N-Intrinsic Functions

Intrinsic Function	Generated Code	Analogous L-Intrinsic
`N_mac(accum acc, int x, int y)`	`mac.acc x, y`	`L_maca, L_macb`
`N_macn(accum acc, int x, int y)`	`macn.acc x, y`	`L_macna, L_macnb`
`N_mac2(accum acc, long x, long y)`	`mac2.acc x,y`	`L_mac2a, L_mac2b`
`N_mul(accum acc, int x, int y)`	`mul.acc x, y`	`L_mula, L_mulb`
`N_muln(accum acc, int x, int y)`	`muln.acc x, y`	None
`N_norm_l(int ret, long a)`	`norm.e ret, a`	`norm_l`
`N_norm_s(int ret, int a)`	`norm ret, a`	`norm_s`
`N_extract_h(int ret, long a)`	`ret = a[31:16]`	`extract_h`
`N_deposit_h(long ret, int a)`	`ret[31:16] = a` `ret[15:0] = 0`	`L_deposit_h`
`N_abs_l(long ret, long a)`	`abs.e ret, a`	`L_abs`
`N_abs_s(int ret, int a)`	`abs ret, a`	`abs_s`
`N_round_l(long ret, long a)`	`round.e ret, a`	`round`
`N_shla_l(long ret, int a)`	`shla.e ret, a`	`L_shla`
`N_shla_s(int ret, int a)`	`shla ret, a`	`shla`

3.5.1 Vector N-Intrinsics

The ZDCC compiler also provides N-Intrinsics for common vector operations. They are shown in Table 3.9. The vector N-Instrinsics will produce more efficient code than the equivalent non-vector code.

Table 3.9 Vector N-Intrinsics

N-Intrinsic [1]	Functionality [2]
`N_vc_mac(accum acc, int *vec1, int inc1, int cnst, int len)`	`for (i=0; i<len; i++) { N_mac(acc,cnst,vec1[i*inc1]); }`
`N_vc_macn(accum acc, int *vec1, int inc1, int cnst, int len)`	`for (i=0; i<len; i++) { N_macn(acc,cnst,vec1[i*inc1]); }`
`N_vv_mac(accum acc, int *vec1, int inc1, int *vec2, int inc2, int len)`	`for (i=0; i<len; i++) { N_mac(acc,vec1[i*inc1],vec2[i*inc2]) }`
`N_vv_macn(accum acc, int *vec1, int inc1, int *vec2, int inc2, int len)`	`for (i=0; i<len; i++) { N_macn(acc,vec1[i*inc1],vec2[i*inc2]); }`

1. All increment values (inc1, inc2) must be –2, –1, 1, or 2.
2. The actual code generated will be more efficient than the functionally-equivalent code in this column.

Important: If you use vector N-Intrinsics at optimization level 3 (-O3), you must also use the `-fno-inline` option. Functions with vector N-Intrinsics must not be inlined, since these intrinsics create labels. If these labels are inlined, they are duplicated and cause an error.

3.5.2 ETSI Functions

The SDCC's N-Intrinsics also allow access to processor-supported ETSI functionality, although the interface is different. For example, the ETSI code:

```
y = norm_l(x);
```

can be rewritten with N-Intrinsics as:

```
N_norm_l(y,x);
```

Another approach that preserves the ETSI defined interface would be to use `N_norm_l` to implement the ETSI function. For example, `norm_l` could be implemented as:

```
static inline int norm_l(long src) {

   int ret;

   N_norm_l(ret,src);

   return(ret);

}
```

You may implement some ETSI functions can be implemented using N-Intrinsics, but you must set mode bits in `%fmode` accordingly. For example, you can implement the ETSI function `L_mac` using `N_mac`, but you must also set the SAT and Q15 mode bits in `%fmode`. This correspondence between N-Intrinsics and ETSI functions is shown in Table 3.10.

Table 3.10 ETSI to N-Intrinsic Mapping

		fmode[1] Register Bits			
ETSI Function	**N-Intrinsic**	**sat**	**q15**	**sre**	**mre**
abs_s	N_abs_s	x	x	x	x
extract_h	N_extract_h	x	x	x	x
L_abs	N_abs_l	x	x	x	x
L_deposit_h	N_deposit_h	x	x	x	x
L_mac	N_mac	1	1	x	0
L_macN	N_mac	0	1	x	0
L_msu	N_macn	1	1	x	0
L_msuN	N_macn	0	1	x	0
L_mult	N_mul	x	1	x	0
L_shl	N_shla_l	1	x	x	x
mac_r	N_mac	1	1	x	1

Table 3.10 ETSI to N-Intrinsic Mapping (Cont.)

		fmode[1] Register Bits			
ETSI Function	**N-Intrinsic**	**sat**	**q15**	**sre**	**mre**
msu_r	N_macn	1	1	x	1
mult	N_mul	x	1	x	0
mult_r	N_mul	1	1	x	1
norm_l	N_norm_l	x	x	x	x
norm_s	N_norm_s	x	x	x	x
round	N_round_l	x	x	x	x

1. 1 = Set, 0 = Cleared, x = Don't Care

3.6 Circular Buffers

The `cbuf.h` header file provides the interface to the circular buffers. The header file's macros generate the necessary assembly instructions.

To use a circular buffer, a pointer must be declared, the circular buffer boundaries must be set, and the circular buffer must be enabled. With SDCC/ZDXCC the pointer must be in r14 or r15.

```
register int *pt asm("r14");
set_r14_cbuf(low,high);
enable_r14_cbuf;
```

With ZDCC, the pointer must be in a0 - a3.

```
register int *pt asm("a2");
set_cbuf(CBUF_ID,low,high);
enable_cbuf(CBUF_ID);
```

`CBUF_ID` must be `A0_CBUF`, `A1_CBUF`, `A2_CBUF` or `A3_CBUF`.

A circular buffer must have at least 4 ints or 2 longs.

Circular buffers can be disabled using the following macros with SDCC/ZDXCC:

```
disable_rn_cbuf;
```

For ZDCC the macro is:

```
disable_cbuf(CBUF_ID);
```

There are special macros defined within `cbuf.h` to access the elements in a circular buffer. These macros are the same for all compilers.

```
load_int_cbuf(dst,pt,inc)
store_int_cbuf(src,pt,inc)

load_long_cbuf(dst,pt,inc)
store_long_cbuf(src,pt,inc)
```

The *inc* parameter determines the number of elements to increment the pointer pt. The *inc* parameter must be a constant rather than a variable. For `load_int_cbuf` and `store_int_cbuf`, *inc* must be in the range 1–50. For `load_long_cbuf` and `store_long_cbuf`, *inc* must be in the range 1–25.

It is legal to access a value pointed to by *pt* using **pt*, so an increment value of 0 is not needed.

The *dst* and *src* parameters are variables used for the destination and source values, respectively. Note that these parameters are not pointers to a location where the destination/source will be stored/accessed, but to the variables that will actually be stored/accessed.

Note: You must disable circular-buffer arithmetic immediately after the final use of *pt*, because the compiler may reuse the register containing *pt* for other purposes. The code generated in this case would not expect the register to have circular arithmetic.

Because the registers supporting circular-buffer functionality are not saved and restored by function calls/returns, circular buffers should not be used with code containing function calls.

3.7 Accessing Control Registers

Macros have been defined in the header file <creg.h> to simplify accessing control registers.

`read_creg``(creg,var) -` Puts the value of a control register into `var.`

`write_creg``(creg,val) -` Puts a value, which can be a variable or an immediate, into a control register. The `val` argument can be made by or-ing together the following masks for the following registers:

Macros have also been defined to manipulate specific bits of control registers.

`bitset_creg``(creg,bitnum)`

`bitclear_creg``(creg,bitnum)`

`bitinvert_creg``(creg,bitnum)`

The bitnumber and value arguments can be filled with macros which have been defined to the approiate value. The bitnumber and mask to access a specific bit has been defined to “bit name”_[MASK|BIT]. For example, to set the Q15 bit of %fmode, use the following macro:

```
bitset_creg(%fmode,Q15_BIT);
```

3.8 Q15 Support

CC supports the Q15 data type. To use Q15 arithmetic, the q15 mode bit in the %fmode register must be set, as follows:

```
   bitset_creg(%fmode,Q15_BIT);
```

The q15 mode bit affects Q15 multiplies and the N-Instrinsics `N_mul`, `N_mac`, `N_macn`, `N_mac2`, and the vector intrinsics.

Q15 arithmetic can be disabled as follows:

```
   bitclear_creg(%fmode,Q15_BIT);
```

This release of the SDK does not support Q15 division.

The code produced for the Q15 data type is equivalent to that produced for the `int` data type, except for the following three cases:

- The product of two Q15s is calculated with a `mul` instruction rather than an `imul` instruction.

- The 16-bit result of a Q15 product is the high-order 16 bits of the result produced by a `mul` instruction. The 16-bit result of an `int` product is the low-order 16 bits of the result produced by an `imul` instruction.
- The product of two Q15 constants is not simplified by the compiler.

The `fprintf` and `printf` functions have been extended to allow printing of the q15 data type. A “%q” format specifier will print a 16-bit value in fixed point notation.

3.9 Inline Assembly

Inline assembly that references C variables can be generated by using the `asm` directive.

3.9.1 Syntax

The basic syntax of the `asm` directive is:

```
asm(   “parameterized assembly” :

       output variable, ... :
       input expression, ... :
       explicitly clobbered register, ... );
```

3.9.2 Parameterized Assembly

The *`parameterized assembly`* consists of a text string containing the desired assembly output with parameters that will be replaced with registers or constants according to the arguments in *`output variable`* and *`input expression`*. The syntax of a parameter is shown in Table 3.11.

Table 3.11 Parameter Output Syntax

Format	Output
%*n*	register name or constant
%m*n*	accumulator name
%o*n*	high register name

In the table above, `n` is the index of the desired argument in *`output variable`* or *`input expression`*. The three formats—`%`, `%m`, and `%o`—control the way an argument is printed in the generated assembly. For example, a variable of `long` type that the compiler has placed in r1 and r0 will be printed as `r0` when the `%` format is specified, as a when the `%m` format is specified, or as `r1` when the `%o` format is specified.

3.9.3 Variables and Expressions

The syntax for an *`output variable`* and *`input expression`* is:

"*`constraint`***"** **(***`expression`***|***`variable`***)**

A *`constraint`* is used to describe the requirements that an instruction places on an argument. For example, an instruction that requires an argument to be in a particular register would put a constraint on that argument to ensure that the argument is placed in an allowed register.

In example 3 in Section 3.9.5, "Examples of `asm` Directive", the `acc` variable is constrained to be in an accumulator register. The supported constraints are shown in Table 3.12.

Table 3.12   Argument Constraints

Constraint	Effect	Availability
=	output variable	All compilers
r	general-purpose register	All compilers
e	address register	ZDCC
h	index register	ZDCC
c	accumulator register	All compilers
n	constant	All compilers
<n>	same as indexed argument	All compilers

Note that a constant argument can be used with an `r` constraint. The SDK copies the constant to a register and uses the register as the argument. You can combine constraints, which can be useful for instructions that allow different types of arguments. For example, the `shla` instruction can accept either a register or an immediate argument.

The appropriate constraint for this argument would be `rn`. In example 4 in Section 3.9.5, “Examples of `asm` Directive”, the input parameter is constrained to be either a register or an immediate. Sometimes it is necessary for two arguments to be in the same register. This requirement can be described by constraints. The first argument should be described with whatever constraint is appropriate, and the second argument’s constraint must be the index of the first argument. For example, the first argument of the `add` instruction is an output/input argument. You must list this argument as an *`output variable`* and an *`input expression`*. The constraint of this argument when it appears as an *`input expression`* should be the index of the argument when it appears as an *`output variable`*. In example 3 in Section 3.9.5, “Examples of `asm` Directive”, the output argument and the first argument illustrate this technique.

3.9.4 Explicitly Clobbered Registers

The syntax for an *`explicitly clobbered register`* is:

“*`register name`*”

This entry tells the compiler that the assembly code generated will clobber the specified register. Thus the generated assembly code may use the specified register for scratch purposes.

3.9.5 Examples of `asm` Directive

The examples in the subsections below illustrate the usage of the `asm` directive.

3.9.5.1 Example 1

```
asm(“norm.e %0, %1”:

   “=r” (ret) :

   “r” (a));
```

The example shown above has one output argument, `ret`, and one input expression, a. If the variable `ret` is in r0 and the variable a is in r4, this directive produces:

```
   norm.e r0, r4
```

3.9.5.2 Example 2

```
asm(“abs r5, %1\n\tst r5, %0” : :
   “e” (addr), “r” (val) :
   “r5”);
```

The example shown above stores the absolute value of `val` at `addr`. Two instructions are generated by this directive. There are two input expressions and no output arguments. Note that register r5 is clobbered by this directive. If `addr` is in a0 and `val` is in r15, this directive produces:

```
   abs r5, r15
   st r5, a0
```

3.9.5.3 Example 3

```
asm(“mac.%m0 %2, %o2” :
   “=c” (acc) :
   “0” (acc), “r” (val));
```

The example above adds the 32-bit product of the high and low 16 bits of `val` to `acc`. Note that the high part of `val` is obtained with the `%o2` operand and that the accumulator is printed with the `%m0` operand. Also note that `acc` is both an input and an output argument, and that the constraint for `acc` when it appears as an output argument is c, an accumulator, and when it appears as an input argument is `0`, which tells the compiler that these two arguments must be in the same location. If `acc` is in r0 and `val` is in r3r2, the following code is generated:

```
   mac.a r2, r3
```

3.9.5.4 Example 4

```
asm(“mov %%smode, %0” : :
   “rn” (val));
```

The example shown above sets `%smode` to `val`. Note that %smode is not specified as a clobbered register, because %smode has no meaning to the compiler. If `val` is a symbolic constant with the value 3, the following code is generated:

```
   mov %smode, 3
```

You can find additional examples of using the asm directive in the header file `N_Intrinsic.h`.

3.9.5.5 Example 5

```
asm(“bits %smode, 7”);
```

The example shown above sets bit 7 in `%smode`. This example illustrates the general rule that if the assembly statement contains an argument (as in Example 4, which contains the argument `%0`), a reference to a register must contain an additional per cent (%) sign. Example 5 contains no argument, so a single % preceding smode is used.

3.9.6 Optimization of Inline Assembly

For purposes of optimization, the compiler assumes that inline assembly has no effect except to modify the output variables. Thus inline assembly can be removed by optimization if none of the output variables is subsequently used. Inline assembly that must not be deleted or significantly moved should contain the keyword `volatile` following the `asm` directive, as shown below.

```
asm volatile(“bits %smode, 7”);
```

The `volatile` keyword is implicit for inline assembly with no output variables. Thus, the use of `volatile` in the above example is redundant.

3.10 Assembly Optimizer and Handwritten Assembly

The assembly optimizers can be used to automatically generate the prologue and epilogue for an assembly function and then to schedule the entire function.

```
sdopt -asm assemblyfile
```

The output will be placed in standard output. The assembly optimizers expect input of the following format:

```
!PROLOGUE(<function name>)
       <function body>
!EPILOGUE
```

This will be transformed by the assembly optimizer to:

```
.set    REGSAVE_SIZE_<function name> <stack space used>
<function name>:
__FUNC_START_<function name>:
<optimized assembly code with prologue/epilogue>
__FUNC_END_<function name>:
ret
```

All registers that must be preserved according to the C calling convention will be preserved. Note that the name REGSAVE_SIZE_<*function name*> can be used if the size of the stack space used by the prologue/epilogue is needed. Any input in the assembly file outside of the

`!PROLOGUE` and `!EPILOGUE` markers will be copied without modification.

3.11 Debugging Options

You can debug code compiled using the GCC-supplied `-g` option, which generates debugging information. You can attempt debugging with optimization turned on, though optimization makes debugging difficult. When debugging optimized code, use the `-mno_sdopt` option, because `sdopt`, `zdxopt` and `zdopt` do not preserve the location of debugging information.

Using the `-g` option with optimization modifies the code generated in two ways. First, the debugging version of the C library is linked, rather than the optimized version. Second, leaf functions save and restore %rpc (without the `-g` option, this save and restore is removed by optimization). The `-g` option saves and restores this register, because the debugger requires it to examine the call stack.

3.12 Code Statistics

CC creates four labels and symbols that are useful in analyzing the code generated by the compiler.

Every function will have a label placed on its first instruction and after its last instruction with the following formats:

`__FUNC_START_<`*`function name`*`>`

`__FUNC_END_<`*`function name`*`>`

The difference of these two labels will give the code size of a function. A function will also have a label placed on its return instruction:

`__FUNC_EXIT_<`*`function name`*`>`

This label is used for function profiling. Every function will also have an absolute symbol that shows the number of words of stack space used per invocation of the function.

`__FUNC_FRAME_SIZE_<`*`function name`*`>`

3.13 Example Compilations

3.13.1 Example 1

```
cc test.c -Tdata=0x1
```

This command invokes the C compiler, assembler, and linker and produces an executable file with the default name `a.out`.
The **`-Tdata=0x1`** command places the data at address 0x1 to prevent a NULL pointer from being a valid pointer.

3.13.2 Example 2

```
cc -c test.c
```

This command invokes the C compiler and assembler only, producing an object file with the default name `test.obj` (Windows) or `test.o` (UNIX).

3.13.3 Example 3

```
cc -S test.c
```

This command invokes the C compiler only, producing an assembly file with the default name `test.s`.

3.13.4 Example 4

```
cc -O3 test.c
```

This command invokes the C compiler with the highest level of optimization, that is, including all level -02 optimizations, as well as function inlining and loop unrolling. The assembler and linker are also invoked, and the output is an executable file with the default name `a.out`.

Chapter 4
Assembler

The SDK Assembler (SDAS/ZDAS) is based on the GNU assembler, AS, from the Free Software Foundation. It is described in *Using AS: The GNU Assembler,* by Dean Elsner, et. al., Free Software Foundation, January 1994. The description of SDAS/ZDAS in this chapter, for the most part, includes only the differences from AS. SDAS is the assembler for the ZSP400 architecture. ZDAS is the assembler for the ZSPG2 architecture. In this chapter, unless otherwise noted, SDAS refers to both the ZSP400 and ZSPG2 assemblers.

The assembler is invoked from the shell using the following command:

`sdas [`*`options`*`]` ***`sourcefile`***

SDAS processes an assembly source file with the `.s` file extension and produces a relocatable object file in ELF format with the default file extension `.obj` (Windows) or `.o` (UNIX).

4.1 Assembly Language Syntax

The basic format of a SDK assembly language statement is:

`[` *`label`*`: ]    [` *`statement`* `] [ !`*`comment`* `]`

Labels are identifiers that start at the beginning of a line, with no leading spaces or tabs, and end with a colon. Identifiers begin with a letter (case is significant) or an underscore, and can continue with more letters, digits, and underscores. Assembly language instructions can be on the same line as a label.

Examples:

```
Start:                  !“Start” is a label
start:                  !“start” is another (different) label
```

```
         bnz start    !“start” is a label reference
loop:    add r0, r1   !“loop” is a label
         bnz Start:   ! Illegal reference (extra colon)
End                   ! Illegal label (missing colon)
```

Symbols beginning with ‘L’ are locally resolved, and are therefore not visible to the linker or to other modules.

Assembler statements can be assembler directives or assembly language instructions. Assembler directives start with a period (‘.’).

Comments start with an exclamation mark (`!`) and continue until the end of the line. The symbol ‘#’ at the beginning of the line indicates that it is a comment.

Note that files with the `.S` extension can be assembled using sdcc, which causes the C preprocessor to run before the assembler. This enables users to use C-style comments and `#defines` in their assembly code. However using a `-g` option will not cause any debug symbol generation, since the source file is an assembly program. To turn on debug information for an assembly program with a `.S` extension, you can use sdcc with the `-Wa` and `-dbg` options (the -dbg option is described in Section 4.1.1.4, “Debugging Option (-dbg),” page 4-3).

All assembly programs must be contained within a section.

Putting `.section ".text", "ax"` before any assembly code ensures that the code gets assembled into the .text section. Please refer to the GNU assembler manual for more information on the section syntax and flag definitions.

4.1.1 Assembler Options

Please refer to the GNU assembler manual for a full description of all options available to the assembler. A few of the more frequently-used options as well as the options specific for the SDK are described below.

4.1.1.1 Suppress warnings (-W)

This options prevents warnings from the assembler from being displayed on the screen.

4.1.1.2 Output file (-o)

Using `-o` *`objfile`* assembles the output into the object file specified. By default, if you do not use the `-o` option, the resulting object file is named `a.out`.

4.1.1.3 Include path (-I)

The `-I` *`dir`* option is used to add the specified directory to the search list used by `.include` directives.

4.1.1.4 Debugging Option (-dbg)

The `-dbg` option adds debugging information to the executable file, which allows you to debug the source file rather than the disassembled text. The usage is:

```
sdas -dbg test.s
```

where `test.s` is the name of the assembly file.

4.1.2 Assembler Directives

The following subsections describe some frequently-used assembler directives, as well as those that are specific to the SDK assembly language.

4.1.2.1 .walign

The `.walign` directive aligns the location counter on the next word boundary specified by an integer argument. If the location counter is already aligned, no action is taken. Intervening words are filled with nop instructions. For example,

```
.walign 32       ! Align to the next 32-word boundary.
```

4.1.2.2 .wspace

The `.wspace` directive allocates space in a segment as specified by an integer argument. The location counter is incremented, regardless of alignment. For example,

```
.wspace 7        ! Increment the location counter by seven.
```

An optional fill value can also be given. If no fill value is given, the space will be filled with zeroes.

```
.wspace 7, 0xd800! Create 7 words of 0xd800
```

4.1.2.3 .word

The `.word` directive allows a user to specify zero or more comma separated values to be assembled into memory.

4.1.2.4 .global

The `.global` directive is used to declare a global symbol. If this directive is not used, a symbol defined in a partial program is visible only within its scope. The `.global` directive makes the symbol visible to the linker.

4.1.2.5 .section

The .section directive assembles the code following it into the section name specified.

Example: .section, “.text”, “ax”

This defines a section named “.text” - the characters following it tell the assembler that the code following the directive is allocatable and is a part of the instruction memory. Please refer to the GNU assembler manual for more information.

Although GNU assembler documentation says unnamed sections go to the default .text section, it is necessary to specify sections explicitly for the ZSP SDK tools.

4.1.3 Assembler Special Cases

For all instructions that require a register pair, the even register must be specified as the operand. For the ZSP400 assembler only, If an odd register is specified, the even register of the register pair is used as the actual operand in the instruction, and the assembler displays a warning message. With the ZSPG2 assembler, **`zdas`**, an odd register will not be converted to an even register and an error will be message will be shown.

For the ZSP400 architecture, a target function must be placed at an even address. If the value is odd, an error message is displayed. A function can be forced to start on an even address by using the `.walign 2` directive. For the ZSPG2 architecture, there are no alignment requirements for call targets.

Chapter 5
Linker

The SDK Linker (SDLD/ZDLD) is based on the GNU linker, LD, from the Free Software Foundation. LD is described in *Using LD: The GNU Linker,* by Steve Chamberlain, Free Software Foundation, January 1994. SDLD is the linker for the ZSP400 architecture. ZDLD is the linker for the ZSPG2 architecture. Unless otherwise noted, SDLD refers to both the ZSP400 and ZSPG2 linkers.

The linker processes the object files generated by the assembler (designated with the `.obj` extension on Windows or `.o` extension on UNIX) and produces an executable file in ELF format with the default name `a.out`.

The linker is invoked from the shell using the following command:

```
sdld [options] sourcefile
```

5.1 Sections

By default, the linker generates .text, .data and .bss sections. The .text sections contains code, .data contains data, and .bss contains uninitialized data. If there are additional user-defined sections specified in the linker script file, the linker will generate them also.

By default, .bss follows .data in Data memory unless relocated using a linker script command.

The following section names have special meaning only on the ZSP400 linker:

- `.exttext_0` through `.exttext_15`
- `.extdata_0` through `.extdata_15`

Code or data in these sections is placed in the appropriate external instruction or data memory, with the particular external page selected by the number in the section name.

On the ZSP400 architecture, the offset of a `call immediate` instruction must be even. If the assembler cannot resolve this offset, the linker will. If the offset is odd, the linker displays an error message. Because the assembler will automatically align `call immediate` instructions on an even address, this error occurs only if the call target was on an odd address. To resolve this error, align the call target on an even address, using the `.walign 2` directive.

5.1.1 Symbols

By default, program execution begins at `__start`. The entry point can be altered by specifying an alternate address, using the `-e` option. For example, the following command will cause execution to begin at address 0xabcd:

```
sdld -e 0xabcd
```

The C stack region will always be set to the internal data memory. The linker uses four symbols for stack-range checking:

- `__stack_start`: beginning of C stack, default setting is 0xF7FF with SDLD and 0xFFEFFF with ZDLD.
- `__stack_end`: ending address of C stack
- `__stack_size.linker_defined`: stack size calculated by linker
- `__stack_size.user_required`: user required stack size set in command line option

You can inspect the values of these symbols in the map file.

The value of the symbol `__stack_start` or `__stack_end` can be set in a linker script file or by using the command-line option `defsym sym=Value`.

The user-required stack size can be set using the command-line option `stack_size=Value`. The linker will report an error when the stack size is inadequate.

5.1.2 Linker command file

A linker command file (also called a linker script file) is a file containing linker commands that explicitly define symbols and locate sections in memory. A linker command file can be specified when the linker is invoked. An example linker command file is shown below.

```
SECTIONS
{
.text 0x2000: {*(.text)}
.data 0x3000: {*(.data)}
vectors 0x0000: {*(vectors)}
}
```

The example above declares the output sections .text, .data, and vectors. Each output section is formed by the corresponding input sections from all files (as indicated by the '`*`').

Refer to the GNU ld man page for more information.

5.1.3 Linker options

The following subsections describe some frequently-used linker options, as well as those that are specific to the SDK assembly language.

Option	**Description**
-T *linkercommandfile*	Replaces the linker's default script file with the specified linkercommandfile.
-o *outputfile*	Names the output file. By default, the output file name is `a.out`.
-l *archive*	Adds *archive* file archive to the list of files to link. The linker will search for files `lib`*`archive`*`.a` for every archive specified using this option.
-L searchdir	Adds *searchdir* to the list of directories to search for archive libraries and linker scripts. Multiple paths can be specified by using the -L option multiple times.
-M	Prints the link map to `stdout`. A link map contains information on the mapping of symbols.

Option	Description
--defsym *symbol=expression*	Creates a global symbol in the output file containing the absolute address specified by the expression. This option can be used multiple times to create multiple symbols. Valid formats for expression are hexadecimal constants or the names of existing symbols.
-Tbss *addr*	Locate the .bss section at the address specified by *`addr`*.
-Ttext *addr*	Locate the .text section at the address specified by *`addr`*.
-Tdata *addr*	Locate the .data section at the address specified by *`addr`*.

Chapter 6
Utilities

This chapter describes the SDK utility programs.

The SDK provides additional utilities for manipulating files that are generated by the tools during project creation. These SDK-specific utilities, described in Table 6.1, replace their GNU counterparts. Tools for the ZSP400 architecture start with an “sd” prefix. Tools for the ZSPG2 architecture start with a “zd” prefix. Unless otherwise specified, the description of a utility applies to both the ZSP400 and ZSPG2 versions of the tools.

Table 6.1 SDK Utilities and GNU Counterparts

Utility	GNU Equivalent	Function
`sdar` `zdar`	`ar`	Creates, modifies, and extracts files from an archive.
`sdnm` `zdnm`	`nm`	Lists symbols from object files.
`sdobjdump` `zdobjdump`	`objdump`	Displays information from object files.
`sdranlib` `zdranlib`	`ranlib`	Generates an index for an archive.
`sdstrings` `zdstrings`	`strings`	Prints the printable characters in the files.
`sdsize` `zdsize`	`size`	Lists section sizes and total size of object file.
`sdstrip` `zdstrip`	`strip`	Discards symbols from object files.
`sdobjcopy` `zdobjcopy`	`objcopy`	Copies and translates object files.

6.1 sdar

Format

```
sdar [-]p[mod [relpos]] archive [member...]
```

Description

`sdar` creates, modifies, and extracts from archives. An *archive* is a single file holding a collection of other files in a structure that allows you to retrieve the original individual files (called *members* of the archive). The original files' contents, mode (permissions), timestamp, owner, and group are preserved in the archive, and can be restored on extraction.

When you specify the modifier `s`, `sdar` creates an index to the symbols defined in relocatable object modules in the archive. Once created, this index is updated in the archive whenever sdar makes a change to its contents (save for the 'q' update operation). An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

You may use '`sdnm -s`' or '`sdnm --print-armap`' to list this index table. If an archive lacks the table, another form of `ar` called `sdranlib` can be used to add just the table.

Options

The *p* keyletter specifies what operation to execute. It may be any of the following, but you must specify only one of them:

Table 6.2 sdar p Keyletter Options

Option	**Description**
`d`	Deletes modules from the archive. Specify the names of modules to be deleted as *`member`*`...`; the archive is untouched if you specify no files to delete. If you specify the ‘`v`’ modifier, `ar` lists each module as it is deleted.
`p`	Prints the specified members of the archive, to the standard output file. If the ‘`v`’ modifier is specified, show the member name before copying its contents to standard output. If you specify no member arguments, all the files in the archive are printed.
`r`	Inserts the files member... into archive (with replacement). This operation differs from ‘`q`’ in that any previously existing members are deleted if their names match those being added. If one of the files named in member... does not exist, ar displays an error message, and leaves undisturbed any existing members of the archive matching that name. By default, new members are added at the end of the file; but you may use one of the modifiers ‘`a`’, ‘`b`’, or ‘`i`’ to request placement relative to some existing member. The modifier ‘`v`’ used with this operation elicits a line of output for each file inserted, along with one of the letters ‘`a`’ or ‘`r`’ to indicate whether the file was appended (no old member deleted) or replaced.
`t`	Displays a table listing the contents of archive, or those of the files listed in member... that are present in the archive. Normally only the member name is shown; if you also want to see the modes (permissions), timestamp, owner, group, and size, you can request that by also specifying the ‘`v`’ modifier. If you do not specify a member, all files in the archive are listed. If there is more than one file with the same name (say, ‘`fie`’) in an archive (say ‘`b.a`’), ‘`ar t b.a fie`’ lists only the first instance; to see them all, you must ask for a complete listing--in our example, ‘`ar t b.a`’.
`x`	Extracts members (named member) from the archive. You can use the ‘`v`’ modifier with this operation, to request that ar list each name as it extracts it. If you do not specify a member, all files in the archive are extracted.

A number of modifiers (*`mod`*) may immediately follow the *p* keyletter, to specify variations on an operation’s behavior:

Table 6.3 sdar p Keyletter Modifiers

Option	Description
`f`	Truncates names in the archive. GNU ar will normally permit file names of any length. This will cause it to create archives which are not compatible with the native ar program on some systems. If this is a concern, the ‘`f`’ modifier may be used to truncate file names when putting them in the archive.
`o`	Preserves the original dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.
`u`	Normally, ‘`ar r`’`...` inserts all files listed into the archive. If you would like to insert only those of the files you list that are newer than existing members of the same names, use this modifier. The ‘u’ modifier is allowed only for the operation ‘`r`’ (replace). In particular, the combination ‘qu’ is not allowed, since checking the timestamps would lose any speed advantage from the operation ‘q’.
`q`	Quick append at end of files

6.2 sdstrip

Format

```
sdstrip
   [-R sectionname | --remove-section=sectionname]
   [-s | --strip-all]
   [-S | -g | --strip-debug]
   [-N symbolname | --strip-symbol=symbolname]
   [-o file]
   [-p |--preserve-dates]
   [--help]
   objfile ...
```

Description

`sdstrip` discards all symbols from the object files *objfile*. The list of object files may include archives. At least one object file must be specified. `sdstrip` modifies the files named in its argument, rather than writing modified copies under different names.

Options

Table 6.4 sdstrip Options

Option	Description
`--help`	Shows a summary of the options to strip and exit.
`-R` *`sectionname`* | `--remove-section=`*`sectionname`*	Removes the named section from the file. You may give this option more than once. Note that using this option inappropriately may make the object file unusable.
`-R` *`sectionname`* | `--remove-section=`*`sectionname`*	Removes any section named *`sectionname`* from the output file. You may give this option more than once. Note that inappropriate use of this option inappropriately may make the output file unusable.
`-s | --strip-all`	Removes all symbols.
`-S | -g | --strip-debug`	Removes debugging symbols only.
`-N` *`symbolname`* | `--strip-symbol=`*`symbolname`*	Removes symbol *`symbolname`* from the source file. You may give this option more than once, and may be combined with other strip options.
`-o` *`file`*	Puts the stripped output in *`file`*, rather than replacing the existing file. If you use this argument, you can specify only one *`objfile`* argument.

6.3 sdranlib

Format

`sdranlib` *`archive`*

Description

The sdranlib utility generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file.

You may use ‘`sdnm -s`’ or ‘`sdnm --print-armap`’ to list this index.

An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

6.4 sdnm

Format

```
sdnm     [-g | -s | -A | -o | -u | -l ] objfile
```

Description

The sdnm utility lists the symbols from object files *objfile*. If no object files are given as arguments, sdnm assumes the file a.out.

Options

Table 6.5 sdnm Options

Option	Description
-A | -o | --print-file-*name*	Precedes each symbol by the name of the input file where it was found, rather than identifying the input file once only before all of its symbols.
-g | --extern-only	Displays only external symbols.
-p | --no-sort	Prints the symbols in the order they are encountered rather than sorting them first.
-s | --print-armap	When listing symbols from archive members, includes the index, which is a mapping (stored in the archive by ar or ranlib) of what modules contain definitions for what names.
-t radix | --radix=*radix*	Uses *radix* as the radix for printing the symbol values. It must be ‘d’ for decimal, ‘o’ for octal, or ‘x’ for hexadecimal.
-u | --undefined-only	Displays only undefined symbols (those external to each object file).
-l | --line-numbers	Uses debug information to display filename and line number for symbols.

6.5 sdsize

Format

```
sdsize [ -A |B | --format=compatibility ][ -x | --
radix=number ][ objfile... ]
```

Description

The sdsize utility lists the section sizes, and the total size, for each of the object or archive files `objfile` in its argument list. By default, one line of output is generated for each object file or each module in an archive.

`objfile`... are the object files to be examined. If none are specified, the file `a.out` will be used.

Options

Table 6.6 sdsize Options

Option	Description
`-A` | `-B` | `--format=`*`compatibility`*	Using one of these options, you can choose whether the output from `sdsize` resembles output from System V UNIX size (using '`-A`', or '`--format=sysv`'), or Berkeley Software Distribution (BSD) size (using '`-B`', or '`--format=berkeley`'). The default is the one-line format similar to BSD format.
`--help`	Shows a summary of acceptable arguments and options.
`-d` | `-o` | `-x` | `--radix=`*`number`*	Using one of these options, you can control whether the size of each section is given in decimal ('`-d`', or '`--radix=10`'); octal ('`-o`', or '`--radix=8`'); or hexadecimal ('`-x`', or '`--radix=16`'). In '`--radix=number`', only the three values (8, 10, 16) are supported.

Example

Here is an example of formatting the output from `sdsize` closer to System V conventions:

```
    sdsize --format=SysV file1

  file1 :
```

```
section       size        addr
.text       294880        8192
.data        81920      303104
.bss         11592      385024
Total       388392
```

6.6 sdstrings

Format

```
sdstrings [-min-len] [-n min-len] [-t radix]
   [--print-file-name] [--bytes=min-len][--radix=radix]
   file...
```

Description

For each file given, the sdstrings utility prints the printable character sequences that are at least 4 characters long (or the number given with the options below) and are followed by an unprintable character. By default, only strings from the initialized and loaded sections of object files are printed; for other types of files, it prints the strings from the entire file.

`sdstrings` is mainly useful for determining the contents of nontext files.

Options

Table 6.7 sdstrings Options

Option	Description
`-f | --print-file-name`	Prints the name of the file before each string.
`-min-`*`len`* `| -n` *`min-len`* `| --bytes=`*`min-len`*	Prints sequences of characters that are at least *`min-len`* characters long, instead of the default 4.
`-t` *`radix`* `| --radix=`*`radix`*	Prints the offset within the file before each string. The single character argument specifies the radix of the offset:'o' for octal, 'x' for hexadecimal, or 'd' for decimal.

6.7 sdobjdump

Format

```
sdobjdump

   [ -d | --disassemble ]

   [ -f | --file-headers ]

   [ -j section | --section=section ]

   [ -t | --syms ]

   [ -h | --section-headers ]

   [ --start-address=address ]

   [ --stop-address=address ]

   [ --help ]

   objfile...
```

Description

The sdobjdump utility displays information about one or more object files. The options control what particular information to display. This information is most useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work.

objfile... are the object files to be examined. When you specify archives, objdump shows information on each of the member object files.

Options

The long and short forms of options, shown here as alternatives, are equivalent. At least one option from the list must be given.

Table 6.8 sdobjdump Options

Option	Description
`-d` | `--disassemble`	Displays the assembler mnemonics for the machine instructions from `objfile`. This option only disassembles those sections which are expected to contain instructions.
`-f` | `--file-header`	Displays summary information from the overall header of each of the `objfile` files.
`-h` | `--section-header` | `--header`	Displays summary information from the section headers of the object file. You may relocate file segments to nonstandard addresses, for example by using the `-Ttext`, `-Tdata`, or `-Tbss` options to `ld`.
`--help`	Prints a summary of the options to objdump and exit
`-j` name | `--section=`*name*	Displays information only for named section.
`--start-address=`*address*	Starts displaying data at the specified address. This affects the output of the `-d`, `-r` and `-s` options.
`--stop-address=`*address*	Stops displaying data at the specified address. This affects the output of the `-d`, `-r` and `-s` options.
`-t	--syms`	Prints the symbol table entries of the file. This is similar to the information provided by the ‘`nm`’ program.

6.8 sdobjcopy

Format

```
sdobjcopy
   [ -O bfdname | --output-target=bfdname ]
   [ -b byte | --byte=byte ]
   [ -i interleave | --interleave=interleave ]
   [ --gap-fill=val ]
   [ --pad-to=address ]
   [ --set-start=val ] [ --adjust-start=incr ]
   infile [outfile]
```

Description

The sdobjcopy utility copies the contents of an object file to another object file. It uses the GNU BFD Library to read and write the object files. It can write the destination object file in a format different from that of the source object file. The exact behavior of `sdobjcopy` is controlled by command-line options.

`sdobjcopy` generates S-records if you specify an output target of '`srec`' (use '`-O srec`').

`sdobjcopy` generates binary output if you specify an output target of '`binary`' (use '`-O binary`').

`sdobjcopy` generates a raw binary file if you specify an output target of 'binary' (e.g., use '`-O binary`'). When `sdobjcopy` generates a raw binary file, it will essentially produce a memory dump of the contents of the input object file. All symbols and relocation information will be discarded. The memory dump will start at the load address of the lowest section copied into the output file.

When generating an S-record or a raw binary file, it may be helpful to use '`-`*S*' to remove sections containing debugging information. In some cases '-R' will be useful to remove sections which contain information which is not needed by the binary file.

`infile`

`outfile`

The source and output files, respectively. If you do not specify `outfile`, `objcopy` creates a temporary file and destructively renames the result with the name of `infile`.

Options

Table 6.9 sdobjcopy Options

Option	**Description**
`-O` *bfdname* | `--output-target=`*bfdname*	Write the output file using the object format bfdname.
`-b` *`byte`* | `--byte=`*`byte`*	Keep only every byteth byte of the input file (header data is not affected). *byte* can be in the range from 0 to interleave-1, where interleave is given by the `-i` or `--`*`interleave`* option, or the default of 4. This option is useful for creating files to program ROM. It is typically used with an srec output target.
`-i` *`interleave`* | `--interleave=`*`interleave`*	Copy only one out of every *interleave* bytes. Select which byte to copy with the `-b` or `--byte` option. The default is 4. `objcopy` ignores this option if you do not specify either `-b` or `--byte`.
`--gap-fill` *val*	Fill gaps between sections with *val*. This operation applies to the load address (LMA) of the sections. It is done by increasing the size of the section with the lower address, and filling in the extra space created with *`val`*.
`--pad-to` *address*	Pad the output file up to the load address *`address`* by increasing the size of the last section. The extra space is filled in with the value specified by `--gap-fill` (default zero).
`--set-start` *val*	Set the address of the new file to *`val`*. Not all object file formats support setting the start address.

Chapter 7
ZISIM Simulator

This chapter describes the SDK ZSP architecture simulator.

The ZSP SDK functional-accurate simulator, ZISIM, simulates the behavior of the LSI40xZ series of ZSP devices, ZSP400, and ZSPG2 architecture-based designs at the architectural level, including the memory model, the operand register file, and the control register file.

7.1 Using ZISIM

ZSIM can be accessed as a target through the debugger or as a stand-alone program. This chapter describes the interface to ZISIM as a stand-alone program. ZISIM can be used in batch mode or interactively, as described in the following subsections. The commands supported in both modes of operation are described in Section 7.2, "ZISIM Commands," page 7-4.

7.1.1 Batch Mode

The simulator can be invoked in batch mode from the command line using the -exec option, as shown below.

```
   % zisim400 executeable_file -exec [options] for ZSP400
architecture
   % zisimg2 executable_file -exec [options] for ZSPG2
architecture
```

The simulator can also be invoked in batch mode using a script file containing ZSIM commands that load, execute, and gather results for a specified executable. Script files may contain any valid ZISIM commands. Comments must be preceded by the comment specifier (#). ZISIM ignores all commands between the # character and the end of line. ZISIM also ignores empty lines.

A simple script file that turns-on instruction tracing and then executes the program test.exe is shown below.

```
load test.exe
enable trace write
run 100000
exit
```

Assuming the file batch.scr contains the commands shown above, you can generate a trace file for test.exe as follows:

```
   % zisim400 -s batch.scr > test.trace (Unix for ZSP400
architecture)
   % zisimg2 -s batch.scr > test.trace (Unix for ZSPG2
architecture)
   C:\zisim400 -s batch.scr > test.trace (Windows for ZSP400
architecture)
   C:\zisimg2 -s batch.scr > test.trace (Windows for ZSPG2
architecture)
```

Refer also to Section 7.2.21, “script,” page 7-15.

7.1.2 Interactive Mode

In interactive mode, ZISIM is invoked from the shell using the following command:

zisim400 [*executable_file*] [*options*]

zisimg2 [executable_file] options

An executable file may or may not be specified, followed by zero or more command-line options separated by spaces The executable file is a ZSP binary file generated using the SDK compiler, assembler, and linker tools, as explained in other chapters of this document. ZISIM processes the source file according to the specified command-line options (refer to Table 7.1). If no options are specified, ZISIM initializes itself, then prompts the user with the ZISIM prompt:

```
zisim{1}>
```

The simulator is now ready to accept and respond to ZISIM commands, which are described in Section 7.2, “ZISIM Commands,”. An executable file may be loaded from within ZISIM using the load exe command.

An example interactive simulation session is described in Section 7.4, "Example Session Using ZISIM". Refer also to the description of using ZISIM use as the target of the SDK's Debugger in Section 9.2.1, "Functional-Accurate Simulator Connection."

Table 7.1 ZISIM Command-line Options

Option	Description
`-c` *NUM*	Limits number of executed instructions to *NUM*. By default, *NUM* = 2000000000. Execution continues until a breakpoint is reached or the number of executed instructions hit the limit. Use this option to ensure termination of an algorithm.
`-h`	Prints brief usage summary.
`-i` *mode_register=value*	Initializes an architectural control (mode) register with specified value. Note that the control register is written without its usual percent (%) sign, and there are no spaces around the equal sign (=). For example, the option to set %SMODE control register is: `-i smode=0x1234`. The option to set r0 register is `-i r0=0x9876`. Refer to Appendix B, "ZSP400 Control Registers" for information on ZSP400 core-based device control registers.
`-m`	Enables memory trace. ZISIM prints a trace of the execution program to standard output whenever a write to a memory occurs. The format of this output is similar to option `-t`.
`-noiboot`	Fetches instructions from external ROM space. If you do not specify this option, instructions are fetched from internal ROM space. ROM is mapped from 0xf800 to 0xffff. This option is specific to zisim400.
`-radix {dec|hex}`	Displays data in specified radix, either decimal or hexadecimal.
-reg	Enables register trace. All the architectural registers will be displayed after executing an instruction.
`-s` *sourcefile*	Reads all the simulator commands from file.
`-t`	Enables flow trace. ZISIM prints a trace of the executing program to standard output. The information printed includes the instruction sequence number, instruction address, the disassembled instruction and operands, and the resulting architectural state. Example output for the `-t` option is shown in Section 7.4, "Example Session Using ZISIM," page 7-24.
`-exec`	Invokes the simulator in noninteractive mode.
`-v`	Prints version number and exit.

7.2 ZISIM Commands

This section describes commands recognized by the ZISIM command line. Table 7.2 provides a brief summary of commands. The output of any ZISIM command can be sent to a file using the standard redirection identifier (>). For example, the command show `attr > filename` dumps the output of the show command to `filename`.

Table 7.2 ZISIM Command Summary

Command	Modifier	Argument	Description
`alias`	–	[*`tag command_sequence`*]	Creates alias (*`tag`*) for command sequence.
`clear`	`break`	*breakpoint_number*	Clears specified breakpoint.
	`dmem`	{`int` | `ext`} *`addr size`*	Clears internal or external data memory.
	`imem`	{`int` | `ext`} *`addr size`*	Clears internal or external instruction memory.
	`stats`	–	Clears statistics information.
`disable`	`break`	*breakpoint*	Disables specified breakpoint.
	`trace`	{`mem` | `reg` | `write`}	Disables run-time instruction tracing.
`dump`	`dmem`	{`int` | `ext`} *`filename addr size`*	Dumps internal or external data memory range to a text file.
	`imem`	{`int` | `ext`} *`filename addr size`*	Dumps internal or external instruction memory range to a text file.
`enable`	`break`	*breakpoint_number*	Enables breakpoint.
	`trace`	{`mem` | `reg` | `write`}	Enables run-time instruction tracing.
`exit`	–	–	Exits simulation session.
`fill`	`dmem`	{`int` | `ext`} *`addr size value`*	Fills internal/external data memory range with *`value`*.
	`imem`	{`int` | `ext`} *`addr size value`*	Fills internal/external instruction memory range with *`value`*.
(Sheet 1 of 3)			

Table 7.2 ZISIM Command Summary (Cont.)

Command	Modifier	Argument	Description
`help`	–	{category | command}	Prints list of commands in a category or command usage.
`load`	dmem	{int | ext} *`filename addr size`*	Loads internal/external data memory from file.
	exe	*`filename`*	Loads ZSP executable into instruction memory from file.
	imem	{int | ext} *`filename addr size`*	Loads internal/external instruction memory from file.
`reset`	–	{hard | soft}	Resets simulator.
`run`	–	[*`number_of_instructions`*]	Runs for specified number of simulation instructions.
`script`	–	*`filename`*	Loads and execute ZISIM script file.
`set`	attr	{history | radix | run} *`value`*	Assigns *`value`* to specified attribute.
	break	pc *`addr`*	Creates a new breakpoint at the specified PC address.
	break	*`symbol label`*	Creates a new breakpoint at the specified label.
	reg	*`register value`*	Assigns *`value`* to specified register.
(Sheet 2 of 3)			

Table 7.2 ZISIM Command Summary (Cont.)

Command	Modifier	Argument	Description
`show`	`attr`	{`run` | `history` | `radix` | `version`}	Shows value of the specified attribute.
	`bits`	*register*	Displays the bit-level states for the specified register.
	`break`	–	Displays list of defined breakpoints.
	`dmem`	{`int` | `ext`} *`addr size`*	Shows contents of a region of internal/external data memory.
	`imem`	{`int` | `ext`} *`addr size`*	Shows contents of a region of internal/external instruction memory.
	`reg`	{*`category`* | *`reg`*}...	Shows contents of register or register set.
	`stats`	[*`opcode`*]	Shows current run-time statistics.
	`trace`	–	Shows trace information during simulation.
`step`	–	–	Advances simulation by one instruction. Same as `run 1`.
`unalias`	–	*alias*	Deletes *`alias`*.
(Sheet 3 of 3)			

Table 7.3 ZISIM400 specific commands

Command	Modifier	Argument	Description
set	size	[dmem|imem] size	Set internal instruction or internal data memory size starting from 0. Default size is maximum value of 0xf800 words.
show	size	[dmem|imem]	Show size of internal instruction or data memory

Table 7.4 ZISIMG2 specific commands

Command	**Modifier**	**Argument**	**Description**
set	size	[dmem|imem] [int|ext] beg_value end_value	Set the size of internal/external instruction or data memory starting from beg_value to end_value including the boundary. Each memory block could overlap one another. Default value for each of them is from 0 to 0x00ffffff words.
show	size	[dmem|imem] [int|ext]	Show the current size of internal/external instruction or data memory.

7.2.1 alias

The `alias` command allows the user to create ZISIM commands by aliasing new commands to existing commands or sequences of commands. Sequences of commands must be contained in quotes and separated by semicolons. Issuing the `alias` command without arguments shows all current aliases.

Format

```
alias tag command_sequence
```

Examples

```
zisim{32} **alias r0 show reg r0**
zisim{32} **alias adv “step ; show pipe ; show reg gpr”**
zisim{32} **alias**
adv     step ; show pipe ; show reg gpr
r0      show reg r0
zisim{33}
```

7.2.2 clear break

This command deletes a breakpoint from the current list of defined breakpoints. The breakpoint number is assigned when a breakpoint is set. Use the `show break` command to display a list of breakpoints.

Format

```
clear break breakpoint_number
```

Example

```
zisim{32} clear break 5
```

7.2.3 clear dmem

This command clears the contents of internal or external data memory. User specifies internal or external memory, the starting address, and the size of the region to clear.

Format

```
clear dmem {int|ext} addr size
```

Example

```
zisim{32} clear dmem int 0x1000 0x0100
```

7.2.4 clear imem

This command clears the contents of internal or external instruction memory. User specifies internal or external memory, the starting address, and the size of the region to clear.

Format

```
clear imem {int|ext} addr size
```

Example

```
zisim{32} **clear imem ext 0x7000 0x1000**
```

7.2.5 clear stats

This command clears all run-time statistic information.

Format

```
clear stats
```

7.2.6 disable break

This command disables a breakpoint from the list of active breakpoints. Use the show break command to display a list of current breakpoints.

Format

```
disable break breakpoint_number
```

Example

```
zisim{32} disable break 4
```

7.2.7 disable trace

This command disables specified trace. See the `enable trace` command for a description of the trace types.

Format

```
disable trace {mem|reg|write}
```

Examples

```
zisim{32} disable trace pipe
zisim{32} disable trace reg
```

7.2.8 dump dmem

This command generates a text file representing the contents of the specified address range of internal or external data memory. The user specifies internal or external memory, the starting address, and the size of the region to dump.

Format

```
dump dmem {int|ext} filename addr size
```

Example

```
zisim{32} dump dmem ext data.dat 0x0000 0xffff
% cat data.dat
0000   /* 0x0000 */
0000   /* 0x0001 */
0000   /* 0x0002 */
0000   /* 0x0003 */
0000   /* 0x0004 */
0000   /* 0x0005 */
0000   /* 0x0006 */
...
28e2   /* 0x00fd */
2f6a   /* 0x00fe */
325d   /* 0x00ff */
%
```

7.2.9 dump imem

This command generates a text file representing the contents of the specified address range of internal or external instruction memory. The user specifies internal or external memory, the starting address, and the size of the region to dump.

Format

```
dump imem {int|ext} filename addr size
```

Example

```
zisim{32} dump imem int imem.dat 0x1000 0x30

% cat imem.dat
0000   /* 0x1000 */
0000   /* 0x1001 */
0000   /* 0x1002 */
0000   /* 0x1003 */
...
0000   /* 0x102c */
0000   /* 0x102d */
0000   /* 0x102e */
0000   /* 0x102f */
%
```

7.2.10 enable break

This command enables a breakpoint from the current list of defined breakpoints. Use the show break command to display a list of current breakpoints.

Format

```
enable break breakpoint_number
```

Example

```
zisim{32} enable break 1
```

7.2.11 enable trace

This command enables a predefined trace type. There are three types of predefined run-time tracing. Run-time traces generate text output instruction by instruction. The three trace types are:

- `mem`

  Displays address and data for any memory location which is updated. Information is generated after the instruction is executed.

- `reg`

  Displays all registers and register values every instruction.

- `write`

  Displays architectural state changes associated with memory or registers for each instruction.

Format

```
enable trace {mem|reg|write}
```

Example

```
zisim{32} enable trace write
```

7.2.12 exit

This command terminates the current simulation session.

Format

```
exit
```

7.2.13 fill dmem

This command fills internal or external data memory range with specified value. User specifies internal or external memory, the starting address, and the size of the region to fill.

Format

```
fill dmem {int|ext} addr size value
```

Example

```
zisim{32} fill dmem ext 0x1000 0xff 0x0505
```

7.2.14 fill imem

This command allows you to specify internal or external memory, the starting address, and the size of the region to fill.

Format

```
fill imem {int|ext} addr size value
```

Example

```
zisim{32} fill imem ext 0x1000 0xff 0x0505
```

7.2.15 help

This command displays help information. Help is available for individual commands as well as for command categories. Specifying a command displays the description and usage for that command. Requesting help for a specified category displays the instructions associated with that category. Commands are categorized according to their function (for instance, all show commands).

Issuing the `help` command with no other specifiers displays help on the command categories.

Format

```
help [category|command]
```

Examples

```
zisim{32} help
```

```
zisim{32} **help all**
zisim{32} **help show**
zisim{32} **help show reg**
```

7.2.16 load dmem

This command loads internal or external data memory from specified text file. User specifies internal or external memory, the starting address, and the size of the region to load. The format of the text file should be the same as the file produced by the dump command. The first column contains the data that will be loaded, with each data on a single line. Data must be in hex format with out 0x prefix. Comments must be enclosed by '`/* */`'.

Format

```
load dmem {int|ext} filename addr size
```

Example

```
zisim{32} load dmem int data.dat 0x1000 0x0fff
```

The output format of the file is:

```
%cat data.dat
2ce5   /* 0x0000 */
3c3f   /* 0x0001 */
2000   /* 0x0002 */
3006   /* 0x0003 */
a00f   /* 0x0004 */
80c0   /* 0x0005 */
...
```

7.2.17 load exe

This command loads a valid ZSP executable into instruction memory. This command performs the same function as specifying the executable filename when ZISIM is invoked. Without the filename specified, this command reloads the previous executable program into memory.

Format

```
load exe {filename}
```

Example

```
   zisim{32} load exe test.exe
or
   zisim{32} load test.exe
```

7.2.18 load imem

This command loads internal or external instruction memory from specified text file. You must specify internal or external memory, the starting address, and the size of the region to load. You must ensure that the format of the text file is the same as the file produced by the `dump` command. The first column contains the data that will be loaded, with each data on a single line. Data must be
in hex format without the `0x` prefix. Comments must be enclosed by '`/* */`'.

Format

```
load imem {int|ext} filename addr size
```

Example

```
% cat inst.txt
2ce5   /* 0x0000 */
3c3f   /* 0x0001 */
2000   /* 0x0002 */
3006   /* 0x0003 */
a00f   /* 0x0004 */
80c0   /* 0x0005 */
bc4c   /* 0x0006 */
6f4c   /* 0x0007 */

zisim{32} load imem int inst.txt 0x1000 8
```

7.2.19 reset

This command resets the state of the simulator. A soft reset initializes all aspects of the simulator except the memory. A hard reset also initializes memories. Issuing the `reset` command without options performs a soft reset.

Format

```
reset [soft|hard]
```

Examples

```
zisim{32} reset soft
zisim{32} reset hard
```

Note that the `reset` command does not reload the program into memory. In order to restart the program, perform one of the following sequence of commands:

```
zisim{32} reset
zisim{32} set reg pc <start_address>
```

or

```
zisim{32} reset hard; load
zisim{33} load
```

`Note: zisimg2 doesn’t support reset soft feature.`

7.2.20 run

This command advances the simulator the specified number of instructions. The simulator uses the value of the `run` attribute if no instruction count is specified. Simulation halts if instruction count is reached, the maximum instruction count is reached, or a system halt occurs.

Format

```
run [number_of_instructions]
```

Examples

```
zisim{32} run
zisim{32} run 100
```

7.2.21 script

This command loads and processes the script file. Script files may contain any valid ZISIM commands. Comments are allowed in the script file; the comment specifier is the # character. ZISIM ignores all commands between the # character and the end of line. Empty lines are also ignored.

Format

```
   script filename
```

Example

```
   zisim{32} script standard.scr
```

Sample Script File

A simple script is shown below.

```
# This example script demonstrates how to turn on
# instruction tracing using a command.
load test.exe
enable trace write
run
exit
```

7.2.22 set attr

The `set attr` command allows you to set three internal ZISIM variables. Table 7.5 shows the configurable ZISIM attributes.

Table 7.5 Configurable ZISIM Attributes

Attribute	Value	Description
`history`	any integer	Number of commands to maintain in history buffer.
`radix`	[int | hex]	Radix (integer or hexadecimal) used to generate output.
`run`	any integer	Default instruction count for the `run` command (when issuing the `run` command with no argument). If undefined, the default value of the run attribute is 2000000000.

Format

```
   set attr [history|radix|run] value
```

Examples

```
   zisim{32} set attr run 1000
   zisim{32} set attr radix hex
```

7.2.23 set break

This command creates and enables a new breakpoint at specified address. Execution halts when the PC reaches the specified address. When a new breakpoint is created, ZISIM tags it with a breakpoint number which is used for other breakpoint commands (use the `show break` command to view a list of current breakpoints).

Format

```
set break pc addr
set break symbol label
```

Example

```
zisim{2} set break pc 0x0010
Breakpoint 1 on PC at address 0x0010
zisim{3} set break symbol main
Breakpoint 2 on PC at address 0xf9b9 of main
```

7.2.24 set reg

This command assigns a value to the specified register.

Format

```
set reg register value
```

Example

```
zisim{32} set reg r0 0x1234
```

7.2.25 set size

7.2.25.1 zisim400

This command sets the size of internal data memory or instruction memory. The default size of internal data or instruction memory is 63488 words (62K words), which is also the maximum size that can be set.

This command does not apply to external memory. (The simulator has 1M words for each external instruction and external data memory.)

Format

```
set size {dmem|imem} size
```

Examples

```
   zisim{32} set size dmem 0x4000
```

This command sets the size of internal data memory to 16 Kwords.

```
   zisim{32} set size imem 0x4000
```

This command sets the size of internal instruction memory to 16 Kwords

7.2.25.2 zisimg2

This command sets the size of internal/external data memory or instruction memory. The default size of internal/external data or instruction memory is 0xffffffwords (16M words) starting from 0, which is also the maximum size that can be set.

This command does not apply to external memory. (The simulator has 1M words for each external instruction and external data memory.)

Format

```
   set size {dmem|imem} {int|ext} beg_value end_value
```

Examples

```
   zisim{32} set size dmem int 0 0xffff
```

This command sets the size of internal data memory to 16 Kwords.

```
   zisim{32} set size imem int 0 0xffff
```

This command sets the size of internal instruction memory to 16 Kwords.

.

7.2.26 show attr

This command shows the value of the specified attribute. You can view the value of the three attributes which are configurable with the `set attr` command as well as view version information for ZISIM.

Format

```
   show attr {run|history|radix|version}
```

Example

```
zisim{1} show attr run
zisim{2} show attr history
zisim{3} show attr radix
zisim{4} show attr version
```

7.2.27 show bits

This command displays the bit field values for the specified register. Do not use the % specifier for control registers.

Format

```
show bits register
```

Example

```
zisim{32} show bits hwflag
hwflag = 0x0000
        er: 0
        ex: 0
        ir: 0
         z: 0
        gt: 0
        ge: 0
         c: 0
       gsv: 0
        sv: 0
        gv: 0
         v: 0
```

7.2.28 show break

This command displays the list of currently defined breakpoints.

Format

```
show break
```

Example

```
zisim{32} show break
Num ID Address   Status
-----------------------
 2  PC  0x2000   Active
 1  PC  0xf9b9   Active
```

7.2.29 show dmem

This command displays a range of internal or external data memory. You must specify internal or external memory, the starting address, and the size of the region to display. The default settings for the `show dmem` command are shown in Table 7.6.

Table 7.6 Default Arguments for show dmem

Argument	Value
{int | ext}	int
addr	0x0
size	16

Format

```
   show dmem {int|ext} addr size
```

Example

```
   zisim{32} show dmem int 0xf000 0x10
```

```
For zisimg2, user can use a symbol instead of an absolute
address value.
   zisim{1} show dmem int array1 20
```

7.2.30 show imem

This command displays a range of internal or external instruction memory. User specifies internal or external memory, the starting address, and the size of the region to show. The *size* and *addr* fields may be omitted, in which case defaults are used. The default settings for the `show imem` command are shown in Table 7.7.

Table 7.7 Default Arguments for show imem

Argument	Value
{int | ext}	int
addr	0x0
size	16

Format

```
show imem {int|ext} [addr] [size]
```

Example

```
zisim{32} show imem int 0xf000 0x10
```

For zisimg2, user can use a symbol instead of an absolute address value.

```
zisim{1} show imem int foo_function 20
```

7.2.31 show reg

This command displays the value of a specified register or the value of a category of registers. More than one category and/or register can be specified. The register categories are:

- `gpr`

  All general purpose registers, r0–r15.

- `cfg`

  All control registers (such as %smode and %hwflag). Do not include the percent sign (%) in the register name.

- `addr`

  All address and index registers for the ZSPG2 architecture. Thus, it is specific for zisimg2.

Format

```
show reg {category|register} ...
```

Examples

```
zisim{32} show reg
zisim{32} show gpr
zisim{32} show cfg r0
zisim{32} show gpr hwflag smode
```

7.2.32 show size

7.2.32.1 zisim400

This command shows the size of internal data or instruction memory. The output is not affected by the radix attribute.

Format

```
show size {dmem|imem}
```

Examples

```
zisim{32} show size dmem
zisim{32} show size imem
```

7.2.32.2 zisimg2

This command shows the size of internal/external data or instruction memory. The output is not affected by the radix attribute.

Format

```
show size {dmem|imem}{int|ext}
```

Examples

```
zisim{32} show size dmem int
zisim{32} show size imem int
```

7.2.33 show stats

This command displays run-time statistics collected by ZISIM. If no argument is specified, ZISIM displays overall statistical information. If the opcode argument is specified, ZISIM displays instruction opcode statistics.

Format

```
show stats [opcode]
```

Examples

```
zisim{32} show stats
zisim{32} show stats opcode
```

7.2.34 show trace

This command shows currently enabled/disabled trace information. Traces currently set to `ON` are enabled during simulation.

Format

```
show trace
```

Example

```
zisim{32} show trace
***(info) Supported trace information:
 - Instruction trace: OFF
 - Register trace:    OFF
 - Memory trace:      OFF
zisim{33}> enable trace write
***(info) Instruction trace is ON.
zisim{34}> show trace
***(info) Supported trace information:
 - Instruction trace: ON
 - Register trace:    OFF
 - Memory trace:      OFF
```

7.2.35 step

This command single-steps the simulator. Issuing the `step` command is equivalent to issuing the command `run 1`.

Format

```
step
```

Example

```
zisim{32} step
```

7.2.36 unalias

This command deletes an alias. (Use the `alias` command to display a list of currently defined aliases.)

Format

```
unalias alias
```

Example

```
zisim{32} unalias adv
```

7.3 I/O Port Usage

ZISIM400 models serial I/O as a memory-mapped device. Programs perform terminal I/O by reading from and writing to the appropriate address locations. The simulator defines two serial ports and one host processor interface (HPI) port. Each port has a transmit buffer and a receive buffer. Table 7.8 shows the memory addresses and corresponding files for the I/O ports for the LSI402ZX, LSI403Z, and ZSP400-core based devices.

Table 7.8 I/O Device Memory Map and Associated Files

	Read		**Write**	
I/O Port	**Address**	**File**	**Address**	**File**
Serial Port 0	0xF901	`sp0in`	0xF900	`sp0out`
Serial Port 1	0xFA01	`sp1in`	0xFA00	`sp1out`
Host Interface Port	0xFB01	`hpiin`	0xFB00	`hpiout`

The format of input and output files is the same. Data must be in decimal digits, with each data on a single line. If the input file is not present in the current running directory at the time of the request, the simulator will print an error message to standard output and exit.

7.4 Example Session Using ZISIM

This section contains an example simulation session using ZISIM in interactive mode.

In the example simulation, `demo.exe` is invoked using the `-t` (enable trace) command-line option. Trace information is displayed in five fields:

```
(0) 0x2000 2cfb movl      r12, 0xfb       ! r12 = 0x00fb
```

- The first field is the instruction sequence number (in parenthesis).

- The second field is the program counter (PC) of the executed instruction.
- The third field is the instruction opcode.
- The fourth field is the disassembled instruction, including operands.
- The fifth field describes the result of the executed instruction.

The trace shown in this example is for the ZSP400 core. The text is linked and loaded at 0x2000.

```
(shell prompt) zisim400 demo.exe -t
*************************************************
               ZISIM     1.206
                  ZSP400
          Instruction Set Simulator

                  LSI Logic
*************************************************
***(info) Starting address: 0x2000
.text   : Loading to INT-INST memory ... 0x2000 -> 0x2950 (0x0951)
.data   : Loading to INT-DATA memory ... 0x0001 -> 0x005f (0x005f)
Loading "demo.exe" successfully.
zisim{1}_
```

If you do not specify a test for initialization, you can load a test from the ZISIM command line. Check the contents of the instruction memory to confirm proper loading of the test. These steps are demonstrated below.

```
zisim{1}show imem int 0x2000 4
0x2000   0x2cfb  movl      r12, 0xfb
0x2001   0x3cf7  movh      r12, 0xf7
0x2002   0xa6d0  mov       r13, 0x0
0x2003   0x2460  movl      r4, 0x60
zisim{2}> _
```

Instruction fetch begins at the entry point you specify in an executable program. You can change this before execution begins by setting the PC to the desired value using the `set reg` command.

The simulator output below demonstrates use of the PC breakpoint: a breakpoint is set for address 0x10 and the simulator advances until the PC reaches address 0x10.

```
zisim{3}> set break pc 0x2050
Breakpoint 1 on PC at address 0x2050
zisim{4}> set break symbol main
Breakpoint 2 on PC at address 0x2010 of main
zisim{5}> run
(0) 0x2000 2cfb movl      r12, 0xfb       !            r12 = 0x00fb
(1) 0x2001 3cf7 movh      r12, 0xf7       !            r12 = 0xf7fb
(2) 0x2002 a6d0 mov       r13, 0x0        !            r13 = 0x0000
(3) 0x2003 2460 movl      r4, 0x60        !             r4 = 0x0060
(4) 0x2004 3400 movh      r4, 0x0         !             r4 = 0x0060
(5) 0x2005 bc54 mov       r5, r4          !             r5 = 0x0060
(6) 0x2006 a051 add       r5, 0x1         !         hwflag = 0x0030
(6) 0x2006 a051 add       r5, 0x1         !             r5 = 0x0061
(7) 0x2007 6054 st        r5, r4, 0       ! INT-DATA[0x0060] = 0x0061
(8) 0x2008 bb1d mov       rpc, r13        !            rpc = 0x0000
(9) 0x2009 2510 movl      r5, 0x10        !             r5 = 0x0010
(10) 0x200a 3520 movh      r5, 0x20        !             r5 = 0x2010
(12) 0x200c a750 call      r5              !            rpc = 0x200d
(PC BREAKPOINT #2)................ Instruction Count=000013 PC=0x2010
zisim{6}> show reg gpr
              r0 = 0x0000              r1 = 0x0000
              r2 = 0x0000              r3 = 0x0000
              r4 = 0x0060              r5 = 0x2010
              r6 = 0x0000              r7 = 0x0000
              r8 = 0x0000              r9 = 0x0000
             r10 = 0x0000             r11 = 0x0000
             r12 = 0xf7fb             r13 = 0x0000
             r14 = 0x0000             r15 = 0x0000
zisim{7}> disable trace write
```

After the final command, the simulator will no longer print the instruction flow trace.

```
zisim{8}> run
Hello World!
(SYSTEM HALT)..................... Instruction Count=000673 PC=0x200e
```

Execution halts when a breakpoint is reached, a system halt occurs, or the maximum instruction count is reached. A system halt refers to setting halt mode as defined by the %smode control register. Execution statistic information can be seen by using show `stats` command.

```
zisim{9}> show stats
     673  instructions executed
      88  load instructions   ( 13.08%)
      65  -  single           (  9.66%)
      23  -  double           (  3.42%)
      56  store instructions  (  8.32%)
      37  -  single           (  5.50%)
      19  -  double           (  2.82%)
     104  discontinuities     ( 15.45%)
      15  -  calls            (  2.23%)
      63  -  conditional      (  9.36%)
      10  -  agnx             (  1.49%)
      25  mispredicts         ( 39.68% of conditional branch)
```

Terminate the simulation session with the exit command.

```
zisim{10}> exit
***(info) Exiting ZISIM.
```

Chapter 8
ZSIM Simulator

This chapter describes the ZSP SDK cycle-accurate architecture simulator.

The ZSP SDK simulator ZSIM is a cycle-accurate simulator for ZSP400 and ZSPG2 architecture-based devices. ZSIM models the architectural features necessary for cycle-by-cycle tracing of architectural state, including the execution pipeline, instruction and data caches, internal and external instruction/data memories, and register files.

8.1 Using ZSIM

ZSIM can be accessed as a target through the debugger or as a stand-alone program. This chapter describes the interface to ZSIM as a stand-alone program. ZSIM can be used in batch mode or interactively, as described in the following subsections. The commands supported in both modes of operation are described in Section 8.2, "ZSIM Commands," page 8-5.

8.1.1 Batch Mode

The simulator can be invoked in batch mode from the command line using the -exec option, as shown below.

```
% zsim[400/g2] executeable_file -exec [options]
```

The simulator can also be invoked in batch mode using a script file containing ZSIM commands that load, execute, and gather results for a specified executable. Script files may contain any valid ZSIM commands. Comments are allowed and must be preceded by the comment specifier (#). ZSIM ignores all commands between the # character and the end of line. ZSIM also ignores empty lines.

A simple script file that turns on instruction tracing and then executes the program test.exe is shown below.

```
load test.exe
enable trace write
run 100000
exit
```

Assuming the file batch.scr contains the commands shown above, a trace file for test.exe could be generated as follows:

```
   % zsim400 -s batch.scr > test.trace (Unix for ZSP400
architecture)
   % zsimg2 -s batch.scr > test.trace (Unix for ZSPG2
architecture)
   C:\zsim400 -s batch.scr > test.trace (Windows for ZSP400
architecture)
   C:\zsimg2 -s batch.scr > test.trace (Windows for ZSPG2
architecture)
```

Refer also to Section 8.2.26, "script," page 8-21.

8.1.2 Interactive Mode

In interactive mode, ZSIM is invoked from the command line using the following command:

For ZSP400 architecture:

`zsim400 [`*`executable_file`*`] [`*`options`*`]`

For ZSPG2 architecture:

```
zsimg2 [executeable_file] [options]
```

You may optionally specify an executable file, followed by zero or more command-line options, which must be separated by spaces

The executable file is a ZSP binary file generated using the SDK compiler, assembler, and linker tools, as explained in other chapters of this document. ZSIM processes the source file according to the specified command-line options (refer to Table 8.1).

If no options are specified, ZSIM initializes itself, then prompts the user with the ZSIM prompt:

```
zsim{1}>
```

The simulator is now ready to accept and respond to ZSIM commands, which are described in Section 8.2, “ZSIM Commands” on page 8-5. An executable file may be loaded from within ZSIM using the `load exe` command.

An example interactive simulation session is described in Section 8.4, “Example Session Using ZSIM” on page 8-34. Refer also to the description of using ZSIM use as the target of the SDK’s Debugger in Section 9.2.2, “Cycle-Accurate Simulator Connection,” page 9-4.

Table 8.1 ZSIM Command-line Options

Option	Description
`-exec`	Invokes the simulator in noninteractive mode.
`-c` *`num`*	Specifies maximum cycle count. Execution aborted after *num* cycles.
`-h`	Prints brief usage summary.
`-i` *`mode_register=value`*	Initializes an architectural control (mode) register with specified value. The control register is written without its usual percent (%) sign, and there are no spaces around the equal sign (=). For example, the option to set %smode control register is: `-i smode=0x1234`. The option to set r0 register is `-i r0=0x9876`. Refer to Appendix B, "ZSP400 Control Registers" for information on ZSP400 core-based device control registers or Appendix D“ZSPG2 Control Registers”
`-m`	Turns on memory trace.
`-p`	Turns on pipeline trace.
`-pf`	Turns on all profile information.
`-pfiu`	Turns on instruction unit profile information.
`-pfpipe`	Turns on pipeline unit profile information.
(Sheet 1 of 2)	

Table 8.1 ZSIM Command-line Options (Cont.)

Option	Description
`-q`	Suppresses startup banner.
`-radix` {dec | hex}	Displays data in the specified radix, either decimal (dec) or hexadecimal (hex).
`-reg`	Turns on register trace.
`-s` *`sourcefile`*	Executes the specified script file following initialization.
`-t`	Turns on instruction trace.
`-v`	Prints ZSIM version number.
(Sheet 2 of 2)	

Table 8.2 Command-line Options for zsim400

Options	Description
`-wed` *`num`*	Sets EXT-DATA memory wait state to be *`num`*. Default is 1.
`-wei` *`num`*	Sets EXT-INST memory wait state to be *`num`*. Default is 1.
`-sid` *`num`*	Sets INT-DATA memory size to be *`num`*. Default is 63488 words.
`-sii` *`num`*	Sets INT-INST memory size to be *`num`*. Default is 63488 words.
`-mempcr num`	Sets the MEMPCR address to be *`num`*. Default is 0xf807.
`-nomempcr`	Indicates that the system does not have MEMPCR.
`-noiboot`	Sets the IBOOT signal LOW to boot from external ROM. If this option is not specified, instructions are fetched from internal ROM space. ROM is mapped from 0xf800 to 0xffff.
`-pfdu`	Turns on data unit profile information.

Table 8.3 Command-line Options for zsimg2

Options	Description
-pflsu	Turn on Load Store Unit profile information
-tic	Turn on instruction cache trace every cycle
-svtadd ADDR	Set system vector table address to be ADDR
-idealmss	Use ideal memory subsystem with zero delay for internal memory and no checking for banking conflict between 2 data access ports.
-bimlib LIBNAME	Use bus interface library LIBNAME to run in co-simulation enviroment such as SWIFT or CVE Seamless.
-cpilib LIBNAME	Use co-processor library LIBNAME. SDK tools comes with an example G711 co-processor library called libzcpig711.so on Solaris or libzcpig711.dll on Windows platform.

8.2 ZSIM Commands

The ZSIM commands are described briefly in Table 8.4 and in detail in the following subsections.

The output of any ZSIM command can be sent to a file using the standard redirection identifier (>). For example, the command `show attr > mydisplay` writes the output of the show command in the file `mydisplay`.

Table 8.4 ZSIM Command Summary

Command	Modifier	Argument	Description
`alias`	–	[*tag command_sequence*]	Creates alias (*tag*) for command sequence.
(Sheet 1 of 4)			

Table 8.4 ZSIM Command Summary (Cont.)

Command	Modifier	Argument	Description
`clear`	`break`	*`breakpoint_number`*	Clears specified breakpoint.
	`dmem`	{`int` | `ext`} *`addr size`*	Clears internal or external data memory.
	`icache`	–	Clears instruction cache.
	`imem`	{`int` | `ext`} *`addr size`*	Clears internal or external instruction memory.
	`stats`	–	Clears statistic information.
`disable`	`break`	*`breakpoint_number`*	Disables specified breakpoint.
	`profile`	[`du` | `iu` | `pipe`]	Disables profile information.
	`trace`	{`pipe` | `reg`}	Disables run-time tracing.
`dump`	`dmem`	{`int` | `ext`} *`filename addr size`*	Dumps internal or external data memory to a text file *`filename`*.
	`imem`	{`int` | `ext`} *`filename addr size`*	Dumps internal or external instruction memory to a text file *`filename`*.
`enable`	`break`	*`breakpoint_number`*	Enables breakpoint.
	`profile`	{`iu` | `pipe`}	Enables module profile information.
	`trace`	{`mem` | `pipe` | `reg` | `write` | `icache`}	Enables run-time cycle tracing.
`exit`	–	–	Exits simulation session.
`fill`	`dmem`	{`int` | `ext`} *`addr size value`*	Fills internal/external data memory segment with *`value`*.
	`imem`	{`int` | `ext`} *`addr size value`*	Fills internal/external instruction memory segment with *`value`*.
`help`	–	{*`category`* | *`command`*}	Prints list of commands in a category or command usage.
`istep`	–	-	Advances the simulator by one instruction.
(Sheet 2 of 4)			

Table 8.4 ZSIM Command Summary (Cont.)

Command	Modifier	Argument	Description
load	dmem	{int | ext} *filename addr*	Loads internal/external data memory from file.
	exe	*filename*	Loads ZSP executable into instruction memory.
	imem	{int | ext} *filename addr*	Loads internal/external instruction memory from file.
reset	hard	{hard | soft}	Reset simulator (hard or soft).
run	–	[*number_of_cycles*]	Runs for specified number of simulation cycles.
script	–	*filename*	Loads and executes ZSIM script file.
set	attr	{history | radix | run} *value*	Assigns *value* to specified attribute.
	break	pc *addr*	Creates a new breakpoint at the specified PC address.
	break	symbol label	Creates a new breakpoint at the specified label.
	reg	*register value*	Assigns *value* to specified register.
(Sheet 3 of 4)			

Table 8.4 ZSIM Command Summary (Cont.)

Command	Modifier	Argument	Description
`show`	`attr`	{`history` | `radix` | `run` | `version`}	Shows value of the specified attribute.
	`bits`	*`register`*	Displays the bit-level states for the specified register.
	`break`	–	Shows list of defined breakpoints.
	`dmem`	{ `int` | `ext`} *`addr size`*	Shows contents of a region of internal/external data memory.
	`icache`	–	Shows current instruction cache contents.
	`imem`	{`int` | `ext`} *`addr size`*	Shows contents of a region of internal/external instruction memory.
	`pipe`	–	Shows contents and state of execution pipeline.
	`profile`	–	Displays supported profile information.
	`reg`	{*`category`* | *`reg`*}...	Shows contents of register or register set.
	`rule`	–	Shows the affected grouping rule in the current cycle.
	`size`	{`dmem` | `imem`}	Shows size of internal data or instruction memory.
	`stats`	–	Shows current run-time statistics.
	`trace`	–	Shows the current status of all tracing attributes.
`step`	–	–	Advances simulation by one cycle. Same as `run 1`.
`unalias`	–	*`alias`*	Deletes *`alias`*.
(Sheet 4 of 4)			

Table 8.5 ZSIM400 specific commands

Command	Modifier	Argument	Description
clear	dcache	-	Clear data cache.
set	delay	[edata|einst] num	Sets wait state for external memory. Default for both external data and instruction memory is 1.
set	size	[dmem|imem] size	Set internal instruction or internal data memory size starting from 0. Default size is maximum value of 0xf800 words.
show	size	[dmem|imem]	Show size of internal instruction or data memory
show	dcache	-	Show data cache contents.
enable	profile	du	Enable profile information on Data Unit.
disable	profile	du	Disable profile information on Data Unit.

Table 8.6 ZSIMG2 specific commands

Command	Modifier	Argument	Description
set	latency	[dmem|imem] [int|ext] num	Set wait state latency for internal/external instruction or data memory. Default value for internal memory is 2 and external memory is 5.
show	latency	[dmem|imem] [int|ext]	Show wait state latency for internal/external instruction or data memory.
set	size	[dmem|imem] [int|ext] beg_value end_value	Set the size of internal/external instruction or data memory starting from beg_value to end_value including the boundary. Each memory block could overlap one another. Default value for each of them is from 0 to 0x00ffffff words.
show	size	[dmem|imem] [int|ext]	Show the current size of internal/external instruction or data memory.
show	operands	instruction_number	Show operand values of an instruction number. Instruction number can be obtained by looking at the output of "show pipe" command.
show	stats	grouping	Display the statistic of grouping rule.
enable	profile	lsu	Turn on profile information of Load Store Unit.
disable	profile	lsu	Turn off profile information of Load Store Unit.

8.2.1 alias

This command creates an alias for a ZSIM command. This command allows you to customize the ZSIM commands by aliasing new commands to existing commands or sequences of commands. Sequences of commands must be contained in quotes and separated by semicolons. Issuing the `alias` command without arguments displays all current aliases.

Format

```
alias [tag] [command_sequence]
```

Examples

```
zsim{32} alias r0 show reg r0
zsim{32} alias adv “step ; show pipe ; show reg gpr”
zsim{32} alias
adv     step ; show pipe ; show reg gpr
r0      show reg r0
zsim{33}
```

8.2.2 clear break

This command deletes a breakpoint from the current list of defined breakpoints. The breakpoint number is assigned when a breakpoint is set. Use the `show break` command to display a list of breakpoints.

Format

```
clear break breakpoint_number
```

Example

```
zsim{32} clear break 5
```

8.2.3 clear dcache

This command invalidates the contents of the data cache.

Format

```
clear dcache
```

Example

```
zsim{32} **clear dcache**
```

8.2.4 clear dmem

This command clears the contents of internal or external data memory. User specifies internal or external memory, the starting address, and the size of the region to clear.

Format

```
clear dmem {int|ext} addr size
```

Example

```
zsim{32} clear dmem int 0x1000 0x0100
```

8.2.5 clear icache

This command clears the contents of the instruction cache.

Format

```
clear icache
```

Example

```
zsim{32} clear icache
```

8.2.6 clear imem

This command clears the contents of internal or external instruction memory. User specifies internal or external memory, the starting address, and the size of the region to clear.

Format

```
clear imem {int|ext} addr size
```

Example

```
zsim{32} clear imem ext 0x7000 0x1000
```

8.2.7 clear stats

This command clears all the run-time statistical information, which includes the cycle count, the number of executed instructions, and the number of instructions that are being grouped in the pipe.

Format

```
clear stats
```

Example

```
zsim{32} clear stats
```

8.2.8 disable break

This command disables a breakpoint from the current list of active breakpoints. (Use the `show break` command to display current list.)

Format

```
disable break breakpoint_number
```

Example

```
zsim{32} disable break 4
```

8.2.9 disable profile

This command disables specified type of profile information. If no profile type is specified, the command will disable all types. Profile types are described in Section 8.2.14, “enable profile,” page 8-15.

Format

```
disable profile [du|iu|pipe]
```

Examples

```
zsim{32} disable profile du
zsim{32} disable profile iu
zsim{32} disable profile pipe
zsim{32} disable profile
```

8.2.10 disable trace

This command disables specified type of trace. Trace types are described in Section 8.2.15, “enable trace,” page 8-16.

Format

```
disable trace type
```

Examples

```
zsim{32} disable trace pipe
zsim{32} disable trace reg
```

8.2.11 dump dmem

This command generates a text file representing the contents of the specified address range of the internal or external data memory. The user specifies internal or external memory, the starting address, and the size of the region to dump.

Format

```
dump dmem {int|ext} filename addr size
```

Example

```
zsim{32} dump dmem ext data.dat 0x0000 0x100

% cat data.dat
0000   /* 0x0000 */
0000   /* 0x0001 */
0000   /* 0x0002 */
0000   /* 0x0003 */
0000   /* 0x0004 */
0000   /* 0x0005 */
0000   /* 0x0006 */
...
28e2   /* 0x00fd */
2f6a   /* 0x00fe */
325d   /* 0x00ff */
%
```

8.2.12 dump imem

This command generates a text file representing the contents of the specified address range of the internal or external instruction memory.

The user specifies internal or external memory, the starting address, and the size of the region to dump.

Format

```
dump imem {int|ext} filename addr size
```

Example

```
zsim{32} dump imem int imem.dat 0x1000 0x30

% cat imem.dat
0000   /* 0x1000 */
0000   /* 0x1001 */
0000   /* 0x1002 */
0000   /* 0x1003 */
...
0000   /* 0x102c */
0000   /* 0x102d */
0000   /* 0x102e */
0000   /* 0x102f */
%
```

8.2.13 enable break

This command enables a breakpoint from the current list of defined breakpoints. See Section 8.2.28, “set break,” page 8-23, for a description of how to create a breakpoint.

Format

```
enable break breakpoint_number
```

Example

```
zsim{32} enable break 1
```

8.2.14 enable profile

This command enables a predefined trace type. Run-time traces generate text output representing the state of the architecture on a cycle-by-cycle basis. There are three types of predefined run-time tracing:

- du

  Displays information from the data unit of the ZSP400 architecture, such as data cache hits and the du_imem_read signal.

- `iu`

  Displays information from the instruction unit, such as instruction cache hits and the `iu_imem_read` signal.

- `pipe`

  Displays information from the pipeline unit, such as cycle-by-cycle grouping rule information.

- `lsu`

  Displays information from the load/store unit of ZSPG2 architecture..

Format

```
   enable profile {du|iu|pipe|lsu}
```

Examples

```
   zsim{1} enable profile du
***(info) Data Unit profile information is ON.
   zsim{2} enable profile iu
***(info) Instruction Unit profile information is ON.
   zsim{3} enable profile pipe
***(info) Pipeline Unit profile information is ON.
```

8.2.15 enable trace

This command enables a predefined trace type. Run-time traces generate text output representing the state of the architecture on a cycle-by-cycle basis. There are four types of predefined run-time tracing:

- `mem`

  Displays address and data for any memory location which is updated. Information is generated in the cycle in which the write occurs.

- `pipe`

  Displays the entire pipeline in every cycle.

- `reg`

  Displays all registers and values in every cycle.

- `write`

Displays architectural state changes associated with memory or registers for each cycle.

Format

```
enable trace {mem|pipe|reg|write}
```

Example

```
zsim{32} enable trace write
```

8.2.16 exit

This command terminates the current simulation session.

Format

```
exit
```

Example

```
zsim{32} exit
```

8.2.17 fill dmem

This command fills internal or external data memory range with specified value.

Format

```
fill dmem {int|ext} addr size value
```

Example

```
zsim{32} fill dmem ext 0x1000 0xff 0x0505
```

8.2.18 fill imem

This command fills internal or external instruction memory range with specified value.

Format

```
fill imem {int|ext} addr size value
```

Example

```
zsim{32} fill imem ext 0x1000 0xff 0x0505
```

8.2.19 help

This command displays help information about commands. Commands are categorized according to their function. Requesting help without specifiers displays help on the command categories; requesting help for a specified category displays the instructions associated with that category. Specifying a particular command displays the description and usage for that command.

Format

```
help [category|command]
```

Examples

```
zsim{32} help
zsim{32} help all
zsim{32} help show
zsim{32} help show reg
```

8.2.20 istep

This command steps the program instruction by instruction. By default, this command is aliased to `is`.

For zsimg2, user can specify a number to indicate number of instructions to be executed.

Format

```
istep
```

or

```
is
```

Examples

```
zsim{22}> istep
CYCLE=000012 PC=0x200c
0x2008  mov       rpc, r13
zsim{23}> is
CYCLE=000012 PC=0x200c
0x2009  movl      r5, 0x10
zsim{24}>
CYCLE=000013 PC=0x200c
0x200a  movh      r5, 0x20
```

```
    zsim{25}>
    CYCLE=000013 PC=0x200c
    0x200b  nop
    zsim{26}>
    CYCLE=000015 PC=0x200d
    0x200c  call      r5
    zsim{27}>
    CYCLE=000020 PC=0x2014
    0x2010  mov       r13, rpc
```

8.2.21 load dmem

This command loads internal or external data memory from the specified text file. You must specify internal or external memory, the starting address, and the size of the region to load. You must ensure that the format of the text file is the same as the file produced by the dump command. The first column contains the data that will be loaded, with each data on a single line. Data must be in hex format without the 0x prefix. Comments must be enclosed by '`/* */`'.

Format

```
    load dmem {int|ext} filename addr size
```

Example

```
    zsim{32} load dmem int data.dat 0x1000 20
```

The output format of the file is:

```
%cat data.dat
2ce5   /* 0x0000 */
3c3f   /* 0x0001 */
2000   /* 0x0002 */
3006   /* 0x0003 */
a00f   /* 0x0004 */
80c0   /* 0x0005 */
...
```

8.2.22 load exe

This command loads a valid ZSP executable into instruction memory. This command performs the same function as specifying the executable filename when ZSIM is invoked.

Format

```
load exe filename
```

Example

```
zsim{32} load exe test.exe
```

or

```
zisim{32} load test.exe
```

8.2.23 load imem

This command loads internal or external instruction memory from specified text file. You must specify internal or external memory, the starting address, and the size of the region to load. You must ensure that the format of the text file is the same as the file produced by the dump command. The first column contains the data that will be loaded, with each data on a single line. Data must be in hex format without the 0x prefix. Comments must be enclosed by '`/* */`'.

Format

```
load imem {int|ext} filename addr size
```

Example

```
% cat inst.txt
2ce5   /* 0x0000 */
3c3f   /* 0x0001 */
2000   /* 0x0002 */
3006   /* 0x0003 */
a00f   /* 0x0004 */
80c0   /* 0x0005 */
bc4c   /* 0x0006 */
6f4c   /* 0x0007 */


zsim{32} load imem int imem.txt 0x1000 8
```

8.2.24 reset

This command resets the state of the simulator. The default is a soft reset, which initializes all aspects of the simulator except the instruction memory. A hard reset performs full initialization.

Format

```
reset [soft|hard]
```

Examples

```
zsim{32} reset
zsim{32} reset hard
```

Important: The `reset` command does not reload the program into memory. In order to restart the program, perform one of the following sequence of commands:

```
zsim{32} reset
zsim{32} set reg pc <start_address>
```

or

```
zsim{32} reset hard; load
```

`Note: zsimg2 doesn't support soft reset feature any more.`

8.2.25 run

This command advances the simulator for the specified number of cycles. If no cycle count is specified, the default cycle count defined for the `run` attribute is used (refer to Section 8.2.27, “set attr,” page 8-22). Simulation halts if cycle count is reached, the maximum cycle count is reached, or a system halt occurs.

Format

```
run [number_of_cycles]
```

Examples

```
zsim{32} run
zsim{32} run 100
```

8.2.26 script

This command loads and processes script file. The script file may contain any valid ZSIM commands. Comments are allowed in the script file, preceded by the hash (#) character. ZSIM ignores all commands between the # character and the end of line. Empty lines are also ignored.

Format

```
   script filename
```

Example

```
   zsim{32} script standard.scr
```

Example Script File

```
# This example script demonstrates how to turn on
# instruction and pipeline tracing using a command.
load test.exe
enable trace write
enable trace pipe
run
exit
```

8.2.27 set attr

The `set attr` command allows you to set three internal ZSIM attributes. These configurable attributes are shown in Table 8.7.

Table 8.7 Configurable ZSIM Attributes

Attribute	Value	Description
history	any integer	Number of commands to maintain in history buffer.
radix	{dec | hex}	Radix (decimal or hexadecimal) used to generate output.
run	any integer	Default cycle count for the `run` command (when issuing the `run` command with no argument). If undefined by the `set attr` command, the default `run` value is 100000 cycles.

Format

```
   set attr attribute value
```

Examples

```
   zsim{32} set attr run 1000
   zsim{32} set attr radix hex
```

8.2.28 set break

This command creates and enables a new breakpoint at specified address. Breakpoints can be set for the program counter. Execution halts at the cycle when the instruction at the specified address is in the set of instructions which are about to be executed in the pipeline's E stage.

When a new breakpoint is created, it is tagged with a breakpoint number which is used by other breakpoint commands. Use the `show break` command to display a list of current breakpoints.

Format

```
   set break pc addr
   set break symbol label
```

Example

```
   zsim{1} set break pc 0x0010
Breakpoint 1 on PC at address 0x0010
   zsim{2} set break symbol main
Breakpoint 2 on PC at address 0xf9b9 of main
```

8.2.29 set delay

This command sets the delay wait state of external data memory or instruction memory. The default delay value is 1 for both external data and instruction memory.

The wait state is the number of cycles between requesting data and having it returned. For example, wait state equals 1 means that data is returned 1 cycle after it is requested.

Format

```
   set delay {edata | einst} num
```

Example

```
   zsim{1} set delay edata 10

   zsim{2} set delay einst 20
```

Note: This command is specific to zsim400

8.2.30 set latency

This command sets the delay wait state of internal/external data memory or instruction memory. The default delay value is 2 for both internal data and instruction memory. The default delay value is 5 for both external data and instruction memory.

The wait state is the number of cycles between requesting data and having it returned. For example, wait state equals 2 means that data is returned 2cycles after it is requested.

Format

```
set latency {imem | dmem} {int | ext} num
```

Example

```
zsim{1} set latency dmem int 10
```

```
zsim{2} set latency dmem ext 20
```

Note: This command is specific to zsimg2

8.2.31 set reg

This command assigns a new value to the specified register.

Format

```
set reg register value
```

Example

```
zsim{32} set reg r0 0x1234
```

8.2.32 set size

8.2.32.1 zsim400

This command sets the size of internal data memory or instruction memory. The default size of internal data or instruction memory is 63488 words (62K words), which is also the maximum size that can be set.

Important: This command does not apply to external memory. (The simulator has 1M words for each external instruction and external data memory.)

Format

```
set size {dmem|imem} size
```

Examples

```
zsim{1} set size dmem 0x4000
zsim{2} set size imem 0x3000
```

8.2.32.2 zsimg2

This command sets the size of internal/external instruction or data memory from a begin value to an end value. The boundary is inclusive. The default size for each of the 4 memory types is the maximum value from 0 to 0x00ffffff words (16M words). A word is a 16-bit value for the ZSPG2 architecture.

Format

```
set size {dmem|imem} {int|ext} beg_value end_value
```

Examples

```
zsim{1} set size dmem int 0 0xffff
zsim{2} set size imem int 0 0xffff
zsim{3} set size dmem ext 0 0x00fffff
zsim{4} set size imem ext 0 0x00fffff
```

8.2.33 show attr

This command displays the value of the specified attribute. See `set attr` for a list of defined attributes. Note that the version attribute can only be used with the show `attr` command; it cannot be used with the `set attr` command.

Format

```
show attr {history|radix|run|version}
```

Example

```
zsim{32} show attr run
```

8.2.34 show bits

This command displays the bit field values for the specified register. When specifying control registers, do not include the percent (%) sign.

Format

```
show bits register
```

Example

```
zsim{32} show bits hwflag
hwflag = 0x0000
      er: 0
      ex: 0
      ir: 0
       z: 0
      gt: 0
      ge: 0
       c: 0
     gsv: 0
      sv: 0
      gv: 0
       v: 0
```

8.2.35 show break

This command displays the list of currently defined breakpoints.

Format

```
show break
```

Example

```
zsim{32} show break
```

8.2.36 show dcache

This command displays the current contents of the data cache.

Format

show dcache

Example

For zsim400 simulator

```
   zsim{1}> show dcache
R13 - D$[ 0]: ------ I  ------  ------  ------  ------
R13 - D$[ 1]: ------ I  ------  ------  ------  ------
R13 - D$[ 2]: ------ I  ------  ------  ------  ------
R14 - D$[ 3]: ------ I  ------  ------  ------  ------
R14 - D$[ 4]: ------ I  ------  ------  ------  ------
R14 - D$[ 5]: ------ I  ------  ------  ------  ------
R15 - D$[ 6]: ------ I  ------  ------  ------  ------
R15 - D$[ 7]: ------ I  ------  ------  ------  ------
R15 - D$[ 8]: ------ I  ------  ------  ------  ------
UL  - D$[ 9]: ------ I  ------  ------  ------  ------
UL  - D$[10]: ------ I  ------  ------  ------  ------
UL  - D$[11]: ------ I  ------  ------  ------  ------
UL  - D$[12]: ------ I  ------  ------  ------  ------
UL  - D$[13]: ------ I  ------  ------  ------  ------
UL  - D$[14]: ------ I  ------  ------  ------  ------
UL  - D$[15]: ------ I  ------  ------  ------  ------
UL  - D$[16]: ------ I  ------  ------  ------  ------
```

The first 9 lines are dedicated for linked load of r13, 14, and 15 register respectively. The next 8 lines are used for any unlinked load. The first "------" column is showing the address of the line. The next column indicates the line invalid 'I' or valid. The next 4 columns are showing the data contains in that line.

For zsimg2

```
> zsim{1}> show dcache
D$[ 0]   0x000001  ------ ------ ------ ------ ------ ------
------ ------ lru[0]
D$[ 1]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
D$[ 2]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
D$[ 3]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
D$[ 4]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
D$[ 5]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
D$[ 6]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
D$[ 7]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
D$[ 8]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
```

```
D$[ 9]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
D$[10]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
D$[11]   0x000001  ------ ------ ------ ------ ------ ------
------ ------
The first column shows the address and the next 8 columns
contain data.
```

8.2.37 show dmem

This command displays a range of internal or external data memory. The user specifies internal or external memory, the starting address, and the size of the region to display. The default settings for the show dmem command are shown in Table 8.9.

Table 8.8 Default Arguments for show dmem

Argument	Value
{int | ext}	int
addr	0x0
size	16

Format

```
    show dmem {int|ext} addr size
```

Example

```
    zsim{32} show dmem int 0xf000 0x10
```

```
For zsimg2, user can use a symbol instead of an absolute
address.
    zsim{1} show dmem int array1 20
```

8.2.38 show icache

This command displays the current contents of the instruction cache.

Format

```
    show icache
```

Example

```
zsim{32} show icache
```

8.2.39 show imem

This command displays a range of internal or external instruction memory. The *size* and *addr* fields may be omitted, in which case defaults are used. The default settings for the `show imem` command are shown in Table 8.9.

Table 8.9 Default Arguments for show imem

Argument	Value
{int | ext}	int
addr	0x0
size	16

Format

```
show imem {int|ext} [addr] [size]
```

Example

```
zsim{1} show imem int 0xf000 0x10
```

```
For zsimg2, user can use symbol instead of absolute address
value.
   zsim{1} show imem int foo_function 20
```

8.2.40 show pipe

This command shows the contents of all stages of the pipeline.

Format

```
show pipe
```

Example

```
zsim{32} show pipe
```

8.2.41 show profile

This command shows the current status (enabled/disabled) for each profile type.

Format

```
show profile
```

Example

```
zsim{32} show profile
***(info) Supported profile information:
- Instruction Unit: OFF
- Data Unit:        OFF
- Pipeline Unit:    OFF
```

8.2.42 show reg

This command displays the values of a category of registers or the value of the specified register. You can list more than one category and/or register. The register categories are:

- `gpr`

  All general purpose registers, r0–r15.

- `cfg`

  All control registers (such as %smode and %hwflag). Do not include the percent (%) sign in the control register name.

- `addr`

  All address and index registers for the ZSPG2 architecture. Thus, it is specific for zsimg2.

Format

show reg [*category*|*register*] ...

Examples

```
zsim{32} show reg
zsim{32} show reg r0
zsim{32} show reg hwflag smode (Do not include the percent
                (%) sign.)
```

8.2.43 show rule

This command displays the affected grouping rule for the current cycle.

Format

```
show rule
```

Examples

```
zsim{32} show pipe
CYCLE: 8
------------------------------------------ F(4:2)
(13)000d:5448:0:mac2.a    r4.e, r8.e
(12)000c:788f:0:lddu      r8.e, r15, 2
(11)000b:784e:1:lddu      r4.e, r14, 2
(10)000a:9a00:1:xor.e     r0.e, r0.e
------------------------------------------ G(4:2)
(9)0009:2d18:0:movl      r13, 0x18
(8)0008:3f00:0:movh      r15, 0x0
(7)0007:3d01:1:movh      r13, 0x1
(6)0006:3e00:1:movh      r14, 0x0
------------------------------------------ R(0:0)
------------------------------------------ E(1:1)
(5)0005:ad02:1:bits      fmode, 2
------------------------------------------ W(1:1)
(4)0004:d700:1:movl      guard, 0x0
zsim{33} show rule
Active grouping rule in current cycle: 23. Only two
instructions requiring an alu or one instruction that
requires both the alus can be grouped.
```

8.2.44 show size

Show size of internal data or instruction memory. The output is not affected by the radix attribute.

Format

```
show size {dmem|imem}{int|ext}
```

Examples

```
zsim{32} show size dmem int
The size of internal data memory is 0xf800 words.
zsim{32} show size imem int
The size of internal instruction memory is 0xf800 words.
```

8.2.45 show stats

Display all the run-time statistics generated by ZSIM. If no argument is specified, ZISIM displays overall statistical information. If the opcode argument is specified, ZISIM displays instruction opcode statistics.

Format

```
show stats
```

Example

```
zsim{32} show stats
zsim{32} show stats opcode
```

8.2.46 show trace

Show currently enabled/disabled trace information. Traces currently set to `ON` are enabled during simulation.

Format

```
show trace
```

Example

```
zsim{32} show trace
***(info) Supported trace information:
- Instruction trace: OFF
- Pipeline trace:    OFF
- Register trace:    OFF
- Memory trace:      OFF
zsim{33} enable trace pipe
***(info) Pipeline trace is ON.
zsim{34} show trace
***(info) Supported trace information:
- Instruction trace: OFF
- Pipeline trace:    ON
- Register trace:    OFF
- Memory trace:      OFF
```

8.2.47 step

Single-step the simulator. Issuing the `step` command is equivalent to issuing the command `run 1`.

Format

```
step
```

Example

```
zsim{32} step
```

8.2.48 unalias

Deletes an alias.

Format

```
unalias [alias]
```

Example

```
zsim{32} unalias adv
```

8.3 I/O Port Usage

ZSIM400 models serial I/O as a memory-mapped device. Programs perform terminal I/O by reading from and writing to the appropriate address locations. The simulator defines two serial ports and one host processor interface (HPI) port. Each port has a transmit buffer and a receive buffer. Table 8.10 shows the memory addresses and corresponding files for the I/O ports for the LSI402ZX, LSI403Z, and ZSP400-core based devices.

Table 8.10 I/O Device Memory Map and Associated Files

	Read		Write	
I/O Port	**Address**	**File**	**Address**	**File**
Serial Port 0	0xF901	`sp0in`	0xF900	`sp0out`
Serial Port 1	0xFA01	`sp1in`	0xFA00	`sp1out`
Host Interface Port	0xFB01	`hpiin`	0xFB00	`hpiout`

The format of input and output files are the same. Data must be in decimal digits, with each data on a single line. If the input file is not present in the current running directory at the time of the request, the simulator will print an error message to standard output and exit.

ZSIM400 also supports user-specified I/O ports. You can create a library containing peripheral devices and then use it in place of the default library in the directory `$SDSP_HOME/sdspI/bin`, which is created when the ZSP SDK tools are installed. The peripheral library is called `libzperiph.dll` on Windows and `libzperiph.so` on Solaris platforms. For information on writing the peripheral library, refer to the *ZSIM Peripheral API Reference Guide,* document DB06-000299-00.

8.4 Example Session Using ZSIM

This section contains an example simulation session using ZSIM in interactive mode.

```
zsim{1}> load exe test.exe
***(info) Starting address: 0x2000
.text   : Loading to INT-INST memory ... 0x2000 -> 0x2950 (0x0951)
.data   : Loading to INT-DATA memory ... 0x0001 -> 0x005f (0x005f)
Loading "test.exe" successfully.
```

The contents of the instruction memory can be checked to confirm proper loading of the test:

```
zsim{2}> show imem int 0x2000 4
0x2000   0x2cfb  movl      r12, 0xfb
0x2001   0x3cf7  movh      r12, 0xf7
0x2002   0xa6d0  mov       r13, 0x0
0x2003   0x2460  movl      r4, 0x60
zsim{3}> _
```

Before execution cycles begin, you can check to make sure that the pipeline and caches are empty:

```
zsim{3}> show pipe
------------------------------------------ F(0:0)
------------------------------------------ G(0:0)
------------------------------------------ R(0:0)
------------------------------------------ E(0:0)
------------------------------------------ W(0:0)
```

As shown above, the five stages of the execution pipeline are identified with a single letter – F (Fetch/decode), G (Group), R (Read), E (Execute), and W (Write Back) – followed by two integers representing the number of instructions currently in that stage and the number of instructions that will advance to the next stage in the following cycle.

```
zsim{4}> show icache
I$[0]: ------ I ------ I ------ I ------ I ------
I$[1]: ------ I ------ I ------ I ------ I ------
I$[2]: ------ I ------ I ------ I ------ I ------
I$[3]: ------ I ------ I ------ I ------ I ------
I$[4]: ------ I ------ I ------ I ------ I ------
I$[5]: ------ I ------ I ------ I ------ I ------
I$[6]: ------ I ------ I ------ I ------ I ------
I$[7]: ------ I ------ I ------ I ------ I ------
```

In the above example, the 8 lines of the instruction cache are shown to be empty . The first column contains the address (4 word boundary) and the remaining 4 columns contain the corresponding instruction opcodes. An ‘I’ to the left of a cell indicates an invalid instruction.

```
zsim{5}> show dcache
R13 - D$[ 0]: ------ I  ------  ------  ------  ------
R13 - D$[ 1]: ------ I  ------  ------  ------  ------
R13 - D$[ 2]: ------ I  ------  ------  ------  ------
R14 - D$[ 3]: ------ I  ------  ------  ------  ------
R14 - D$[ 4]: ------ I  ------  ------  ------  ------
R14 - D$[ 5]: ------ I  ------  ------  ------  ------
R15 - D$[ 6]: ------ I  ------  ------  ------  ------
R15 - D$[ 7]: ------ I  ------  ------  ------  ------
R15 - D$[ 8]: ------ I  ------  ------  ------  ------
UL  - D$[ 9]: ------ I  ------  ------  ------  ------
UL  - D$[10]: ------ I  ------  ------  ------  ------
UL  - D$[11]: ------ I  ------  ------  ------  ------
UL  - D$[12]: ------ I  ------  ------  ------  ------
UL  - D$[13]: ------ I  ------  ------  ------  ------
UL  - D$[14]: ------ I  ------  ------  ------  ------
UL  - D$[15]: ------ I  ------  ------  ------  ------
UL  - D$[16]: ------ I  ------  ------  ------  ------
```

The 17 lines of the data cache are shown to be empty in the above example. The first column contains the address (4-word boundary) and the remaining 4 columns contain data values. An ‘I’ to the left of a data line indicates that the corresponding data line is invalid.

Continuing with the example, as execution proceeds, the pipeline and instruction cache reflect changes expected by instruction flow:

```
zsim{6}> run 4 ; show pipe
CYCLE=000004 PC=0x2000
CYCLE: 4
```

```
   ---------------------------------------- F(4:1)
   (7)2007:6054:0:st        r5, r4, 0
   (6)2006:a051:0:add       r5, 0x1
   (5)2005:bc54:0:mov       r5, r4
   (4)2004:3400:1:movh      r4, 0x0
   ---------------------------------------- G(4:1)
   (3)2003:2460:0:movl      r4, 0x60
   (2)2002:a6d0:0:mov       r13, 0x0
   (1)2001:3cf7:0:movh      r12, 0xf7
   (0)2000:2cfb:1:movl      r12, 0xfb
   ---------------------------------------- R(0:0)
   ---------------------------------------- E(0:0)
   ---------------------------------------- W(0:0)
zsim{7}> show icache
I$[0]: 0x2000 V 0x2cfb V 0x3cf7 V 0xa6d0 V 0x2460
I$[1]: 0x2004 V 0x3400 V 0xbc54 V 0xa051 V 0x6054
I$[2]: ------ I ------ I ------ I ------ I ------
I$[3]: ------ I ------ I ------ I ------ I ------
I$[4]: ------ I ------ I ------ I ------ I ------
I$[5]: ------ I ------ I ------ I ------ I ------
I$[6]: ------ I ------ I ------ I ------ I ------
I$[7]: ------ I ------ I ------ I ------ I ------
zsim{8}> _
```

The simulator output below demonstrates use of the PC breakpoint. A breakpoint is set for address 0x10 and the simulator is advanced. Execution halts when the instruction associated with the breakpoint address reaches the Group stage. The state of the pipeline and operand registers are shown after the breakpoint halt occurs.

```
zsim{8}> set break sym main
Breakpoint 1 on PC at address 0x2010 of main
zsim{9}> enable trace write
***(info) Instruction trace is ON.
zsim{10}> run
<6> (0) 0x2000 2cfb movl      r12, 0xfb       !              r12 = 0x00fb
<7> (1) 0x2001 3cf7 movh      r12, 0xf7       !              r12 = 0xf7fb
<7> (2) 0x2002 a6d0 mov       r13, 0x0        !              r13 = 0x0000
<8> (3) 0x2003 2460 movl      r4, 0x60        !               r4 = 0x0060
<9> (4) 0x2004 3400 movh      r4, 0x0         !               r4 = 0x0060
<10> (5) 0x2005 bc54 mov       r5, r4          !               r5 = 0x0060
<11> (6) 0x2006 a051 add       r5, 0x1         !           hwflag = 0x0030
<11> (6) 0x2006 a051 add       r5, 0x1         !               r5 = 0x0061
<11> (7) 0x2007 6054 st        r5, r4, 0       ! INT-DATA[0x0060] = 0x0061
<12> (9) 0x2009 2510 movl      r5, 0x10        !               r5 = 0x0010
<13> (8) 0x2008 bb1d mov       rpc, r13        !              rpc = 0x0000
<13> (10) 0x200a 3520 movh      r5, 0x20        !               r5 = 0x2010
<14> (12) 0x200c a750 call      r5              !              rpc = 0x200d
(PC BREAKPOINT #1)...................... CYCLE=000020 PC=0x2014
```

Trace information is displayed in six fields:

- The first field is the cycle count number (enclosed by '< >').
- The second field is the instruction sequence number (in parenthesis).
- The third field is the program counter (PC) of the executed instruction.
- The fourth field is the instruction opcode.
- The fifth field is the disassembled instruction, including operands.
- The sixth field describes the result of the executed instruction.

```
zsim{11}> run 7; show pipe
<20> (13) 0x2010 2501 movl      r5, 0x1         !              r5 = 0x2001
<20> (14) 0x2011 b91d mov       r13, rpc        !             r13 = 0x200d
<21> (15) 0x2012 3500 movh      r5, 0x0         !              r5 = 0x0001
<21> (16) 0x2013 6fdc stu       r13, r12, -1    ! INT-DATA[0xf7fb] = 0x200d
<21> (16) 0x2013 6fdc stu       r13, r12, -1    !             r12 = 0xf7fa
<22> (17) 0x2014 a0cf add       r12, 0xffff     !          hwflag = 0x0040
<22> (17) 0x2014 a0cf add       r12, 0xffff     !             r12 = 0xf7f9
<22> (19) 0x2016 1060 call      0x20d6          !             rpc = 0x2017
<23> (18) 0x2015 615c st        r5, r12, 1      ! INT-DATA[0xf7fa] = 0x0001
<25> (20) 0x20d6 a641 mov       r4, 0x1         !              r4 = 0x0001
<26> (21) 0x20d7 b91d mov       r13, rpc        !             r13 = 0x2017
<26> (22) 0x20d8 6fdc stu       r13, r12, -1    ! INT-DATA[0xf7f9] = 0x2017
<26> (22) 0x20d8 6fdc stu       r13, r12, -1    !             r12 = 0xf7f8
<27> (23) 0x20d9 bc6c mov       r6, r12         !              r6 = 0xf7f8
CYCLE=000027 PC=0x20dc
CYCLE: 27
   ---------------------------------------- F(4:3)
   (33)20ea:bc34:0:mov       r3, r4
   (32)20e9:6f7c:1:stu       r7, r12, -1
   (31)20e8:b910:1:mov       r0, rpc
   (30)20e7:6b2c:1:stdu      r2.e, r12, -2
   ---------------------------------------- G(4:3)
   (29)20e6:3d00:0:movh      r13, 0x0
   (28)20e5:6b0c:1:stdu      r0.e, r12, -2
   (27)20e4:2d68:1:movl      r13, 0x68
   (26)20dc:1004:1:call      0x20e4
   ---------------------------------------- R(1:1)
   (25)20db:a063:1:add       r6, 0x3
   ---------------------------------------- E(2:2)
   (24)20da:725c:1:ld        r5, r12, 2
   (23)20d9:bc6c:1:mov       r6, r12
   ---------------------------------------- W(2:2)
   (22)20d8:6fdc:1:stu       r13, r12, -1
   (21)20d7:b91d:1:mov       r13, rpc
zsim{12}> show reg gpr
              r0 = 0x0000              r1 = 0x0000
              r2 = 0x0000              r3 = 0x0000
              r4 = 0x0001              r5 = 0x0001
              r6 = 0xf7f8             r7 = 0x0000
              r8 = 0x0000             r9 = 0x0000
             r10 = 0x0000            r11 = 0x0000
             r12 = 0xf7f8            r13 = 0x2017
             r14 = 0x0000            r15 = 0x0000
zsim{14}>
```

Execution halts when a breakpoint is reached, the maximum cycle count is reached, or a system halt occurs. A system halt refers to the halt mode as defined by the power level (lvl) field in the DSP's %smode control register.

A simulation session is terminated with the exit command.

```
zsim{12}> exit
***(info) Exiting ZSIM.
%_
```

Chapter 9
Debugger

This chapter describes the SDK source and assembly-level Debugger for the ZSP400 and ZSPG2 architectures.

The SDK Debugger, SDBUG, is based on the GNU Debugger (GDB) from the Free Software Foundation. GDB is described in *Debugging with GDB: The GNU Source Level Debugger*, by Richard Stallman, *et*. *al*., Free Software Foundation, January 1994. The description of SDBUG in this chapter, for the most part, includes only the differences from GDB.

For Windows 95/98/NT platforms, the debugger can be accessed using the ZSP Integrated Development Environment, as described in Chapter 10, "ZSP Integrated Development Environment (ZSP IDE)." This chapter describes the debugger's standard GNU command-line interface, available for all platforms.

9.1 Using SDBUG

SDBUG is invoked from the command line as follows:

```
<debugger name> [options] [executable_file]
```

where debugger name is the name of the desired debugger as listed in Table 9.1.

Table 9.1 Debugger Names

Debugger Name	Use when debugging...
sdbug400	code written for devices based on the ZSP400 architecture.
zdxbug	code originally written for devices based on the ZSP400 architecture, but cross-compiled for the ZSPG2 architecture.
zdbug	code designed for devices based on the ZSPG2 architecture.

The above command both invokes and initializes the debugger.

SDBUG-only command-line options are listed in Table 9.2. All other SDBUG options are described in Stallman, *et. al.*

Table 9.2 SDBUG-Only Options

Option	Description	Availability
`-mempcr=ADDR`	Sets the address of the mempcr register.	sdbug400
`-no_mempcr`	Specifies that the hardware target has no MEMPCR register	sdbug400
`-jtag_type=TYPE`	Gives priority to the detection of the JTAG interface specified. TYPE can be either pci (Corelis PCI JTAG), pcmcia (Corelis PCMCIA JTAG), or raven (Macraigor Raven) By default, SDBUG first attempts to use the PCMCIA JTAG card, then the PCI JTAG card, then the Macraigor Raven interface..	sdbug400
`-jtag_mapfile=FILE`	Makes the debugger look for the map file FILE, rather than the default called “mapfile” in the current directory and SDSP_HOME/sdspI/misc.	sdbug400

Use the following command to load the symbol table from the executable file:

```
(sdbug) file a.out
```

Next select SDBUG’s target execution environment (as described in the following section). For example, to target the cycle-accurate simulator:

```
(sdbug) target zsim
```

Use the following command to load the text and data sections of the executable file:

```
(sdbug) load a.out
```

Now you are ready to debug your program using the standard GDB commands.

9.2 SDBUG Execution Environments

The debugger supports four execution environments:

- Functional-accurate software simulation on the host (using ZISIM)
- Cycle-accurate software simulation on the host (using ZSIM)
- Target hardware, connected through the serial port
- Target hardware, connected through a JTAG controller (Windows 95/98/NT platforms only)

These environments are described in the following subsections.

9.2.1 Functional-Accurate Simulator Connection

The ZISIM target simulator is invoked by the following command:

```
(sdbug) target sim [option...]
```

where option is any of the simulator options described in Table 7.1 on page 7-1.

With this connection, program execution is performed by the functional-accurate simulator, ZISIM, under the control of the debugger. The debugger examines the simulator state to process queries from the user.

SDBUG uses the functional-accurate simulator commands to select information that is requested from the executing program by the ZISIM simulator. These commands are listed in Table 9.3 and described in detail in Section 7.2, “ZISIM Commands,” page 7-4.

The format for simulator commands using ZISIM is:

```
(sdbug) sim simulator-command
```

Table 9.3 SDBUG Target ZISIM Simulator Commands

Command	Description
`clear-stats`	Resets the statistics.
`close` *`filename`*	Closes file *`filename`*.[1]
`help`	Displays the list of simulator commands that can be invoked.
`max_number_of_files` *`number`*	Sets the maximum number of files that can be opened at the same time to *`number`*.[1]
`memory_download` *`filename addr size`*	Writes *`size`* of items to memory *`addr`* from file *`filename`*.[1]
`memory_upload` *`filename addr size`*	Reads *`size`* of items from memory *`addr`* to file *`filename`*.[1]
`print-stats`	Prints statistics such as instruction mix, load, store, discontinue, and mispredicts to `stdout`.
`reg-off`	Sets the simulator register tracing off.
`reg-on`	Sets the simulator register tracing on.
`trace-off`	Sets the simulator trace off.
`trace-on`	Sets the simulator trace on.
`clear-opcode`	Resets statistics of opcode usage.
`print-opcode`	Prints statistics of opcode usage.

1. This command may also be invoked without specifying the target name. See Section 9.3.1, “Generic Target-Specific Commands” on page 9-11 for details.

9.2.2 Cycle-Accurate Simulator Connection

The ZSIM target simulator is invoked by the following command:

```
(sdbug) target zsim
```

With this connection, the cycle-accurate simulator (ZSIM) executes your program under the control of the debugger. The debugger examines the simulator state to process queries from the user.

The cycle-accurate simulator commands are used to select information that is requested from the executing program by the ZSIM simulator. These commands are listed in Table 9.4 and described in detail in Section 8.2, "ZSIM Commands," page 8-5.

The format for ZSIM commands is:

```
(sdbug) zsim simulator-command
```

Table 9.4 SDBUG Target ZSIM Commands

Command	Description
`clear-stats`	Resets the general statistics.
`clear-opcode`	Resets the opcode usage statistics.
`close` *`filename`*	Closes file *`filename`*.[1]
`help`	Displays the list of simulator commands that can be invoked.
`max_number_of_files` *`number`*	Sets the maximum number of files that can be opened at the same time to *number*.[1]
`memory_download` *`filename addr size`*	Writes *size* of items to memory *`addr`* from file *`filename`*.[1]
`memory_upload` *`filename addr size`*	Reads *size* of items from memory *`addr`* to file *`filename`*.[1]
`pfdu-off`	Turns off data unit profile information.
`pfdu-on`	Turns on data unit profile information.
`pfiu-off`	Turns off instruction unit profile information.
`pfiu-on`	Turns on instruction unit profile information.
`pfpipe-off`	Turns off pipeline unit profile information.
`pfpipe-on`	Turns on pipeline unit profile information.
`pipe-off`	Sets the simulator pipeline off.
(Sheet 1 of 2)	

Table 9.4 SDBUG Target ZSIM Commands (Cont.)

Command	Description
`pipe-on`	Sets the simulator pipeline on.
`print-dcache`	Prints contents of data cache to `stdout`.
`print-icache`	Prints contents of instruction cache to `stdout`.
`print-opcode`	Prints instruction opcode history to `stdout`.
`print-pipe`	Prints contents of the pipeline to `stdout`.
`print-profile`	Prints collected profile information to `stdout`.
`print-rule [# | all]`	Prints grouping rule to `stdout`[2].
`print-stats`	When cycle count is on, prints statistics to `stdout`.
`print-stats-inc`	Prints incremental statistics information to `stdout`.
`pf` *`functionName start end`*	Collects profile information for *`functionName`* from *`start`* to *`end`* addresses. Follow by `profile-on` command to turn on the profile collector.
`profile-func`	Collects profile information for all functions in the program. Follow by the `profile-on` command to turn on the profile collector.
`profile-off`	Turns off profile collector.
`profile-on`	Turns on profile collector
`reg-off`	Sets the simulator register tracing off.
`reg-on`	Sets the simulator register tracing on.
`trace-off`	Sets the simulator trace off.
`trace-on`	Sets the simulator trace on.
(Sheet 2 of 2)	

1. This command may also be invoked without the target name. See Section 9.3.1, “Generic Target-Specific Commands” on page 9-11 for details.
2. The optional arguments only work in sdbug400. zdbug and zdxbug only supports the display of the grouping rules that are currently active.

9.2.2.1 User-Specified Profiling

When used with the cycle-accurate simulator, the debugger supports profiling of selected areas of your project code. To use this feature, you must define the regions to be profiled using the following pair of assembler directives in your source code:

asm("\n__FUNC_START_*region_name*:");

<*code to be profiled*>

asm("\n__FUNC_EXIT_*region_name*:");

The profiling can then be enabled using the following commands:

```
(sdbug) profile-func
(sdbug) profile-on
```

Execute the program by typing:

```
(sdbug) run
```

Display the profiling statistics using:

```
(sdbug) print_profile
```

With respect to profiling, the `profile-func` command will treat *region_name* just like a function. Note that for function profiling to operate correctly, execution that passes through the `start` label must also pass through the `exit` label.

9.2.3 UART Connection

The UART connection is invoked by the following commands:

```
(sdbug) set remotebaud [baud_rate]
(sdbug) target sdsp-remote serial_port
```

The required baud rate can be specified when setting `remotebaud`. The default baud rate setting is 38400.

To use this connection, your target evaluation board must be able to support UART-based debugging with appropriate hardware and firmware. In addition, your target must be booted from flash memory that contains the UART debug code. For instructions on programming the flash memory, refer to the application note, *Programming the Flash.* To ensure that your EB402 Evaluation Board is booted from (external) flash memory, set the IBOOT pin LOW. Refer also to the *EB402 Evaluation Board User's Guide*.

Use the SDBUG commands in Table 9.5 to communicate with the target board though the serial port connection.

The format for serial port commands is:

```
(sdbug) sdsp-remote sdsp-remote-command
```

Table 9.5 SDBUG UART Connection Commands

Command	**Description**
`close file` *`filename`*	Close file *`filename`*.[1]
`help`	List UART connection commands.
`max_number_of_files` *`number`*	Specify the maximum number of files that can be opened at the same time.[1]
`memory_download` *`filename addr size`*	Write *`size`* of items to memory *`addr`* from file *`filename`*. *`addr`* can be a label.[1]
`memory_upload` *`filename addr size`*	Read *`size`* of items from memory *`addr`* to file *`filename`*. *`addr`* can be a label.[1]

1. This command may also be invoked without the target name. See Section 9.3.1, "Generic Target-Specific Commands" on page 9-11 for details.

9.2.4 JTAG Controller Connection

To use the JTAG connection, you must install a Corelis PCI or PCMCIA Type II Boundary Scan Controller card on your Windows machine and install a cable connecting it to your evaluation board.

Note: The JTAG target is available only for Windows 95/98/NT platforms.

The JTAG target is invoked by the following commands:

```
(sdbug) jtag set_clk 2 0 0

(sdbug) target jtag
```

The first command is required to set the parameters for the JTAG clock (TCK) on the Corelis Boundary Scan Controller card, where the first parameter (2) specifies the base clock oscillator to be used (50 MHz), the second parameter (`0`) disables the clock prescaler, and the third parameter (`0`) is used as the clock divisor (divide by 2). (These are the default settings for boards running at 100 MHz and above.) The second command establishes the connection.

Refer to the *Corelis Software Development Kit User's Manual* for information on supported JTAG clock speeds.

The JTAG commands described in Table 9.6 are used to select information that is requested from the target using the JTAG connection.

The format for JTAG commands is:

```
(sdbug) jtag jtag-command
```

Table 9.6 SDBUG JTAG Commands

Command	**Description**
`close` *`filename`*	Close file *`filename`*.[1]
`help`	List JTAG commands.
`set_clk` *`val1 val2 val3`*	Sets the JTAG clock according to the JTAG interface in question. With the Corelis JTAG interfaces, the values are base clock oscillator, prescaler enable, and clock divisor, respectively. For Macraigor Raven, it would be the speed value followed by two zeros. Generally speaking, fthe JTAG clock speed should be approximately 1/10th to 1/20th of the ZSP clock speed.
`raven_lpt` *`port`*	Tells the debugger to use LPT 1, 2, or 3 as the Raven LPT port.
`max_number_of_files` *`number`*	Specify the maximum number of files that can be opened at the same time.[1]
`memory_download` *`filename addr size`*	Write *`size`* of items to memory *`addr`* from file *`filename`*. *`addr`* can be a label.[1]
`memory_upload` *`filename addr size`*	Read *`size`* of items from memory *`addr`* to file *`filename`*. *`addr`* can be a label.[1]

1. This command may also be invoked without the target name. See Section 9.3.1, "Generic Target-Specific Commands" on page 9-11 for details.

9.2.4.1 Hardware-Assisted Debugging

The JTAG target environment supports hardware-assisted debugging. The format for a hardware-assisted debugging command is:

```
(sdbug) hw hardware_assisted_debugging_command
```

Important: **All breakpoints must be disabled before using hardware-assisted debugging**. Only one breakpoint may be set, and when it is set, any previously-set breakpoint is deactivated. You cannot perform I/O during hardware-assisted debugging.

Important: **Hardware-assisted debugging will function correctly only with the correct map file for the specific part being debugged.** The SDK comes with the map file for LSI402ZX rev. 1 (mapfile), LSI402ZX rev. 2 (mapfile_rev2), and LSI403LP (mapfile_403lp); if your application uses a different processor, please contact the vendor for the correct map file. The default map file loaded is mapfile. To change the map file used, either copy the new map file to the directory the debugger is inovked in as “mapfile,” or copy to the current directory or $SDSP_HOME/sdspl/misc and use the --jtag_mapfile command line option to specify the map file to use.

The commands available for hardware-assisted debugging are shown in Table 9.7.

Table 9.7 Hardware-Assisted Debugging Commands

Command	**Description**
`enable_ice`	Enable hardware-assisted debugging.
`resume`	Resume execution.
`step` *`n`*	Step *n* cycles.
`insn_addr_brk` *`addr`*	Set a breakpoint when executing an instruction at *`addr`*.
`st_addr_brk` *`addr`*	Set a breakpoint when storing to *`addr`*.
`st_data_brk` *`data`*	Set a breakpoint when storing the value *`data`*.

Table 9.7 Hardware-Assisted Debugging Commands (Cont.)

Command	Description
`st_addr_and_data_brk` *`addr data`*	Set a breakpoint when storing *`data`* to *`addr`*.
`st_addr_or_data_brk` *`addr data`*	Set a breakpoint when storing to *`addr`* or storing the value *`data`*.
`disable_brk`	Disable hardware breakpoint.
`return_to_sw_dbg`	Returns to software debug mode. Must have executed in hardware debug mode for at least one cycle in order for this to work.

9.3 Debugger Commands – Special Cases

Some SDBUG commands have special cases, which are described in the following subsections. For more information on the usage of any command, issue the `help` command at the (sdbug) prompt.

9.3.1 Generic Target-Specific Commands

To make test scripts that need to run under multiple targets more generic, the hardware and software target-specific commands `memory_upload`, `memory_download`, `close`, and `max_number_of_files` may now be used without their target prefixes after the `target` has been specified.

For example, the command:

```
(sdbug) jtag max_number_of_files 1
```

may be replaced by

```
(sdbug) max_number_of_files 1
```

within a script after you have issued the `target` command.

9.3.2 Backtrace Command

To use the backtrace command, you must adhere to the calling conventions described in Section 3.2, “Compiler Conventions.” To use this command to display the call stack, set breakpoints on the function name. This command may display incorrect results when the debugger is halted inside a function prologue or epilogue.

9.3.3 Info Registers Command

9.3.3.1 sdbug400, zdxbug

To use this command, the %rpc register must be stored on the stack, even for leaf functions. Otherwise, the compiler returns incorrect values for the %pc and %rpc registers when traversing the stack. Refer to Section 3.2, "Compiler Conventions."

9.3.3.2 zdbug

The code still needs to following the compiler convention, though the convention has now been changed. Refer to Section 3.2, "Compiler Conventions." for details.

9.3.4 Breakpoint Command

SDBUG reserves the use of pc value zero. If two breakpoints are inadvertently set at pc value zero, the debugger will loop while trying to execute the instruction. If a breakpoint has to be set at pc value zero, set only one breakpoint at that address.

9.3.5 Print Command

The `print` command is typically used to display the values of variables and arrays. It may also be used to display the values in any memory location.

9.3.6 Set Command

The `set` command is used to change the state of the processor or the debugger. It can be used to change any register value, the value of any word in any memory, or the value of any variable.

Keep in mind that with the cycle-accurate simulator (ZSIM), the `set` command may not operate correctly if it is used to change the contents of a register that will be used by an instruction currently in the pipeline—if the instruction is in a pipeline stage older than Group (G), the instruction may read the old value. Also, using the ZSIM `set` to modify a memory location that has already been loaded into the data cache will modify both the data cache and the memory. (With the UART and JTAG targets, modifying memory will not affect the data cache.)

9.3.7 Cycle-Step Command

The `cycle-step` command is only available for use with the cycle-accurate simulator (ZSIM). This command causes the simulator to advance the pipeline cycle-by-cycle.

Format:

```
    cycle-step #
```

Example:

```
    (sdbug) cycle-step 10
```

The simulator will be advanced by 10 clock cycles.

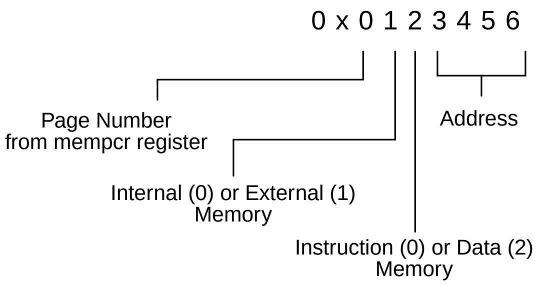
9.3.8 Accessing Memory with the Debugger

9.3.8.1 sdbug400, zdxbug

Debugger commands use memory addresses that are seven hexadecimal digits in length.

The address format is shown in Figure 9.1. The seventh (leftmost and most-significant) digit is the page number (0x0–0xF) from the mempcr register, the sixth digit selects between internal (0) or external (1) memory, the fifth digit selects instruction (0) or data (2) memory, and the first four (rightmost and least-significant) digits are the normal 16-bit address. If any of the three most-significant digits are omitted from an address, they are assumed to be zero.

Figure 9.1 Debugger Memory Addressing (sdbug400, zdxbug)


0 x 0 1 2 3 4 5 6
Page Number from mempcr register
Address
Internal (0) or External (1) Memory
Instruction (0) or Data (2) Memory

<u>Note:</u> All other ZSP SDK tools and linker scripts use four-digit addressing. The debugger is the only tool that uses seven-digit memory addressing.

Some examples of debugger memory addressing are shown below:

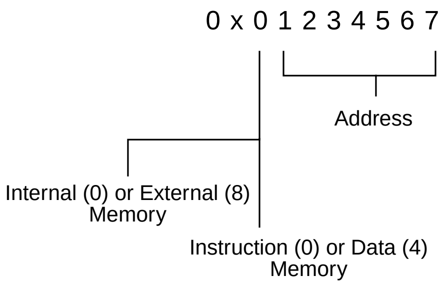
0x0001000	Internal instruction at address 0x1000
0x0022000	Internal data at address 0x2000
0x0103000	Page 0, external instruction memory at address 0x3000
0x2124000	Page 2, external data memory at address 0x4000
0xa105000	Page 10, external instruction memory at address 0x5000

9.3.8.2 zdbug

Debugger commands use memory addresses that are eight hexidecimal digits in length.

The address format is shown in Figure 9.2. The eighth (leftmost and most-significant) digit's fourth bit (0x80000000) selects between internal (0) or external (1) memory, the eighth digit's third bit (0x40000000) selects instruction (0) or data (1) memory. The other seven digits are used to determine the address. If any of the leftmost digits are ommitted from an address, they are assumed to be zero.

Figure 9.2 Debugger Memory Addressing (zdbug)


0 x 0 1 2 3 4 5 6 7
Address
Internal (0) or External (8) Memory
Instruction (0) or Data (4) Memory

Note: All other ZSP SDK tools and linker scripts use 24-bit addressing. The debugger is the only tool that uses 30-bit addressing.

Some examples of debugger memory addressing are shown below:

0x00001000	Internal instruction at address 0x1000
0x40002000	Internal data at address 0x2000
0x80003000	External instruction memory at address 0x3000
0xC0004000	External data memory at address 0x4000
0x30000000	Internal Instruction at address 0x300000000
0xF0000100	External data memory at address 0x30000100

9.4 Dynamic Breakpoints

Command-line debugging supports dynamic breakpoints for all target execution environments while in software debug mode. Dynamic breakpoints are set by pressing `cntl-C`.

9.5 Example Debugging Sessions

This section contains two examples demonstrating the use of SDBUG. The first example uses the functional-accurate simulator, ZISIM. The second example uses the JTAG controller connection for hardware-assisted debugging.

9.5.1 Example 1

In this sample debugging session, the executable is built from the C and assembly programs shown in Appendix A, "Example Programs" The name of the executable is demo.exe, and the start address is 0x1000. The target is set to the functional-accurate simulator (ZISIM) for the LSI402Z. The complete command name is used the first time the command is invoked (for example, backtrace); subsequent invocations use the abbreviated command name (bt).

```
(shell) sdbug400
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=sparc-sun-solaris2.6 --target=sdsp-zsp-elf"...
(sdbug) file demo.exe
Reading symbols from demo.exe...done.
(sdbug) target sim
Connected to the simulator.
(sdbug) load demo.exe
.text   : 0x   0 .. 0x  cd ... Loading
.data   : 0x  cd .. 0x  cf ... Loading
Transfer rate: 3312 bits in <1 sec.
(sdbug) breakpoint main
Breakpoint 1 at 0x13: file demo.c, line 9.
(sdbug) b func_1
Breakpoint 2 at 0x56: file func1.s, line 9.
(sdbug) b func_2
Breakpoint 3 at 0x89: file func2.c, line 4.
(sdbug) b func_3
Breakpoint 4 at 0x70: file func1.s, line 50.
(sdbug) run
Starting program: /user/Tools/MyProject02/demo.exe

Breakpoint 1, main () at demo.c:9
9          char ch = ’A’;
(sdbug) list
4
5       int t=500;
6
7       main()
8       {
9          char ch = ’A’;
10         int i,j = 100,k;
11
12         for (i=0; i< 2; i++) {
13           func_2();
```

```
(sdbug) step
10         int i,j = 100,k;
(sdbug) print j
$1 = 0
(sdbug) p i
$2 = 0
(sdbug) continue
Continuing.

Breakpoint 3, func_2 () at func2.c:4
4        int x=0,n=0;
(sdbug) next
5        while(n < 20)
(sdbug) n 5
25          t1 = x;
(sdbug) backtrace
#0  func_2 () at func2.c:25
#1  0x21 in main () at demo.c:13
(sdbug) up
#1  0x21 in main () at demo.c:13
13          func_2();
(sdbug) down
#0  func_2 () at func2.c:25
25          t1 = x;
(sdbug) info reg r2 r3 r12 rpc pc
r2             0x0     0
r3             0x0     0
r12            0xf7f3  -2061
rpc            0x21    33
pc             0xc0    192
(sdbug) c
Continuing.

Breakpoint 2, func_1 () at func1.s:14
14             mov     r5, r4
Current language:  auto; currently asm
(sdbug) l
9              mov     r13, %rpc
10             stu     r13, r12, -1
11
12             /** END PROLOGUE **/
13
14             mov     r5, r4
15             ld      r4, r5
16             mov     r6, 500
17             cmp     r4, r6          /*  *t <= 500;  */
18             bgt     L2
(sdbug) s 6
20             mov     r6, 100
(sdbug) info breakpoints
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x00000013 in main at demo.c:9
        breakpoint already hit 1 time
```

```
2   breakpoint     keep y   0x00000056 func1.s:9
        breakpoint already hit 1 time
3   breakpoint     keep y   0x00000089 in func_2 at func2.c:4
        breakpoint already hit 1 time
4   breakpoint     keep y   0x00000070 func1.s:50
(sdbug) **delete 4**
(sdbug) **b demo.c:23**
Breakpoint 5 at 0x3b: file demo.c, line 23.
(sdbug) **c**
Continuing.

Breakpoint 3, func_2 () at func2.c:4
4         int x=0,n=0;
(sdbug) **n 3**
9              x += 5;
(sdbug) **bt**
#0  func_2 () at func2.c:9
#1  0x21 in main () at demo.c:13
(sdbug) **c**
Continuing.

Breakpoint 2, func_1 () at func1.s:14
14              mov     r5, r4
(sdbug) **disable 2 3**
(sdbug) **c**
Continuing.

Breakpoint 5, main () at demo.c:23
23         while (i < 20) {
(sdbug) **p i**
$3 = 2
(sdbug) **p j**
$4 = 100
(sdbug) **c**
Continuing.

Breakpoint 5, main () at demo.c:23
23         while (i < 20) {
(sdbug) **d 5**
(sdbug) **c**
Continuing.
(SYSTEM HALT)................................................ PC=0x000e
Total Instructions: 1384

Program exited normally.
(sdbug) **exit**
```

9.5.2 Example 2

This example illustrates the use of hardware-assisted debugging with the JTAG connection. The example program hw_dbg.s is shown in Appendix A, "Example Programs"

```
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin32 --target=sdsp-zsp-elf".
(sdbug) file a.out
Reading symbols from a.out...done.
(sdbug) jtag set_clk 2 0 0
(sdbug) target jtag
Connected to the target JTAG.
(sdbug) load
.data: 0x   1 .. 0x   1 ... Loading
.text: 0x   0 .. 0x  ce ... Loading
(sdbug) hw enable_ice
(sdbug) hw insn_addr_brk 0x11
(sdbug) run
Starting program: hardware_debug.out
Connected to the target JTAG.
.data: 0x   1 .. 0x   1 ... Loading
.text: 0x   0 .. 0x  ce ... Loading
Before:
      r0:0000            r4:0000            r8:0000           r12:0000
      r1:0000            r5:0000            r9:0000           r13:0000
      r2:0000            r6:0000           r10:0000           r14:0000
      r3:0000            r7:0000           r11:0000           r15:0000

  %fmode:0000      %hwflag:0004           %pc:0000       %timer1:0000
     %tc:0000        %ireq:0060          %rpc:0000        %loop2:0000
  %imask:0000          c10:0000          %tpc:ffff        %loop3:0000
    %ip0:0000          c11:0000      %cb0_beg:0000           c27:0000
    %ip1:0000        %vitr:0000      %cb1_beg:0000           c28:0000
  %loop0:0000          c13:0000      %cb0_end:0000           c29:0000
  %loop1:0000        amode:0000      %cb1_end:0000          %dei:0000
  %guard:0000       %smode:0200       %timer0:0000          %ded:0000

Host: Waiting to scan out of target 6024 bits
Host: Writing scan command
Host: Scanned out of target 6024 bits ffff
Successfully entered HW Debug mode ...

(sdbug) i r 14
r14            0x00
(sdbug) i r 15
```

```
r15            0x00
(sdbug) i r pc
pc             0x1319
(sdbug) hw st_data_brk 0xab02
(sdbug) hw resume
Host: Scanning into target 6024 bits
Host: Finished scanning into target 6024 bits
Host: Waiting to scan out of target 6024 bits
Host: Writing scan command
Host: Scanned out of target 6024 bits ffff
(sdbug) i r 14
r14            0x44
(sdbug) i r 15
r15            0x00
(sdbug) i r pc
pc             0x3048
(sdbug) hw resume
Host: Scanning into target 6024 bits
Host: Finished scanning into target 6024 bits
Host: Waiting to scan out of target 6024 bits
Host: Writing scan command
Host: Scanned out of target 6024 bits ffff
(sdbug) i r 14
r14            0x77
(sdbug) i r 15
r15            0x00
(sdbug) i r pc
pc             0x4569
(sdbug) hw st_addr_brk 0x2000
(sdbug) hw resume
Host: Scanning into target 6024 bits
Host: Finished scanning into target 6024 bits
Host: Waiting to scan out of target 6024 bits
Host: Writing scan command
Host: Scanned out of target 6024 bits ffff
(sdbug) i r 14
r14            0x88
(sdbug) i r 15
r15            0x00
(sdbug) i r pc
pc             0x4c76
(sdbug) hw st_addr_and_data_brk 0x2001 0xab01
(sdbug) hw resume
Host: Scanning into target 6024 bits
Host: Finished scanning into target 6024 bits
Host: Waiting to scan out of target 6024 bits
Host: Writing scan command
Host: Scanned out of target 6024 bits ffff
(sdbug) i r 14
r14            0xd13
(sdbug) i r 15
r15            0x22
(sdbug) i r pc
```

```
pc              0x82130
(sdbug) **quit**
```

Chapter 10 ZSP Integrated Development Environment (ZSP IDE)

Version 4.0 of SDK Tools features a new Graphical Interface Integrated Development Environment for ZSP software project management, referred to as ZSP IDE. ZSP IDE is a productivity-enhancing tool for users of ZSP Processors, allowing easy setup, build, and debug of ZSP software projects. This chapter will focus on managing project structure and building executable ZSP programs. The ZSP IDE Debugger chapter describes the graphical user interface for the debugger.

Features of ZSP IDE –

- Workspaces to organize projects and default settings
- ZSP Project Build Support - G2, G1/G2, ZSP400
- Compatibility - Backward-compatible with Version 3.2 Projects.
- Windows and UNIX (planned) platforms
- Multiple Projects in same directory
- Build Output linked to Source File View
- Parallel Debug Manager

System Requirements – ZSP IDE requires PC/Windows 95/98/2000.

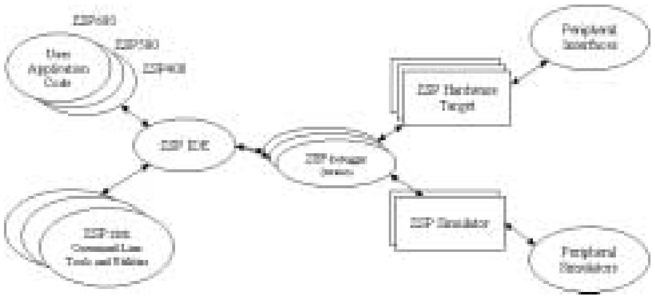
Although unsupported at this time, the IDE will be available on Solaris platforms as well in the future.

This section is organized as follows: ZSP IDE Overview, Workspace Overview, Project Overview, and detailed functional information.

10.1 ZSP IDE Overview

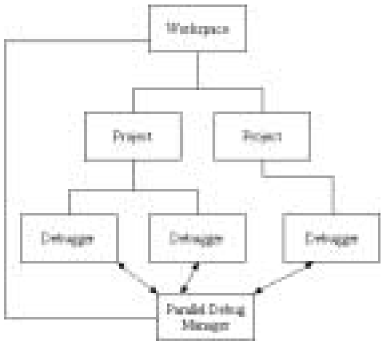
ZSP IDE provides an integrated tool suite for ZSP software developers by managing projects, building code, and debuging for all ZSP processors and supported targets. The graphical user interface allows easy project setup for users with minimal familiarity with ZSP tools and hardware.

Figure 10.1 ZSP IDE Tools Suite Implementation



10.1.1 Introduction to Workspaces and Projects

Figure 10.2 ZSP IDE Workspace



10.1.1.1 Project

The basic element of each ZSP software project is an executable file. Each executable file is managed by ZSP IDE based on settings that are created within ZSP IDE and stored in a project file. Project settings include all information needed to build and debug an executable:

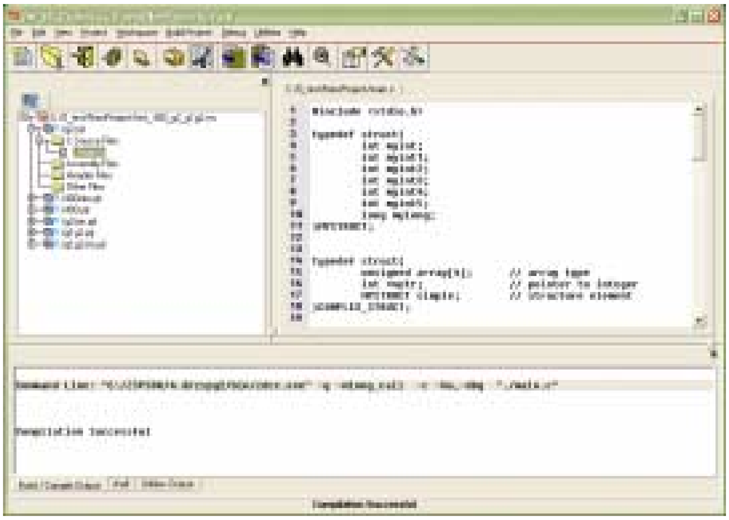
- Target ZSP Architecture
- Compiler Settings
- Include and Archive File directories
- Assembler Settings
- Debugger Settings
- IDE Debugger Window Settings

10.1.1.2 Workspace

A workspace may contains any grouping of projects with any combination of processor settings and debug targets. The workspace component of ZSP IDE allows maintenance of default settings for its component projects.

10.1.1.3 IDE

Figure 10.3 ZSP IDE Main Window



```
#include <stdio.h>

typedef struct{
        int myInt;
        int myInt1;
        int myInt2;
        int myInt3;
        int myInt4;
        int myInt5;
        long myLong;
}MYSTRUCT;


typedef struct{
        unsigned array[5];      // array type
        int *myptr;             // pointer to integer
        MYSTRUCT simple;        // structure element
}COMPLEX_STRUCT;
```

```
Command Line: [illegible]

Compilation Successful
```

The main window layout of ZSP IDE contains the main menu, toolbar, project tree, source file editing area and output/utility windows.

All of the main functions of ZSP IDE are available through the main menu. The most commonly used functions from the main menu are also accessible throught the toolbar. The project tree displays the workspace and project structure, allows opening of source files for editing, and provides quick access to pertinent menu functions through popup menus.

At the bottom of the ZSP IDE main screen is the output window which displays the output of build and compile commands. Additional tabs grouped with the output window in the lower section provide a basic operating system shell interface and an output window for post-processing functions (such as object dump utility) or for custom commands. The shell tab allows operating system command line capability from within the IDE. The Utility Output tab displays output of utility commands available from within the IDE.

The bottom part of the IDE shows status information. The current cursor location in the editor window is also reflected in this status area.

10.2 Working With Workspaces and Projects

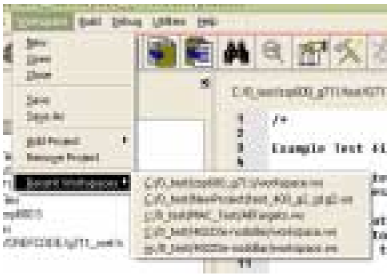
10.2.1 Working With Workspaces

The purpose of a workspace is to organize and to provide default settings for a Project or group of Projects. New and existing Projects may be added to a workspace. A Project may belong to multiple workspaces.

A set of default properties can be set for a workspace. Any new project added to the workspace will inherit the default settings of the workspace. These settings may later be altered by the user. When an existing project is added to a workspace, the user is given the option to either keep existing project settings or inherit the default settings of the workspace.

The Workspace menu has sub-menus to open, close and save workspace files. It also has sub-menus to add new or existing projects to a workspace. You can also delete projects from a workspace. A history of the previous workspaces visited is also available to quickly switch between workspaces. Only one workspace may be active at any time. Switching to a different workspace will close the existing workspace and the component projects. If a source file was altered and not yet saved, a warning is issued and the user is provided with an option to save changes before switching to a different workspace.

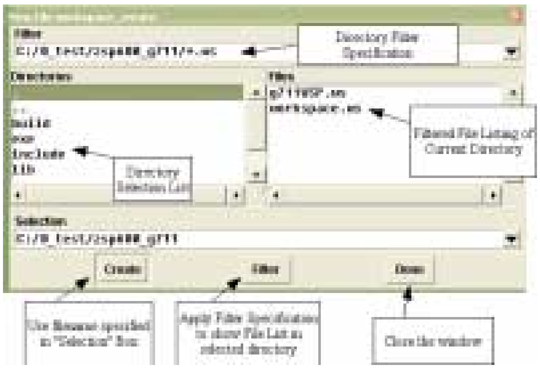
Figure 10.4 Recent Workspaces List



10.2.1.1 Creating a new workspace

A new workspace can be created by selecting Workspace>New from the IDE Main Menu. A dialog box is displayed to specify the filename for the new workspace.

Figure 10.5 File Selection Dialog


Directory Filter Specification
Filtered File Listing of Current Directory
Directory Selection List
Use filename specified in "Selection" Box
Apply Filter Specification to show File List in selected directory
Close the window
Create
Filter
Done

The workspace filename will be taken from the Selection Entry in the Dialog Box. You may type or cut and paste the selected pathname directly into the Selection entry box and select the Create command button to create the new workspace, or you may navigate to a new directory from the currently selected one. The Files area shows the filtered contents of the selected directory when the Filter command button is selected. To limit file extensions shown in the file selection box, type the filter specification into the Filter Entry area and select the Filter button at the bottom of the Dialog Box. To change the directory, select from the directory selection box or type the directory into the Filter Entry area. To create a new directory, enter the desired path into the Selection entry area. The new directory will be created when the Create command button is selected. Workspace file names will always be specified by the filename extension ".ws". This is not modifiable. If another extension or no extension is specified, then the .ws extension will be created for the base filename entered.

Note: Filenames may not contain space characters.

10.2.1.2 Open a Workspace

An existing workspace can be opened using the same procedure as described in the previous section.

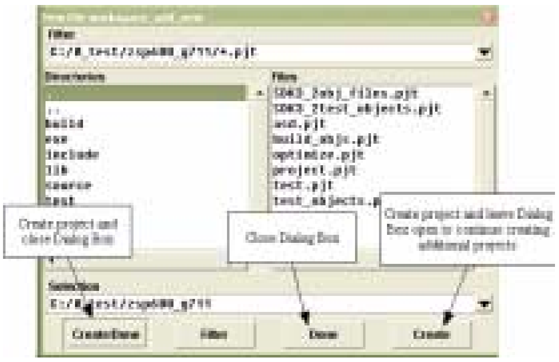
10.2.1.3 Save a Workspace

Select Save from Workspace menu to save the current workspace.

Select Save As from the Workspace menu to display the file selection dialog box to save a workspace with a different name or in a different directory. The new workspace becomes the current session after executing "Workspace -> Save As".

10.2.1.4 Add Projects to a Workspace

To add new or existing projects to a workspace, select "Add Project" from the Workspace menu. The file selection dialog for projects is similar to that used for creating workspaces. The default filename extension for project files is ".pjt".



It is also possible to add multiple projects to a workspace without closing the dialog box by selecting the Create command button for each project to be added. When all projects have been added, select Done to close the dialog box.

10.2.1.5 Delete a Project from a Workspace

Select the project to be deleted from the workspace in the project explorer window then select Workspace->Remove Project.

10.2.1.6 Close a Workspace

To close a workspace select Workspace->Close from the Workspace menu. Before closing a workspace, the user is prompted to save any unsaved files.

10.2.2 Working With Projects

A project is a container for source files, object files, executable files, build settings and debugger settings.

Each project's settings are stored in a file with a .pjt extension. It is not necessary for the constituent files to be resident in the same directory as the project. The project can be moved as long as the paths to the source files are correct. Source files, header files, libraries and object modules can be shared across multiple projects. Multiple project files may exist in the same directory.

The Project menu has sub-menus to open, close and save project files, and a sub-menu to add new or existing files to a project. You can also delete files from a project. A history of projects recently visited is available to quickly move between projects. Only one project can be active at any time. Switching to a different project will close the existing project. If a source file was altered, a warning is issued and the user is provided with an option to save changes before switching to a different project.

Figure 10.6 Project Menu

10.2.2.1 Creating a New Project

To create a new project within a workspace, you may select either from the main menu Workspace>Add Project>New Project or from the Project Tree popup menu over the active workspace node Add Project> New Project.

A dialog box similar to that described for workspaces will be displayed and you may create the new project using the same methods.

10.2.2.2 Opening an Existing Project

A Project can be opened by the following options

Step 1. Project -> Open. The file selection dialog is displayed.

Step 2. One can browse to the appropriate directory and specify the project file (.pjt file) to be opened.

Step 3. Click OK

This will open the selected project. All associated component source, header, and object files will be shown in the project explorer pane.

10.2.2.3 Saving a Project

To save a project, select Save from the Project menu.

To save a project to a new project file name, select Save As from the Project Menu. A dialog box is displayed to save the project with a different name or in a different directory. The new name is immediately reflected in the project explorer window and the new project becomes active.

10.2.2.4 Add Files to a Project

To add new or existing files to a workspace, select “Add File” from the Project menu. The file selection dialog for files is similar to that used for creating projects. There is no default filename extension for files. The initial filter in the file selection dialog suggests source files. Edit the filter specification in the filter entry to display listings of files with alternate extensions.

It is also possible to add multiple files to a project without closing the dialog box by selecting the Create command button for each file to be added. When all files have been added, select Done to close the dialog box.

10.2.2.5 Delete Files from project

To delete a file from a project, use the popup menu over the file you would like to remove to select “Remove From Project”. You may also select the file to be deleted from the project explorer window then select Project->Remove Files from the main menu.

10.2.2.6 Close a project

To close a project select Project->Close from the Project menu. Before closing project the user is prompted to save any unsaved files.

10.3 Project Settings

Selecting Build->Settings or Debug->Settings from the main menu invokes the display of the Settings Dialog. If a workspace node is selected in the project tree then the Settings Dialog will reflect the default settings for the workspace.

The Settings Dialog contains a tabbed notebook view that contains all of the settings for a project, including settings for the ZSP compiler, assembler, linker, debugger, and GUI debugger preferences. These tabs are described below. You may page between the various tabs on the Settings Dialog and make changes. When the changes are complete for all of the tabs, select “Save and Exit” to save the settings to the project file and close the Dialog Box. Select “Exit without Saving” to discard the changes.

10.3.1 Build methodology and Project Tree Structure

The ZSP IDE project tree partitions project files into “folders” based on filename extensions. Source files which have extension of “.c” for C sources are added to the “C Source Files” folder. Source files which have a “.S” extension are assembly sources that require C preprocessing while a “.s” extension indicates an assembly source file which does not require preprocessing. Both “.S” and “.s” are inserted into the IDE Project Tree in “Assembly Source Files” folder. Include files which have extensions of “.h” or “.inc” will be added to the project tree “Include Files” folder. Additionally, when a file with a “.h” or “.inc” extension is added to the project, ZSP IDE will provide a prompt allowing the directory containing the files to the Include Directories list. Files with any other extension than those described here are inserted into the project in the “Other Files” folder and will not be part of the build process.

The ZSP IDE invokes the appropriate ZSP compiler (SDCC ZDCC ZDXCC) based on the processor type selected in the Settings dialog. The ZSP compiler invokes each of the component processes that complete the build process. Options may be specified to direct the behavior or the compiler, assembler, and linker. These are specified in the Settings panel.

10.3.2 Compiler/Assembler Settings

The Compiler Settings tab is the primary control for each project. The processor architecture selected in the Compiler Settings tab controls the entire set of underlying command line tools and utilities. The three available architecture choices are

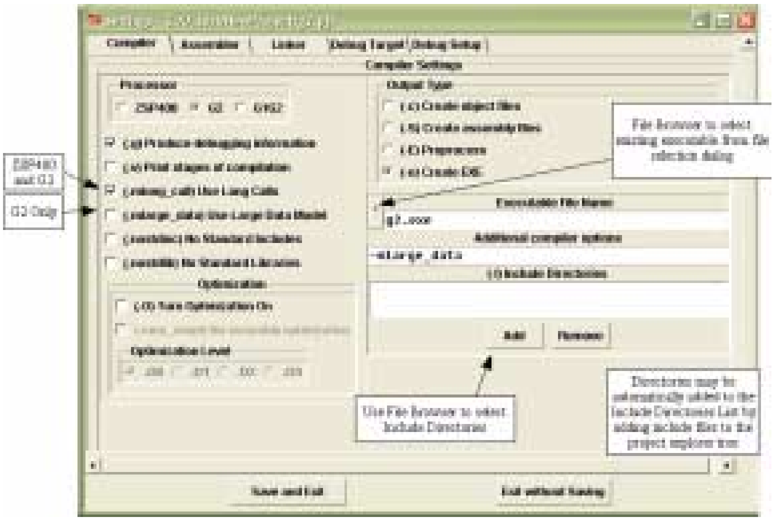
- G2 - This option selects the ZSP G2 architecture.
- ZSP400 - This option selects the ZSP400 architecture.
- G1G2 - This option is provided to enable building ZSP400 code for processors based on ZSP G2 architecture.

ZSP400 is the first generation ZSP architecture. This setting will work for all ASSPs based on this core (example LSI402ZX, LSI403Z, LSI403LP).

ZSPG2 is the next generation architecture in the ZSP roadmap. It has many new instructions, new resources and a bigger address range. It is assembly compatible with the ZSP400.

At this time there is a dual mac core called ZSP500 that is being designed based on the ZSPG2 architecture. It supports a 24-bit address range and is a 4 issue machine. The simulators in the toolchain support the ZSP500 in a cycle accurate and instruction accurate modes. Refer to the ZSP400 and ZSPG2 manuals for more information. Select G2 to compile for ZSP500 or G1/G2 to compile ZSP400 source code for G2.

Figure 10.7 Compiler Settings



Compiler Settings

Processor: ZSP400 [✓ G2] G162

[✓] (-g) Produce debugging information

(-v) Print stages of compilation

[✓] (-mlong_call) Use Long Calls

(-mlarge_data) Use Large Data Model

(-nostdinc) No Standard Includes

(-nostdlib) No Standard Libraries

Optimization

(-O) Turn Optimization On

Optimization Level: [✓ -O0] -O1 -O2 -O3

Output Type

(-c) Create object files

(-S) Create assembly files

(-E) Preprocess

[✓ (-o) Create EXE]

Executable File Name: g2.exe

Additional compiler options: -mlarge_data

(-I) Include Directories

ZSP400 and G2

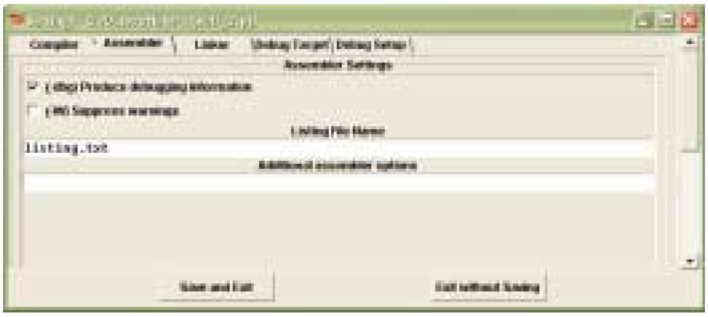
G2 Only

File Browser to select existing executable from file selection dialog

Use File Browser to select Include Directories

Directories may be automatically added to the Include Directories List by adding include files to the project explorer tree

Figure 10.8 Assembler Settings



Assembler Settings

[✓] (-dbg) Produce debugging information

(-W) Suppress warnings

Listing File Name: listing.txt

Additional assembler options

The following table describes the other options that control the Compiler and Assembler.

Table 10.1 Compiler/Assembler Options

Option (Command Line Equivalent)		Description
Produce debugging information (-g)		This option instructs the compiler to produces debugging information for source-level debugging.
Print stages of compilation (-v)		This option instructs the compiler to print the commands executed in stages of compilation, and to print the version number of the tools before compilation.
Optimization (-O number)		This option instructs the compiler to produce optimized code. Select optimization level 0-3. See the compiler section of this document for more details regarding optimization levels and the impact of optimizaton on debugging capabilities.
No assembly optimization (-mno_sdopt)		This option supresses back-end optimization that is otherwise automatically performed on compiler-generated assembly code.
Use Long calls (-mlong_call)		This option will tell the compiler to use register-based calls (long calls). These calls can be optimized where possible if back-end optimization is enabled.
Use Large Data Model (-mlarge_data)		The large data model has no requirements on the size or placement of the data and bss sections.
Additional compiler options Text Box (option)		This option specifies any compiler options that do not have a check box in the Compiler/Assembler options tab. Separate multiple options with spaces.
Output Options	Create object files (-c)	This option instructs the compiler to compile and assemble the source files and produces object file(s) only (no linking is performed).
	Create assembly files (-s)	This option instructs the compiler to stop after compilation and produces assembly code files for each C source file specified.
	Preprocess files (-E)	This option stops compilation after the preprocessing stage and redirects the preprocessed output to standard output.
	Create executable (-o)	This option instructs the compiler to compile all sources and link objects into the executable file specified in the Executable File Name entry.
	Executable File Name	Specify the name of the executable file you want here.

Table 10.1 Compiler/Assembler Options

Option (Command Line Equivalent)	Description
No standard includes (-nostdinc)	This option directs the compiler not to search the standard system directories for header and include files.
Include Directories (-I)	Include Directories is a list of directories that the compiler will search for header and include files.
No Standard Libraries (-nostdlib)	This option forces the compiler to not use the standard system startup files or libraries during linking.
Suppress warnings (-W)	This option suppresses warning messages.
Listing option (-a)	This option produces a listing file. The listing file includes high-level source information, assembly instruction information, and symbol information. Type a filename in the text box to save the listing to a file. The listing is sent to standard output if no filename is specified.
Additional compiler options Text Box (option)	This option specifies any compiler options that do not have a check box in the Compiler/Assembler options tab. Separate multiple options with spaces.
Produce debugging information (-dbg)	This option includes debugging symbols in the object file to allow source-level debugging of assembly files.
Additional assembler options Text Box (option)	This option specifies any assembler options that do not have a check box in the Compiler/Assembler options tab. Separate multiple options with spaces.

The following table describes the options that control the linker.

10.3.3 Linker Settings

The Linker Settings window provides detailed control over link behavior. See the Linker section of this manual for more detail.

a. Entry Point - The Entry Point directive to the linker specifies the starting address or label of the executable. The default is the label "__start" (provided in crt0.obj for C programs). For assembly programs you may specify the entry point to be any valid label or address, or you may accept the default which is the start of the ".text" section.

b. Locate Stack - The Locate Stack defines the __stack_start symbol that determines the starting address of the program stack pointer.

c. Define Symbols - You may define other symbols with this option.

d. Code Section (-Ttext) - You may specify the starting address for the ".text" section by entering a valid address in this entry box.

e. Data Section (-Tdata) - You may specify the starting address for the initialized data section by entering a valid address in this entry box.

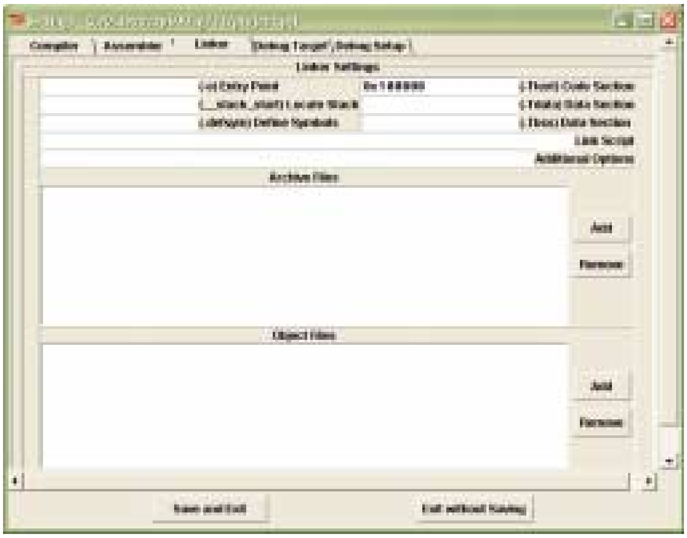
f. BSS Section (-Tbss) - You may specify the starting address for the uninitialized data section by entering a valid address in this entry box.

g. Link Script - If you need more control over the locations of sections in you executable, you may specify a link script file in this entry. If you specify a relative pathname, it should be relative to your project directory.

h. Additional Options - Specify any additional options for linking.

i. Archive Files - Archive Files is a list of archive files to be linked with your project's object files to produce the executable.

j. Object Files - Object Files is a list of object files to be linked with your project's object files to product the execuatable.



For archive and object files, you may invoke a file browser to select the files by selecting the appropriate “Add” command button. Likewise, remove a file from the list by selecting it with the mouse and then selecting the “Remove” command button.

Table 10.2 Linker options

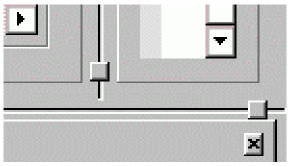
Option (Command Line Equivalent)	**Description**
Entry Point (-e)	This option specifies a symbol for beginning execution of the program. The default entry point is __start. Enter the symbol in the text box.
Locate Stack (__stack_start)	This entry defines the symbol __stack_start for the Linker to explicitly locate the program stack.
Define symbols (-defsym)	Creates a symbol in the output file containing the absolute address specified by the expression. Enter the symbol and the expression in the text box, using the following syntax: symbol=expression. Note that spaces are not allowed next to the ‘=’ sign.
Code Section(-Ttext)	This entry specifies the starting address for the text segment of the output file. Default value is 0x0
Data Section (-Tdata)	This entry specifies the starting address of the data segment of the output file. Default value is 0x0.
Data Section (-Tbss)	This entry specifies the starting address of the uninitialized data segment of the output file. Default value is 0x0.
Link Script	This entry specifies a filename to be used as a Linker Command File. Filename extension must not conflict with source / object file name extensions.
Object Files	Specifies external object files to be linked with the project’s object files. Select Add button to select new object files from a File Selection Dialog box. To remove an object file from the list, select the entry with the mouse and then select Remove.
Archive files List Box (-L)	Specifies external archive files to be linked with the project’s object files. Select Add button to select new archive files from a File Selection Dialog box. To remove an archive file from the list, select the entry with the mouse and then select Remove.
Additional options	This entry specifies any linker options that do not have a check box in the Linker options tab. Separate multiple options with spaces.

10.4 ZSP IDE Detailed Description

10.4.1 Paned Window Controls

The IDE Main Window is divided into resizable sections by Paned Window Controls. IDE screen area displayed in the Paned Window may may be resized by dragging the handles of the paned window controls that separate the screen areas.

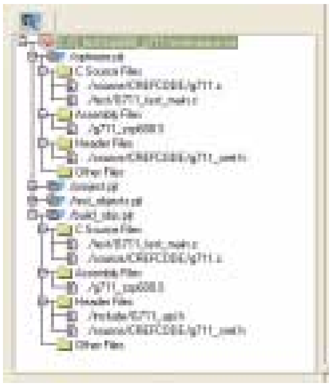
Figure 10.9 Paned Window Handles



10.4.2 Project Tree

The Project Tree component of the ZSP IDE shows a hierarchical view of the files included in your projects and workspace. The Project Tree also provides the primary means of selecting the active project or workspace component.

Figure 10.10 ZSP IDE Project Tree

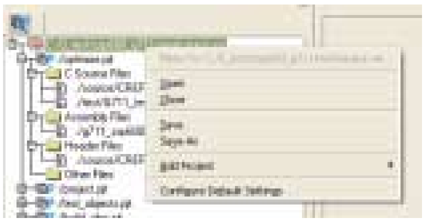


Select the workspace node of the tree to customize default settings for your workspace. Default settings are applied to new projects when they are created within your workspace. Default settings may also be applied to existing projects when they are added to your workspace. To apply default settings to existing projects, select the checkbutton labeled "Use Workspace Settings" in the Preferences Window. To invoke the Preferences Window select Preferences from the View Menu. See Section 10.4.3.5, "View Menu," page 10-21 for details on the preferences window.

Select a project or file from the tree to activate the project file as the Current Project. The Current Project is the project affected by Build and Debug operations.

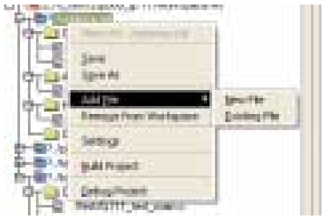
Select a file from the Project Tree to edit the file in the Edit Window.

A popup menu is available for the workspace node of the project tree. To invoke the Workspace Popup Menu, click the right mouse button over the workspace node of the tree.

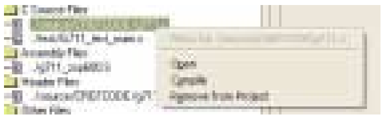


The Workspace Popup Menu shows the name of the workspace followed by shortcuts to workspace menu items from the main menu.

A popup menu is available for a project node of the project tree in the same fashion as above.



A popup menu is available for the filenode of the project tree in the same fashion as above.



10.4.3 Main Menu

The Main Menu provides access to major functions of ZSP IDE such as opening, closing, and maintaining workspaces, projects and files, as well as building and debugging projects

10.4.3.1 Operating the Main Menu

Main menu items may be selected either by left-clicking with the mouse or by typing on the keyboard using menu accelerator keys (Underlined character in the menu name). To invoke the menus from the keyboard, depress the ALT key and the accelerator key for the Main Menu item concurrently. This will display the pull down subitem menu from which you can make further selections without using the ALT key. You may also use the Up, Down, Left, and Right arrow keys to navigate through the menus, terminating your choice with either the Enter key to confirm or the Escape key to cancel your selection.

10.4.3.2 Main Menu Functions

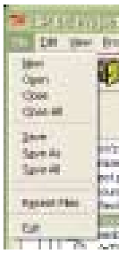
The ZSP IDE Main Menu provides the following submenus:

- File Menu
- Edit Menu
- View Menu
- Project Menu
- Workspace Menu
- Build Menu
- Debug Menu
- Help Menu

10.4.3.3 File Menu

The File Menu is used for operations on text files such as source files, include files, batch files, or any other text file. This menu is provided to open new or existing files and save and close active files.

Figure 10.11 ZSP IDE File Menu



A file opened using the file menu does not automatically belong to the active project. A file needs to be explicitly added to a project as described in the section on Projects.

A file may be opened and edited even if no workspace or project is active.

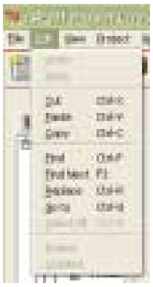
10.4.3.4 Edit Menu

A simple editor is included in the IDE. The Edit Menu provides options that may be useful during editing. It is fairly intuitive to use and provides standard edit functionality like cut, copy, paste, indent, outdent, find, replace, select all, undo and redo.

The Edit functions are applicable to a file that is being edited in the Edit Window. They are not applicable for projects, workspaces and directories and will result in errors if used for anything but File editing.

Short cut keys are also available for common edit functions.

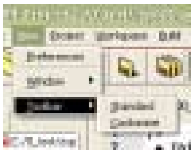
Figure 10.12 ZSP IDE Edit Menu



10.4.3.5 View Menu

The view menu is available to selectively display and customize ZSP IDE Screen components.

Figure 10.13 ZSP IDE View Menu

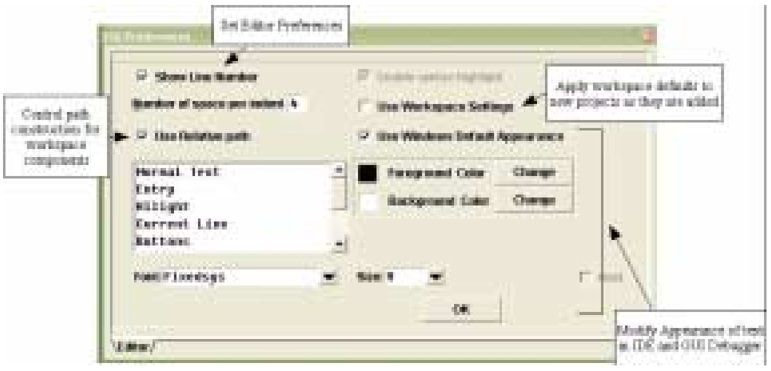


View Preferences

A user may set IDE enviromnent preferences by selecting View->Preferences. The preferences window offers options to alter editor settings in a tab labeled “Editor”. Colors, Text style, line number and other preferences can be set in this window.

The checkbutton labeled “Use workspace settings” controls the default project settings when a new project is created. If it is checked, then the project will be created with the default workspace settings, otherwise the project will be created with generic defaults.

The checkbutton labeled “Use Relative Path” controls the type of path that is created within the workspace and projects. If it is not checked then absolute paths will be used for workspace components (projects and files, include directories, etc.) Otherwise relative paths are used. Relative path heirarchy begins with the workspace, (which is always an absolute path). Projects are relative to the workspace. Files and other project component paths are relative to the project directory.


Set Editor Preferences
Apply workspace defaults to new projects as they are added
Control path construction for workspace components
Modify Appearance of text in IDE and GUI Debugger
Editor
Change
Change
OK

☑ Show Line Number

Number of spaces per indent: 4

☑ Use Relative path

☐ Use Workspace Settings

☑ Use Windows Default Appearance

Normal Text
Entry
Hilight
Current Line
Buttons

■ Foreground Color

□ Background Color

Font: Fixedsys

Size: 9

After finishing setting the preferences, click OK to save these settings.

View Window

View->Window provides the option to display or hide the Project Explorer set of tabs and the Output set of tabs. A check mark to the left of the item denotes if the window is active. The setting is toggled each time an item is selected.

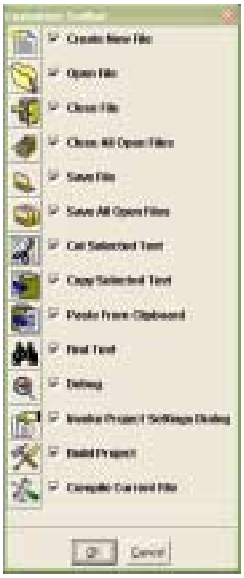
View Toolbar

The Toolbar Buttons icons displayed in the toolbar at the top of the IDE window can be customized to a user’s liking.

There is a default that comes up as the standard toolbar. A user may select View->Toolbar->customize to customize the toolbar. When customize is selected a window titled “Customize Toolbar” pops up that shows the various options available for customizing. On clicking OK, the toolbar will be altered to display the customized settings.

Select View->Toolbar->Customize to display the Customize Toolbar Window which allows selection of toolbar buttons to be displayed.

Figure 10.14 Customize Toolbar



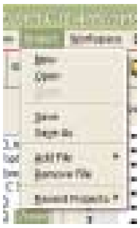
Switch back to the standard settings by selecting View->Toolbar->Standard.

A check mark to the left of the item denotes if the selection is active. The setting is toggled each time an item is selected.

10.4.3.6 Project Menu

The Project Menu allows projects to be created and maintained.

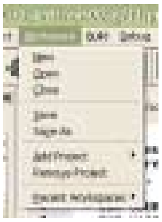
Figure 10.15 ZSP IDE Project Menu



10.4.3.7 Workspace Menu

The Workspace Menu allows Workspaces to be created and maintained

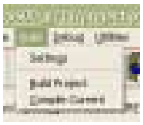
Figure 10.16 ZSP IDE Workspace Menu



10.4.3.8 Build Menu

The Build Menu invokes the ZSP IDE Build process and allows customization of Project Build Parameters.

Figure 10.17 ZSP IDE Build Menu



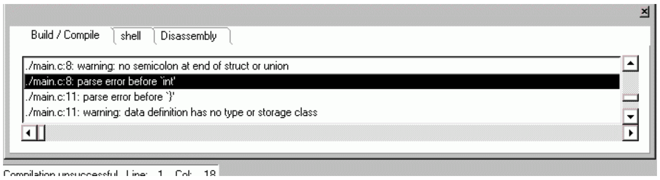
Build Project

Once a project is created and the constituent files added to it, the build settings which control the options with which the underlying tools (compiler, assembler, linker) are invoked can be set and the executable can be built.

Build project will build the executable, using the options specified in the Project Settings window. This functionality is also available from the popup menu on the project tree when a project file is the selected node.

When building the executable, build messages will be displayed on the output window in the Build / Compile Output tab if enabled.

Figure 10.18 Build / Compile Output Window



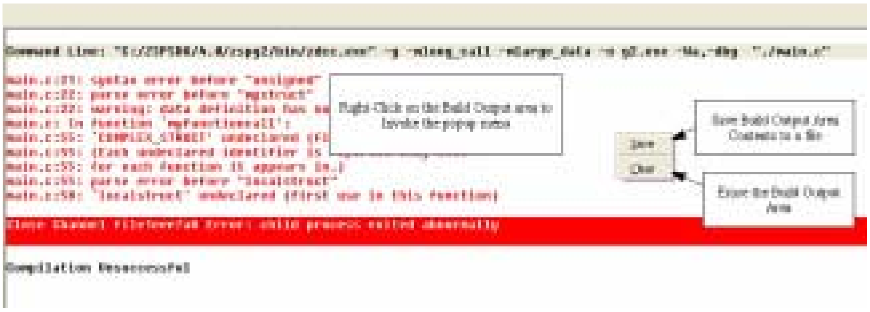
```
./main.c:8: warning: no semicolon at end of struct or union
./main.c:8: parse error before `int'
./main.c:11: parse error before `}'
./main.c:11: warning: data definition has no type or storage class
```

Compilation unsuccessful Line: 1 Col: 18

The Build Output Window displays all the standard error or output of the process of building or compiling a project. The output can be saved by right clicking on the window. If errors in building are shown in the Build Output Window, you may easily display the source file and line containing the error in the Edit Window by double-clicking with the left mouse button on the line in the Build Output Window.

A popup menu is available within the Build Output Window to save or clear the window contents.

Figure 10.19 Build Output Window Popup Menu



```
Command Line: "E:/ZSP500/4.0/zspg2/bin/zdcc.exe" -g -mlong_call -mlarge_data -o g2.exe -Wa,-dbg "./main.c"

main.c:21: syntax error before "unsigned"
main.c:22: parse error before "mystruct"
main.c:22: warning: data definition has no [illegible]
main.c: In function `myfunctioncall':
main.c:55: `COMPLEX_STRUCT' undeclared [illegible]
main.c:55: (Each undeclared identifier is [illegible]
main.c:55: for each function it appears in.)
main.c:55: parse error before "localstruct"
main.c:58: `localstruct' undeclared (first use in this function)
Close Channel FileIsDone Failed Error: child process exited abnormally

Compilation Unsuccessful
```

Right-Click on the Build Output area to Invoke the popup menu
Save
Clear
Save Build Output Area Contents to a file
Erase the Build Output Area

Settings

Select Settings to customize the parameters to be used for building your project. This functionality is also available from the popup menu on the project tree when the project file is the selected node.

Compile Current

Select “Compile Current” to compile the currently selected source file. The ZSP compiler is invoked with the -c option and an object file will be produced with the same base name as the input file and an extension of “.obj”. This functionality is also available from the popup menu on the project tree when the source file is the selected node.

10.4.3.9 Debug Menu

The Debug Menu provides configuration and control of Project Debugging.

Figure 10.20 ZSP IDE Debug Menu

Use Default Target

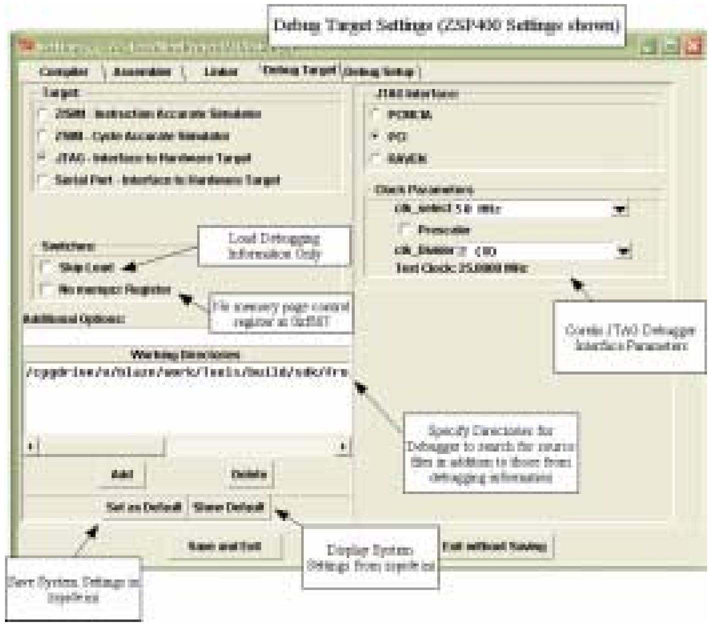
Select Use Default Target from the Debug Menu to ignore the Target Settings in the project file and use instead the default settings you have saved for your computer. Use Default Target is a system setting and is not a component of the project. The default target setting is stored in the ZSP Tools Program directory in a file named zspide.ini.

Target Settings

Select Target Settings from the Debug Menu to display the Debug Target Settings Panel

Debug Settings displays the valid target types for the processor type that is specified in your project's Build Options. For ZSP400, valid Debug Targets are ZSIM, ZISIM, and Hardware Targets. For G2, valid Debug targets are software simulators ZSIM and ZISIM.

Figure 10.21 Debug Settings


Debug Target Settings (ZSP400 Settings shown)
Load Debugging Information Only
No memory page control register at 0xffff
Config JTAG Debugger Interface Parameters
Specify Directories for Debugger to search for source files in addition to those from debugging information
Display System Settings from zspide.ini
Save System Settings in zspide.ini
Compiler
Assembler
Linker
Debug Target
Debug Setup
Add
Delete
Set as Default
Show Default
Save and Exit
Exit without Saving

Target
- ZSIM - Instruction Accurate Simulator
- ZSIM - Cycle Accurate Simulator
- [✓ JTAG - Interface to Hardware Target]
- Serial Port - Interface to Hardware Target

JTAG Interface
- PCMCIA
- [✓ PCI]
- RAVEN

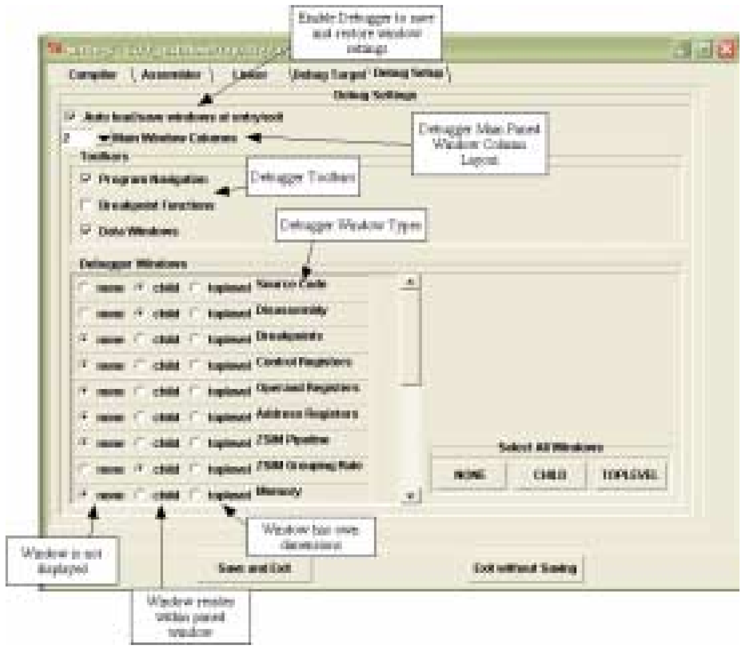
Clock Parameters
clk_select 5.0 MHz
Prescale
clk_divider 2 (/4)
Test Clock: 25.0000 MHz

Switches
- Skip Load
- No memory Register

Additional Options:

Working Directories
/cygdrive/e/blaze/work/tools/build/sdk/fro

Figure 10.22 Debug Window Settings



Debug Settings

[✓] Auto load/save windows at entry/exit
2 Main Window Columns

Toolbars
- [✓] Program Navigation
- [ ] Breakpoint Functions
- [✓] Data Windows

Debugger Windows

none	[✓ child]	toplevel	Source Code
none	[✓ child]	toplevel	Disassembly
[✓ none]	child	toplevel	Breakpoints
[✓ none]	child	toplevel	Control Registers
[✓ none]	child	toplevel	Operand Registers
[✓ none]	child	toplevel	Address Registers
[✓ none]	child	toplevel	ZSIM Pipeline
none	[✓ child]	toplevel	ZSIM Grouping Rule
[✓ none]	child	toplevel	Memory

Run

Select Run to launch the ZSP IDE Debugger using the selected processor and debug target settings.

Invoke PDM

Select Invoke PDM from the Debug Menu to run the Parallel Debug Manager component of ZSP IDE. PDM allows concurrent debugging of projects. PDM is valid when a workspace is active and operates on all projects selected from within the current workspace. See Section 10.7, "Parallel Debug Manager," page 10-33 for more information on this feature.

10.4.3.10 Utilities Menu

objdump

Select "objdump" from the Utilities Menu to invoke the objdump dialog box.

The Object File Utility dialog box shows information about object files. The default object file is the compiler output file from the currently selected project. You may select another object file from a file selection dialog for processing by selecting the "Choose File" command button.

Figure 10.23 Object File Utility

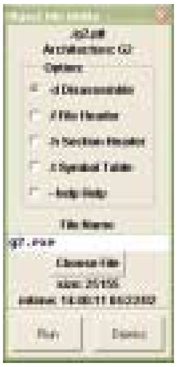
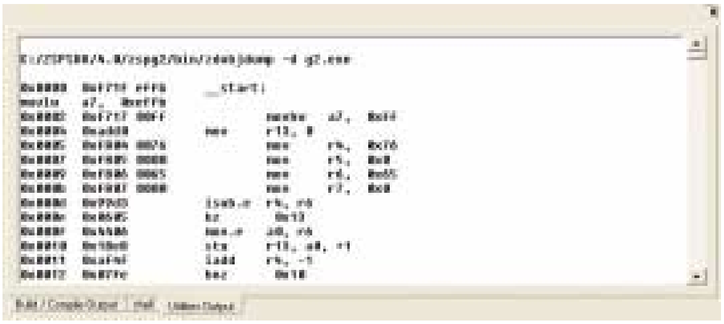


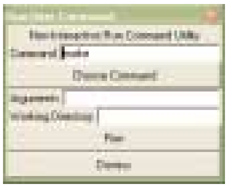
Figure 10.24 Utility Output Window Showing Disassembled Code



```
E:/ZSP500/4.0/zspg2/bin/zdobjdump -d g2.exe

0x0000  0xf7ff effb     __start:
movlo   a7,  0xeffb
0x0002  0xf7f7 00ff            movhx  a7,  0xff
0x0004  0xad00          mov    r13, 0
0x0005  0xf804 0076            mov    r4,  0x76
0x0007  0xf805 0000            mov    r5,  0x0
0x0009  0xf806 0065            mov    r6,  0x65
0x000b  0xf807 0000            mov    r7,  0x0
0x000d  0x7b23          lsub.e r4, r6
0x000e  0x0505          bz      0x13
0x000f  0x5506          mov.e  a0, r6
0x0010  0x18e0          stx    r13, a0, +1
0x0011  0xaf4f          ladd   r4, -1
0x0012  0x07fe          bnz     0x10
```

User Command

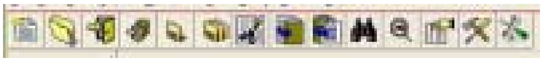
Select User Command from the Utilities menu to display a dialog box that allows execution of a custom command to be executed.



10.4.4 Toolbar

The Toolbar provides easy access to commonly used functions of ZSP IDE.

Table 10.3 ZSP IDE Toolbar



The following functions are available through the toolbar.

New File

Select the “New File” toolbar button to create a new text file in the IDE editor window. The new file is not automatically included in the current working project. If you wish the new file to be a project component, use the “Add File” option either from the main menu Project Menu or from the project tree popup menu over the selected project.

Open File

Select the “Open File” toolbar button to open an existing text file in the IDE editor window. The opened file is not automatically included in the current working project. If you wish the opened file to be a project component, use the “Add File” option either from the main menu Project Menu or from the project tree popup menu over the selected project.

Close

Select the “Close File” toolbar button to close the text file that is the being edited in the IDE edit window.

Close All

Select the “Close All” toolbar button to close all of the text files that are the being edited in the IDE edit window.

Save

Select the “Save” toolbar button to save the file that is currently being edited in the editor window.

Save All

Select the “Save All” toolbar button to save all of the files that are present in the editor window and that have been modified.

Cut

Select the “Cut” toolbar button to cut selected text from the editor window.

Copy

Select the “Copy” toolbar button to copy the selected text from the editor window into the clipboard buffer.

Paste

Select the “Paste” toolbar button to paste the contents of the clipboard at the insertion point in the text file in the edit window.

Find

Select the “Find” toolbar button to invoke the Find Dialog, which allows searching the current source file for the desired text.

Settings

Select the “Settings” toolbar button to display the Settings window for the currently selected project or workspace.

Build

Select the “Build” toolbar button to invoke the build tools using the settings from the currently selected project

Compile

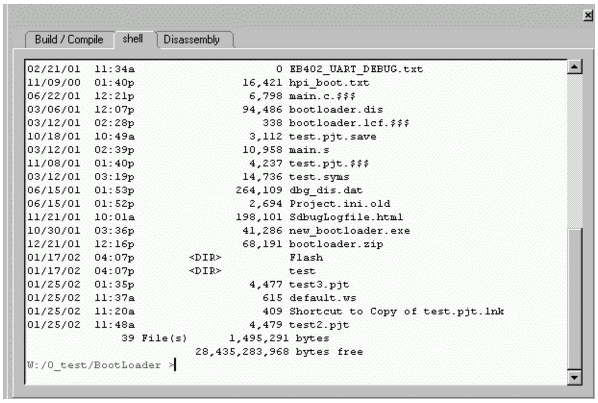
Select the “Compile” toolbar button to compile the currently selected source file.

Debug

Select the “Debug” toolbar button to invoke the GUI debugger for the currently selected project.

10.5 Shell Window

Figure 10.25 Shell Window

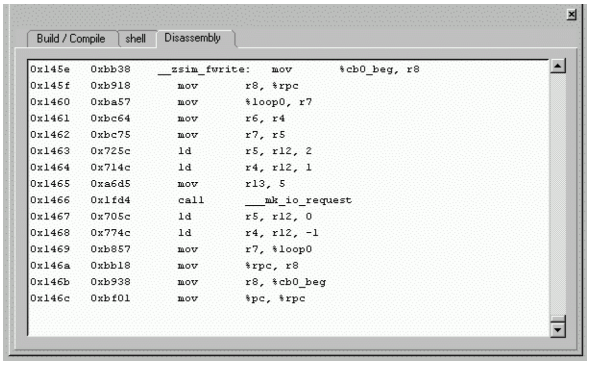


```
02/21/01  11:34a                     0 EB402_UART_DEBUG.txt
11/09/00  01:40p                16,421 hpi_boot.txt
06/22/01  12:21p                 6,798 main.c.$$$
03/06/01  12:07p                94,486 bootloader.dis
03/12/01  02:28p                   338 bootloader.lcf.$$$
10/18/01  10:49a                 3,112 test.pjt.save
03/12/01  02:39p                10,958 main.s
11/08/01  01:40p                 4,237 test.pjt.$$$
03/12/01  03:19p                14,736 test.syms
06/15/01  01:53p               264,109 dbg_dis.dat
06/15/01  01:52p                 2,694 Project.ini.old
11/21/01  10:01a               198,101 SdbugLogfile.html
10/30/01  03:36p                41,286 new_bootloader.exe
12/21/01  12:16p                68,191 bootloader.zip
01/17/02  04:07p        <DIR>          Flash
01/17/02  04:07p        <DIR>          test
01/25/02  01:35p                 4,477 test3.pjt
01/25/02  11:37a                   615 default.ws
01/25/02  11:20a                   409 Shortcut to Copy of test.pjt.lnk
01/25/02  11:48a                 4,479 test2.pjt
              39 File(s)      1,495,291 bytes
                         28,435,283,968 bytes free
W:/0_test/BootLoader >
```

This is a window, where you can type dos commands

10.6 Disassembly Window

Figure 10.26 Disassembly Window



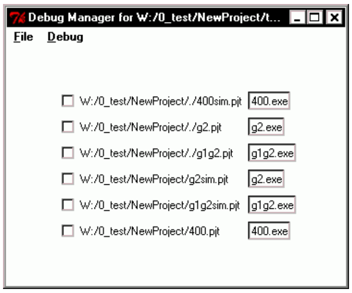
```
0x145e   0xbb38    __zsim_fwrite:   mov       %cb0_beg, r8
0x145f   0xb918       mov       r8, %rpc
0x1460   0xba57       mov       %loop0, r7
0x1461   0xbc64       mov       r6, r4
0x1462   0xbc75       mov       r7, r5
0x1463   0x725c       ld        r5, r12, 2
0x1464   0x714c       ld        r4, r12, 1
0x1465   0xa6d5       mov       r13, 5
0x1466   0x1fd4       call      ___mk_io_request
0x1467   0x705c       ld        r5, r12, 0
0x1468   0x774c       ld        r4, r12, -1
0x1469   0xb857       mov       r7, %loop0
0x146a   0xbb18       mov       %rpc, r8
0x146b   0xb938       mov       r8, %cb0_beg
0x146c   0xbf01       mov       %pc, %rpc
```

The disassembly window shows disassembled code sections and is generated from the executable file. To populate the Disassembly Window, select Disassembly from the View Menu.

10.7 Parallel Debug Manager

When PDM starts, a configuration window is displayed from which you may select the projects from within your workspace that you would like to debug. Selected projects display a checkmark in the selection box.

Figure 10.27 Parallel Debug Manager Setup Window

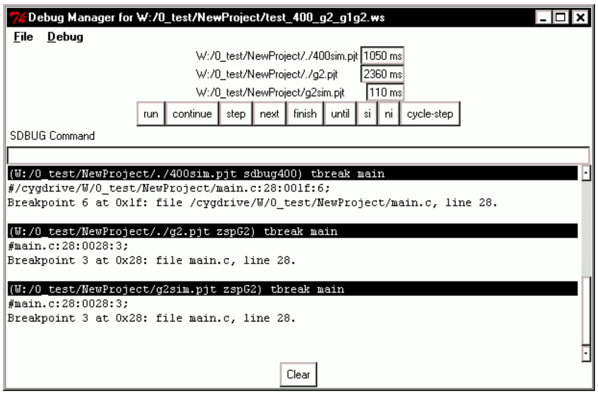


☐ W:/0_test/NewProject/./400sim.pjt	400.exe
☐ W:/0_test/NewProject/./g2.pjt	g2.exe
☐ W:/0_test/NewProject/./g1g2.pjt	g1g2.exe
☐ W:/0_test/NewProject/g2sim.pjt	g2.exe
☐ W:/0_test/NewProject/g1g2sim.pjt	g1g2.exe
☐ W:/0_test/NewProject/400.pjt	400.exe

Select Run from the Debug menu choice to start debugging. The PDM window changes to debugging mode and ZSP IDE Debuggers are launched for each of the projects selected. Each debugger may be controlled independently using its own controls, or all debuggers may execute the same commands as directed by the PDM Control Window.

PDM Controls include command buttons from the Debug Execute menus and a command prompt and output window. Commands that are typed into the command prompt will have output displayed in the PDM output window for each of the projects being debugged.

Figure 10.28 Parallel Debug Manager Control Window


Debug Manager for W:/0_test/NewProject/test_400_g2_g1g2.ws
File
Debug
W:/0_test/NewProject/./400sim.pjt
1050 ms
W:/0_test/NewProject/./g2.pjt
2360 ms
W:/0_test/NewProject/g2sim.pjt
110 ms
run
continue
step
next
finish
until
si
ni
cycle-step
SDBUG Command
Clear

```
(W:/0_test/NewProject/./400sim.pjt sdbug400) tbreak main
#/cygdrive/W/0_test/NewProject/main.c:28:001f:6;
Breakpoint 6 at 0x1f: file /cygdrive/W/0_test/NewProject/main.c, line 28.

(W:/0_test/NewProject/./g2.pjt zspG2) tbreak main
#main.c:28:0028:3;
Breakpoint 3 at 0x28: file main.c, line 28.

(W:/0_test/NewProject/g2sim.pjt zspG2) tbreak main
#main.c:28:0028:3;
Breakpoint 3 at 0x28: file main.c, line 28.
```

10.8 Help Menu

10.9 Editor

The ZSP IDE Editor is a window where you can write your code. It allows basic editing functionality

10.10 ZSP IDE File Formats

ZSP IDE produces a number of files when you create and compile a project or Workspace. These can be categorized as follows

Table 10.4

Extension	Description
.c	C Source file
.S or .s	Assembly source File
.h	Header File
.pjt	Project File
.ws	Workspace File
.exe	Executable file

Chapter 11
ZSP IDE Debugger

This chapter describes how to use the ZSP IDE Debugger, a graphical debugging environment for developers using the ZSP family of Digital Signal Processor Cores.

Version 4.0 of SDK Tools features a new Graphical Interface Integrated Development Environment for ZSP software project management, referred to in this document as ZSP IDE. The debug component of ZSP IDE is the focus of this chapter and is referred to as ZSP IDE Debugger.

ZSP IDE Debugger is a menu-driven user interface to the ZSP Debugger. It provides a user-friendly graphical interface that allows navigation through your code while showing program and processor information for debugging purposes. The ZSP IDE Debugger allows setting breakpoints, examining registers and variables, watching source level variables, examining memory. Commands may be entered to be executed by the Command Line Debugger. The capability to automatically save your current debug settings and restore them at startup allows quick setup for each debugging session.

The ZSP IDE Debugger is an integral component of the ZSP IDE executable (ZSPide.exe). The Debugger is configured and invoked from the IDE Debug Menu to operate on the IDE Current Project. The Debugger component of the IDE may be run independently from the IDE by using a separate executable (guidebug.exe). When running the Debugger in this way, a Project File may be loaded through the Debugger File Menu.

Features of ZSP IDE Debugger –

- Processor Support - ZSP G2 Architecture, ZSP400 Architecture, and G1/G2 (to use ZSP400 source code for processors based on ZSP G2 architecture.)
- Compatibility - Backwards-compatible with Version 3.2 Projects.

- Windows and UNIX (planned) Debugger platforms
- Support for multiple targets
- Processor Register Windows - Operand, Control, Address Registers
- Displays cycle-accurate simulator information, code statistics, code profile, instruction grouping rules
- Concurrent Source and Disassembly level debugging
- 40-Bit Register display
- Multiple sessions may run concurrently
- Command-Line Debugger interface

Underlying Command Line Tools – Behind the ZSP IDE Debugger is a command line interface to a GNU Debugger (sdbug, zdbug, zdxbug) for the ZSP processor. Each currently supported debug target (ZSP Core) uses a separate configuration of the Command Line Debugger.

Table 11.1 Command Line Debugger Executables

Target	**Command Line Debugger**
ZSP400	sdbug400.exe
G2	zdbug.exe
G1G2	zdxbug.exe

Target Interfaces

Table 11.2 Debugger Targets

Simulator targets
Cycle accurate simulator (zsim)
Instruction level simulator (zisim)
Hardware Targets
Corelis PCMCIA based JTAG connector
Corelis PCI based JTAG connector
UART (Serial Port)

ZSP IDE Debugger supports the JTAG hardware target for ZSP400, UART (Serial Port) hardware target for ZSP400, ZISIM instruction-accurate simulator for ZSP400, G2, and G1G2, and ZSIM cycle-accurate simulator for ZSP400, G2, and G1G2.

11.1 GUI Debugger Overview

11.1.1 Main Window

The Main Window comprises a Title Bar, Menu Bar, Tool Bar(s), Status Area, and Debugging Window area in which Debugging Windows may be displayed.

11.1.2 Title Bar - Project File Name Display

When a project is loaded, the name of the project file is displayed in the Main Window Title Bar.

11.1.3 Window Area

Debugging Windows are displayed in the window area in the center of the Main Window. The Main Window configuration adds new Debugging Windows by splitting the available window size into panes that are resized by adjusting the handle on the separator between the windows. Alternatively, Debugging Windows may each be separated from the Main Window (see Section 11.1.7.3, “Top Level Window Presentation,” page 11-8).

11.1.4 Status Area

The Status Area at the bottom of the Main Window shows general information throughout the debugging session, such as the target processor, debug target, executable file name, and debugging status.

11.1.5 Main Menu

The Main Menu provides access to major functions of the debugger such as controlling breakpoints, executing navigation commands, displaying Debugging Windows.

11.1.5.1 Operating the Main Menu

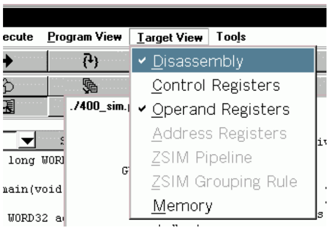
Main menu items may be selected either by left-clicking with the mouse or by typing on the keyboard using menu accelerator keys (Underlined character in the menu name). To invoke the menus from the keyboard, depress the ALT key and the accelerator key for the Main Menu item concurrently. This will display the pull down subitem menu from which you can make further selections without using the ALT key. You may also use the Up, Down, Left, and Right arrow keys to navigate through the menus, terminating your choice with either the Enter key to confirm or the Escape key to cancel your selection.

11.1.5.2 Controlling Debugging Windows Through the Main Menu

Debugging Windows display program and/or debugging target information. Debugging Windows may be selected for viewing through the Main Menu checkbutton menu items.

Debugging Window Menu Checkmarks – When a Debugging Window is displayed, the corresponding Main Menu item displays a checkmark in front of the menu text field

Figure 11.1 Menu Checkmarks For Debugging Windows



11.1.6 Main Toolbars

Toolbars are available as menu shortcuts to provide access to commonly used debugging features.

11.1.6.1 Available toolbars

Toolbars exist for the following areas:

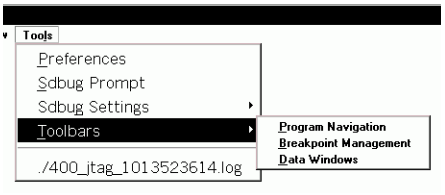
- Program navigation (Execute Menu shortcuts)

- Breakpoints (Breakpoint Menu shortcuts)
- Windows (Debugging Window menu shortcuts)

11.1.6.2 Invoking Toolbars

Select Toolbars from the Tools Menu and select the desired toolbar by name to toggle the display of the toolbar below the menu in the Main Window.

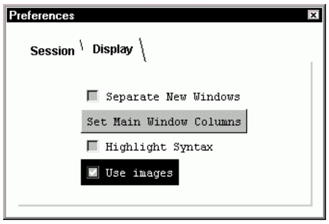
Figure 11.2 Tools Menu - Invoke Toolbars



11.1.6.3 Modifying Toolbar Appearance

Toolbar Buttons may be viewed with text or icon annotation. To view the button annotation as text, select Preferences from the Tools Menu to display the Preferences Window then unselect the “use images” checkbutton. Figure 11.4 and Figure 11.5 illustrate the appearance of the toolbar for each of these annotation modes.

Figure 11.3 Preferences - Use Images For Toolbar Buttons



Each Toolbar Button has a text description that is displayed when the mouse cursor is present on the button providing additional information regarding that button’s functionality.

Figure 11.4 Toolbar Buttons With Text Annotation

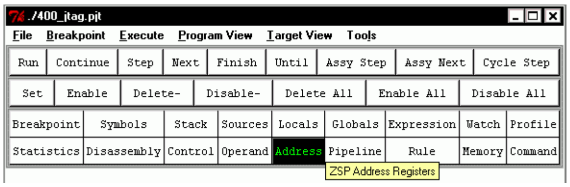
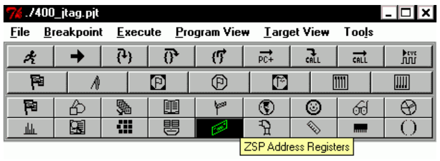

./400_jtag.pjt
File Breakpoint Execute Program View Target View Tools
Run Continue Step Next Finish Until Assy Step Assy Next Cycle Step
Set Enable Delete- Disable- Delete All Enable All Disable All
Breakpoint Symbols Stack Sources Locals Globals Expression Watch Profile
Statistics Disassembly Control Operand Address Pipeline Rule Memory Command
ZSP Address Registers

Figure 11.5 Toolbar Buttons With Image Annotation



11.1.7 Debugging Windows (General)

Debugging Windows comprise the following types (described in detail in later sections):

- C/Assembly Program Windows
  – Source Code
  – Breakpoint List
  – Debugging Symbols
  – Call Stack
  – Local Variables
  – Global Variables
  – Expression
  – Watch
  – ZSIM Statistics
  – ZSIM Profile
- Target system windows
  – Disassembly Code
  – Control Registers

  - Operand Registers
  - Address Registers (G2)
  - Memory
  - ZSIM Grouping Rule
  - ZSIM Pipeline
- Tools Preferences
- Command Line Interface

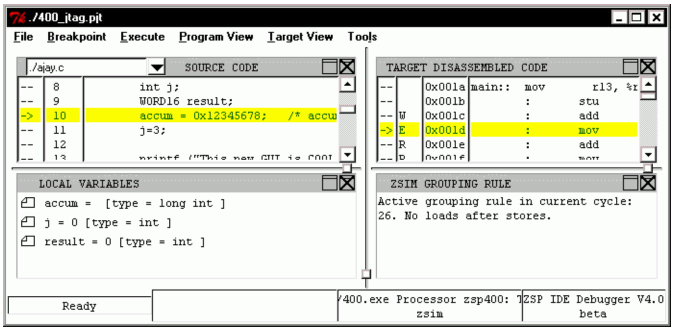
11.1.7.1 Debugging Window Operation

Debugging Windows are displayed by selecting the appropriate menu item from the Main Menu or by selecting the appropriate button from the Window Toolbar. To remove the window from the display, invoke the menu item again to remove the checkmark, close the window by clicking on the “X” icon, or deselect the associated Toolbar Button.

11.1.7.2 Debugging Windows Paned Window Presentation

Debugging Windows are presented by default in a Paned Window view as child windows within the Main Window. In this configuration, all windows appear at the same level, ie. no separate Debugging Windows. Each Debugging Window may be separated from the Paned Window (see Section Debugging Window Top Level Preference - Page 9 and Section Changing Debugging Window View Mode - Page 9)

Figure 11.6 Debugger Paned Window



```
--   8          int j;
--   9          WORD16 result;
->   10         accum = 0x12345678;    /* accu
--   11         j=3;
--   12
```

```
--     0x001a main::  mov      r13, %r
--     0x001b         :        stu
-- W   0x001c         :        add
-> E   0x001d         :        mov
-- R   0x001e         :        add
```

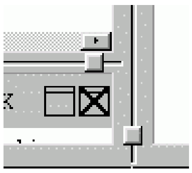
```
accum =  [type = long int ]
j = 0 [type = int ]
result = 0 [type = int ]
```

```
Active grouping rule in current cycle:
26. No loads after stores.
```

```
Ready    /400.exe Processor zsp400: zsim    ZSP IDE Debugger V4.0 beta
```

Paned Window Operation – Windows displayed in the Paned Window may may be resized by dragging the handles of the paned window controls that separate the rows and columns of the Debugging Window area.

Figure 11.7 Paned Window Handles



Resizing columns affects all windows in that column while resizing rows only modifies one window plus its vertical neighbor.

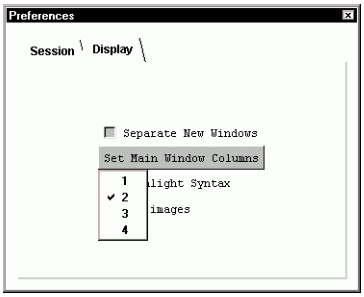
Paned Window Configuration – The presentation of windows in the Paned Window may be configured in 1-4 columns by selecting Preferences from the Tools Menu and “Main Window Columns” from the Preferences Window Display Tab. To change the number of columns displayed during a session,

Step 1. Set the desired number of columns in the preferences panel

Step 2. Save the debugging session (File > Save > Session)

Step 3. Reload the debugging session (File > Load > Session)

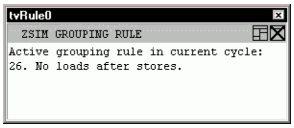
Figure 11.8 Preferences - Set Main Window Columns



11.1.7.3 Top Level Window Presentation

Top Level presentation of a Debugging Window displays that window as a separate Top Level window.

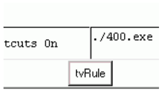
Figure 11.9 Top Level Debugging Window



```
Active grouping rule in current cycle:
26. No loads after stores.
```

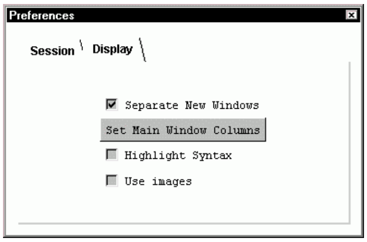
Top Level focus control – Top Level Debugging Windows that are obscured by other graphics on the screen may be brought into focus for viewing by selecting the corresponding Window Button on the toolbar at the bottom of the Paned Window.

Figure 11.10 Top Level Window Focus Control



Debugging Window Top Level Preference – New Debugging Windows may be automatically configured for Top Level presentation by selecting Preferences from the Tools Menu and then selecting the checkbox labeled "Separate New Windows" in the Display Tab of the Preferences Window.

Figure 11.11 Preferences - Separate New Window



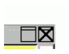
☑ Separate New Windows
☐ Highlight Syntax
☐ Use images

11.1.7.4 Changing Debugging Window View Mode

Each of the Debugging Windows may be changed to and from Top Level or Paned Windows or may be closed by selecting the appropriate window icon at the upper right corner of that Debugging Window’s submenu area.

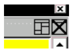
Click the left mouse button on the Window icon to separate the window into a Top Level Window. Click the left mouse button on the “X” icon to close the window.

Figure 11.12 Display Controls for Paned Window



Click the left mouse button on the window icon to join the Top Level Window into the Paned Window. Click the left mouse button on the “X” icon to close the window.

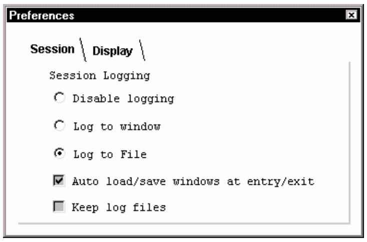
Figure 11.13 Display Controls for Top Level Window



11.1.7.5 Autoload Debugging Windows Preference

When restarting a debugging session, the windows displayed in the previous session may be automatically displayed by selecting Preferences from the Tools Menu then selecting the "Autoload/save windows at entry/exit" checkbox from the Session Tab of the Preferences Window.

Figure 11.14 Preferences - Autoload Windows



Session Logging

( ) Disable logging

( ) Log to window

(●) Log to File

[✓] Auto load/save windows at entry/exit

[ ] Keep log files

Display settings are saved as part of the project data when Autoload is selected. This includes all of the window preferences selections and all of the debugging windows that are open when the debugger is closed.

11.2 Detailed Descriptions

11.2.1 Main Menu

11.2.1.1 File Menu

File operations available through the File Menu include:

- loading and saving debugging sessions
- loading an executable for debugging
- loading and saving memory images
- script recording and playback

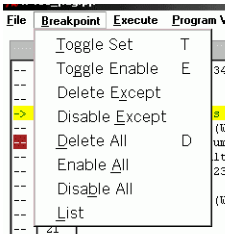
11.2.1.2 Breakpoint Menu

Breakpoints allow program execution to stop at specified code locations so that processor and program information may be examined during debugging. Each line of source or disassembly code may be specified as a Breakpoint. When a Breakpoint is enabled, program execution will be stopped when the line of code is scheduled as the next instruction. When a Breakpoint is disabled, program execution is not stopped at the line but continues past the breakpoint. Breakpoint Operations available through the Breakpoint Menu include:

- toggling breakpoints at the currently selected source line
- enabling and disabling a breakpoint at the currently selected source line.
- disabling or deleting breakpoints at all except the currently selected line
- deleting, enabling, or disabling all breakpoints
- toggling display of the breakpoint listing window

Breakpoints are indicated in the Source and Disassembly Windows in the left-most column of the window. An Enabled Breakpoint is indicated by a red highlight in this area of the line. A Disabled Breakpoint is indicated by a gray highlight.

Figure 11.15 Breakpoint Menu



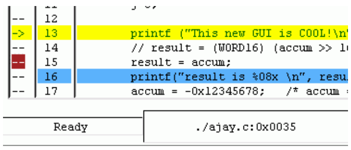
Current Selection Line – When setting a breakpoint from the breakpoint menu, the breakpoint will be set at the Current Selection Line.

At the completion of each program navigation step (eg breakpoint reached, single step executed, etc) the Current Selection Line is the highlighted program line.

The Current Selection Line for Breakpoint Operations may be set in either the Source Window or Disassembly Window. Left-click the mouse with the mouse pointer over the desired line and that line will become the Current Selection Line. Alternatively, you may use the up and down arrow keys to select the previous or next line of code as the Current Selection Line.

When the Current Selection Line is selected with the mouse or keyboard, the address of the Current Selection Line is displayed in the status bar at the bottom of the Paned Window, the appropriate line/lines is/are highlighted in both the Source Code and Disassembly Windows, and subsequent Breakpoint Operations will be applied to that line.

Figure 11.16 Source Code Window Current Selection Line



```
printf ("This new GUI is COOL!\n'
// result = (WORD16) (accum >> l
result = accum;
printf("result is %08x \n", resu
accum = -0x12345678;   /* accum
```

Breakpoint Toolbar (Menu alternative) – Each of the breakpoint functions except the listing is available from a toolbar that is displayed in the Main Window. To display the Breakpoint Toolbar select Toolbars from the Tools Menu and then select Breakpoint Management from the Toolbars cascade menu.

Toolbar settings are saved and restored for each debugging session when “auto load/save windows at entry/exit” is selected in Debugging Preferences.

Breakpoint Menu –

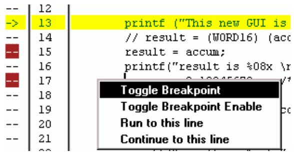
Toggle Set
When a breakpoint is 'Toggle Set' by the "Toggle Set" menu choice, the debugger checks for the existence of a breakpoint at the current line. If a breakpoint exists, it is deleted. If a breakpoint does not exist, then one is created at the current line.

Alternatives to Breakpoint Menu 'Toggle Set':

- Source and Disassembly Window Popup Menus "Toggle Breakpoint"
- Source and Disassembly Window Breakpoint Area (left-most column of the window) left-click
- Keyboard Shortcut "T or t"

Example of Source Code Window breakpoint controls and displays

Figure 11.17 Source Code Window Breakpoints



```
--
->      printf ("This new GUI is
--      // result = (WORD16) (acc
--      result = accum;
--      printf("result is %08x \r
--
--
--
--
--
```

Toggle Enable
When a breakpoint is 'Toggle Enabled' by the "Toggle Enable" menu choice, the debugger checks for the existence of a breakpoint at the current line. If a breakpoint does not exist, then one is created at the current line and enabled. If a breakpoint exists, the debugger checks for the enabled state of the breakpoint. If it is enabled then the breakpoint is set to disabled and vice-versa.

Alternatives to Breakpoint Menu 'Toggle Enable'

- Source and Disassembly Window Popup menus "Toggle Breakpoint"
- Keyboard Shortcut "E or e"

Delete Except
Selecting "Delete Except" from the Breakpoint Menu causes all breakpoints to be deleted except at the current line. If no breakpoint exists at the current line, then a breakpoint at the current line is created.

Disable Except
Selecting "Disable Except" from the Breakpoint Menu causes all breakpoints to be disabled except at the current line. If no breakpoint exists at the current line, then a breakpoint at the current line is created.

Delete All
Selecting "Delete All" from the Breakpoint Menu causes all breakpoints to be deleted.

Enable All
Selecting "Enable All" from the Breakpoint Menu causes all existing breakpoints to be enabled.

Disable All
Selecting "Disable All" from the Breakpoint Menu causes all existing breakpoints to be disabled.

List
Selecting "List" from the Breakpoint Menu displays a Debugging Window showing details of breakpoints currently set.

Note: For details about the Breakpoint List Window, see Section , “Breakpoint List,” page 11-22

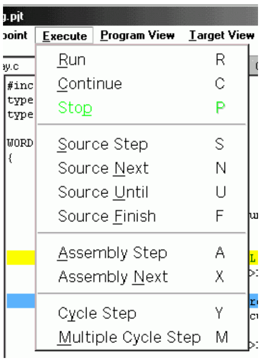
11.2.1.3 Execute Menu

The Execute Menu provides access to commonly used navigation features for debugging.

- Run
- Continue
- Stop
- Source Step
- Source Next
- Source Until

- Source Finish
- Assembly Step
- Assembly Next
- Cycle Step
- Multiple Cycle Step

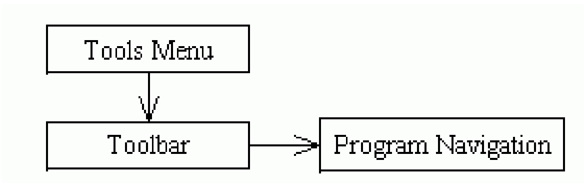
Figure 11.18 Execute Menu



Alternative to execute menu for execute functions – Additional means of navigation are:

- Program Navigation Toolbar
- Keyboard shortcut keys
- Popup menu on source and disassembly Debugging Windows

Program Navigation Toolbar
Each of the execute functions is available from a toolbar that is invoked from the Tools Menu. To turn on the Program Navigation Toolbar, select:



Keyboard Shortcut Keys
Keyboard shortcut keys allow single-keystroke navigation through program execution**.**

Table 11.3 Keyboard Shortcuts

Key	**Action**
F2 R r	Run
F3 C c	Continue
F4 S s	Step
F5 N n	Next
F6 A a	Assembly Step
F7 X x	Assembly Next
I i	finish
U u	Until
O o	stop
Y y	Cycle-Step
M m	Multiple Cycle-Step

Popup Execution Functions
Selecting a source or disassembly line and using the right-click popup menu allows run or continue to that line.

Execute Menu Functions –

Run
Run causes the program to be run from the start.

Continue
Continue causes the program to be run from the current position.

Step
Step causes the program to advance from the current source position to the next source line for which debugging information exists. If the source file does not exist, the Disassembly Window will be invoked for navigation through the debug execution steps. If the current source is assembly

code then Step advances by one assembly instruction, stepping info function calls.

Next
Next causes the program to advance from the current source position to the next source line. If the current source position is a function call then the function is stepped over. Otherwise the behavior is the same as Step. If the current source is assembly code then Next advances by one assembly instruction, stepping over function calls.

Assembly Step
Assembly step advances program execution by an assembly-level instruction. Assembly step will follow calls to step into functions.

Assembly Next
Assembly next advances program execution by an assembly-level instruction. Assembly next will step over calls and will not step into functions.

Finish
Finish completes execution of a function and returns to the line following the function call.

Until
Until continues running until a source line past the current line in the current stack frame is reached.

Stop
Stop causes a dynamic breakpoint to be executed in a running program. Program execution is halted and current state of the program and processor is reflected in the Debugging Windows.

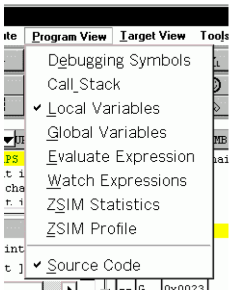
Cycle-Step
Cycle-step advances program execution by one processor clock cycle. Cycle-step is available for the ZSIM simulator target only. Depending on instruction grouping, more than one assembly instruction may be executed in a Cycle-Step.

Multiple Cycle-Step
Multiple Cycle-step advances program execution by a user defined number of processor clock cycles. Multiple Cycle-step is available for the ZSIM simulator target only.

11.2.1.4 Program View Menu

The Program View Menu controls program-related windows. To display a window, select it from the menu. When the window is displayed, a checkmark is placed next to the window description. See Section 11.2.2.1, "C/Assembly Program Windows," page 11-19 for detailed window information.

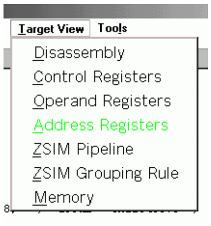
Figure 11.19 Program View Menu



11.2.1.5 Target View Menu

The Target View Menu controls target hardware-related windows. To display a window, select it from the menu. When the window is displayed, a checkmark is placed next to the window description. See Section 11.2.2.2, "Target Windows," page 11-27 for detailed window information.

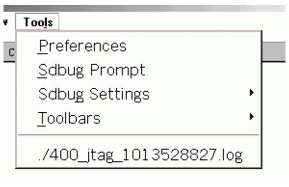
Figure 11.20 Target View Menu



11.2.1.6 Tools Menu

The Tools Menu provides customization of views for each project, access to a Command Line Debugger Interface, display of target settings, selection of toolbars to be displayed in the Main Window, and log file display. See Section 11.2.2.3, “Tools Windows and Functions,” page 11-33 for more information on the Tools Menu items.

Figure 11.21 Tools Menu



11.2.1.7 Help Menu

The Help Menu provides help.

11.2.2 Debugging Window Detailed Descriptions

11.2.2.1 C/Assembly Program Windows

Available from the Program View menu or from the Window Toolbar, the Program Windows display data pertinent to execution of a program. Available Program Windows include:

- Source Code Window
- Breakpoint List Window
- Debugging Symbols Window
- Call Stack Window
- Local Variables Window
- Global Variables Window
- Expression Window
- Watch Window
- ZSIM Profile Window
- ZSIM Statistics Window

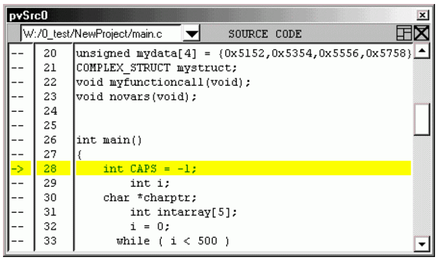
Source Code Window – The Source Code window displays the program source files for debugging. The location of the program source files are obtained from the debugging information in the loaded executable. Additional directories may be searched for source files by using the Working Directories specification in the Project Settings dialog of the IDE.

Accessing Source Code Window
The Source Code Window is accessible through the Program View Menu by selecting “Source Code”.

Program execution tracking
Tracking of program execution is visible through the Source and Disassembly Windows. The Current Line is highlighed as the next instruction to be executed.

Source Code Window Display
The Source Code Window is populated based on information reported from the Command Line Debugger. When the Source Code Window is created, all source files known to the Command Line Debugger are inserted into the file selection pulldown box when the Source Code Window is created. The content of the source files are read from their files and displayed in the Source Code Window either when you select the file for viewing from the file selection pulldown box or when program execution enters that source code file.

Figure 11.22 Source Code Window



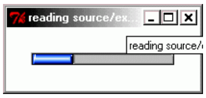
W:/0_test/NewProject/main.c

```
--  unsigned mydata[4] = {0x5152,0x5354,0x5556,0x5758}
--  COMPLEX_STRUCT mystruct;
--  void myfunctioncall(void);
--  void novars(void);
--
--
--  int main()
--  {
->      int CAPS = -1;
--          int i;
--      char *charptr;
--          int intarray[5];
--          i = 0;
--        while ( i < 500 )
```

Source Code Window Syntax Highlighting
If the project preferences indicate that syntax highlighting is desired, each file will be highlighted at creation.

Source Code Window progress bar
While source file loading or highlighting is in progress, a progress bar is displayed to inform the user of the status of the operation. If the source file is a Top Level window then the progress bar is also displayed as a Top Level window, otherwise the progress bar is displayed in the Main Window status area.

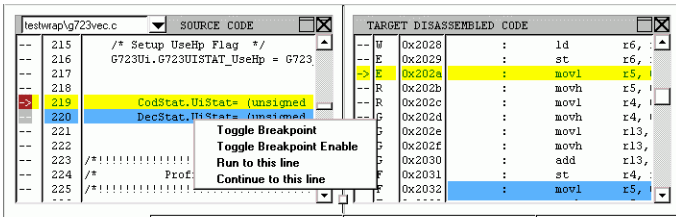
Figure 11.23   Progress Bar Window



Source Code Window Components

The Source Code Window contains columns for breakpoint information, pipeline stage (ZSIM target only) line number, and source code text. The window submenu contains a source file listing drop-down box in the Source Code Window Menu. The source file drop-down box lists all of the source files known to the Command Line Debugger.

Figure 11.24   Source Code Window (shown with Disassembly Window)



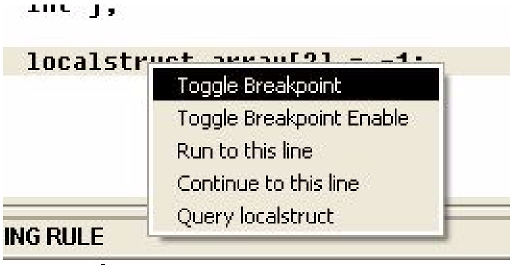
```
--   215      /* Setup UseHp Flag  */
--   216      G723Ui.G723UISTAT_UseHp = G723
--   217
--   218
->   219           CodStat.UiStat= (unsigned
--   220           DecStat.UiStat= (unsigned
--   221
--   222
--   223  /*!!!!!!!!!!!!!!
--   224  /*            Prof
--   225  /*!!!!!!!!!!!!!!
```

```
-- W  0x2028    :    ld     r6,
-- E  0x2029    :    st     r6,
-> E  0x202a    :    movl   r5,
-- R  0x202b    :    movh   r5,
-- R  0x202c    :    movl   r4,
-- G  0x202d    :    movh   r4,
   G  0x202e    :    movl   r13,
   G  0x202f    :    movh   r13,
   G  0x2030    :    add    r13,
   F  0x2031    :    st     r4,
   F  0x2032    :    movl   r5,
```

- Source Code Window Breakpoint Area
  The breakpoint area shows enabled breakpoints in red, disabled breakpoints in gray and the current line is indicated by an ascii arrow.
- Source Code Window Line Highlighting
  The Source Code Window has two important items highlighted for user information, one being the Current Program Execution Line and the other being the User-Selected Line.
  – Source Code Window Current Execution Line
    Current Program Execution Line - indicated by a highlighted background on the code and line number areas.
  – Source Code Window User-Selection Line
    User-Selection Line - indicated by a blue band over the line selected. The line may be selected by clicking the left mouse button on the desired line. The line may also be selected by using the keyboard up and down arrow keys. The source code filename and instruction address are displayed in the Main Window status bar when the user selects a line. If the 'Target View > Disassembly' Window is displayed when the user selects

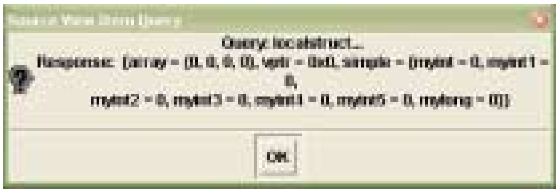
a source code line, then the associated disassembly lines are also marked with the same color blue band and brought into view.

- Source Code Window Popup Menu
  The popup menu for the source code or Disassembly Window is invoked by right-clicking the mouse over the code area. The popup menu allows you to toggle a breakpoint or breakpoint enable at the selected line, run from start to the selected line, or continue from the current execution point to the selected line. Run and continue to the selected line is implemented by saving the breakpoints, setting a break at the selected line and then executing run or continue as specified.



The source code window popup menu also allows a command-line debugger query to be performed using the word beneath the mouse pointer as the query expression.

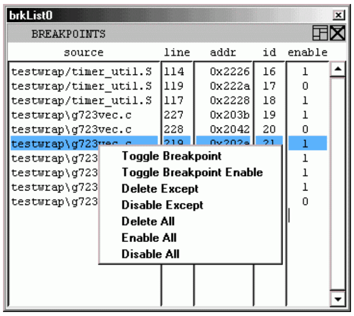
Figure 11.25 Example Source Code Popup Query Result



```
Query: localstruct...
Response: {array = {0, 0, 0, 0}, vptr = 0x0, simple = {myint = 0, myint1 = 0,
myint2 = 0, myint3 = 0, myint4 = 0, myint5 = 0, mylong = 0}}
```

Breakpoint List – Selecting "Breakpoint -> List" from the Main Window causes a Debugging Window window to be displayed showing details of breakpoints currently set.

Figure 11.26 Breakpoint List Window



source	line	addr	id	enable
testwrap/timer_util.S	114	0x2226	16	1
testwrap/timer_util.S	119	0x222a	17	0
testwrap/timer_util.S	117	0x2228	18	1
testwrap\g723vec.c	227	0x203b	19	1
testwrap\g723vec.c	228	0x2042	20	0
testwrap\g723vec.c	219	0x202a	21	1
testwrap\g723				1
testwrap\g723				1
testwrap\g723				1
testwrap\g723				0

For each existing breakpoint, the breakpoint list shows:

- source code file name
- source code file line number
- instruction address
- command line debugger's breakpoint identification number
- breakpoint enable state

Selecting a Breakpoint Line
Left-click on a line in the breakpoint list to select that breakpoint as the current line for Breakpoint Operations.

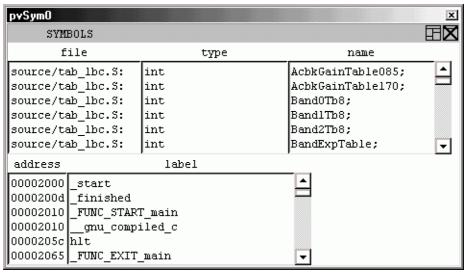
Actions on Selecting a Breakpoint Line
When a breakpoint line is selected from the list, if the Source Code and/or Disassembly Windows are shown, the breakpoint line is highlighted and brought into focus in these windows.

Operations Available for a Selected Breakpoint Line
Right-click on a line in the breakpoint list to display a popup menu of breakpoint operations that may be applied to the selected Breakpoint.

Saving of Breakpoints
Breakpoints are saved and restored with the project session when Autoload is selcted from the Preferences Window.

Debugging Symbols Window – Debugging Symbols are available for browsing using the Debugging Symbols Window. Two types of information are presented, program data symbols and program instruction symbols.

Figure 11.27 Debugging Symbols Window



pvSym0

SYMBOLS

file	type	name
source/tab_lbc.S:	int	AcbkGainTable085;
source/tab_lbc.S:	int	AcbkGainTable170;
source/tab_lbc.S:	int	Band0Tb8;
source/tab_lbc.S:	int	Band1Tb8;
source/tab_lbc.S:	int	Band2Tb8;
source/tab_lbc.S:	int	BandExpTable;

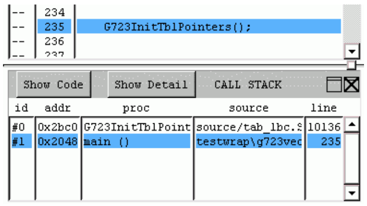
address	label
00002000	_start
0000200d	_finished
00002010	_FUNC_START_main
00002010	__gnu_compiled_c
0000205c	hlt
00002065	_FUNC_EXIT_main

- Program Data Symbols
  The Symbols Window lists variables that are global, indicates the source file in which they are defined and the data type associated with the variable.
- Program Instruction Symbols
  The Symbols Window lists instruction labels for the program being debugged and the associated addresses.

The Debugging Symbols Window is only populated when it is invoked, since it will not change within the debugging session.

Call Stack – To display a program's Call Stack, select Call Stack from the Program View Menu.

Figure 11.28 Call Stack Window



```
--  234
--  235   G723InitTblPointers();
--  236
--  237
```

id	addr	proc	source	line
#0	0x2bc0	G723InitTblPoint	source/tab_lbc.S	10136
#1	0x2048	main ()	testwrap\g723vec	235

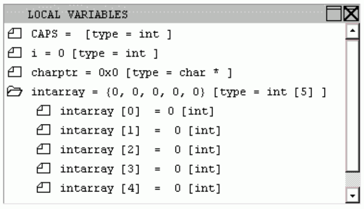
Call Stack Code Viewing
To view the code associated with one of the stack levels displayed, select that line in the Stack Window and select the Show Code button. The Source and Disassembly Windows will display the associated code.

Call Stack Details Popup
The Show Detail on the Stack Window menu shows details in a popup window so that information exceeding the display area may be easily examined. The detailed information includes the Stack Level, Address,

Procedure (name and arguments), Source File name, Source File line number.

Local Variables – To display local variables, select Local Variables from the Program View Menu. The Local Variables Window shows all variables that are in the local scope.

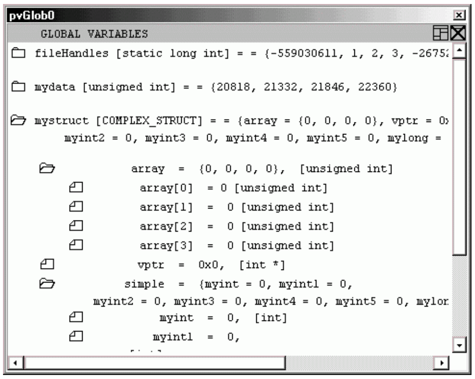
Figure 11.29   Local Variables Window



```
LOCAL VARIABLES
CAPS =  [type = int ]
i = 0 [type = int ]
charptr = 0x0 [type = char * ]
intarray = {0, 0, 0, 0, 0} [type = int [5] ]
    intarray [0]  = 0 [int]
    intarray [1]  =  0 [int]
    intarray [2]  =  0 [int]
    intarray [3]  =  0 [int]
    intarray [4]  =  0 [int]
```

Global Variables – A view of global variables is available from the Main Menu by selecting 'Program View > Globals'. The Global Variables Window shows all variables that are global in scope.

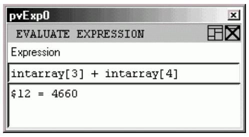
Figure 11.30   Global Variables Window



```
pvGlob0
GLOBAL VARIABLES
fileHandles [static long int] = = {-559030611, 1, 2, 3, -2675

mydata [unsigned int] = = {20818, 21332, 21846, 22360}

mystruct [COMPLEX_STRUCT] = = {array = {0, 0, 0, 0}, vptr = 0x
     myint2 = 0, myint3 = 0, myint4 = 0, myint5 = 0, mylong =

              array  =  {0, 0, 0, 0},  [unsigned int]
                 array[0]  = 0 [unsigned int]
                 array[1]  =  0 [unsigned int]
                 array[2]  =  0 [unsigned int]
                 array[3]  =  0 [unsigned int]
                vptr  =  0x0,  [int *]
              simple  =  {myint = 0, myint1 = 0,
          myint2 = 0, myint3 = 0, myint4 = 0, myint5 = 0, mylon
                    myint  =  0,  [int]
                   myint1  =  0,
```

Expression – To have the debugger evaluate and display a single expression at each display refresh, use the Evaluate Expression Window. To invoke the Evaluate Expression Window, select Evaluate Expression from the Program View Menu. Type the expression you would like evaluated into the entry area and expression will be evaluated and displayed after each execution step

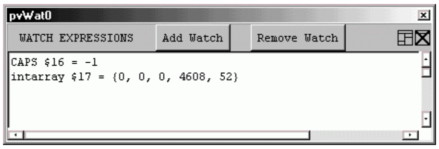
Figure 11.31 .Expression Window



```
intarray[3] + intarray[4]
$12 = 4660
```

Watch – To have the debugger evaluate and display multiple expressions at each display refresh, use the Watch Expression Window. To invoke the Watch Expression Window, select Watch Expression from the Program View Menu. Add expressions to watch using the Add Watch button in the Watch Expression Window. Remove expressions from the Watch Expression Window by selecting the existing expression and selecting the Remove Watch button.

Figure 11.32 Watch Expressions Window



```
CAPS $16 = -1
intarray $17 = {0, 0, 0, 4608, 52}
```

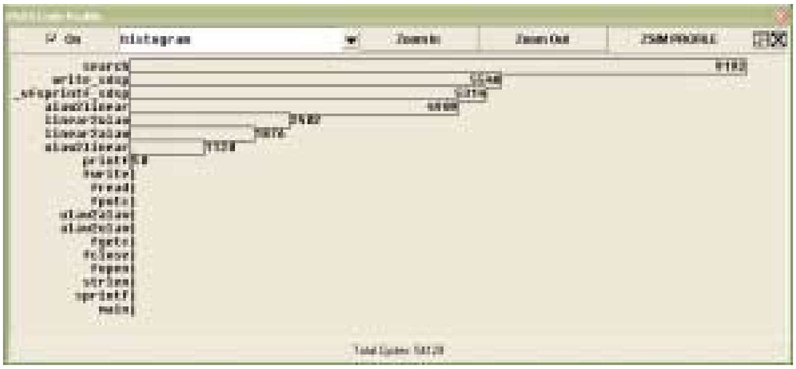
ZSIM Target Windows – ZSIM Debugging windows are available when ZSIM is selected as the target in the IDE Debug>Setup window.

- ZSIM Profile

  A view of the code execution profile is available for the ZSIM target by selecting Profile from the Program View Menu. The menubar of the Profile Window includes a checkbutton to turn function profiling off and on and a checkbutton to select incremental mode, which shows only the functions executed since the last navigational step.

  The Profile Window shows each function name that is available for profiling, the histogram, cumulative and calls information reported by ZSIM. A bargraph chart is displayed with data type selectable from a drop-down selection box.

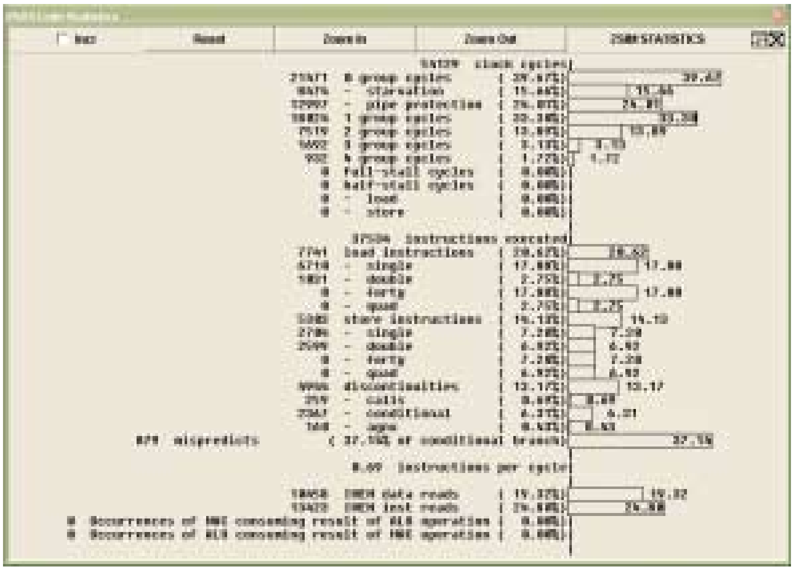
Figure 11.33 ZSIM Profile Window



- ZSIM Statistics

  A view of code execution statistics is available for the ZSIM target by selecting Statistics from the Program View Menu.

Figure 11.34 ZSIM Statistics Window



```
                              54129  clock cycles
           21471  0 group cycles      [ 39.67%]
            8476  -  starvation       [ 15.66%]
           12997  -  pipe protection  [ 24.01%]
           18026  1 group cycles      [ 33.30%]
            7519  2 group cycles      [ 13.89%]
            1682  3 group cycles      [  3.11%]
             932  4 group cycles      [  1.72%]
               0  full-stall cycles   [  0.00%]
               0  half-stall cycles   [  0.00%]
               0  -  load             [  0.00%]
               0  -  store            [  0.00%]

             37506  instructions executed
            7741  load instructions   [ 20.63%]
            6710  -  single           [ 17.88%]
            1031  -  double           [  2.75%]
               0  -  forty            [ 17.88%]
               0  -  quad             [  2.75%]
            5302  store instructions  [ 14.13%]
            2706  -  single           [  7.20%]
            2596  -  double           [  6.92%]
               0  -  forty            [  7.20%]
               0  -  quad             [  6.92%]
            4940  discontinuities     [ 13.17%]
             259  -  calls            [  0.69%]
            2367  -  conditional      [  6.31%]
             160  -  sgos             [  0.43%]
     879  mispredicts        ( 37.14% of conditional branch)

              0.69  instructions per cycle

           10458  DMEM data reads     [ 19.32%]
           13423  DMEM inst reads     [ 24.80%]
  0  Occurrences of MAC consuming result of ALU operation [  0.00%]
  0  Occurrences of ALU consuming result of MAC operation [  0.00%]
```

11.2.2.2 Target Windows

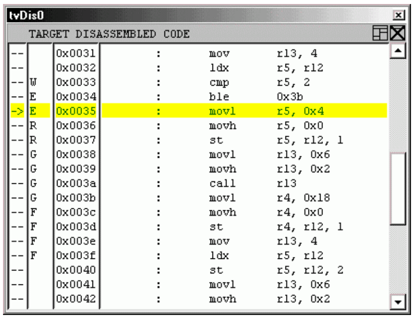
Available from the Target View Menu or from the Window Toolbar, the Target Windows display data pertinent to the state of the processor after each navigational step in the debug session. Available Target Windows include

- Disassembly Code Window
- Control Registers Window
- Operand Registers Window
- Address Registers Window (G2 only)
- Memory Window
- ZSIM Pipeline Window
- ZSIM Grouping Rule Window

Disassembly Window – The Disassembly Window shows disassembled instructions from the target's program memory. The address range of the Disassembly Window includes all instructions in the current scope. As execution proceeds, the Disassembly Window is repopulated as necessary.

The Disassembly Window comprises left to right, a Breakpoint column, pipeline stage column (for ZSIM target only), address column, and disassembled code. The next line to execute is indicated by an ASCII-styled arrow in the breakpoint column.

Figure 11.35   Disassembly Window

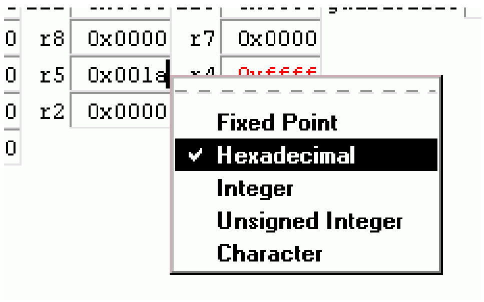


```
--    0x0031    :    mov     r13, 4
--    0x0032    :    ldx     r5, r12
-- W  0x0033    :    cmp     r5, 2
-- E  0x0034    :    ble     0x3b
-> E  0x0035    :    movl    r5, 0x4
-- R  0x0036    :    movh    r5, 0x0
-- R  0x0037    :    st      r5, r12, 1
-- G  0x0038    :    movl    r13, 0x6
-- G  0x0039    :    movh    r13, 0x2
-- G  0x003a    :    call    r13
-- G  0x003b    :    movl    r4, 0x18
-- F  0x003c    :    movh    r4, 0x0
-- F  0x003d    :    st      r4, r12, 1
-- F  0x003e    :    mov     r13, 4
-- F  0x003f    :    ldx     r5, r12
--    0x0040    :    st      r5, r12, 2
--    0x0041    :    movl    r13, 0x6
--    0x0042    :    movh    r13, 0x2
```

Register Window General Description

Three types of register windows, Control Register Window, Operand Register Window, and Address Register Window (G2 Only) are available to display and modify the processor registers. These windows have similar functionality. Each item in a Register Window may be edited by left-clicking in the item to set the input focus, typing in the desired value followed by depressing the enter key. The new value is sent to the Debugger when the enter key is pressed. The Register Window is then refreshed to validate the entry. Each item in the Register Window may be formatted independently of the other items by right-clicking on the item to invoke the popup format menu.

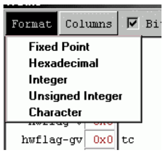
Figure 11.36 Register Element Popup Format Menu



Register Windows each contain a subwindow menu that includes the following functions.

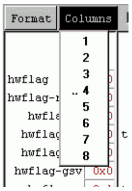
- Format
  The Format Menu in a Register Window allows reformatting of data for all of the visible registers to one of the following formats:
  - Fixed Point (for 16, 32, or 40-bit numbers)
  - Hexadecimal
  - Integer
  - Unsigned Integer
  - ASCII Character

Figure 11.37 Register Window Format Menu



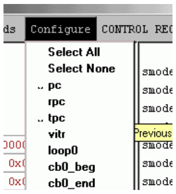
- Columns
  The Columns Menu in the Register Window allows arrangement of the individual registers in the Window into 1-8 columns.

Figure 11.38 Register Window Columns Menu



- Configure
  The Configure Menu item in the Register Window allows selection of individual registers to be displayed in the window by selecting them from a list.

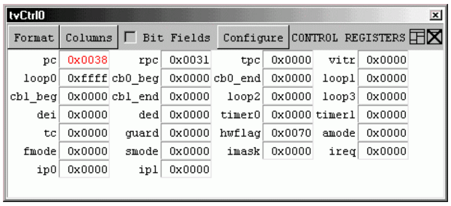
Figure 11.39 Register Window Configure Menu



Control Registers –

The Control Registers Window provides access to the target processor's control registers.

Figure 11.40 Control Register Window - Standard Mode



pc	0x0038	rpc	0x0031	tpc	0x0000	vitr	0x0000
loop0	0xffff	cb0_beg	0x0000	cb0_end	0x0000	loop1	0x0000
cb1_beg	0x0000	cb1_end	0x0000	loop2	0x0000	loop3	0x0000
dei	0x0000	ded	0x0000	timer0	0x0000	timer1	0x0000
tc	0x0000	guard	0x0000	hwflag	0x0070	amode	0x0000
fmode	0x0000	smode	0x0000	imask	0x0000	ireq	0x0000
ip0	0x0000	ip1	0x0000				

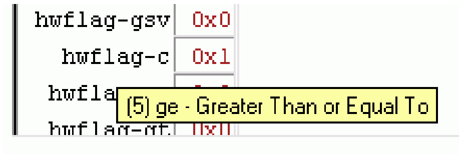
In addition to the common Register View submenu items, the Control Register Window also provides examination and modification capabilities for individual bit fields within each of the Control Registers. The individual bit fields may be edited in the same manner as described in the general Register Window description above.

- Bit Fields
  The Bit Fields checkbox menu item in the Control Register Submenu

Window turns on the display of individual bitfields for the visible control registers.

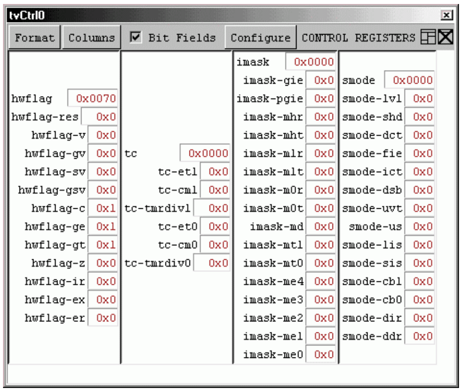
Each of the Control Register and Bit Field entries displayed in the Control Register Window is labeled with a mnemonic abbreviations of the register name. The full name and bit position(s) if appropriate are displayed in a popup text box when you move the mouse pointer over the entry or label.

Figure 11.41 Control Register Bitfield Entry Annotation



```
hwflag-gsv  0x0
  hwflag-c  0x1
(5) ge - Greater Than or Equal To
```

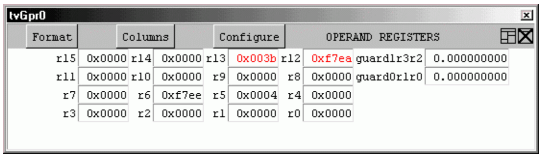
Figure 11.42 Control Register Window - Bitfield Mode



```
                                                imask       0x0000
                                                 imask-gie  0x0   smode       0x0000
hwflag      0x0070                              imask-pgie  0x0   smode-lvl   0x0
hwflag-res  0x0                                  imask-mhr  0x0   smode-shd   0x0
  hwflag-v  0x0                                  imask-mht  0x0   smode-dct   0x0
 hwflag-gv  0x0   tc           0x0000            imask-mlr  0x0   smode-fie   0x0
 hwflag-sv  0x0          tc-etl  0x0             imask-mlt  0x0   smode-ict   0x0
hwflag-gsv  0x0          tc-cml  0x0             imask-m0r  0x0   smode-dsb   0x0
  hwflag-c  0x1   tc-tmrdivl   0x0               imask-m0t  0x0   smode-uvt   0x0
 hwflag-ge  0x1          tc-et0  0x0              imask-md  0x0    smode-us   0x0
 hwflag-gt  0x1          tc-cm0  0x0             imask-mtl  0x0   smode-lis   0x0
  hwflag-z  0x0   tc-tmrdiv0   0x0               imask-mt0  0x0   smode-sis   0x0
 hwflag-ir  0x0                                  imask-me4  0x0   smode-cbl   0x0
 hwflag-ex  0x0                                  imask-me3  0x0   smode-cb0   0x0
 hwflag-er  0x0                                  imask-me2  0x0   smode-dir   0x0
                                                 imask-mel  0x0   smode-ddr   0x0
                                                 imask-me0  0x0
```

Operand Registers – Operand Registers Window provides access to the target processor's operand registers. Menu items in the operand register Window include Format, Columns, and Configure functionality described above in the general Register Window description.

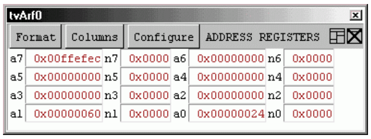
Figure 11.43 Operand Register Window



r15	0x0000	r14	0x0000	r13	0x003b	r12	0xf7ea	guard1r3r2	0.000000000
r11	0x0000	r10	0x0000	r9	0x0000	r8	0x0000	guard0r1r0	0.000000000
r7	0x0000	r6	0xf7ee	r5	0x0004	r4	0x0000		
r3	0x0000	r2	0x0000	r1	0x0000	r0	0x0000		

Address Registers (G2) – Address Registers Window provides access to the target processor's address and index registers. Menu items in the operand register Window include Format, Columns, and Configure functionality described above in the general Register Window description.

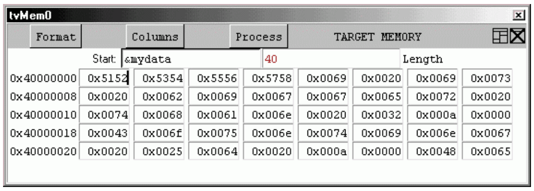
Figure 11.44 Address Register Window



a7	0x00ffefec	n7	0x0000	a6	0x00000000	n6	0x0000
a5	0x00000000	n5	0x0000	a4	0x00000000	n4	0x0000
a3	0x00000000	n3	0x0000	a2	0x00000000	n2	0x0000
a1	0x00000060	n1	0x0000	a0	0x00000024	n0	0x0000

Memory – The Memory Window provides access to the target processor's memory. Menu items in the memory Window include Format and Columns functionality described above in the general Register Window description. except that memory may displayed in up to 16 columns.

Figure 11.45 Memory Window



Start: &mydata 40 Length

0x40000000	0x5152	0x5354	0x5556	0x5758	0x0069	0x0020	0x0069	0x0073
0x40000008	0x0020	0x0062	0x0069	0x0067	0x0067	0x0065	0x0072	0x0020
0x40000010	0x0074	0x0068	0x0061	0x006e	0x0020	0x0032	0x000a	0x0000
0x40000018	0x0043	0x006f	0x0075	0x006e	0x0074	0x0069	0x006e	0x0067
0x40000020	0x0020	0x0025	0x0064	0x0020	0x000a	0x0000	0x0048	0x0065

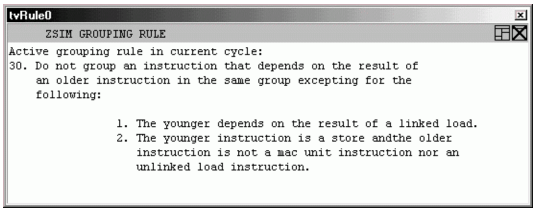
Start address for the memory Window may be an address or debugging symbol.

ZSIM Target Windows – ZSIM Debugging Windows are available when ZSIM is the current debugging target.

- Grouping Rule Window

The Grouping Rule Window displays ZSIM instruction grouping information. The rule displayed applies to instructions currently in the grouping stage at the pipeline.

Figure 11.46 ZSIM Grouping Rule Window

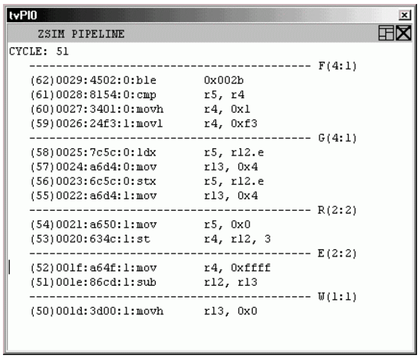


```
Active grouping rule in current cycle:
30. Do not group an instruction that depends on the result of
    an older instruction in the same group excepting for the
    following:

                1. The younger depends on the result of a linked load.
                2. The younger instruction is a store andthe older
                   instruction is not a mac unit instruction nor an
                   unlinked load instruction.
```

- ZSIM Pipeline Window

  The ZSIM Pipeline Window displays ZSIM pipeline information.

Figure 11.47 ZSIM Pipeline Window



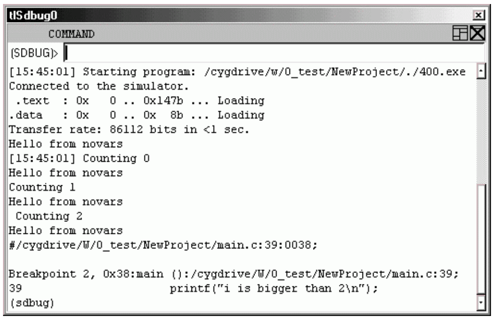
```
CYCLE: 51
   ------------------------------------------ F(4:1)
   (62)0029:4502:0:ble       0x002b
   (61)0028:8154:0:cmp       r5, r4
   (60)0027:3401:0:movh      r4, 0x1
   (59)0026:24f3:1:movl      r4, 0xf3
   ------------------------------------------ G(4:1)
   (58)0025:7c5c:0:ldx       r5, r12.e
   (57)0024:a6d4:0:mov       r13, 0x4
   (56)0023:6c5c:0:stx       r5, r12.e
   (55)0022:a6d4:1:mov       r13, 0x4
   ------------------------------------------ R(2:2)
   (54)0021:a650:1:mov       r5, 0x0
   (53)0020:634c:1:st        r4, r12, 3
   ------------------------------------------ E(2:2)
   (52)001f:a64f:1:mov       r4, 0xffff
   (51)001e:86cd:1:sub       r12, r13
   ------------------------------------------ W(1:1)
   (50)001d:3d00:1:movh      r13, 0x0
```

11.2.2.3 Tools Windows and Functions

Preferences Window – The Preferences Window provides customization of your project session preferences

Command Line Debugger Window – The Command Line Debugger Window provides direct access to the Command Line Debugger. Commands entered in the command entry box are passed to the Command Line Debugger and the response from each command is presented in the output window.

Figure 11.48 Command Line Window

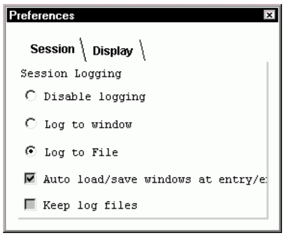


```
(SDBUG)>
[15:45:01] Starting program: /cygdrive/w/0_test/NewProject/./400.exe
Connected to the simulator.
 .text  : 0x   0 .. 0x147b ... Loading
.data   : 0x   0 .. 0x  8b ... Loading
Transfer rate: 86112 bits in <1 sec.
Hello from novars
[15:45:01] Counting 0
Hello from novars
Counting 1
Hello from novars
 Counting 2
Hello from novars
#/cygdrive/W/0_test/NewProject/main.c:39:0038;

Breakpoint 2, 0x38:main ():/cygdrive/W/0_test/NewProject/main.c:39;
39                    printf("i is bigger than 2\n");
(sdbug)
```

11.2.2.4 Using Session Logging

The Session Logging functionality of the ZSP IDE debugger captures communications with the underlying Command Line Debugger for informational purposes. To configure Session Logging, open the Preferences Window by Selecting “Preferences” from the Tools Menu.

Figure 11.49 Preferences Window - Logging



Session Logging

- ( ) Disable logging
- ( ) Log to window
- (●) Log to File
- [✓] Auto load/save windows at entry/e
- [ ] Keep log files

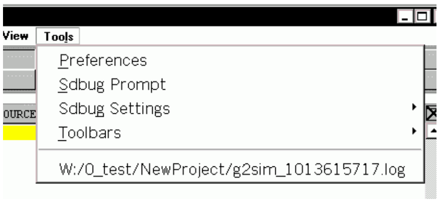
Session Log Types – The Session Log may be disabled by selecting the radiobutton labeled “Disable logging” in the Preferences Window. This setting is recommended for the best speed performance of the debugging environment.

The Session Log may be directed to a window by selecting the radiobutton labeled “Log to Window” in the Preferences Window. Logging to a window provides continuous non-interactive update throughout the debugging session. Logging to a window is faster than logging to a file. There is no permanant Session Log record when logging to a window.

The Session Log may be directed to a file for a permanent Session Log record by selecting the radiobutton labeled “Log to file” in the

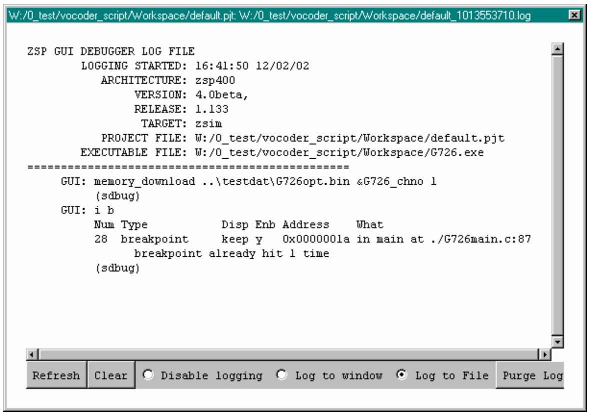
Preferences Window. When Session Logging is recording to a file, the Log File Name is appended to the Tools Menu (see Figure 11.50). To view the Log File, select the Log File from the Tools Menu. If you want to retain log files after your debugging session exits, select the checkbutton labeled “Keep Log Files” in the Preferences Window. Otherwise the logfile will be automatically deleted.

Figure 11.50 Tools Menu - Session Log File



The name of the log file is generated automatically and contains the project file name and a number related to the logging start time. Selecting the Log File name from the Tools Menu invokes the Session Log Window.

Figure 11.51 Session Log Window


W:/0_test/vocoder_script/Workspace/default.pjt: W:/0_test/vocoder_script/Workspace/default_1013553710.log
Refresh
Clear
Disable logging
Log to window
Log to File
Purge Log

```
ZSP GUI DEBUGGER LOG FILE
        LOGGING STARTED: 16:41:50 12/02/02
           ARCHITECTURE: zsp400
                VERSION: 4.0beta,
                RELEASE: 1.133
                 TARGET: zsim
           PROJECT FILE: W:/0_test/vocoder_script/Workspace/default.pjt
        EXECUTABLE FILE: W:/0_test/vocoder_script/Workspace/G726.exe
================================================
     GUI: memory_download ..\testdat\G726opt.bin &G726_chno 1
          (sdbug)
     GUI: i b
          Num Type           Disp Enb Address    What
          28  breakpoint     keep y   0x0000001a in main at ./G726main.c:87
                breakpoint already hit 1 time
          (sdbug)
```

Log Window Controls

- Refresh - When logging to a file, the refresh button reads the log file into the Log Window text area
- Clear - Clears the Log Window text area

- Log Type Radiobuttons - Radiobuttons labeled “Disable Logging”, “Log To Window”, and “Log To File” have the same functionality as their counterparts in the Preferences Window. The presence of these radiobuttons allows logging to be easily reconfigured when in use.
- Purge Log File - Each time the logging mode changes to “Log to File” a new log file is created and the log file name is updated on the Tools Menu. The “Purge Log Files” button deletes all log files in your project directory (ie those with a .log extension).

Appendix A
Example Programs

This appendix contains two example programs, demo.c and hw_dbg.s, that are referred to in previous chapters of this document. The first example is a program project that combines C and assembly-language modules. The second example is a program used in hardware-assisted debugging.

A.1 Example Program: demo.c

This example is a C program in the file demo.c. It calls another C function, func2, in the file func2.c. It also calls two assembly functions, func1 and func3, in the assembly file func1.s.

```
int func_1 (int *t);
void func_2 ();
int func_3 ();

int t=500;

main()
{
   char ch = ‘A’;
   int i,j = 100,k;

   for (i=0; i< 2; i++) {
     func_2();
     k = func_1 (&j);
     if (k) {
       j = func_3() + 100;
     }
     else {
       j = 100;
     }
```

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

```
   }

   while (i < 20) {
     k++;
     i++;
   }
}
```

Example Program: `func2.c`

```
int t1;
void func_2 ()
{
  int x=0,n=0;
  while(n < 20)
  {
     switch(n) {
     case 0:
       x += 5;
       n =1;
       break;
     case 1:
       x = x <<4;
       n = 4;
       break;
     case 17:
       x = x ^ 13;
       n = 20;
       break;
     default:
       x++;
       n++;
       break;
     }
     t1 = x;
  }
```

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Example Program: `funcl.s`

```
          .segment “text”

   .globl _func_1
   .walign 2
_func_1:

   /** PROLOGUE **/

   mov    r13, %rpc
   stu    r13, r12, -1

   /** END PROLOGUE **/

   mov    r5, r4
   ld     r4, r5
   mov    r6, 500
   cmp    r4, r6       /*  *t <= 500;  */
   bgt    L2
   ld     r4, r5
   mov    r6, 100
   add    r4, r6       /*   *t += 100;  */
   st     r4, r5
   mov    r4, 1
   br     L1
L2:
   mov    r4, 0
   br     L1
L1:

   /** EPILOGUE **/

   bitc   %imask, 15
   nop
   add    r12, 1
   ldu    r13, r12, 1
   mov    %rpc, r13
   add    r12, -1
   bits   %imask, 15
   ret

   /** END EPILOGUE **/
```

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

```
    .extern_t
    .globl _func_3
    .walign 2
_func_3:

    /** PROLOGUE **/

    mov    r13, %rpc
    stu    r13, r12, -1

    /** END PROLOGUE **/

    mov    r5, 300
    lda    r4, _t
    ld     r4, r4
    shll   r4, 1
    add    r4, r5       /**  k = i + 2 * t  **/
    add    r4, r5
    lda    r6, _t
    ld     r6, r6
    add    r4, r6
    br     L3
L3:

    /** EPILOGUE **/

    bitc   %imask, 15
    nop
    add    r12, 1
    ldu    r13, r12, 1
    mov    %rpc, r13
    add    r12, -1
    bits   %imask, 15
    ret

    /** END EPILOGUE **/
```

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

A.2 Example Program hw_dbg.s

This example illustrates hardware-assisted debugging. It consists of one assembly file, hw_dbg.s.

```
.section ".text"
   .global __start
__start:
    bits   %smode, 6
    mov    r0, 0xab00
    mov    r1, 0xab01
    mov    r2, 0xab02
    mov    r3, 0xab03
    mov    r4, 0xab04
    mov    r5, 0xab05
    mov    r14, 0
    mov    r15, 0
    nop
    nop
    nop
    nop
    nop
    add    r14, 1
    mov    r13, 0x2000
    mov    r12, 0x2001
    nop
    nop
    nop
    nop
    nop
    add    r14, 1
    st     r0, r13
    nop
    nop
    nop
    nop
    nop
    add    r14, 1
    st     r1, r13
    nop
    nop
    nop
    nop
    nop
    add    r14, 1
```

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

```
st     r2, r13
nop
nop
nop
nop
nop
add    r14, 1
st     r0, r13
nop
nop
nop
nop
nop
add    r14, 1
st     r1, r13
nop
nop
nop
nop
nop
add    r14, 1
st     r2, r13
nop
nop
nop
nop
nop
add    r14, 1
st     r0, r13
nop
nop
nop
nop
nop
add    r14, 1
st     r1, r13
nop
nop
nop
nop
nop
add    r14, 1
st     r2, r13
nop
nop
nop
nop
nop
```

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

```
add     r14, 1
st      r0, r13
nop
nop
nop
nop
nop
add     r14, 1
st      r1, r13
nop
nop
nop
nop
nop
add     r14, 1
st      r2, r13
nop
nop
nop
nop
nop
nop
nop
nop
nop
nop
add     r15, 1
st      r0, r12
nop
nop
nop
nop
nop
add     r15, 1
st      r1, r12
nop
nop
nop
nop
nop
add     r15, 1
st      r2, r12
nop
nop
nop
nop
nop
add     r15, 1
```

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

```
st      r0, r12
nop
nop
nop
nop
nop
add     r15, 1
st      r1, r12
nop
nop
nop
nop
nop
add     r15, 1
st      r2, r12
nop
nop
nop
nop
nop
add     r15, 1
st      r0, r12
nop
nop
nop
nop
nop
add     r15, 1
st      r1, r12
nop
nop
nop
nop
nop
add     r15, 1
st      r2, r12
nop
nop
nop
nop
nop
add     r15, 1
st      r0, r12
nop
nop
nop
nop
nop
```

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

```
add     r15, 1
st      r1, r12
nop
nop
nop
nop
nop
add     r15, 1
st      r2, r12
nop
nop
nop
nop
nop
bitc    %smode, 6
halt
```

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Appendix B ZSP400 Control Registers

The ZSP400 control registers are listed in Table B.1.

Table B.1 ZSP400 Control Registers

Register Reference Number	Control Register	Register Description
0	`%fmode`	Functional Mode Register
1	`%tc`	Timer Control Register
2	`%imask`	Interrupt Mask Register
3	`%ip0`	Interrupt Priority Register 0
4	`%ip1`	Interrupt Priority Register 1
5	`%loop0`	Loop 0 Register
6	`%loop1`	Loop 1 Register
7	`%guard`	Guard Bits for {r1 r0} and {r3 r2}
8	`%hwflag`	Condition Codes
9	`%ireq`	Interrupt Request Register
10	`reserved`	–
11	`reserved`	–
12	`%vitr`	Viterbi Traceback Register
13	`reserved`	–
14	`%amode`	Addressing Mode Register
15	`%smode`	System Mode Register
(Sheet 1 of 2)		

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Table B.1 ZSP400 Control Registers (Cont.)

Register Reference Number	Control Register	Register Description
16	%pc	Program Counter
17	%rpc	Return Program Counter
18	%tpc	Trap Return Program Counter
19	%cb0_beg	Circular Buffer 0 Begin Address
20	%cb1_beg	Circular Buffer 1 Begin Address
21	%cb0_end	Circular Buffer 0 End Address
22	%cb1_end	Circular Buffer 1 End Address
23	%timer0	Timer0
24	%timer1	Timer1
25	%loop2	Loop 2 Register
26	%loop3	Loop 3 Register
27	reserved	–
28	reserved	–
29	reserved	–
30	%dei	Device Emulation Instruction Register
31	%ded	Device Emulation Data Register
(Sheet 2 of 2)		

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Appendix C ZSPG2 Control Registers

The G2 control registers are listed in Table C.1.

Table C.1 G2 Control Registers

Register Reference Number	Control Register	Register Description
0	`%fmode`	Functional Mode Register
1	`%tc`	Timer Control Register
2	`%imask`	Interrupt Mask Register
3	`%ip0`	Interrupt Priority Register 0
4	`%ip1`	Interrupt Priority Register 1
5	`%loop0`	Loop 0 Register
6	`%loop1`	Loop 1 Register
7	`%guard`	Guard Bits for {r1 r0} and {r3 r2}
8	`%hwflag`	Condition Codes
9	`%ireq`	Interrupt Request Register
10	%cb2_beg	Circular buffer 2 Begin Address
11	%cb2_end	Circular buffer 2 Begin Address
12	`%vitr`	Viterbi Traceback Register
13	`%shwflag`	Sticky Condition Codes
14	%amode	Address Mode Register
15	%smode	System Mode Register
(Sheet 1 of 2)		

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Table C.1 G2 Control Registers (Cont.)

Register Reference Number	Control Register	Register Description
16	`%pc`	Program Counter
17	`%rpc`	Return Program Counter
18	`%tpc`	Trap Return Program Counter
19	`%cb0_beg`	Circular Buffer 0 Begin Address
20	`%cb1_beg`	Circular Buffer 1 Begin Address
21	`%cb0_end`	Circular Buffer 0 End Address
22	`%cb1_end`	Circular Buffer 1 End Address
23	`%timer0`	Timer0
24	`%timer1`	Timer1
25	`%loop2`	Loop 2Register
26	`%loop3`	Loop 3 Register
27	`%cb3_beg`	Circular Buffer 3 Begin Address
28	`%cb3_end`	Circular Buffer 3 End Address
29	`reserved`	–
30	`%dei`	Device Emulation Instruction Register
31	`%ded`	Device Emulation Data Register
(Sheet 2 of 2)		

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Appendix D
L-Intrinsic Functions

This appendix describes the Long Intrinsic functions (L-Intrinsics) that were included in Version 1.0 of the SDK compiler and that are currently supported for backward compatibility. The L-Intrinsics are no longer implemented within the compiler itself, but rather with a header file, dsp.h. Note that although the L-Intrinsics are supported, you should develop new code using the N-Intrinsics, described in Chapter 3, "C Cross Compiler," Section 3.5, "N-Intrinsics," page 3-16.

To use the L-Intrinsic functions, add the following line to all your C files:

```
#include <dsp.h>
```

The compiler implements the L-Intrinsic functions using the assembly instructions shown in Table D.1.

Table D.1 Long Intrinsic Functions

Intrinsic Function	Underlying Instruction
L_mula	mul.a
L_maca	mac.a
L_macna	macn.a
L_mac2a	mac2.a
L_mulb	mul.b
L_macb	mac.b
L_macnb	macn.b
L_mac2b	mac2.b

The long argument for the L_maca, L_macb, L_macna, L_macnb, L_mac2a, and L_mac2b intrinsic functions is copied to the appropriate accumulator register, which is {r0,r1} for the .a versions and {r2, r3} for the .b versions.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

The compiler generates code to copy the arguments to the proper accumulator registers, if required. Eliminating the steps required in copying the arguments minimizes execution time. Copying the arguments is not required if:

- The long argument already exists in the appropriate accumulator (for example, if you call L_maca with a variable declared as type accum_a).

Execution time can also be minimized by not requiring the result to be copied to its destination. Copying the result is not required if:

- The destination for the intrinsic function's result is already the target for the instruction used to implement the intrinsic function (for example, if L_maca returns a value to a variable declared as type accum_a)

For example, the following code is legal:

```
accum_b b;
int x,y;
...
b = L_maca(b,x,y);
```

However, it is more efficient to use:

```
b = L_macb(b,x,y);
```

In the first case (b = L_maca(b,x,y)), two copies are required—one to move {r3 r2} to {r1 r0} for the argument, and another to move {r3 r2} to {r1 r0} to the destination. The second case (b = L_macb(b,x,y)) requires no extra copies.

Note that a call to an L_*a function clobbers any variable declared with an accum_a, and a call to an L_*b function clobbers any variable declared with an accum_b. In the following example, the value of variable a is equivalent to b after the L_maca function call:

```
accum_a a;
accum_b b;
int x,y;
a = 0;
...
b = L_maca(b,x,y);
```

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Note: It is not guaranteed that a will have the same value as b in future versions of the SDK compiler.

Long L_mula (int var1, int var2)	This function returns a 32-bit result of the multiplication of a 16-bit variable `var1` with a 16-bit variable `var2`, with one shift left.
Long L_mulb (int var1, int var2)	This function returns a 32-bit result of the multiplication of a 16-bit variable `var1` with a 16-bit variable `var2`, with one shift left.
Long L_maca (long var3, int var1, int var2)	This function multiplies the 16-bit variable `var1` by the 16-bit variable `var2` and shifts the result left by 1. This 32-bit result is added to the 32-bit variable `var3` with saturation and returns the 32-bit result.
Long L_macb (long var3, int var1, int var2)	This function multiplies the 16-bit variable `var1` by the 16-bit variable `var2` and shifts the result left by 1. This 32-bit result is added to the 32 bit variable `var3` with saturation and returns the 32-bit result.
Long L_macna (long var3, int var1, int var2	This function multiplies the 16-bit variable `var1` by the 16-bit variable `var2` and shifts the result left by 1. This 32-bit result is subtracted by the 32-bit variable `var3` with saturation and returns the 32-bit result.
Long L_macnb (long var3, int var1, int var2)	This function multiplies the 16-bit variable `var1` by the 16-bit variable `var2` and shifts the result left by 1. This 32-bit result is subtracted by the 32-bit variable `var3` with saturation and returns the 32-bit result.
Long L_mac2a (long var3, long var1, long var2)	The lower 16 bits of the variable `var1` is multiplied with the lower 16 bits of the variable `var2`. The higher 16 bits of the variable `var1` is multiplied with the higher 16 bits of variable `var2`, and the two 32-bit results are added to the variable `var3`, which is the return value.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Long L_mac2b (long var3, long var1, long var2)	The lower 16 bits of the variable `var1` is multiplied with the lower 16 bits of the variable `var2`. The higher 16 bits of the variable `var1` is multiplied with the higher 16 bits of variable `var2`, and the two 32-bit results are added to the variable `var3`, which is the return value.
Long norm_l (long var1)	This function produces the number of left shifts required to normalize a 32-bit variable `var1`. The number is a 32-bit result.
int norm_s (int var1)	This function produces the number of left shifts required to normalize a 16-bit variable `var1`. The number is a 16-bit result.
Long L_deposit_h (int var1)	This function returns a 32-bit result, where the high-order 16 bits is the input 16-bit variable `var1`, and the low-order 16 bits are zeroed.
int extract_h (long)	This function returns a 16-bit result which is the high-order 16 bits of the 32-bit input.
Long L_abs (long var1)	This function returns a 32-bit result which is the absolute value of the 32-bit variable `var1`. Note that abs (0x8000) returns 0x7FFF.
int abs_s (int var1)	This function returns a 16-bit result which is the absolute value of the 16-bit variable `var1`. Note that `abs.s` (0x8000) returns 0x7FFF.
int round (long)	This function returns a 16-bit result. The result is obtained by rounding the lower 16 bits of the 32-bit input number and storing it in the higher 16 bits with saturation. This value is then shifted right by 16 bits to obtain the result.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Appendix E
Signal Processing Library

The library, `libalg.a`, contains some basic functionality commonly used in signal processing. The interface to `libalg.a` is contained in `alg.h`, which can be accessed with:

```
#include <alg.h>
```

To use this library, it must be linked in with a -lalg switch on the link line.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.1 API Specification Auto-correlation Library Function on G2

E.1.1 Auto-correlation

Synopsis

```
void lib_autocorr(*Struct_AUTOCOR)
```

***Struct_AUTOCOR**	Pointer to the Auto-correlation Structure

Input

The input variables that are to be passed through the AUTOCOR structure:

short DataSize	Length of the input data
short InputData	Input data array of size Datasize*2
short NumberOfLags	Number of auto-correlation lags needed
short Scale	Factor to use in scaling the partial products

Return Value

None

Output

The output is returned as a field in the AUTOCOR structure

short AutoCorrData	Array to hold the Auto-correlation values

Description

This function implements the auto-correlation of the input data (InputData) and stores the computed correlation lags in an array (AutoCorrData). The number of correlation lags are specified by NumberOfLags. As the number of lags are small, a direct sum-of-product algorithm is used for computing the correlation values.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.2 API Specification for Convolutional Encoder Library Function on G2

E.2.1 Convolutional Encoder

Synopsis

```
void lib_convEnc_k9r2(short *inpw, short *outpw, short
Nwords)
```

Input

Short *inpw	Pointer to input data (packed, 16-bit array)
Short Nwords	Size of input array

Return

None

Output

Short *outpw	Pointer to output data (packed, 16-bit array)

Description

This function implements a Convolutional encoder with generating polynomial,

G0 = 1 + D2 + D3 + D4 + D8 (octal 561)
G1 = 1 + D1 + D2 + D3 + D5 + D7 + D8 (octal 753)

and with a constraint length of K=9 and rate of R=1/2.

It employs Block-XOR technique, along with LUT-based sorting and operates on packed words containing input data bits.

Dependencies/Assumptions

This encoder always starts from the zero state.

Assumes that the input data bits are packed into an array of 16-bit words, in a "right-MSB" format, that is, in each word, the LSB has the oldest

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

data. In the final word, if there are fewer than 16 data bits, the MSB part may be filled with zero bits but not essential.

The output encoded bits are available packed into 16-bit words in the same "right-MSB" format. The output array size is twice that of the input array, and any extra bits in the final output word may be ignored.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.3 API Specification for 16bit CRC Library Function on G2

E.3.1 CRC 16bit

Synopsis

```
short lib_crc16(short *inpw, short Nwords)
```

Input

Short *inpw	Pointer to input data (packed, 16-bit array)
Short Nwords	Size of input for which CRC is needed

Output

Short crc16	Computed checksum (16-bit scalar)

Description

This function implements CRC-16 bit checksum calculation, based on the Generating Polynomial

P(D) = D(16) + D(12) + D(5) + 1 (decimal 69,665).

Dependencies/Assumptions

Assumes that the input bits are packed into an array of 16-bit words, in a "right-MSB" format, that is, in each word, the LSB has the oldest data. In the final input word, if there are fewer than 16 data bits, the MSB part may be filled with zero bits but not essential.

The output encoded bits are available packed into one 16-bit word in the same "right-MSB" format.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.4 API Specification for 8bit CRC Library Function on G2

E.4.1 CRC 8bit

Synopsis

```
short lib_crc8(short *inpw, short Nwords)
```

Input

Short *inpw	Pointer to input data (packed, 16-bit array)
Short Nwords	Size of input for which CRC is needed

Output

Short crc8	Computed checksum (16-bit scalar)

Description

This function implements CRC-8 bit checksum calculation, based on the Generating Polynomial

D(8) + D(7) +D(4) + D(3) + D + 1 (decimal 411).

Dependencies/Assumptions

Assumes that the input data bits are packed into an array of 16-bit words, in a "right-MSB" format, that is, in each word, the LSB has the oldest data. In the final input word, if there are fewer than 16 data bits, the MSB part may be filled with zero bits but not essential.

The output encoded bits are available packed into one 16-bit word in the same "right-MSB" format.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.5 API Specification for 32 bit Division Library Function on G2

E.5.1 32 bit Division

Synopsis

```
Result32 lib_div32( Num32, Den32  )
```

Input

Int Num32	32 bit positive integer
Int Den32	32 bit positive integer

Return

Int Result32	Q31 Fractional number

Description

Performs a 32 bit fractional division between two 32 bit positive integers

Result32 = Num32/Den32

The technique is a 32 bit implementation of the 16 bit divide step instruction "diva"

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.6 API Specification for IIR Library Function on G2

E.6.1 IIR

Synopsis

```
void lib_IIR(short *indata, short *coef, short *state, short N)
```

Input

Short *indata	Pointer to input data.
Short *coef	Coefficient vector.
Short *state	Intermediate state of the filter.
Short N	Length of the input data vector.

Return

None

Output

Output is returned in the “indata” input data vector.

Description

This function implements an in-place Infinite Impulse Response (IIR) filter.

Dependencies/Assumptions

The input data is assumed to be in Q1.15 format.

The number of taps in the filter “T” must be a multiple of 2.

Coefficients are stored as -a1/2, -a2/2, b1/2, b2/2, ..., b0/2.

Input data is stored 0, In(0), In(1), ..., In(N).

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.7 API Specification for IIR Biquad Library Function on G2

E.7.1 IIR Biquad

Synopsis

```
void lib_IIRBIQ(short *indata, short *coef, short *state, short N-1)
```

Input

Short *indata	Pointer to input data.
Short *coef	Coefficient vector.
Short *state	Intermediate state of the filter.
Short N-1	Length of the input data vector.

Return

None

Output

Output is returned in the “indata” input data vector.

Description

This function implements an in-place Biquad Infinite Impulse Response (IIR) filter.

Dependencies/Assumptions

The input data is assumed to be in Q1.15 format.

The number of taps in the filter “T” must be a multiple of 2.

Coefficients are stored as -a11/2, a21/2, b11/2, b21/2 -a21/2, a22/2, b21/2, b22/2.

Input data is stored 0, In(0), In(1), ..., In(N).

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.8 API Specification for Inverse Square Root Library Function on G2

E.8.1 Inverse Square Root

Synopsis

Xout lib_invsqrt(Xi)

Input

Short Xi Q14 number in the range 0x1000 (0.25) < Xi < 0x7fff (1.99999)

Return

short Xout Q14 number in the range 0x1000 (0.25) < Xi < 0x7fff (1.99999)

Description

Calculate the inverse square root of an input Xi.

Xout = 1/sqrt(Xi)

Technique employs a look up table to obtain a first approximation to Xout.

The approximation Xout is then used by following recursive algorithm to calculate a more precise value for Xout.

Xout = (3/2)*Xout - (Xi * Xout^3)/2

Three iterations of the above algorithm are performed

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.9 API Specification for Synthesis Lattice Filter Library Function on G2

E.9.1 Synthesis Lattice Filter

Synopsis

```
short lib_lattice(short *b, short n, short *k)
```

Input

Short *b	Array of filter coefficients
Short n	Number of data samples
Short *k	Array of filter coefficients

Output

Short f	Result of forward synthesis

Description

This function implements a Lattice filter. The lattice is a synthesis filter which calculates the following loop:

```
f -= b[n - 1] * k[n - 1];
for (i = n - 2; i >= 0; i--) {
f -= b[i] * k[i];
b[i + 1] = b[i] + (k[i] * f);
                 {
```

where “n” is the order for the filter, “k” and “b” are coefficients and “f” is the “forward result”

The variables f, b[i],k[i] and k are in q15 format.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.10 API Specification for Real Block FIR Library Function on G2

E.10.1 Real Block FIR

Synopsis

```
void lib_realblockfir(*FIR)
```

***RBF_CFG_Type**	Pointer to a configuration type structure

Input

int *x	Address of input array, length>=N.
int *h	Address of coefficients, length>=T. Coefficients stored in reverse order h(T-1) ... h(0).
int N	Number input samples in x to filter. N must be multiple of 4.
int T	Number of filter taps (length of h). T must be multiple of 4 and T>=8.

Output

int *y	Address of output array, length>=N
int *delay_line	Base address of delay line
int *delay_current	Ptr to current addr in delay line (oldest sample)

Description

This function implements a real valued block FIR filter. The N samples of input array (“x”) are filtered with T filter coefficients in array (“h”), and the result is stored in array (“y”).

The input, output, and filter coefficients are 16-bits. The filter coefficients must be stored in reverse order h(T-1) ... h(0).

A delay line is used to hold the history of input data and it is updated each time to contain the latest T samples and point to the oldest of them.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Accumulations are 40 bits with bits 31-16 being the stored result, which will be saturated if SAT is enabled.

Two taps for each of 4 output samples are computed every iteration of the inner loop.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

E.11 API Specification for 256 point FFT Library Function on G2

E.11.1 256 point FFT

Synopsis

```
void lib_FFT256(short *in_data, short *out_data, *twiddles)
void lib_iFFT256(short *in_data, short *out_data, *twiddles)
```

Input

Short *in_data	Pointer to input data
Short *twiddles	Array of Twiddle factors

Return

None

Output

Short *out_data	Computed FFT or iFFT values

Description

This function implements a 256 point complex, Radix-2, decimation-in-time Fast Fourier Transform (FFT) algorithm.

Dependencies/Assumptions

The input and output data are to be stored as Im,Re,Im,Re... and are in natural order.

The input and output data is in Q.15 format.

Twiddle factors have to be recalculated and stored in memory.

Copyright © 1999-2002 by LSI Logic Corporation. All rights reserved.

Index

Customer Feedback

We would appreciate your feedback on this document. Please copy the following page, add your comments, and fax it to us at the number shown.

If appropriate, please also fax copies of any marked-up pages from this document.

Important: Please include your name, phone number, fax number, and company address so that we may contact you directly for clarification or additional information.

Thank you for your help in improving the quality of our documents.

Reader's Comments

Fax your comments to:	LSI Logic Corporation Technical Publications M/S E-198 Fax: 408.433.4333

Please tell us how you rate this document: *ZSP SDK Software Development Kit.* Place a check mark in the appropriate blank for each category.

	Excellent	**Good**	**Average**	**Fair**	**Poor**
Completeness of information	____	____	____	____	____
Clarity of information	____	____	____	____	____
Ease of finding information	____	____	____	____	____
Technical content	____	____	____	____	____
Usefulness of examples and illustrations	____	____	____	____	____
Overall manual	____	____	____	____	____

What could we do to improve this document?

If you found errors in this document, please specify the error and page number. If appropriate, please fax a marked-up copy of the page(s).

Please complete the information below so that we may contact you directly for clarification or additional information.

Name	____________	Date	____________
Telephone	____________	Fax	____________
Title	____________		
Department	____________	Mail Stop	____________
Company Name	____________		
Street	____________		
City, State, Zip	____________		

You can find a current list of our U.S. distributors, international distributors, and sales offices and design resource centers on our web site at

http://www.lsilogic.com/contacts/na_salesoffices.html