USER'S GUIDE

ZSP™ Software Development Kit

May 2003

Revision 4.3.1

LSI LOGIC®

DB15-000126-10

DB15-000126-10, Sixth Edition (May 2003)
This document describes Rev. 4.3.1 of LSI Logic Corporation's ZSP™ Software Development Kit and will remain the official reference source for all revisions/releases of this product until rescinded by an update.

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

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

Preface

This book is the primary reference and user's guide for the ZSP™ Software Development Kit (SDK). The SDK supports digital signal processors based on the ZSP400 core (for example, the LSI402ZX and LSI403LP) 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 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, **ZSP SDK Functional-Accurate Simulator**, describes the SDK functional-accurate simulator.
- Chapter 8, **ZSP SDK Cycle-Accurate Simulator**, describes the SDK cycle-accurate simulator.
- Chapter 9, **Debugger**, describes the SDK debugger.
- Chapter 10, **ZSP MDI Configuration Files**, describes the configuration files for the ZSP SDK MDI libraries.
- Chapter 11, **ZSP Integrated Development Environment**, describes the SDK Project Manager provided by LSI Logic with Windows 98/NT/2000/XP and Solaris versions of the SDK.
- Chapter 12, **ZSP IDE Debugger**, describes the GUI Debugger provided by LSI Logic with Windows 98/NT/2000/XP and Solaris 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_zsp500.a` and `libalg_zsp600.a` libraries.

Related Publications

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

LSI403LP Digital Signal Processor User's Guide, LSI Logic Corporation, order number R14025. Provides detailed information of the LSI403LP 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, November, 1995 / June, 2001. 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.

ZSIM Peripheral API Reference Guide, LSI Logic Corporation, document number DB06-000299-00. Provides information on writing peripheral libraries.

LSI402ZX Development Kit Getting Started Guide, LSI Logic Corporation, document number DB06-000453-01, March, 2003. Provides information on using the LSI402ZX Development Kit.

EB402 Evaluation Board User's Guide, LSI Logic Corporation, document number DB15-000143-01, July, 2001. 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`.

We would like to acknowledge Herschel Technologies for providing the standard floating point library included in this release.

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, Windows NT, Windows XP, and Windows 2000 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.

Notation	Example	Meaning and Use
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.
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

Figures

Tables

Chapter 1
Introduction

The ZSP Software Development Kit (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 98, Windows NT, Windows 2000, Windows XP, and Solaris 2.x platforms. 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.

The major sections in this chapter are:

- Section 1.1, "Overview of the SDK Tools"
- Section 1.2, "Overview of Software Development Using the SDK Tools"

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 auxiliary files.
- The tmp directories are used by the tools for temporary space.

The GNU based tools for the ZSP400 all have an "sd" prefix. The analogous 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 to take advantage of the many high-performance features of the ZSP LSI402ZX and LSI403Z devices and ZSP400-based parts, 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.
readelf	readelf	Display the contents of ELF format files.

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

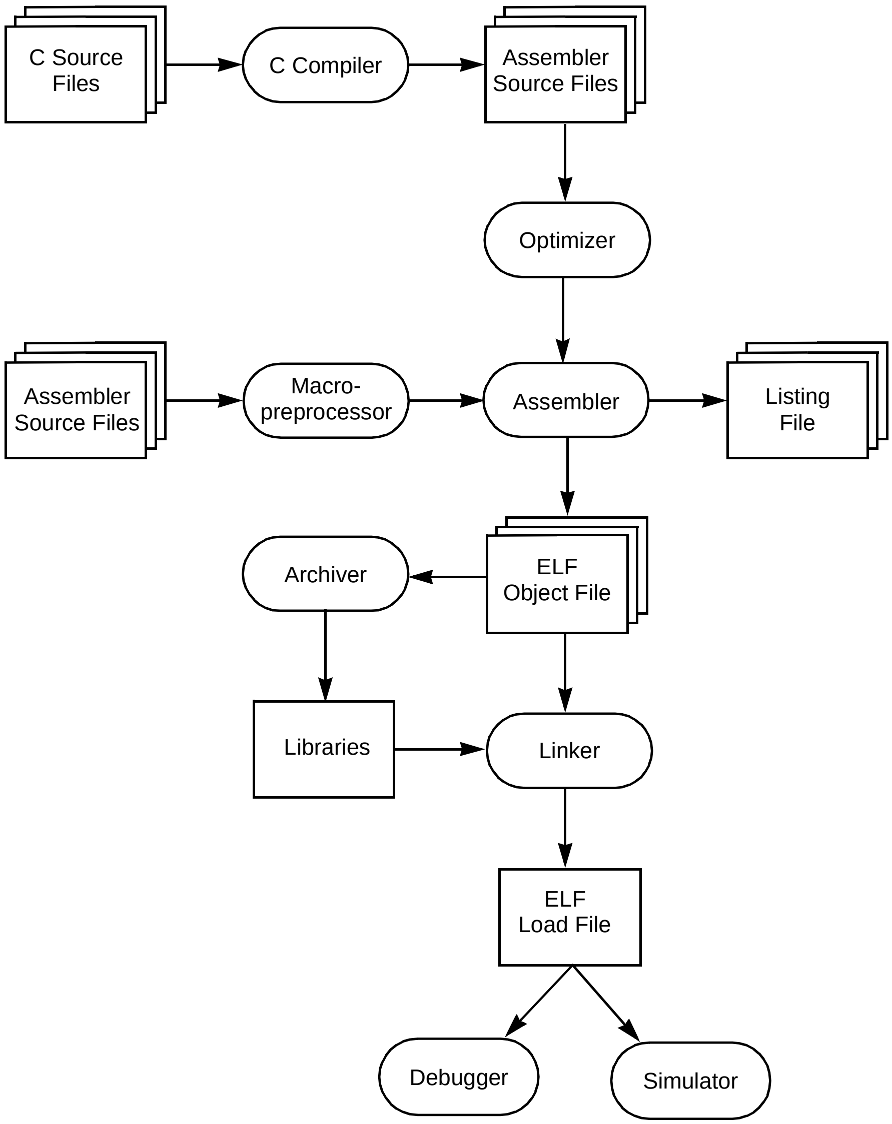
- The compiler's optimizer, sdopt/zdopt/zdxopt, performs additional optimizations to those performed by sdcc/zdcc/zdxcc.
- Both the functional-accurate and cycle-accurate simulators are provided in a standalone form that supports a simple command-line interface and that can be provided in a dynamic linkable format that can be used in conjunction with the debugger.
- The GUI tools include an IDE and a GUI Debugger for both Windows and Solaris platforms.

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 assembler translate the assembly source file into a relocatable object file in the Executable and Linkable Format (ELF) file format (`.obj` for Windows or `.o` for 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 98/NT/2000/XP 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." The tools can also be accessed using the ZSP Integrated Development Environment (ZSP IDE), (Chapter 11), and the ZSP IDE Debugger (Chapter 12).

Figure 1.1 Overview of Software Development



Chapter 2
Installation

The SDK is available for Windows 98, Windows NT, Windows 2000, Windows XP, and Solaris 2.x. The media used to provide the tools is a CD-ROM. This chapter describes how to install the ZSP Software Development Kit. It contains the following major sections:

- Section 2.1, "Contents of the CD-ROM"
- Section 2.2, "Installation on Windows Systems"
- Section 2.3, "Uninstalling the SDK Tools on Windows Systems"
- Section 2.4, "Installation on Solaris Systems"
- Section 2.4, "Installation on Solaris Systems"

2.1 Contents of the CD-ROM

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

Table 2.1 SDK CD-ROM High-Level Directories

Directory	Description
doc	Contains the complete documentation for the SDK tools and the GNU tools. Also includes documentation for the license manager (FLEXlm) and the ZSP Development Kit.
docs	Contains ZSP partners profile information.
`bin`	Contains various executable files used during installation.
`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 98/NT/2000/XP platforms, and examples that can be added to a 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
- 84 Mbytes of disk space

On Windows NT/2000/XP systems, you need administrator privileges to install the ZSP SDK Tools for more than one user account.

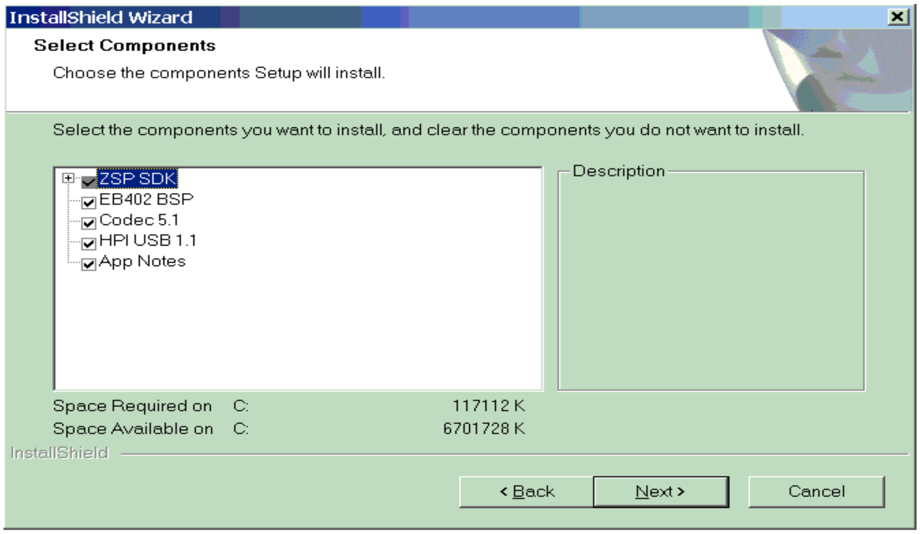
2.2.1 Installing SDK Tools

Important: 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.

The installation process should start automatically and the Select Components dialog box should display, as shown in Figure 2.1.

Figure 2.1 Select Components Dialog Box



If the dialog box does not appear, try running Launch.exe on the CD-ROM. If the CD drive is D:, the program may be found at D:\Launch.exe.

Step 2. Follow the Setup instructions.

The default directory is C:\LSI_Logic\SDK<Version Number>. You can change the default directory, if necessary.

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

The files are installed in subdirectories under C:*Installation_Directory*, where Installation_Directory is the directory specified in Step 2. The subdirectory organization and file descriptions are given in Table 2.2 through Table 2.16.

Table 2.2 Files Installed in C:*Installation_Directory*\doc

Filename	Function
elfread.pdf	Documentation on sdelfread and zdelfread
SDK_<vers>_errata.txt	Errata for ZSP SDK version <vers>
SDK_<vers>_Readme.txt	"Read Me First" file for SDK version <vers>
SDK_<vers>_ReleaseNotes.txt	Release notes for SDK version <vers>
zspsdk_<vers>.pdf	This *User's Guide*

Table 2.3 File Installed in C:*Installation_Directory*\doc\Arch

Filename	Function
peripherial_api.pdf	ZSIM peripheral library API reference guide

Table 2.4 Files Installed in C:*Installation_Directory*\mdi\GNU

Filename	Function
as.pdf	GNU assembler
binutils.pdf	GNU binutils
gcc	GNU Compiler Collection, version 2.95
gcc-3.0	GNU Compiler Collection, version 3.0
gdb	GNU debugger
ld	GNU linker

Table 2.5 Files Installed in C:*Installation_Directory*\mdi

Filename	Function
mdi.dll	Microprocessor Device Interface library for ZSP
CorelisPCI.dll	Probe Support library for Corelis PCI Boundary Scan interface
CorelisPCMCIA.dll	Probe Support library for Corelis PCMCIA Boundary Scan interface

Table 2.6 Files Installed in C:*Installation_Directory*\mdi\Devices

Filename	Function
jtag400.cfg	JTAG configuration file for the ZSP40X family
jtag500.cfg	JTAG configuration file for the ZSP500
zisim400.cfg	ZISIM configuration file for the ZSP40X family
zisim500.cfg	ZISIM configuration file for the ZSP500
zsim400.cfg	ZSIM configuration file for the ZSP40X family
zsim500.cfg	ZSIM configuration file for the ZSP500

Table 2.7 Files Installed in C:*Installation_Directory*\license

Filename	Function
Flexlm_Enduser.pdf	*FlexLM User's Guide for Endusers*
lmborrow.exe	FlexLM utility. See the *FlexLM User's Guide* for details.
lmdiag.exe	
lmdown.exe	
lmgrd.exe	
lmhostid.exe	
lminstall.exe	
lmpath.exe	
lmremove.exe	
lmreread.exe	
lmstat.exe	
lmswitchr.exe	
lmswitchr.exe	
lmtools.exe	
lmutil.exe	
lmver.exe	
zspld.exe	FlexLM vendor daemon for ZSP SDK tools

Table 2.8 Files Installed in C:*Installation_Directory*\mdi\Drivers

Filename	Function
jtagdrv.dll	JTAG driver file for the ZSP40X family
jtagdrvG2.dll	JTAG driver file for the ZSP500
libdrvzisim400.dll	ZISIM driver file for the ZSP40X family

Table 2.8 Files Installed in C:*Installation_Directory*\mdi\Drivers (Cont.)

Filename	Function
libdrvzisim500.dll	ZISIM driver file for the ZSP500
libdrvzsim400.dll	ZSIM driver file for the ZSP40X family
libdrvzsim500.dll	ZSIM driver file for the ZSP500

Table 2.9 Files Installed in C:*Installation_Directory*\sdspI\bin

Filename	Function
readelf.exe	GNU utility for examining an object file
sdelfread.exe	Produces a simple dump of entire contents of an object file
libzisim400.dll	Dynamic link libraries used in sdbug400 for target zisim
libzsim400.dll	
libzperiph.dll	
sdar.exe	Archive utility
sdas.exe	Assembler
sdbug400.exe	Source-level debugger for ZSP400-based devices
sdcc.exe	Driver
sdcc1.exe	Compiler
sdcpp.exe	Preprocessor
sdld.exe	Linker
sdnm.exe	Symbol listing utility
sdobjcopy.exe	Object file copying utility
sdobjdump.exe	Object dump utility
sdopt.exe	Optimizer
sdranlib.exe	Ranlib utility

Table 2.9 Files Installed in C:*Installation_Directory*\\sdspI\bin (Cont.)

Filename	Function
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

Table 2.10 Files Installed in 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
libm.a	Math library

Table 2.11 Files Installed in C:*Installation_Directory*\\sdspI\include

Filename	Function
assert.h	Standard header file
cbuf.h	Circular buffer
creg.h	Control registers
ctype.h	Standard header file
dsp.h	L-Intrinsics
float.h	Floating point support

Table 2.11 Files Installed in C:*Installation_Directory*\sdspI\include (Cont.)

Filename	Function
libsdsp.h	SDSP-specific printing
limits.h	Standard header file
math.h	Math library functions
N_Intrinsic.h	N-Intrinsics
q15.h	Support file (deprecated)
setjmp.h	Standard header files
simios.h	
stdarg.h	
stddef.h	
stdio.h	
stdlib.h	
string.h	
timer_util.h	Timer functions

Table 2.12 Files Installed in C:*Installation_Directory*\zspg2\bin

Filename	Function
readelf.exe	GNU utility for examining an object file.
zdelfread.exe	Produces a simple dump of entire contents of an object file
libcpig711.dll	Dynamic link library used for g711 coprocessor support. Used by zdbug, zdxbug for target zsim, or zsimg2.
libzisimg2.dll	Dynamic link library used in zdbug and zdxbug for target sim or zisimg2
libzidlmssg2.dll	Dynamic link library used in zdbug and zdxbug for target sim or zisimg2

Table 2.12 Files Installed in `C:\Installation_Directory\zspg2\bin` (Cont.)

Filename	Function
`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`	Compiler driver for zdcc
`zdxcc1.exe`	Compiler driver for zdxcc
`zdcpp.exe`	Preprocessor
`zdxcpp.exe`	Preprocessor for zdxcc
`zdld.exe`	Linker
`zdnm.exe`	Symbol listing utility
`zdobjcopy.exe`	Object file copying utility
`zdobjdump.exe`	Object dump utility
`zdopt.exe`	Optimizer
`zdxopt.exe`	Optimizer for ZSP400 to ZSP500 code
`zdranlib.exe`	Ranlib utility
`zdsize.exe`	Size utility
`zdstrings.exe`	String print utility
`zdstrip.exe`	Symbol discarding utility
`zisimg2.exe`	Functional-accurate simulator for ZSP400-based devices

Table 2.13 Libraries Installed in C:*Installation_Directory*\zspg2\lib

Filename	Function
crt0.obj	Startup file
libc.a	C library
libg.a	C library with debug information
libm.a	Math function library
libalg_zsp500.a	Basic signal processing functionality -- optimized for ZSP500 core.
libalg_zsp600.a	Basic signal processing functionality -- optimized for ZSP600 core.

Table 2.14 Libraries Installed in C:*Installation_Directory*\zspg2\liibg1g2

Filename	Function
crt0.obj	Startup file
libc.a	C library
libg.a	C library with debug information
libm.a	Math function library

Table 2.15 Header Files Installed in C:*Installation_Directory*\zspg2\include

Filename	Function
cbuf.h	Circular buffer
ctype.h	Standard header file
creg.h	Control registers
dsp.h	L-Intrinsics
float.h	Floating point

Table 2.15 Header Files Installed in C:*Installation_Directory*\zspg2\include (Cont.)

Filename	Function
libsdsp.h	SDSP-specific printing
limits.h	Standard header file
math.h	Math functions
N_Intrinsic.h	N-Intrinsics
q15.h	Support file
setjmp.h	Standard header files
simios.h	
stdarg.h	
stddef.h	
stdio.h	
stdlib.h	
string.h	

Table 2.16 Files Installed in C:*Installation_Directory*\ide\bin

Filename	Function
zspcat.exe	Used by zspide
djpeg.exe	Used by data graph utility
float.exe	Used by GUI Debugger for floating point data conversion
guidebug_help.exe	Help file for the GUI Debugger.
KILL.EXE	Use by GUI Debugger to kill command line debugger process
plplot510.dll	Used by data graph utility
rls_semaphore.exe	Used by GUI Debugger
tktable.dll	Used by GUI Debugger

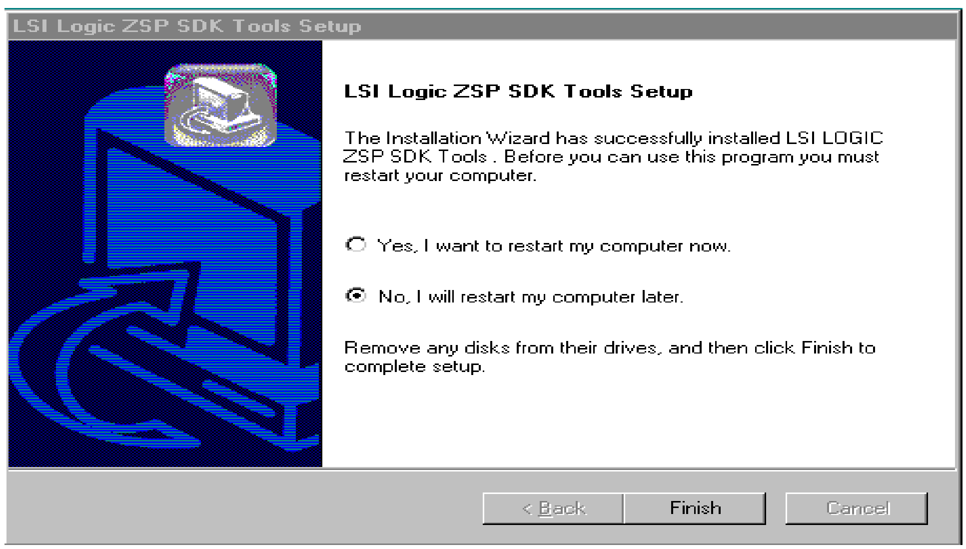
Table 2.16 Files Installed in **C:*Installation_Directory*\ide\bin** (Cont.)

Filename	Function
TLIST.EXE	Used by GUI Debugger to identify command-line debugger process
zdmake.exe	Make utility
zspide.exe	IDE for the ZSP family of processors
zspide_help.exe	Help file for the IDE
zsplic.exe	Licence manager utility

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, as shown in Figure 2.2.

Figure 2.2 System Reboot Prompt



Click Finish to exit from the Setup program.The system is restarted according to the option selected in the preceding Tools Setup 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.

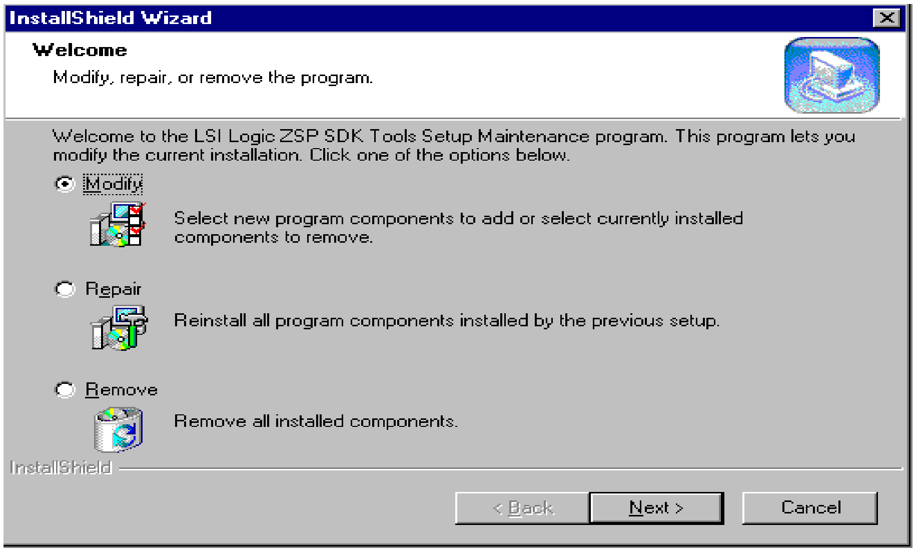
The Control Panel is accessed by clicking on the Start menu, then selecting Settings, then selecting Control Panel.

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

Step 3. Then select the LSI LOGIC SDK tools and click on Add/Remove.

This causes the dialog box shown in Figure 2.3 to appear.

Figure 2.3 Uninstalling the SDK Tools



Step 4. Click on Remove and continue with Next to uninstall the tools.

2.4 Installation on Solaris Systems

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

Step 1. Insert the CD-ROM.

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

Step 2. If `vold` is not running, insert 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/solaris` or `/mnt/cdrom/solaris`:

`/cdrom/solaris/setup`

or

`/mnt/cdrom/solaris/setup`

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/`

Two scripts are provided by the setup routine, `sdk.csh` and `sdk.sh`, that set up the environment for you. From csh, type "`source sdk.csh`" to update your environment variables. Type "`sdk.sh`" from the Bourne shell.

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. To use the simulator or debugger, you must include the environment variable `LD_LIBRARY_PATH` in the installation path. Use the following one-line command:

```
setenv LD_LIBRARY_PATH ${LD_LIBRARY_PATH}:
$SDSP_HOME/sdspI/bin:$SDSP_HOME/zspg2/bin
:$SDSP_HOME/MDI:$SDSP_HOME/ide/bin
```

The Setup program installs the SDK files. Table 2.17 through Table 2.24 list these files and the directories to which they are installed.

Table 2.17 Command-Line Tools Installed in $SDSP_HOME/sdspI/bin

Filename	Function
readelf	GNU utility for examining an object file
sdelfread	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
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

Table 2.18 Libraries Installed in $SDSP_HOME/sdspI/lib

Filename	Function
crt0.o	Startup file
libc.a	C library
libg.a	C library with debug information
liblongc.a	C library with long calls
libm.a	Math functions

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

Table 2.19 Header Files Installed in $SDSP_HOME/sdspI/include

Filename	Function
assert.h	Standard header file
cbuf.h	Circular buffer
creg.h	Control registers
ctype.h	Standard header file
dsp.h	L-Intrinsics
float.h	Floating point support
libsdsp.h	SDSP-specific printing
limits.h	Standard header file
N_Intrinsic.h	N-Intrinsics
math.h	Math functions
q15.h	Support file

Table 2.19 Header Files Installed in `$SDSP_HOME/sdspI/include` (Cont.)

Filename	Function
`setjmp.h`	Standard header file
`simios.h`	
`stdarg.h`	
`stddef.h`	
`stdio.h`	
`stdlib.h`	
`string.h`	
`timer_util.h`	Timer functions

Table 2.20 Command-Line Tools Installed in `$SDSP_HOME/zspg2/bin`

Filename	Function
`readelf`	GNU utility for examining an object file
`zdelfread`	Produces a simple dump of entire contents of an object file
`zdar`	Archive utility
`zdas`	Assembler
`zdbug`	Source-level Debugger for the G2 architecture
`zdcc`	Compiler
`zdcc1`	Compiler
`zdcpp`	Preprocessor
`zdld`	Linker
`zdnm`	Symbol listing utility
`zdobjcopy`	Object file copying utility
`zdobjdump`	Object dump utility
`zdopt`	Optimizer

Table 2.20 Command-Line Tools Installed in $SDSP_HOME/zspg2/bin (Cont.)

Filename	Function
zdxbug	G1-G2 Cross Debugger
zdxcc	G1-G2 Cross Compiler
zdxcpp	G1-G2 Cross Preprocessor
zdxopt	G1-G2 Cross Optimizer
zdranlib	Random library (ranlib) utility
zdsize	Size utility
zdstrings	String print utility
zdstrip	Symbol discarding utility
zisimg2	Functional-accurate simulator for G2-based devices
zsimg2	Cycle-accurate simulator for G2-based devices

Table 2.21 Libraries Installed in $SDSP_HOME/zspg2/lib

Filename	Function
crt0.o	Startup file
libalg_zsp500.a	Basic signal processing functionality -- optimized for ZSP500 core.
libalg_zsp600.a	Basic signal processing functionality -- optimized for ZSP600 core.
libc.a	C library
libg.a	C library with debug information
libm.a	C library with long calls

Table 2.22 Libraries Installed in $SDSP_HOME/zspg2/libg1g2

Filename[1]	Function
crt0.o	Startup file
libc.a	C library
libg.a	C library with debug information
libm.a	C library with long calls

1. These files are referenced when the zdx* cross compilers are invoked.

Table 2.23 Header Files Installed in $SDSP_HOME/zspg2/include

Filename	Function
alg.h	Signal processing
assert.h	Standard header file
cbuf.h	Circular buffer
creg.h	Control registers
ctype.h	Standard header file
dsp.h	L-Intrinsics
float.h	Floating point support
libsdsp.h	SDSP-specific printing
limits.h	Standard header file
N_Intrinsic.h	N-Intrinsics
math.h	Math functions
q15.h	Support file

Table 2.23 Header Files Installed in `$SDSP_HOME/zspg2/include` (Cont.)

Filename	Function
`setjmp.h`	Standard header files
`simios.h`	
`stdarg.h`	
`stddef.h`	
`stdio.h`	
`stdlib.h`	
`string.h`	
`timer_util.h`	Timer functions

Table 2.24 Files Installed in `$SDSP_HOME/ide/bin/`

Filename	Function
`zspide`	IDE for the ZSP family of processors
`zspide_help`	Help file for the IDE
`guidebug_help`	Help file for the GUI debugger

For both the Windows and Solaris setups, additional files and directories are 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.

2.5 License Management

This section describes how to set up licensing for the SDK tools. As of Release 4.3, the SDK toolset is distributed under a license agreement. Licenses must be obtained from LSI before the SDK will function.

For license administration, please also refer to the *FLEXlm End User's Guide*, located on the distribution CD at `/doc/Flexlm_Enduser.pdf`.

2.5.1 Obtaining a License File

For SDK Tools to run, you must now obtain and install a license file.

Permanent license files are obtained directly by either FAX or email to `dsp-license@lsil.com`. To get the license, you must provide the identification for the machine that will be hosting the license manager (zspld). By default, the license manger daemon zspld and FLEXlm utilities program are installed in
`C:\Installation_Directory``\license`.

To obtain this identification string, log onto the machine that will be hosting the license manager and enter the following command:

```
lmutil lmhostid
```

Email the entire output along with the additional required information on the license request form to `dsp-license@lsil.com`. Alternatively, the information can be FAXed to the number on the form.

2.5.2 Starting the License Manager

The ZSP Tools license manager (`zspld`) is designed to run as a background task on one machine in your network as specified in your license file. Once invoked, it runs silently, monitoring and controlling the number of users on your network.

To start the license daemon, type:

```
lmgrd -c <License File> -l zsplic.log
```

where <License File> is the filename of the license file received from LSI.

To shut down the license manager, type:

```
lmdown -c <License File>
```

2.5.3 Setting Environment Variables

After starting the license server, set or append the hostname to the environment variable LM_LICENSE_FILE. This can be a license file name or port@host. For example, if the license manager "zspld" is run on a machine named "somemachine", set LM_LICENSE_FILE to "@somemachine".

License environment variables are set in two different ways:

1. The normal set or setenv commands (or the System Control Panel on Windows NT/2000/XP)
2. The registry ZSPLD_LICENSE_FILE (Windows v6.0+) or in $HOME/.flexlmrc (UNIX v7.0+), which functions like the registry for FLEXlm on UNIX.

On Windows, the FLEXlm registry location is:

```
HKEY_LOCAL_MACHINE\Software\FLEXlm License Manager
```

On UNIX, the equivalent information is stored in $HOME/.flexlmrc.

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` is 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 produces 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 -02 and -03.	sdcc zdcc zdxcc
-mlong_cond_branch	No effect. Retained for backward compatibility.	sdcc zdxcc
-mlong_uncond_branch	No effect. Retained for backward compatibility.	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
-msmall_data	Assume data and bss are placed in first 64K words. (default)	zdcc
-mlarge_data	Make no assumptions about data and bss.	zdcc
-mcheck_stack	Check if stack grows into heap space. If this occurs, print an error message and halt.	sdcc zdcc zdxcc
-m500	Generate code optimized for the ZSP500 core. (default)	zdcc
-m600	Generate code optimized for the ZSP600 core.	zdcc

1. Since the range of a call immediate on ZSPG2 is 16-bits versus 13-bits on ZSP400, the -mlong_call option is less frequently needed for zdxcc and zdcc.

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 either always requires `-mlong_call` or never requires 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`, converts 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 levels greater than -O1 are selected. Note that for debugging 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 is more compact and has 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

Option	Description
`-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 are based on the name of the input file: compiling `foo.c` with `-save-temps` produces `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.

3.2.1 Preprocessing Conventions

The preprocessing symbol **__ZSP__** is defined by sdcc, zdxcc and zdcc. The preprocessing symbol **__ZSP_G2__** is also defined by zdcc.

3.2.2 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 prints a 16-bit value in fixed-point notation. For example, the call:

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

prints:

```
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

Table 3.4 Compiler's Representation of C Data Types (Cont.)

C Data Type	Representation
accum_b	32 bits
float	32 bits
double	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.

3.2.3 Register Usage

3.2.3.1 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
- Scratch registers: r13, %cb0_beg, %cb0_end, %cb1_beg, %cb1_end, %loop0, %loop1, %loop2, %loop3
- Clobbered registers: %hwflag, %vitr
- 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.

3.2.3.2 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, %shwflag, %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, %cb0_beg-%cb3_beg, %cb0_end-%cb3_end
- Clobbered registers: %hwflag, %shwflag, %vitr
- Return registers: a0 for pointer values, r4 for 16-bit return values, and r5r4 for 32-bit non-pointer values.

Stack memory below the stack pointer, a7, may be used by interrupts. This includes the memory location pointed to by the stack pointer. Thus, the stack pointer must never point to memory that needs to be preserved. 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 ANSI C Code[1]		Required Entry Value		Required Value On Return From Call[2]		May be Modified Within Function	
		sdcc zdxcc	zdcc	sdcc zdxcc	zdcc	sdcc zdxcc	zdcc	sdcc zdxcc	zdcc
%amode	ld	yes		0		0		no	
	st	yes		0		0		no	
	cbX	n/a	yes	n/a	0	n/a	0	n/a	yes
%fmode	rez	no		x		x		yes	
	sat[3]	yes	no	0	x	x		yes	
	q15[4]	no		x		x		yes	
	sre[5]	yes		x		x		yes	
	mre[6]	no		x		x		yes	

Table 3.5 Effect of Mode Bits on Compiler-Generated Code (Cont.)

Mode Register	Mode Register Bit	Affects ANSI C Code[1]		Required Entry Value		Required Value On Return From Call[2]		May be Modified Within Function	
		sdcc zdxcc	zdcc	sdcc zdxcc	zdcc	sdcc zdxcc	zdcc	sdcc zdxcc	zdcc
%smode	shd[7]	yes	n/a	x	n/a	pre-serve	n/a	no	n/a
	lis	yes		0	x	0	pre-serve	no	
	sis	yes		0	x	0	pre-serve	no	
	cbX[8]	yes		0		0		yes	no
	dir[9]	yes		x		preserve		no	
	ddr[10]	yes		x		preserve		no	

1. ANSI C code does not use intrinsics, circular buffers, or the q15 data type.
2. This column describes the requirements on an assembly function called from C code.
3. With sdcc/zdxcc, the sat bit of %fmode can affect ANSI C code because of the add and subtract instructions. ANSI C code expects unsaturated arithmetic. If saturated arithmetic is required 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 the -minfer_mac option is used, the compiler also generates mac and macn instructions, which are also affected by the sat bit.
4. With sdcc/zdxcc, the q15 mode bit affects ANSI C code if the -minfer_mac option is used.
5. The sre bit of %fmode affects ANSI C code because of the shra and shra.e instructions. Only perform right shifts of signed variables when the sre bit is cleared.
6. With sdcc/zdxcc, the mre mode bit affects ANSI C code if the -minfer_mac option is used.
7. This bit is ZSP400 specific and selects/deselects the use of shadow registers. Compiled code operates correctly with either shadow registers or nonshadow registers.
8. 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 zdcc. When using sdcc/zdxcc, to prevent these bits from affecting ANSI C code, clear these bits when the portion of code requiring circular buffers is exited.
9. 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.
10. 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.

3.2.4 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 (left to right).

- A pointer value is passed in the first unused register in the following list: a0, a1, a6, r5r4, r7r6, r3r2.
- A 32-bit non-pointer value is passed in the first unused register in the following list: r5r4, r7r6, r3r2, a0, a1, a6.
- A 16-bit value is passed in the first unused register in the following list: r4, r5, r6, r7, r2, r3.
- All other non-structure parameters are passed on the stack.
- Structures larger than 32-bits are passed on the stack. All other structures are passed in the same manner as a non-pointer argument of the same size.
- 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)
```

expects its arguments to be in: r4, r7r6, and r5, respectively.

3.2.5 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

high address	Callee saved registers
	%rpc
	Local variables and temporaries
low address	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
	locals/temps of bar	
	max args of all functions called by bar	
low address	memory location pointed to by r12/a7 (stack pointer)	not part of any stack frame

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.

3.2.6 Example Code for Function Prologue and Epilogue

3.2.6.1 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 is 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 in this epilogue. 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
```

Some functions can restore registers without using r6. This is done by utilizing indexed loads. For example, a leaf function with r8 and r9 stored at stack offsets 1 and 2 can use the following epilogue:

```
ld     r8, r12, 1
ld     r9, r12, 2
ret
```

3.2.6.2 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 is 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
```

3.2.7 Parameter Passing Examples

3.2.7.1 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
```

3.2.7.2 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 are 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 produces 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 fills 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 at location __heap_start, which is placed after the bss section by default. The heap is not allowed to grow beyond __heap_limit. By default this symbol is set to the largest allowed address (0xFFFF for sdcc and zdxcc, and 0xFFFFFF for zdcc), so effectively the heap is not limited unless the user explicitly sets this symbol to a lower value. Note that the memory between __heap_start and __heap_limit is not reserved for the heap, the heap is just guaranteed not to take memory outside of that region. The stack grows from higher to lower addresses, and starts at the address specified by the predefined variable __stack_start – 1, which has a default value of 0xF7FF for sdcc/zdxcc and 0xFFEFFF for zdcc. The symbols controlling the stack and heap can be modified as shown below.

```
    sdcc -Wl,-defsym,__stack_start=0xd000,-
defsym,__heap_start=0xd001,-defsym,__heap_limit=0xd500
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. Run-time libraries are specific to a particular target and their locations are shown in Table 3.8.

Table 3.8 Run-time Library Location

Target	Compiler	Library Location
G1	sdcc	$SDSP_HOME/sdspl/lib
G1G2	zdxcc	$SDSP_HOME/zspg2/libg1g2
G2	zdcc	$SDSP_HOME/zspg2/lib

The SDK compilers automatically link with the version of the library that is appropriate for the intended target. Users who explicitly link in the run-time libraries must be sure to select the library from the correct location.

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.

- I/O functions (stdio.h)

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

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

- The filehandles stdin, stdout, and stderr are supported.
- Pseudo-random number generation functions (stdlib.h)

  rand, rand_r, srand, _lrand, _lrand_float

- Memory allocation functions (stdlib.h)

  calloc, free, malloc

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

Integral division is supported by routines in the run-time libraries. These routines generates a non-maskable interrupt and then halt if division by zero occurs.

In addition to the run-time library support, the header files, ctype.h and assert.h provide support for classifying characters and for debugging code.

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 cause the host program to perform the requested file I/O operation. All host programs and all zdbug targets support file I/O.

The following functions are supported by the floating-point math library, libm.a: `acos`, `asin`, `atan`, `atan2`, `ceil`, `cos`, `cosh`, `exp`, `fabs`, `floor`, `fmod`, `frexp`, `ldexp`, `log10`, `log`, `modf`, `sin`, `sinh`, `sqrt`, `tan`, and `tanh`.

3.5 Timer Support

The header file `timer_util.h` provides support for using the system timers. The ZSP400 and ZSPG2 architectures have two 16-bit timers, `%timer0` and `%timer1`. These timers are countdown timers and operate in two modes, **single-shot** and **auto-reload**. In single-shot mode, the timers count down and stop at 0. In auto-reload mode, the timers are reset to the last initialized value after 0 is reached. The rate at which the timers decrement is controlled by a **prescale divisor**. When the prescale divisor is set to *X*, the timer is decremented once every *X* clock cycles. The prescale divisor can be set to any value from 1-64. The complete interface for using the timers is:

`ZSP_timer_set(`**<timer>**`, unsigned int value)`

`ZSP_timer_mode(`**<timer>**`, `**<mode>**`)`

`ZSP_timer_prescale(`**<timer>**`, unsigned int prescale)`

`ZSP_timer_start(`**<timer>**`)`

```
ZSP_timer_stop(<timer>)
```

```
unsigned int ZSP_timer_read(<timer>)
```

The **<timer>** parameter must be TIMER0 or TIMER1. The **<mode>** parameter must be SINGLE_SHOT or AUTO_RELOAD.

The following example illustrates how to use this interface to time a section of code:

```
#include <stdio.h>
#include <timer_util.h>

main ()
{
  unsigned int timer;
  int a[4000],sum,i;

  ZSP_timer_mode(TIMER1,SINGLE_SHOT);
  ZSP_timer_set(TIMER1,60000);
  ZSP_timer_prescale(TIMER1,5);
  ZSP_timer_start(TIMER1);

  sum=0;
  for (i=0; i<4000; i++)
    sum+=a[i];
  ZSP_timer_stop(TIMER1);

  timer = ZSP_timer_read(TIMER1);
  printf("Elapsed %lu\n",((long)(60000 - timer))*5);
}
```

3.6 N-Intrinsics

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 3.9. The older L-intrinsics are still supported and are described in Appendix D, "L-Intrinsic Functions."

Table 3.9 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`

3.6.1 Vector N-Intrinsics

N-Intrinsics are also provided for common vector operations. They are shown in Table 3.10. The vector N-Intrinsics produce more efficient code than the equivalent non-vector code.

Table 3.10 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 is 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.

3.6.2 ETSI Functions

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 is 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 3.11.

Table 3.11 ETSI to N-Intrinsic Mapping

ETSI Function	N-Intrinsic	fmode[1] Register Bits			
		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
msu_r	N_macn	1	1	x	1
mult	N_mul	x	1	x	0

Table 3.11 ETSI to N-Intrinsic Mapping (Cont.)

ETSI Function	N-Intrinsic	fmode[1] Register Bits			
		sat	q15	sre	mre
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

3.7 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 is stored/accessed, but to the variables that are actually 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 does 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.

3.8 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:

- %fmode: MRE_MASK, SRE_MASK, Q15_MASK, SAT_MASK, REZ_MASK
- %amode: RCA_LD_MASK, RCA_ST_MASK, RCA_REV_MASK, CB0_MASK, CB1_MASK CB2_MASK, CB3_MASK
- %smode: DDR_MASK, DIR_MASK, SIS_MASK, LIS_MASK, US_MASK, UVT_MASK, DSB_MASK, ICT_MASK, FIE_MASK, DCT_MASK, LVL_MASK
- %imask: PGIE_MASK, GIE_MASK

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 appropriate 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);
```

3.9 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 prints a 16-bit value in fixed point notation.

3.10 Inline Assembly

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

3.10.1 Syntax

The basic syntax of the `asm` directive is:

```
asm(   "parameterized assembly" :

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

3.10.2 Parameterized Assembly

The *`parameterized assembly`* consists of a text string containing the desired assembly output with parameters that are replaced with registers or constants according to the arguments in *output variable* and *input expression*. The syntax of a parameter is shown in Table 3.12.

Table 3.12 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 is 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.

3.10.3 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 3.10.5, "Examples of asm Directive", the `acc` variable is constrained to be in an accumulator register. The supported constraints are shown in Table 3.13.

Table 3.13 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 is `rn`. In example 4 in Section 3.10.5, "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 3.10.5, "Examples of asm Directive", the output argument and the first argument illustrate this technique.

3.10.4 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.

3.10.5 Examples of asm Directive

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

3.10.5.1 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
```

3.10.5.2 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
```

3.10.5.3 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
```

3.10.5.4 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.

3.10.5.5 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.

3.10.6 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.

3.11 Assembly Optimizer and Handwritten Assembly

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

sdopt -asm *assemblyfile*

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

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

This is transformed by the assembly optimizer to:

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

Use the option “-asm_pe_only” if only prologue/epilogue generation is desired.

sdopt -asm_pe_only *assemblyfile*

or

zdopt -asm_pe_only *assemblyfile*

All registers that must be preserved according to the C calling convention are 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 is copied without modification.

3.12 Debugging Options

You can debug code compiled using the gcc-supplied `-g` option, which generates debugging information. Optimization levels -O0 and -O1 are fully compatible with debugging. At level -O0 no optimization is done -- at level -O1 only optimizations that preserve debugging capability are performed. At level -O2 debugging should only be done with the `-mno_sdopt` option because the assembly optimizers do not preserve the location of debugging information. Debugging at level -O2 can be confusing, since instructions may be re-ordered, dead-code may be eliminated, etc. However, in most cases the structure of the control-flow graph is preserved, so it is usually possible to use breakpoints to stop at a particular location in a program.

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.

3.13 Code Statistics

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

Every function has 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 gives the code size of a function. A function also has a label placed on its return instruction:

__FUNC_EXIT_<*function name*>

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

__FUNC_FRAME_SIZE_<*function name*>

3.14 Example Compilations

3.14.1 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.

3.14.2 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).

3.14.3 Example 3

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

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

3.14.4 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

This chapter describes the SDK Assembler. The chapter contains the following major sections:

- Section 4.1, "Introduction"
- Section 4.2, "Assembly Language Syntax"

4.1 Introduction

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).

4.2 Assembly Language Syntax

The basic format of an 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.

Files with the .S extension can be assembled using sdcc, which causes the C preprocessor to run before the assembler. This enables you to use C-style comments and #defines in your assembly code. However using a -g option does 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 , "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. Refer to *Using AS: The GNU Assembler* for more information on the section syntax and flag definitions.

4.2.1 Assembler Options

Please refer to *Using AS: The GNU Assembler* 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 following.

Suppress warnings (-W) –

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

Output file (-o) –

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

Include path (-I) –

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

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.

ELF Flag (--defsym g1g2=1) –

This option is only available for **zdas**. When this option is used, the resulting object file has an ELF flag of 0x1000.0000, the flag setting for G1G2 programs. If this option is not set, the resulting object file has an ELF flag of 0x2000.0000, the flag setting for ZSPG2 programs.

4.2.2 Assembler Directives

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

.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.
```

.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 is filled with zeroes.

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

.word –

The .word directive allows you to specify zero or more comma-separated values to be assembled into memory.

.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.

.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. Refer to *Using AS: The GNU Assembler* 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.

4.2.3 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 is not converted to an even register and an error message is 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.

4.2.4 ELF Number and Flags

All ZSP400, ZSPG2, and G1G2 object files and programs have an ELF number of 79. This number is automatically created by the assemblers and linkers. ZSP400 object files and programs have an ELF flag of 0x8000.0000. This is automatically generated by **sdas** and **sdld**. ZSPG2 object files and programs have an ELF flag of 0x2000.0000. This is generated by default by **zdas** and **zdld**. G1G2 object files and programs should have an ELF number of 0x1000.0000. Object files will have this flag only if **zdas** is invoked with a "`--defsym g1g2=1`" option. A program has a G1G2 flag if any of the modules on the link line have a G1G2 flag. The G1G2 compiler, **zdxcc**, automatically uses this option when invoking **zdas**. Assembly files for G1G2 programs that are not assembled with the "`--defsym g1g2=1`" option produce object files with the G2 flag. However, these inappropriately flagged object files can still be used to produce a G1G2 executable.

The ZSP400 linker, **sdld**, produces an error message if any module does not have an ELF number of 79 or if the ELF flag is not 0x8000.0000. The ZSPG2 linker, **zdld**, produces an error message if any module does not have an ELF number of 79 or if the ELF flag is not 0x2000.0000 or 0x1000.0000.

Chapter 5
Linker

This chapter describes the SDK Linker. The major sections in the chapter are:

5.1 Introduction

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
```

5.2 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 generates 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 automatically aligns `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.

5.3 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 causes execution to begin at address 0xABCD:

```
sdld -e 0xabcd
```

The C stack and heap always lie in internal data memory.

- __stack_start: beginning of C stack, default setting is 0xF7FF with sdld and 0xFF.EFFF with zdld.
- __stack_size: user_required. Is an optional symbol that you set, describing the amount of stack space that is required. If this symbol is set, it ensures that the required stack space starting from __stack_start is not allocated to other sections (e.g., data, BSS). Setting this variable does not prevent the heap from growing into this area.
- __heap_start: starting address of C heap. The default value is the end of the BSS, ___bss_end.
- __heap_limit: limit which heap will not grow beyond. Default setting is 0xFFFF with sdld and 0XFF.FFFF with zdld. The space between __heap_start to __heap_limit is not reserved for the heap. The stack can still grow into this area. These values only guarantee that the heap does not grow out of this area.

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

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

5.4 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)}
}
```

You need to supply the previous linker command file to the linker through the ‘-T’ option. Otherwise, it uses the default linker script file.

The previous example 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.

5.5 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 searches 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.
--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`*.

5.6 ELF Number and Flags

All ZSP400, ZSPG2, and G1G2 object files and programs have an ELF number of 79. This number is automatically created by the assemblers and linkers. ZSP400 object files and programs have an ELF flag of 0x8000.0000. This is automatically generated by **sdas** and **sdld**. ZSPG2 object files and programs have an ELF flag of 0x2000.0000. This is generated by default by **zdas** and **zdld**. G1G2 object files and programs should have an ELF number of 0x1000.0000. Object files have this flag only if **zdas** is invoked with a "`--defsym g1g2=1`" option. A program has a G1G2 flag if any of the modules on the link line have a G1G2 flag. The G1G2 compiler, **zdxcc**, automatically uses this option when invoking **zdas**. Assembly files for G1G2 programs that are not assembled with the "`--defsym g1g2=1`" option produce an object file with the G2 flag. However, these inappropriately flagged object files can still be used to produce a G1G2 executable.

The ZSP400 linker, **sdld**, produces an error message if any module does not have an ELF number of 79 or if the ELF flag is not 0x8000.0000. The ZSPG2 linker, **zdld**, produces an error message if any module does not have an ELF number of 79 or if the ELF flag is not 0x2000.0000 or 0x1000.0000.

Chapter 6
Utilities

This chapter describes the SDK utility programs. The chapter contains the following major sections:

6.1 Introduction

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.
`readelf`	readelf	Displays the contents of ELF format files.
`sdelfread` `zdelfread`	none	Displays two sections '.text' and '.data' in hex.

6.2 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.

Table 6.2 sdar p Keyletter Options (Cont.)

Option	Description
`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 normally permits file names of any length. This causes 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 want to insert only 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

6.3 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. 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. Inappropriate use of this option may make the output file unusable.
`-s | --strip-all`	Removes all symbols.
`-S | -g | --strip-debug`	Removes debugging symbols only.

Table 6.4 sdstrip Options (Cont.)

Option	Description
`-N` *`symbolname`* | `--strip-symbol=`*`symbolname`*	Removes symbol *`symbolname`* from the source file. You may give this option more than once, and combine it 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.

6.4 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.

6.5 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.

6.6 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 is 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
```

6.7 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.

6.8 sdobjdump

Format –

```
sdobjdump
   [ -d | --disassemble ]
   [ -f | --file-headers ]
   [ -j section | --section=section ]
   [ -h | --section-headers ]
   [ -s | --full-contents ]
   [ -t | --syms ]
   [ --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.
-s | --full-contents	Display the full contents of any section requested in hex.

6.9 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 essentially produces a memory dump of the contents of the input object file. All symbols and relocation information are discarded. The memory dump starts 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’ is 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.

6.10 readelf

Fomat –

```
readelf
   [ -h | --file-headers ]
   [ -v | --version ]
   [ -H | --help]
```

Description –

`readelf` displays information about one or more ELF format object files. The options control what particular information to display. elffile... are the object files to be examined.

The long and short forms of options, shown here as alternatives, are equivalent.

Options –

Table 6.10 elfread Options

Options	Description
`-h	--file-header`	Displays the information contained in the ELF header at the start of the file.
`-v	--version`	Displays the version number of readelf.
`-H	--help`	Displays the command line options understood by readelf.

Chapter 7
ZSP SDK Functional-Accurate Simulator

This chapter describes the ZSP SDK functional-accurate simulator (ZISIM).

ZISIM simulates the behavior of the ZSP400 and ZSPG2 cores at the architectural level, including the memory model, the operand register file, and the control register file.

This chapter contains the following major sections:

7.1 Using ZISIM

ZISIM 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. Table 7.1 shows available simulators.

Table 7.1 Functional-Accurate Simulators

Name	Use when simulating...
zisim400	code written for ZSP400 architecture.
zisimg2	code written for ZSPG2 architecture.

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]
zisimg2 executable_file -exec [options]
```

The simulator can also be invoked in batch mode using a script file containing ZISIM 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
or
zisimg2 -s batch.scr > test.trace
```

Refer also to Section 7.2.21, “script,” page 7-16.

7.1.2 Interactive Mode

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

`zisim400 [executable_file] [options]`

or

`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 7.2). If no options are specified, ZISIM initializes itself, then displays 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 as the target of the SDK's Debugger in Section 9.2.1, "Functional-Accurate Simulator Connection."

Table 7.2 ZISIM Command-Line Options

Option	Description
`-c` *NUM*	Limits number of executed instructions to *NUM*. By default, *NUM* = 2,000,000,000. 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. 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.
`-ignore`	Ignore run-time warning messages such as uninitialized memory accesses, invalid circular buffer size.
`-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.

Table 7.2 ZISIM Command-Line Options (Cont.)

Option	Description
-reg	Enables register trace. All the architectural registers are 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 7-25.
-exec	Invokes the simulator in noninteractive mode.
-v	Prints version number and exit.
-cl arg1 ... argn	Pass any command line arguments after -cl to the program.

7.2 ZISIM Commands

This section describes commands recognized by the ZISIM command line. Table 7.3 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 7.3 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.

Table 7.3 ZISIM Command Summary (Cont.)

Command	Modifier	Argument	Description
disable	break	*breakpoint*	Disables specified breakpoint.
	trace	{mem | reg | write}	Disables run-time instruction tracing.
	warning	-	Disables run-time warning messages such as uninitialized memory accesses or invalid circular buffer size.
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.
	warning	-	Enables run-time warning messages such as uninitialized memory accesses or invalid circular buffer size.
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*.
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.

Table 7.3 ZISIM Command Summary (Cont.)

Command	Modifier	Argument	Description
set	args	*arg1 arg2 ... argn*	Pass arg1 to argn to the program as run-time arguments.
	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.
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*.

Table 7.4 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 7.5 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 0x00FF.FFFF words.
show	size	*[dmem|imem] [int|ext]*	Show the current size of internal/external instruction or data memory.

7.2.1 alias

This 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. You specify 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. You specify 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. You specify 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. You specify 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 runtime tracing. Run-time traces generate text output instruction by instruction. The three trace types are:

- write

  Displays architectural state changes associated with memory or registers for each instruction in the following formats:

  (seqID) PC Opcode Instruction ! register=value

  (seqID) PC Opcode Store Instruction ! [Memory-Address]=value

  (seqID) PC Opcode Load Instruction ! register=value [Memory-Address]

  (seqID) PC Opcode Branch Instruction ! direction, result

  seqID: unique ascending sequence number for each instruction.

  PC: address of instruction in memory.

  Instruction: disassembled instruction.

  Register: architecture register name.

  Direction: direction for a discontinuity instruction such as branch or conditional execution. Direction is either forward, or backward and the result is either taken or not taken.

  For example,

  ```
  (1) 0x000002 6200 mov %fmode, r0      !fmode=0x0014
  ```

  Instruction `mov %fmode, r0` modifies `%fmode` to value 0x0014.

- `mem`

  Displays address and data for any memory location which is updated. Information is generated after the instruction is executed. This option is a subset of 'enable trace write' because it does not display register updates.

- `reg`

  Displays all registers and register values for every 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. You specify 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 a specified text file. You specify 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 is loaded, with each data on a single line. Data must be in hex format without 0x prefix. Comments must be enclosed by `/* */` characters.

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 a 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 is loaded, with each data on a single line. Data must be in hex format without the 0x prefix. Comments must be enclosed by `/* */` characters.

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
```

The `reset` command does not reload the program into memory. To restart the program, perform one of the following sequences of commands:

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

or

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

Note: `zisimg2` does not support the soft reset 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 the 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 following.

```
# 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

This command allows you to set three internal ZISIM variables. Table 7.6 shows the configurable ZISIM attributes.

Table 7.6 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 2,000,000,000.

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 a 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

This command is slightly different for the two ZSP architectures.

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 0xFF.FFFF words (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 and 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 7.7.

Table 7.7 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, you 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. You specify 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 7.8.

Table 7.8 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, you 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

Like `set size`, this command is slightly different for the two ZSP architectures.

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 7.9 shows the memory addresses and corresponding files for the I/O ports for the LSI402ZX, LSI403Z, and ZSP400-core based devices.

Table 7.9 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 prints an error message to standard output and exits.

7.4 Example Session Using ZISIM

This section contains an example simulation session using ZISIM400 in interactive mode. A simulation session using zisim for other architectures is similar.

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 following.

```
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 following 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 no longer prints 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 sets 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
ZSP SDK Cycle-Accurate Simulator

This chapter describes the ZSP SDK Simulator (ZSIM).

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.

This chapter contains the following major sections:

8.1 Using ZSIM

ZSIM can be accessed either 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 8-6. For the debugger target ZSIM, see Chapter 9, “Debugger.“ Table 8.1 shows available cycle-accurate simulators.

Table 8.1 Cycle-Accurate Simulators

Name	Use when Simulating...
zsim400	code written for ZSP400 architecture.
zsimg2	code written for ZSPG2 architecture.

8.1.1 Batch Mode

The simulator can be invoked in batch mode from the command line using the `-exec` option, as shown following:

```
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 following:

```
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
```

or

```
zsimg2 -s batch.scr > test.trace
```

Refer also to Section 8.2.26, “script,” page 8-26.

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 command line options are processed on a first-come, first-serve basis.

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 8.2).

If no options are specified, ZSIM initializes itself, then displays 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 8-6. 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 8-40. Refer also to the description of using ZSIM as the target of the SDK Debugger in Section 9.2.2, "Cycle-Accurate Simulator Connection," page 9-4.

Table 8.2 ZSIM Command-Line Options

Option	Description
-exec	Invokes the simulator in noninteractive mode.
-c *num*	Specifies maximum cycle count. Execution terminated after *num* cycles.
-h	Prints brief usage summary.
-i *mode_register=value*	Initializes an architectural control (mode) register with the 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 the %smode control register is: `-i smode=0x1234`. The option to set the r0 register is: `-i r0=0x9876`. Refer to Appendix B, "ZSP400 Control Registers," for information on ZSP400 core-based device control registers or Appendix C, "ZSPG2 Control Registers,"
-ignore	Ignores run-time warning messages such as uninitialized memory accesses, invalid circular buffer size.
-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.
-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.
-cl arg1 ... argn	Passes any command-line arguments after -cl to the program.

Table 8.3 Command-Line Options Specific to 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 8.4 Command-Line Options Specific to zsimg2

Options	Description
-pflsu	Turns on Load Store Unit profile information.
-tic	Turns on instruction cache trace every cycle.
-svtadd ADDR	Sets system vector table address to be ADDR.
-idealmss	Uses ideal memory subsystem with zero delay for internal memory and no checking for banking conflict between two data access ports.
-bimlib LIBNAME	Uses bus interface library LIBNAME to run in cosimulation environment such as SWIFT or CVE Seamless.
-msslib LIBNAME	Uses a different memory subsystem library LIBNAME other than default. The default library is `libzmiug2`.
-cpilib LIBNAME	Uses coprocessor library LIBNAME. SDK tools come with an example G711 coprocessor library called `libzcpig711.so` on Solaris or `libzcpig711.dll` on Windows platforms.

8.2 ZSIM Commands

The ZSIM commands are described briefly in Table 8.5 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 8.5 ZSIM Command Summary

Command	Modifier	Argument	Description	400	G2
`alias`	–	[*tag command_sequence*]	Creates alias (*tag*) for command sequence.	x	x
`clear`	break	*breakpoint_number*	Clears specified breakpoint.	x	x
	dcache	–	Invalidates data cache.	x	
	dmem	{int | ext} *addr size*	Clears internal or external data memory.	x	x
	icache	–	Clears instruction cache.	x	
	imem	{int | ext} *addr size*	Clears internal or external instruction memory.	x	x
	pipe	–	Invalidates the pipeline.	x	x
	stats	–	Clears run-time statistics.	x	x
		opcode	Clears opcode run-time statistics.	x	

(Sheet 1 of 6)

Table 8.5 ZSIM Command Summary (Cont.)

Command	Modifier	Argument	Description	400	G2
disable	break	*breakpoint_number*	Disables specified breakpoint.	x	x
	profile	du	Disables data unit profile information.	x	
		iu	Disables instruction unit profile information.	x	x
		lsu	Disables load store unit profile information.		x
		pipe	Disables pipeline unit profile information.	x	x
		resource	Disables resource profile information		x
	trace	{write | pipe | reg|mem}	Disables run-time tracing.	x	x
		icache	Disables instruction cache run-time tracing.		x
	warning	–	Disables run-time warning messages such as uninitialized memory accesses or invalid circular buffer size.	x	x
dump	dmem	{int | ext} *filename addr size*	Dumps internal or external data memory to a text file *filename*.	x	x
	imem	{int | ext} *filename addr size*	Dumps internal or external instruction memory to a text file *filename*.	x	x
(Sheet 2 of 6)					

Table 8.5 ZSIM Command Summary (Cont.)

Command	Modifier	Argument	Description	400	G2
enable	break	*breakpoint_number*	Enables breakpoint.	x	x
	profile	du	Enables data unit profile.	x	
		iu	Enables instruction unit profile.	x	x
		lsu	Enables load store unit profile.		x
		pipe	Enables pipeline unit profile.	x	x
		resource	Enables resource profile.		x
	trace	{mem | pipe | reg | write}	Enables run-time cycle tracing.	x	x
		icache	Enables instruction cache run-time tracing.		x
	warning	–	Enables run-time warning messages such as uninitialized memory accesses or invalid circular buffer size.	x	x
exit	–	–	Exits simulation session.	x	x
fill	dmem	{int | ext} *addr size value*	Fills internal/external data memory segment with *value*.	x	x
	imem	{int | ext} *addr size value*	Fills internal/external instruction memory segment with *value*.	x	x
help	–	{*category* | *command*}	Prints list of commands in a category or command usage.	x	x
istep	–	[number_of_instructions]	Advances the simulator by one instruction for zsim400. For G2, you can specify the number of instructions.	x	x
load	dmem	{int | ext} *filename addr*	Loads internal/external data memory from file.	x	x
	exe	*filename*	Loads ZSP executable into instruction memory.	x	x
	imem	{int | ext} *filename addr*	Loads internal/external instruction memory from file.	x	x
reset	hard	{hard | soft}	Resets simulator (hard or soft – is only applied for zsim400).	x	x
(Sheet 3 of 6)					

Table 8.5 ZSIM Command Summary (Cont.)

Command	Modifier	Argument	Description	400	G2
run	–	[*number_of_cycles*]	Runs for specified number of simulation cycles.	x	x
script	–	*filename*	Loads and executes ZSIM script file.	x	x
set	args	*arg1 arg2 ... argn*	Passes arg1 to argn to the program as run-time arguments.	x	x
	attr	{history | radix | run} *value*	Assigns *value* to specified attribute.	x	x
		addrwidth *value*	Assigns value from 1 to 32 to address width. Default is 24 bits.		x
	break	pc *addr*	Creates a new breakpoint at the specified PC address.	x	x
	break	symbol label	Creates a new breakpoint at the specified label.	x	x
	delay	[edata|einst] num	Sets wait state for external memory. Default for both external data and instruction memory is 1.	x	
	latency	[dmem|imem] [int|ext] num	Sets wait state latency for internal/external instruction or data memory. Default value for internal memory is 1 and external memory is 5.		x
	reg	*register value*	Assigns *value* to specified register.	x	x
	size	*[dmem|imem] size*	Sets internal instruction or data memory size starting from 0. Default size is maximum value of 0xF800 words.	x	
		[dmem|imem] [int|ext] beg_value end_value	Sets 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 0x00FF.FFFF words.		x
(Sheet 4 of 6)					

Table 8.5 ZSIM Command Summary (Cont.)

Command	Modifier	Argument	Description	400	G2
show	attr	{history | radix | run | version}	Shows value of the specified attribute.	x	x
	bits	*register*	Displays the bit-level states for the specified register.	x	x
	break	–	Shows list of defined breakpoints.	x	x
	dcache	–	Show data cache contents.	x	x
	dmem	{ int | ext} *addr size*	Shows contents of a region of internal/external data memory.	x	x
	icache	–	Shows current instruction cache contents.	x	x
	imem	{int | ext} *addr size*	Shows contents of a region of internal/external instruction memory.	x	x
	operand	instruction_number	Shows operand values of an instruction currently in the pipe. Instructionnumber can be obtained by looking at the output of show pipe command.		x
	pred	–	Shows static branch prediction table.		x
	pipe	–	Shows contents and state of execution pipeline.	x	x
	profile	– contents of register or registe	Displays supported profile information.	x	x
	reg	{*category* | *reg*}...	Shows contents of register or register set.	x	x
	rule	–	Shows the affected grouping rule in the current cycle.	x	x
	size	{dmem | imem} [int|ext]	Shows size of data or instruction memory.	x	x
	stats	<incremented | opcode>	Shows current run-time statistics.	x	x
		grouping	Displays the statistic of grouping rule.		x
	trace	–	Shows the current status of all tracing attributes.	x	x
(Sheet 5 of 6)					

Table 8.5 ZSIM Command Summary (Cont.)

Command	Modifier	Argument	Description	400	G2
`step`	–	–	Advances simulation by one cycle. Same as `run 1`.	x	x
`unalias`	–	*`alias`*	Deletes *`alias`*.	x	x
(Sheet 6 of 6)					

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 for ZSP400.

Format –

```
clear dcache
```

Example –

zsim{32} **clear dcache**

8.2.4 clear dmem

This command clears the contents of internal or external data memory. You specify 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. You specify 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 the specified type of profile information. If you do not specify a profile type, the command disables all types. Profile types are described in Section 8.2.14, “enable profile,” page 8-16.

Format –

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

Examples –

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

8.2.10 disable trace

This command disables the specified type of trace. Trace types are described in Section 8.2.15, “enable trace,” page 8-19.

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 listing the contents of the specified address range of the internal or external data memory. Parameters are internal or external memory, file name, 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 listing the contents of the specified address range of the internal or external instruction memory. Parameters are internal or external memory, filename, starting address, and 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 8-27, 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 four types of predefined runtime tracing:

- du

  Displays information from the data unit of the ZSP400 architecture, such as data cache hits and the du_imem_read signal. This is not valid for G2.

- iu

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

  For zsim400, the output looks like:

  ```
  cycle# -   IU_IMEM_READ : address
  cycle# -   ICACHE hits   : address
  ```

  For zsimg2, the output looks like:

```
1       - PFU_COND: NO_VAL_INST_FETCH_PC cond_abort_mode=0
1       - PFU_PC: 0x00f800 pf_addr:0x00f800
1       - PFU_AGN: p0:0, c0:0, a0:0, p1:0, c1:0, a1=0
1       - PFU_cl_unv[1 0 0 0 0 0 0 0] cl_disc_unv[1 0 0 0 0 0 0 0]
1       - PFU C:d1=-1, C:d2=-1, C:d3=-1, C:d4=-1, N:d1=0, N:d2=0, N:d3=0, N:d4=0
1        Grouping Rule: none (0 instructions in G stage).
```

The first column displays the cycle count. The PFU_COND line displays the condition of the PFU state machine and the mode it operates in. The possible states are shown in Table 8.6. The field cond_abort_mode=1 means the PFU receives the reqi_cond_abort signal from the memory subsystem. The PFU_PC line displays the current PC address coming from the ISU (Instruction Sequencing Unit). The PFU_AGN line displays agn status for the loop awakening logic. The next line shows the state of the cl_unv (cache-line unavailable) and cl_disc_unv (cache-line discontinuity unavailable) bits. The last line displays cache line pointers associated with the state machine.

Table 8.6 PFU State Machine

Cond	Description
DEFAULT	Default condition.
WAIT_ON_MSS_RETRY	Prefetch queue is full or memory subsystem asserts retry for a request.
MID32_FETCH_PC	Current PC lands into the middle of a 32-bit instruction.
STRAD_NIC_FETCH_PCP8	The second half of a 32-bit instruction that straddles a cache line is not in cache.
NO_VAL_INST_FETCH_PC	Current PC is not in cache.
PCP8_IC_NO_PREFETCH	Current PC+8 is in cache, no need to prefetch.
PCP8_NIC_PREFETCH	Current PC+8 is not in cache, prefetch that address.
PCP8_NIC_CLUNV_NO_PREFECH	Current PC+8 is not in cache. Machine can not prefetch because that line is unavailable.
VDISC_WAIT	Wait for a loop or register based discontinuity.
VDISC_IC_NO_PREFETCH	Target of an immediate discontinuity is in cache, no need to prefetch.
VDISC_NIC_PREFETCH	Target of an immediate discontinuity is not in cache.
VDISC_NIC_CLUNV_NO_PREFETCH	Target of an immediate discontinuity is not in cache, but it maps to a line that is not available

- `pipe`

  Displays information from the pipeline unit, such as cycle-by-cycle grouping rule information of instructions issued in the G stage.

```
13      (9) 0x0000000e movlw     a4, 0x20                      AGU0
13      (10) 0x00000010 movhw     a0, 0x0                      AGU1
13      Grouping Rule: 19.1 (2 instructions in G stage).
```

The first number is the cycle count. The second number, in parentheses, is the instruction sequence number. The third number, in hexadecimal, is the instruction address. The last column shows the unit in which the instruction is executed.

- `resource`

  Displays information on resource usage of the AGU, ALU, and MAU units for the G2 simulator. The output has the following format:

  `cycle# - insts:#, AGU:#, ALU:#, MAU:#, words transferred:#`

  or

  `cycle# - GR stalls.`

  For the first format line, the first number is the cycle count. The second field displays the number of issued instructions. The third field displays the number of AGU units that are being used. The forth field displays the number of ALU units that are being used. The fifth field displays the number of MAU units that are being used. The sixth field indicates the number of words that are being transferred to or from memory.

  The resource information is collected in the GR stage of the pipeline. The second format line is the output when the GR stage is stalled.

- `lsu`

  Displays information from the load/store unit of ZSPG2 architecture. The first two fields have the following format:

  <cycle#> - <port#>

  The *<cycle#>* field describes the cycle when a transaction is being made. The *<port#>* describes on which port the request is being made. The subsequent fields will be of one the following formats:

```
lsu_mss_req_read <addr> <size> [<pf direction>] <cond> <status> <insn#>
lsu_mss_req_write<addr> <size> <status> <insn#>
lsu_mss_send_data<addr> <insn#>
lsu_mss_get_data<addr> <size> <data> <status> <insn#>
lsu_mss_cond_abort(<cond-cycle#>) <insn#>
```

  The first field describes the action being requested. The possible actions are described in Table 8.7. The *<addr>* field describes the address of the action. The *<size>* field describes the size of the request. The *<pf direction>* field describes the direction of the prefetch. The *<condition>* field describes whether the transaction is conditional or unconditional. The *<status>* field shows if the request was accepted by the MSS (memory subsystem) or needs to be retried. The *<insn#>* field is the sequence number of the instruction associated with the action. The *<cond-cycle#>* field shows in which cycle the core received an abort signal from the MSS.

Table 8.7 LSU Output Description

TAG	Description
lsu_mss_req_read	Request a read from MSS.
lsu_mss_req_write	Request a write to MSS.
lsu_mss_send_data	Send data for req_write.
lsu_mss_get_data	Receive data for req_read.
lsu_mss_cond_abort	Receive cond_abort signal from the MSS to abort any requests in previous and current cycle.

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 runtime tracing:

- write

  Displays architectural state changes associated with memory or registers for each cycle in the following format:

```
<cycle> (seqID) PC Opcode Instruction ! register=value
<cycle> +=+=+=+=+=+=+=+=+=+=+=+ ! register=value
<cycle> (seqID) PC Opcode Store Instruction ! [Memory-Address]=value
<cycle> (seqID) PC Opcode Load Instruction ! register=value [Memory-Address]

<cycle> (seqID) PC Opcode Branch Instruction ! direction, result
```

cycle: Cycle count that the register is modified.

seqID: Unique ascending sequence number for each instruction.

PC: Address of instruction in memory.

Instruction: Disassembled instruction.

Register: Architecture register name.

Direction: Direction for a discontinuity instruction such as branch or conditional execution. Direction is either forward or backward, and the result is either taken or not taken.

+=+=: A register is modified without any associated instruction such as when an interrupt is taken or a timer enable mode.

For example:

```
<13> (1) 0x000002 6200 mov %fmode, r0 !  fmode=0x0014
```

Instruction `mov %fmode, r0` modifies `%fmode` to value 0x0014 at cycle 13.

- `mem`

  Displays address and data for any memory location which is updated. Information is generated in the cycle in which the write occurs. This option is a subset of 'enable trace write' because it does not display register updates.

  ```
  <cycle> (seqID) PC Opcode Instruction [Memory
  Address]=Value
  ```

  For example:

  ```
  <99> (255) 0x00006d 1884 stu r0, a4, 1
  ! [0x00000024]=0x9966
  ```

  Instruction `stu r0, a4,1` writes value 0x9966 to memory location 0x24 at cycle 99.

- `icache`

  Displays the entire instruction cache in every cycle. See `show icache` command for output description. This command is valid only for G2.

- `pipe`

  Displays the entire pipeline in every cycle. See `show pipe` command for output description

- `reg`

  Displays all registers and values in every 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 the internal or external data memory range with the 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 the internal or external instruction memory range with the 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, you can specify the 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 are loaded, with each data on a single line. Data must be in hex format without the `0x` prefix. Comments must be enclosed by `/* */` characters.

Format –

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

Example –

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

The 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

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

8.2.23 load imem

This command loads internal or external instruction 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 is loaded, with each piece of data on a single line. Data must be in hex format without the `0x` prefix. Comments must be enclosed by `/* */` characters.

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. 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 no longer supports the soft reset feature.

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 8-26). Simulation halts if the 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 a 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 and profile using
# a command file.
load test.exe
enable profile pipe # turn on grouping rule info
enable trace write # turn on instruction trace info
enable trace pipe # turn on pipeline info
run
exit
```

Note: The same script can be invoked as a command-line argument to the simulator as shown following.

```
%zsim400 -s standard.scr
```

or

```
%zsimg2 -s standard.scr
```

8.2.27 set attr

This command allows you to set three internal ZSIM attributes. These configurable attributes are described in Table 8.8.

Table 8.8 Configurable ZSIM Attributes

Attribute	Value	Description	400	G2
history	any integer	Number of commands to maintain in history buffer.	x	x
radix	{dec | hex}	Radix (decimal or hexadecimal) used to generate output.	x	x
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.	x	x
addrwidth	any integer from 1 to 32	Number of bits in address bus for G2 architecture.		x

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 the 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 (for zsim400 only)

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
```

8.2.30 set latency (for zsimg2 only)

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 2 cycles 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
```

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

The format of this command is different for the two simulators.

8.2.32.1 set size for 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 (62 Kwords), which is also the maximum size that can be set.

Important: This command does not apply to external memory. (The simulator has 1 Mwords 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 set size for 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 0x00FF.FFFF words (16 Mwords). 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. The `version` and `pred` attribute can be used only with the show `attr` command; they can not be used with the `set attr` command.

Format –

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

Example –

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

8.2.34 show bits

This command displays the bit and 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 the 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 nine lines are dedicated for the linked loads of the r13, 14, and 15 registers respectively. The next eight lines are used for any unlinked load. For each line, the first "------" column shows the address of the line. The next column indicates that the line is invalid (I) or valid. The next four columns show the data contained in that line.

- For the zsimg2 simulator without invoking the option *-idealmss*:

```
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 second column shows the address tag of the line and the next eight columns contain data. Address tag 0x000001 means invalid address tag. '-----' means the cache line is empty. The example shows the initial state of the cache. The symbol `lru[0]` indicates the least recently used cache line.

8.2.37 show dmem

This command displays a range of internal or external data memory. You 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 8.9.

Table 8.9 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, you 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 8.10.

Table 8.10 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, you can use a symbol instead of the 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. An instruction in the pipeline is represented in the following format:

(seqID) Address:Opcode:IssueBit:Disassembled instructions,

where:

- *SeqID*: Unique ascending sequence number for each instruction.
- *Address*: Address of the instruction in memory.
- *Opcode*: Binary opcode of an instruction in hexadecimal digit.
- *IssueBit*: Instruction is issued to the next stage in the following cycle.

For zsim400, the output displays a five-stage pipeline

```
CYCLE: 0
   ------------------------------------------ F(0:0)
   ------------------------------------------ G(0:0)
   ------------------------------------------ R(0:0)
   ------------------------------------------ E(0:0)
   ------------------------------------------ W(0:0)
```

stage(#:#): 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 advance to the next stage in the following cycle.

For zsimg2, the output displays an eight-stage pipeline.

```
CYCLE: 0 <stall>
   ------------------------------------------ FD(0:0)
   ------------------------------------------ GR(0:0)
   ------------------------------------------ RD(0:0)
   ------------------------------------------ AG(0:0)
   ------------------------------------------ M0(0:0)
   ------------------------------------------ M1(0:0)
   ------------------------------------------ EX(0:0)
   ------------------------------------------ WB(0:0)
```

stage(#:#): eight stages of the execution pipeline are identified with double letters – FD (Fetch/decode), GR (Group), RD (Read), AG (Address Generation), M0 (Memory stage 0), M1 (Memory Stage 1), EX (Execute), and WB (Write Back) – followed by two integers representing the number of instructions currently in that stage and the number of instructions that advance to the next stage in the following cycle.

The `<stall>` field next to the cycle number indicates a stall has occurred in the current cycle. Table 8.11 shows all three possible stalls for G2.

Table 8.11 Pipe Stall Description

Stall	Description
Pipe stalls by instruction #	Full pipe stall occurs by the indicated instruction number.
Half pipe stalls AG	Pipe stalls from AG and up.
Half pipe stalls M0	Pipe stalls from M0 and up.

Format –

```
show pipe
```

Example –

```
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
```

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

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}{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

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

Format –

```
show stats
```

Example –

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

8.2.46 show trace

This command shows 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

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

Format –

```
step
```

Example –

```
zsim{32} step
```

8.2.48 unalias

This command 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 8.12 shows the memory addresses and corresponding files for the I/O ports for the LSI402ZX, LSI403Z, and ZSP400-core based devices.

Table 8.12 I/O Device Memory Map and Associated Files

I/O Port	Read		Write	
	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 piece of 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 prints an error message to standard output and exits.

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 zsim400 in interactive mode. A simulation session using ZSIM for other architecture is similar.

```
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 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 (four word boundary) and the remaining four 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 (four word boundary) and the remaining four 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 following demonstrates the 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 debuggers are 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 the SDK debuggers in this chapter, for the most part, includes only the differences from gdb.

For Windows 98/NT/2000/XP platforms, the debuggers can be accessed using the ZSP Integrated Development Environment, as described in Chapter 11, "ZSP Integrated Development Environment." This chapter describes the debuggers' standard GNU command-line interface, available for all platforms.

9.1 Using the Debugger

The debugger 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.

Command-line options only available in the SDK debuggers are listed in Table 9.2. All other options are described in Stallman, *et. al.*

Table 9.2 Special 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, zdxbug zdbug
-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/sdspl/misc.	sdbug400, zdxbug, zdbug

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

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

Next, select the debugger'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 standard gdb commands.

9.2 Debugger 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 probe (Windows 98/NT/2000/XP 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 7.2 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.

Target commands that change the behavior of the subordinate ZISIM instance controlled by the SDK debugger are listed in Table 9.3 and described in detail in Section 7.2, "ZISIM Commands," page 7-4.

The format for sending commands to ZISIM is:

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

Table 9.3 ZISIM Simulator Target 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.
`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 9-13 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.

Target commands that change the behavior of the subordinate ZSIM instance controlled by the SDK debugger are listed in Table 9.4 and described in detail in Section 8.2, “ZSIM Commands,” page 8-6.

The format for ZSIM commands is:

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

Table 9.4 ZSIM Target Commands

Command	Description
`clear-stats`	Resets the general 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.
`pfresource-off`	G2 only. Turns on resource usage information.
`pfresource-on`	G2 only. Turns off resource usage information.
`pipe-off`	Sets the simulator pipeline off.
`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`.

Table 9.4 ZSIM Target Commands (Cont.)

Command	Description
`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
`profile-reset`	Clears all collected profiling information.
`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.

1. This command may also be invoked without the target name. See Section 9.3.1, “Generic Target-Specific Commands” on page 9-13 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 treats *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 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 UART Target 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 9-13 for details.

9.2.4 JTAG Probe Connection

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

Note: The JTAG target is available only for Windows 98/NT/2000/XP 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 JTAG Target 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 is the speed value followed by a zero and the lpt port to use. Generally speaking, the JTAG clock speed should be approximately 1/10th to 1/20th of the ZSP clock speed.
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 9-13 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 hardware 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 invoked in as “mapfile,” or copy to the current directory or $SDSP_HOME/sdspI/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 9.8. For an example on how to use hardware-assisted debugging, refer to Section 9.6.2, "Example 2," page 9-21.

Table 9.7 Hardware-Assisted Debugging Commands for G1

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*.
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	Return to software debug mode. Must have executed in hardware debug mode for at least one cycle in order for this to work.

Table 9.8 Hardware-Assisted Debugging Commands for G2

Command	Description
enable_ice	Enable hardware-assisted debugging.
resume	Resume execution.
step *n*	Step *n* cycles.
insn_addr0_brk *addr* ... insn_addr3_brk *addr*	Set a breakpoint when executing an instruction at *addr*.
disable_insn_addr0_brk ... disable_insn_addr3_brk	Disable instruction address breakpoint.

Table 9.8 Hardware-Assisted Debugging Commands for G2 (Cont.)

Command	Description

`ext0_brk`
`...`
`ext3_brk`
	Set external breakpoint.

`disable_ext0_brk`
`...`
`disable_ext3_brk`
	Disable external breakpoint.

`st_addr_brk` *`addr`*	Set a breakpoint when storing to *`addr`*.
`disable_addr_brk`	Disables hardware data address breakpoint.
`st_data_brk` *`data`*	Set a breakpoint when storing the value *`data`*.
`disable_data_brk`	Disables hardware data value breakpoint.
`ld_addr_and_data_brk` *`addr data`*	Set a breakpoint when loading *`data`* from *`addr`*.
`st_addr_and_data_brk` *`addr data`*	Set a breakpoint when storing *`data`* to *`addr`*.
`ld_addr_or_data_brk` *`addr data`*	Set a breakpoint when loading from *`addr`* or the value *`data`* is loaded.
`st_addr_or_data_brk` *`addr data`*	Set a breakpoint when storing to *`addr`* or storing the value *`data`*.
`ext0_and_ld_addr_and_data_brk` *`addr data`*	Set a breakpoint when loading *`data`* from *`addr`* `and external BP0`.
`ext0_and_st_addr_and_data_brk` *`addr data`*	Set a breakpoint when storing *`data`* to *`addr`* `and external BP0`.
`ext0_and_ld_addr_or_data_brk` *`addr data`*	Set a breakpoint when (loading from *`addr`* or the value *`data`* is loaded) and external BP0.
`ext0_and_st_addr_or_data_brk` *`addr data`*	Set a breakpoint when (storing to *`addr`* or storing the value *`data`*) and external BP0.
`disable_combination_brk`	Disables hardware combination breakpoint.
`disable_brk`	Disable all hardware breakpoints.
`addr_mask`	Sets the bit mask for data address breakpoints.
`data_mask`	Sets the bit mask for data value breakpoints.
`not_addr`	Applies logical NOT to the data address breakpoint.
`not_data`	Applies logical NOT to the data value breakpoint.

Table 9.8 Hardware-Assisted Debugging Commands for G2 (Cont.)

Command	Description
ahb_clk_resume	Resumes the AHB clock without restarting the core.
ahb_clk_stop_en	Makes AHB clock stop when hardware breakpoints are hit.
io_clk_resume	Resumes IO clock without restarting the core.
io_clk_stop_en	Makes IO clock stop when hardware breakpoints are hit.
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 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

The SDK debugger reserves the use of pc value zero. If two breakpoints are inadvertently set at pc value zero, the debugger loops 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 modifies both the data cache and the memory. (With UART and JTAG targets, modifying memory does 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 is advanced by 10 clock cycles.

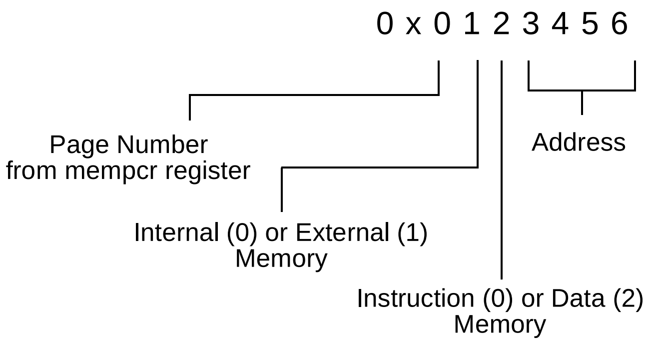
9.3.8 Accessing Memory with the Debugger

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 (left-most 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 (right-most 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)



Note: 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:

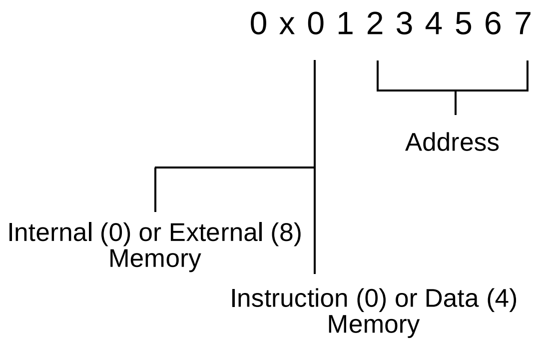
0x0001000	Internal instruction at address 0x1000
0x0022000	Internal data at address 0x2000
0x0103000	Page 0, external instruction memory at address 0x3000
0x2124000	Page 2, external data memory at address 0x4000
0xa105000	Page 10, external instruction memory at address 0x5000

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 (left-most 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 left-most digits are omitted from an address, they are assumed to be zero.

Figure 9.2 Debugger Memory Addressing (zdbug)



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

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 `ctrl-C`.

9.5 Configuration Files

Each target comes with a configuration file that is installed in %SDSP_HOME/MDI/Resources directory. The following table indicates which file belongs to which targets.

Table 9.9 Target Configuration Files

File Name	Target	Use in Debugger
`zisim400.resource`	`sim`	sdbug400
`zsim400.resource`	`zsim`	sdbug400
`jtag400.resource`	`jtag`	sdbug400
`zisim500.resource`	`sim`	zdbug
`zsim500.resource`	`zsim`	zdbug
`jtag500.resource`	`jtag`	zdbug

For a description of all the MDI configuration files and what they do, see Chapter 10, “ZSP MDI Configuration Files.“

9.6 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.

9.6.1 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
```

9.6.2 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 MDI Configuration Files

This chapter describes the configuration files for the ZSP SDK MDI libraries.

The ZSP SDK MDI library supports various hardware and software debug targets used by the SDK debugger. This library uses various configuration files to set itself up. For the most part, the default values in the configuration files are fine, but there are some fields that you may want to change, such as the register mapping file or the clock speed for JTAG.

There are two kinds of configuration files: Device Configuration Files and Device Resource Files, both of which uses a similar syntax.

10.1 Configuration File Basics

The configuration files share the same syntax for both types of configuration files.

10.1.1 Comments

Comments may be put in the configuration file by placing a '#' sign at the beginning of the line. These lines are not processed by the configuration file parser, but are meant to convey information to a human reader.

```
#This is a comment
```

The comment ends at the end of the line.

10.1.2 Section Headers

A section header is a line that begins with an opening bracket ([) and ends with a closing bracket(]), with the name of the section in between the brackets. A section ends with the end of the file or another section header.

```
[Device Libs]
```

The above example declares a new section called "`Device Libs`"

10.1.3 Fields

In each section, various fields are expected to exist. Each field is defined by its case-insensitive name, followed by a equal sign, followed by the field's value. Some fields may be optional, and may be left out of a section when the default values are acceptable.

```
Family = "DSP"
```

The above defines a field called `Family` with the value "`DSP`", a string. There are five possible field types: string fields, endian fields, numeric fields, boolean fields, and memory mapping fields.

10.1.3.1 String Fields

String fields are defined by a double quote and end with a double quote. Some strings may have a fixed maximum length. Strings are case-sensitive.

```
Family = "DSP"
```

The above defines the field `Family` as a string "`DSP`."

10.1.3.2 Endian Fields

Endianness fields may have the value of "`big`" for big endian, or "`little`" for little endian.

```
Endian = little
```

This sets Endianness to little endian.

10.1.3.3 Numeric Fields

A numeric field is defined by one or more digits. For example,

```
Number = 123
```

Defines a field named "Number" with a numeric value of 123. Fields may have restrictions on the range of values it accepts. The number may also be a hexadecimal value. For example,

```
Number  =  0x7B
```

is an equivalent of the previous example, as well.

10.1.3.4 Boolean Fields

Boolean fields have the values of "true" or "false."

```
IsTrue = true
```

Sets IsTrue to true.

10.1.3.5 Memory Mapping Fields

Memory mapping fields are a comma-delimited list of memory addresses and their read and write permissions. The smallest applicable block determines whether an address is readable, writable, or neither. For example,

```
Memory = 0x0-0xffffrw, 0xf800-0xfa00r
```

sets the address 0x0 to 0xf7ff and 0xfa01-0xffff to be both readable and writable, but 0xf800 to 0xfa00 is read-only.

10.2 Device Configuration Files

Device configuration files are in a common format for all available ZSP MDI targets. They reside in the Devices subdirectory where the MDI library happens to be. Device configuration files consist of two sections: a device information section and a device library section. The fields contained in each section are listed below.

10.2.1 Device Information Section

This section contains information obtainable via standard MDI device queries, and is generally meant to inform the user of the target in question.

Table 10.1 Field Listing -- Device Information Section

Field Name	Field Type	Comments
DeviceName	String	Max. length is 80
Family	String	Max. length is 14
FClass	String	Max. length is 14.
FPart	String	Max. length is 14.
FISA	String	Max. length is 14.
Vendor	String	Max. length is 14.
VFamily	String	Max. length is 14.
VPart	String	Max. length is 14.
VPartRev	String	Max. length is 14.
VPartData	String	Max. length is 14. Currently used by the debugger to determine the target type (May be jtag, sim, or zsim)
Endian	Endian	Max. length is 14.

10.2.2 Device Libs Section

This section specifies the exact MDI driver and configuration file to use for this particular device.

Table 10.2 Field Listing -- Device Libs Section

Field Name	Field Type	Comments
Driver	String	Specify a file in the Drivers subdirectory.
Configuration	String	Specify a file in the Resources subdirectory.

10.3 Driver Configuration (Resource) Files

The driver configuration files are stored in the Resources subdirectory where the MDI library happens to be. As their name implies, these are processed by the specific driver libraries. Therefore, the sections and fields that exists in the files can be different from one driver to the next. However, they all use the same syntax, as mentioned above.

In the current version of the SDK, three different types of driver configuration files exist: zsim resource files, zisim resource files, and JTAG resource files.

10.3.1 ZSP400 ZISIM

10.3.1.1 General Settings Section

Table 10.3 Field Listing -- ZSP400 ZSIM General Settings

Field Name	Field Type	Comments
Simulator Library	String	Name of simulator library. File name extension is not required. It will be extended with ".so" on Solaris or ".dll" for Windows.
iboot pin	Number	Set pin IBOOT=1 to start from internal memory location 0xf800. If iboot pin = 0, simulation will start from external memory location 0xf800.

10.3.1.2 Memory Settings Section

Table 10.4 Field Listing -- ZSP400 ZISIM Memory Settings

Field Name	Field Type	Comments
Internal Instruction Memory	Memory Mapping	Only one memory range is supported. The beginning value must be 0, and the maximum ending value is 0xf7ff
Internal Data Memory	Memory Mapping	Only one memory range is supported. The beginning value must be 0, and the maximum ending value is 0xf7ff

10.3.2 ZSP400 ZSIM

10.3.2.1 General Settings Section

Table 10.5 Field Listing -- ZSP400 General Settings

Field Name	Field Type	Comments
Simulator Library	String	Name of simulator library. File name extension is not required. It will be extended with ".so" on Solaris or ".dll" for Windows.
iboot pin	Number	Set pin IBOOT=1 to start from internal memory location 0xf800. If iboot pin = 0, simulation will start from external memory location 0xf800.
MSS Library	String	Name of Memory Subsystem library. File name extension is not required. It will be extended with ".so" on Solaris or ".dll" for Windows. The default library name is "libzmss400".

10.3.2.2 Memory Settings Section

Table 10.6 Field Listing -- ZSP400 ZSIM Memory Settings

Field Name	Field Type	Comments
Internal Instruction Memory	Memory Mapping	Only one memory range is supported. The beginning value must be 0, and the maximum ending value is 0xf7ff
Internal Data Memory	Memory Mapping	Only one memory range is supported. The beginning value must be 0, and the maximum ending value is 0xf7ff
hasmempcr	Number	Indicates that the system has MEMPCR memory mapped register. hasmempcr = 0 indicates that the system doesn't have MEMPCR and the next entry is not required.
mempcr address	Number	Specifies the address of MEMPCR.

10.3.3 ZSP400 JTAG

10.3.3.1 General Settings Section

Table 10.7 Field Listing -- ZSP400 JTAG General Settings

Field Name	Field Type	Comments
Probe Driver	String	Requires the full name of the library. This library is used to talk to the actual JTAG probe hardware.
Probe Speed	Number	Speed settings for the JTAG Probe.
Hardware Mode	Boolean	Whether ZSP400 starts in hardware or software debug mode.
Register Mapfile	String	Mapping for all the register bits in the core scan chain. Used for hardware debug.

10.3.3.2 Memory Settings Section

Table 10.8 Field Listing -- ZSP400 JTAG Memory Settings

Field Names	Field Type	Comments
internal instruction memory	Memory Mapping	
internal data memory	Memory Mapping	
external instruction memory	Memory Mapping	
external data memory	Memory Mapping	

10.3.4 ZSP500/ZSP600 ZISIM

10.3.4.1 General Settings Section

Table 10.9 Field Listing -- ZSP500/ZSP600 ZISIM General Settings

Field Names	Field Type	Comments
Architecture	String	May be either “zsp500” or “zsp600”.
Simulator Library	String	Name of simulator library. File name extension is not required. It will be extended with “.so” on Solaris or “.dll” for Windows.
SVT Address	Number	Specify system vector table address.

10.3.4.2 Memory Settings Section

Table 10.10 Field Listing -- ZSP500/ZSP600 ZISIM Memory Settings

Field Name	Field Type	Comments
internal instruction memory	Memory Mapping	0xffffff is the maximum value
internal data memory	Memory Mapping	0xffffff is the maximum value
external instruction memory	Memory Mapping	0xffffff is the maximum value
external data memory	Memory Mapping	0xffffff is the maximum value

10.3.5 ZSP500/ZSP600 ZSIM

10.3.5.1 General Settings Section

Table 10.11 Field Listings -- ZSP500/ZSP600 ZSIM General Settings

Field Name	Field Type	Comments
Architecture	String	May be either “zsp500” or “zsp600”.
Simulator Library	String	Name of simulator library. File name extension is not required. It will be extended with “.so” on Solaris or “.dll” for Windows.
SVT Address	Number	Specify system vector table address.
Co-Processor Library	String	Name of Co-Processor library. File name extension is not required. It will be extended with “.so” on Solaris or “.dll” for Windows.
Bus Interface Library	String	Name of Bus Interface library. File name extension is not required. It will be extended with “.so” on Solaris or “.dll” for Windows. Specify “none” if no special libraries are needed.
MSS Library	String	Name of Memory Subsystem library. File name extension is not required. It will be extended with “.so” on Solaris or “.dll” for Windows.

10.3.5.2 Memory Settings Section

Table 10.12 Field Listings -- ZSP500/ZSP600 ZSIM Memory Settings

Field Name	Field Type	Comments
internal instruction memory	Memory Mapping	0xffffff is the maximum value
internal data memory	Memory Mapping	0xffffff is the maximum value
external instruction memory	Memory Mapping	0xffffff is the maximum value
external data memory	Memory Mapping	0xffffff is the maximum value

10.3.6 ZSP500/ZSP600 JTAG

10.3.6.1 General Settings Section

Table 10.13 Field Listing -- ZSP500/ZSP600 JTAG General Settings

Field Name	Field Type	Comments
Probe Driver	String	Requires the full name of the library. This library is used to talk to the actual JTAG probe hardware.
Probe Speed	Number	Speed settings for the JTAG Probe.
Hardware Mode	Boolean	Whether ZSP400 starts in hardware or software debug mode.
Register Mapfile	String	Mapping for all the register bits in the core scan chain. Used for hardware debug.

10.3.6.2 Memory Settings Section

Table 10.14 Field Listing -- ZSP500/ZSP600 JTAG Memory Settings

Field Names	Field Type	Comments
internal instruction memory	Memory Mapping	
internal data memory	Memory Mapping	
external instruction memory	Memory Mapping	
external data memory	Memory Mapping	

Chapter 11
ZSP Integrated Development Environment

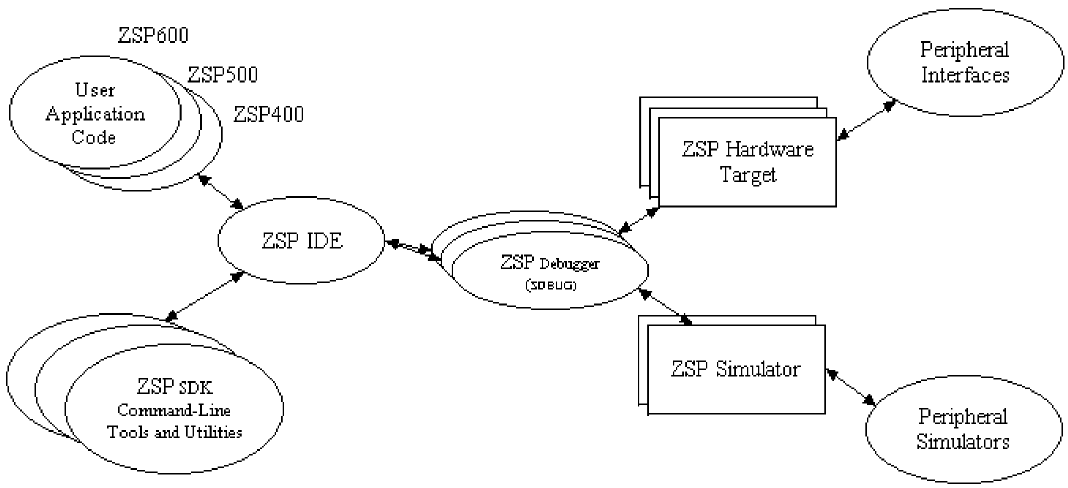
The ZSP Integrated Development Environment (ZSP IDE) is a graphical user interface (GUI) application for ZSP software project management. ZSP IDE enhances productivity for users of ZSP processors, allowing easy setup, build, and debug of ZSP software projects. This chapter focuses on managing project structures and building executable ZSP programs using ZSP IDE. Chapter 9, “Debugger,“ describes the GUI for the debuggers.

This chapter contains the following major sections:

11.1 ZSP IDE Overview

ZSP IDE provides an integrated tool suite for ZSP software developers by managing projects, building code, and debugging for all ZSP processors and supported targets. Refer to Figure 11.1. The graphical user interface allows easy project setup for users with minimal familiarity with ZSP tools and hardware.

Figure 11.1 ZSP IDE Tools Suite Implementation



11.1.1 Features

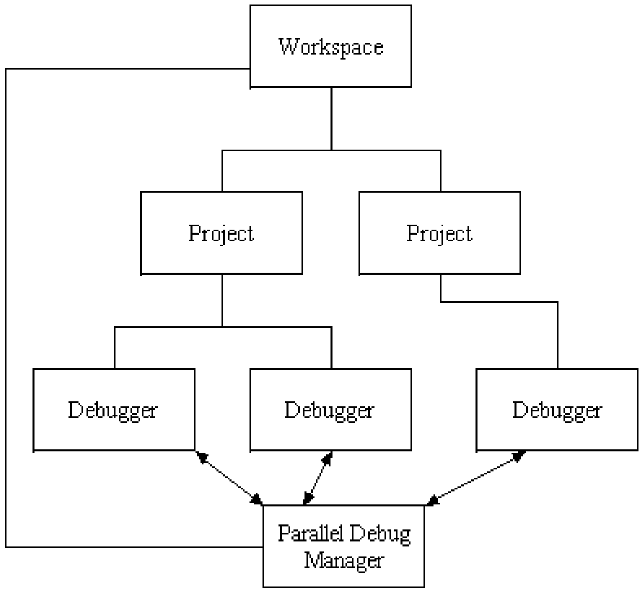
- Workspaces to organize projects and default settings
- ZSP Project Build Support - G2, G1/G2, ZSP400
- Compatibility - Backward-compatible with Version 3.2 projects
- Windows and UNIX (planned) platforms
- Multiple projects in same directory
- Build output linked to Source File View
- Parallel Debug Manager

11.1.2 Introduction to Workspaces and Projects

As shown in Figure 11.2, a workspace may contain 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.

Figure 11.2 ZSP IDE Workspace



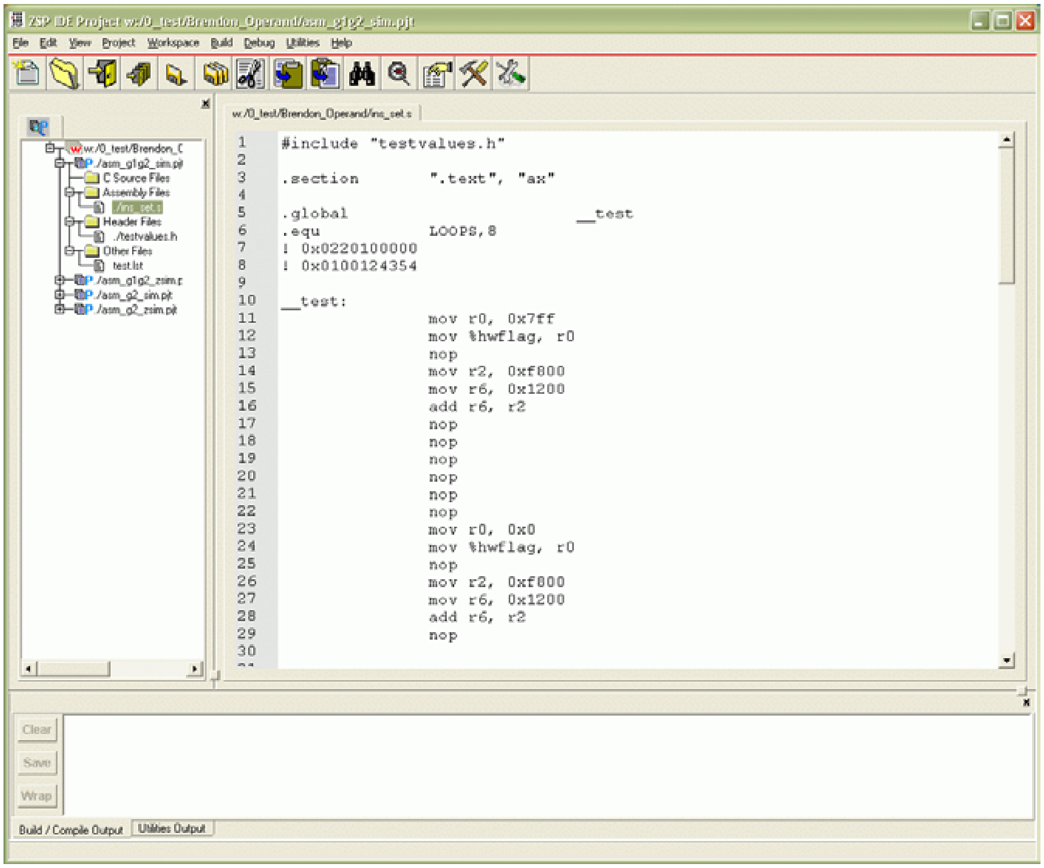
The basic element of each ZSP software project is an executable file. Each executable file is managed by ZSP IDE based on settings that are created within ZSP IDE and stored in a project file. Project settings include all information needed to build and debug an executable:

- Target ZSP architecture
- Compiler settings
- Include and archive file directories
- Assembler settings
- Debugger settings
- IDE Debugger window settings

11.1.2.1 IDE

Figure 11.3 shows the Main window of the IDE.

Figure 11.3 ZSP IDE Main Window



The Main window contains the main menu, toolbar, project tree, source file editing area and output/utility windows.

All of the major functions of ZSP IDE are available through the main menu. The most commonly used functions from the main menu are also accessible through 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. An additional tab grouped with the output window in the lower section provides an output window for post-processing functions (such as object dump utility) or for custom commands. The Utility Output tab displays output of utility commands available from within the IDE.

The bottom part of the Main window shows status information. The current cursor location in the editor window is also reflected in this status area.

11.2 ZSP IDE Filename Extensions

ZSP IDE produces a number of files when you create and compile a project or workspace. These are categorized in Table 11.1.

Table 11.1 ZSP IDE Filename Extension Assignments

Filename Extension	File Description
`.c`	C Source file
`.S` or `.s`	Assembly source file
`.h`	Header file
`.pjt`	Project file
`.ws`	Workspace file
`.exe`	Executable file

11.3 Working With Workspaces and Projects

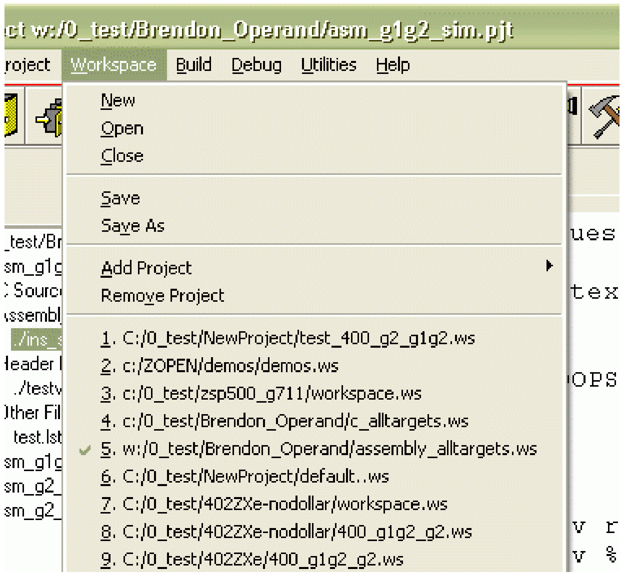
The purpose of a workspace is to organize and to provide default settings for a project or group of projects. New and existing projects may be added to a workspace. A project may belong to multiple workspaces.

11.3.1 Working With Workspaces

A set of default properties exists for each workspace. After the workspace is created, the workspace properties may be modified. Any new project added to the workspace inherits the default settings of the workspace.

The Workspace menu has submenus to open, close and save workspace files. It also has submenus 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 (see Figure 11.4). Only one workspace may be active at any time. Switching to a different workspace closes the existing workspace and the component projects.

Figure 11.4 Recent Workspaces List



11.3.1.1 Creating a New Workspace

To create a new workspace, select Workspace -> New in the IDE Main menu. A dialog box is displayed to enter the filename for the new workspace.

> Note: Filenames may not contain space characters.

11.3.1.2 Opening a Workspace

To open an existing workspace, use the same procedure as described in the previous section.

11.3.1.3 Saving a Workspace

To save the current workspace, select Save in the Workspace menu.

To save a workspace with a different name or in a different directory, select Save As in the Workspace menu to display the File Selection dialog box. The new workspace becomes the current session after executing Workspace -> Save As.

11.3.1.4 Adding Projects to a Workspace

To add new or existing projects to a workspace, select Add Project in 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`.

11.3.1.5 Deleting a Project from a Workspace

To delete a project from a workspace, select the project to be deleted in the Project Explorer window, then select Workspace -> Remove Project.

11.3.1.6 Closing a Workspace

To close a workspace, select Workspace -> Close from the Workspace menu. Before the workspace closes, you are prompted to save any unsaved files.

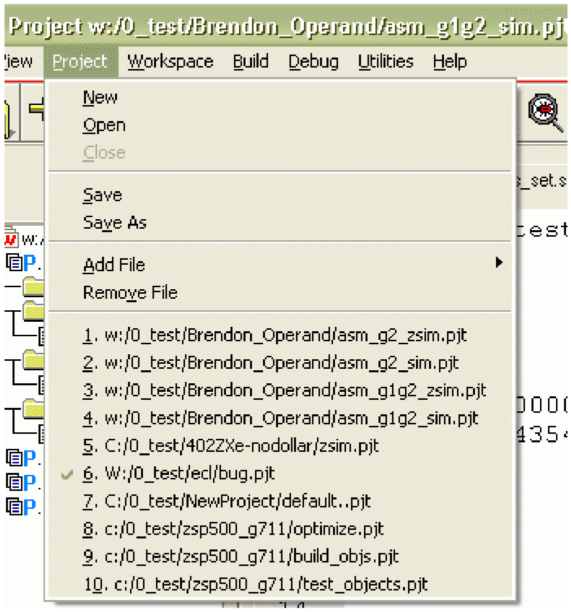
11.3.2 Working With Projects

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, shown in Figure 11.5, has submenus to open, close, and save project files, and a submenu 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 closes the existing project. If a source file was altered, a warning is issued and you are provided with an option to save changes before switching to a different project.

Figure 11.5 Project Menu



11.3.2.1 Creating a New Project

To create a new project within a workspace, select either Workspace -> Add Project -> New Project in the Main menu or Add Project -> New Project in the Project Tree popup menu over the active workspace node.

A dialog box is displayed so you can create the new project.

11.3.2.2 Opening an Existing Project

To open an existing project, follow these steps:

Step 1. Select Project -> Open.

This displays the File Selection dialog box.

Step 2. Browse to the appropriate directory and highlight the project file (`.pjt` file) to be opened.

Step 3. Click OK

This opens the selected project. All associated component source, header, and object files are shown in the Project Explorer pane.

11.3.2.3 Saving a Project

To save a project, select Project -> Save.

To save a project to a new project filename, select Project -> Save As. 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.

11.3.2.4 Adding Files to a Project

To add new or existing files to a project, select Project -> Add File. The File Selection dialog box for files is similar to that used for creating projects. There is no default filename extension for files.

11.3.2.5 Deleting Files from a Project

To delete a file from a project, use the popup menu over the file you want 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.

11.3.2.6 Closing a Project

To close a project, select Project -> Close in the Main menu.

11.4 Project Settings

Selecting Build -> Settings or Debug -> Settings in the main menu displays the Settings dialog box. If a workspace node is selected in the Project Tree, then the Settings dialog box reflects the default settings for the workspace.

The Settings dialog box 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 in the following subsections. You may page between the various tabs in the Settings dialog box 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.

11.4.1 Build Methodology and Project Tree Structure

The ZSP IDE Project Tree partitions project files into folders based on filename extensions.

- Source files with a "`.c`" extension (for C sources) are added to the C Source Files folder.
- Source files with an "`.S`" extension are assembly sources that require C preprocessing.
- Files with an "`.s`" extension are assembly source files, which do not require preprocessing.

Files with extension of "`.S`" or "`.s`" are inserted into the IDE Project Tree in the Assembly Source Files folder. Include files which have extensions of "`.h`" or "`.inc`" are added to the Project Tree's Include Files folder. Additionally, when a file with an "`.h`" or "`.inc`" extension is added to the project, the ZSP IDE provides 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 are not 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 box. The ZSP compiler invokes each of the component processes that complete the build process. You may specify options in the Settings panel to direct the behavior of the compiler, assembler, and linker.

11.4.2 Compiler/Assembler Settings

The Compiler settings tab (see Figure 11.6) 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 works for all ASSPs based on this core (for example, the LSI402ZX, LSI403Z, and LSI403LP).

ZSPG2, the next generation architecture in the ZSP roadmap, has many new instructions, new resources, and a bigger address range. It is assembly compatible with the ZSP400.

The dual mac core called ZSP500 is based on the ZSPG2 architecture. It supports a 24-bit address range and is a four 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 11.6 Compiler Settings

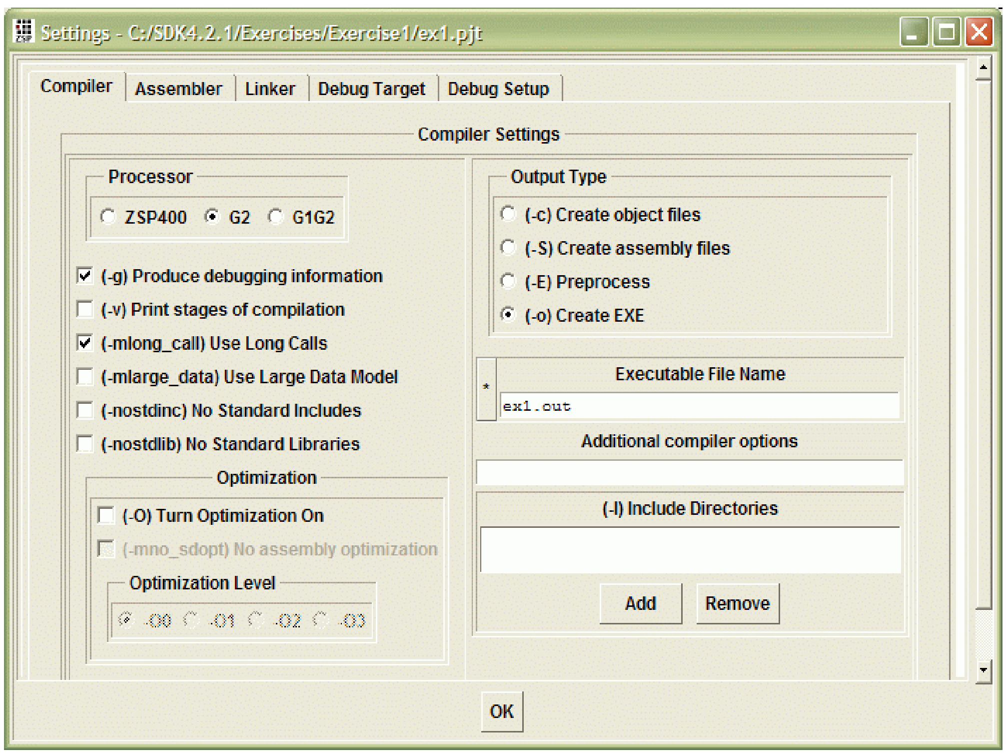


Figure 11.7 shows the Assembler settings tab.

Figure 11.7 Assembler Settings

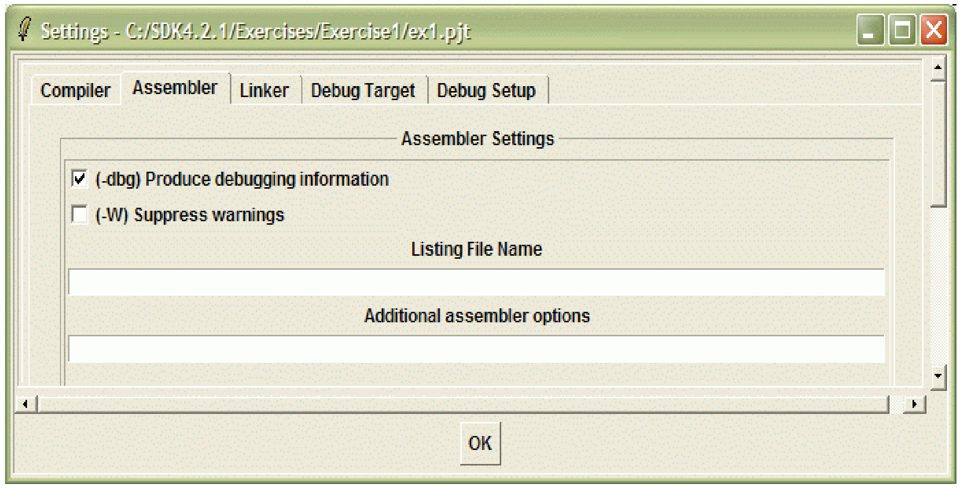


Table 11.2 describes the other settings that control the Compiler and Assembler.

Table 11.2 Compiler/Assembler Settings

Option (Command Line Equivalent)	Description
Produce debugging information (-g)	This option instructs the compiler to produce 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 optimization on debugging capabilities.
No assembly optimization (-mno_sdopt)	This option suppresses back-end optimization that is otherwise automatically performed on compiler-generated assembly code.
Use Long calls (-mlong_call)	This option tells 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.

Table 11.2 Compiler/Assembler Settings (Cont.)

Option (Command Line Equivalent)		Description
Additional compiler options Text Box (option)		This option specifies any compiler options that do not have a check box in the Compiler/Assembler settings tabs. 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.
No standard includes (-nostdinc)		This option directs the compiler not to search the standard system directories for header and include files.
Include Directories (-I)		This is a list of directories that the compiler searches 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.
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 settings tab. Separate multiple options with spaces.

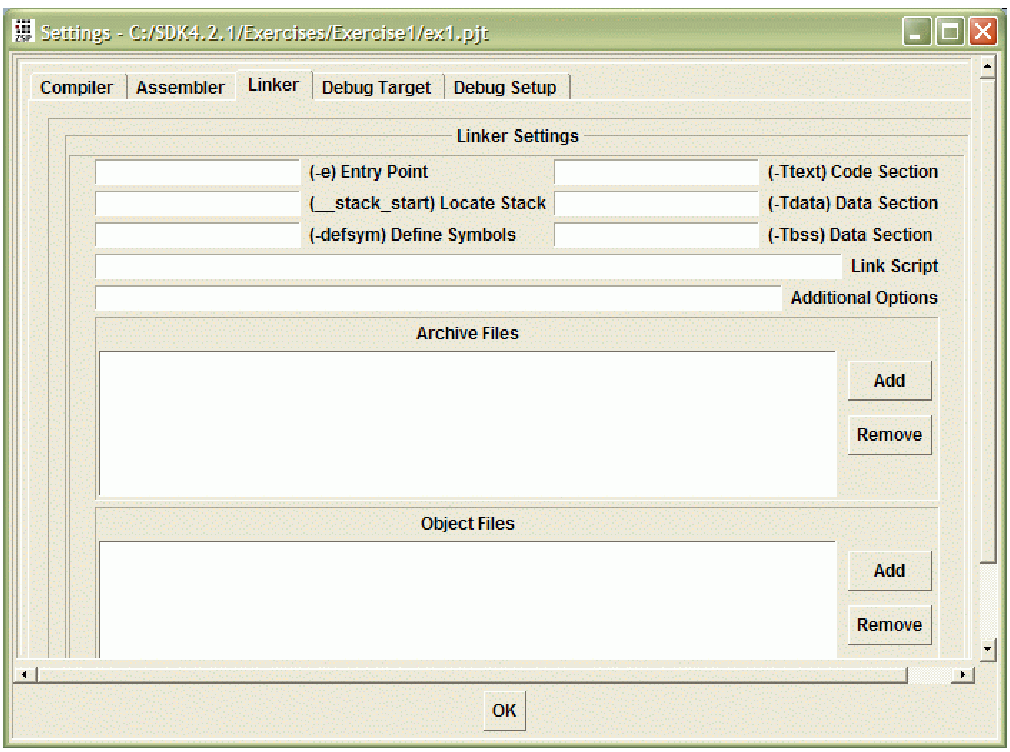
Table 11.2 Compiler/Assembler Settings (Cont.)

Option (Command Line Equivalent)	Description
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 settings tab. Separate multiple options with spaces.

11.4.3 Linker Settings

The Linker Settings window, shown in Figure 11.8, provides detailed control over link behavior. See the Linker section of this manual for more detail.

Figure 11.8 Linker Settings



For archive and object files, you can invoke a file browser to select the files by selecting the appropriate Add command button. To remove a file from the list, select it with the mouse and then select the Remove command button.

Table 11.3 describes the options that control the linker.

Table 11.3 Linker Options

Option (Command Line Equivalent)	Description
Entry Point (-e)	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.
Locate Stack (__stack_start)	This entry defines the `__stack_start` symbol that determines the starting address of the program stack pointer.
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. 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. The default value is 0x0
Data Section (-Tdata)	This entry specifies the starting address of the initialized data segment of the output file. The default value is 0x0.
Data Section (-Tbss)	This entry specifies the starting address of the uninitialized data segment of the output file. The default value is 0x0.
Link Script	This entry specifies a filename to be used as a Linker Command file, if you need more control over the locations of sections in your executable. The filename extension must not conflict with source/object filename extensions. If you specify a relative pathname, it should be relative to your project directory.
Object Files	Specifies external object files to be linked with the project's object files to produce the executable. Select the 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 to produce the executable. Select the 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 Settings tab. Separate multiple options with spaces.

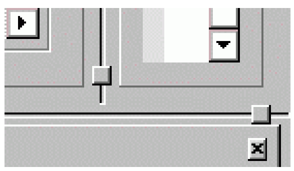
11.5 ZSP IDE Detailed Description

This section provides more detail about the ZSP IDE graphical user's interface.

11.5.1 Paned Window Controls

The IDE Main window is divided into resizable sections by paned window controls, as shown in Figure 11.9. The IDE screen area displayed in the paned window may be resized by dragging the handles of the paned window controls that separate the screen areas.

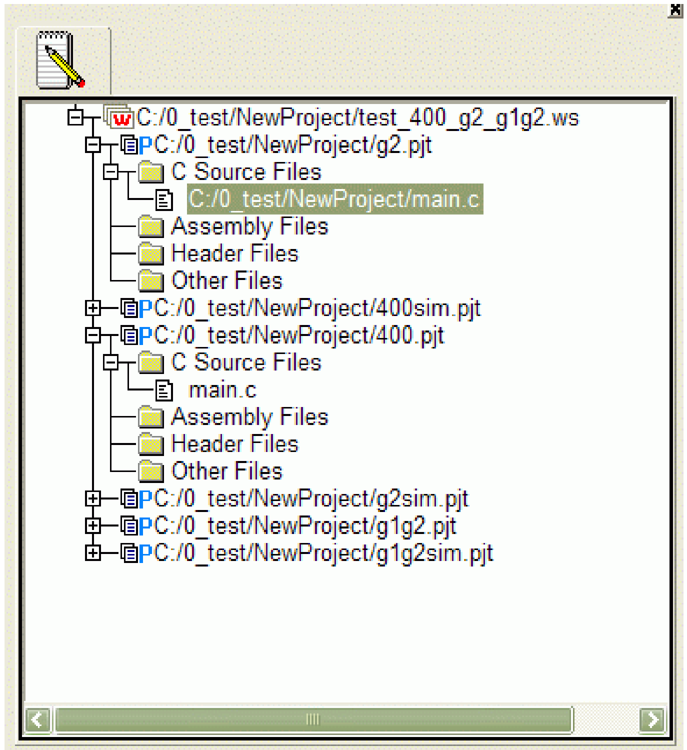
Figure 11.9 Paned Window Handles



11.5.2 Project Tree

The Project Tree component of the ZSP IDE (see Figure 11.10) 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 11.10 ZSP IDE Project Tree



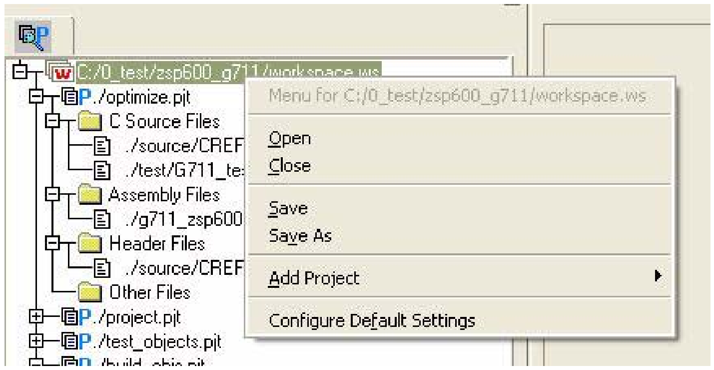
Select the workspace node of the tree to customize default settings for your workspace. Default settings are applied to new projects when they are created within your workspace. Default settings may also be applied to existing projects when they are added to your workspace. To apply default settings to newly created projects, select the checkbutton labeled Use Workspace Settings in the Preferences window. To display the Preferences window, select View -> Preferences. See Section 11.5.3.5, "View Menu," page 11-21 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.

Double-click the left mouse button while the mouse cursor is positioned over a source file in 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. See Figure 11.11.

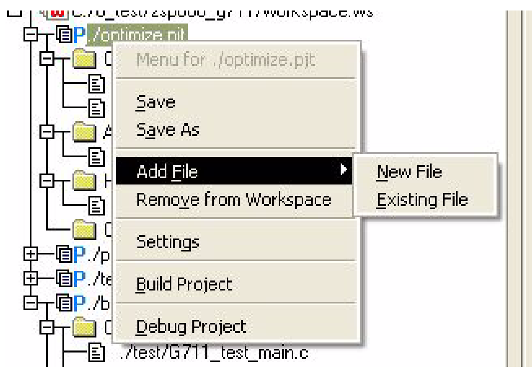
Figure 11.11 Workspace Popup Menu



The Workspace popup menu shows the name of the workspace followed by shortcuts to workspace menu items from the main menu.

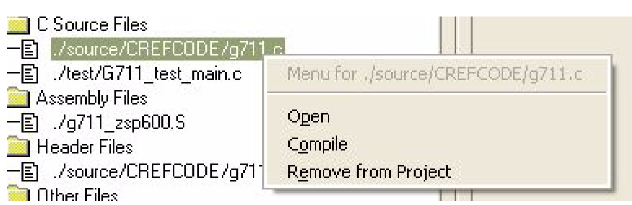
A popup menu is also available for a project node of the Project Tree. Clicking the mouse on a project node causes the menu in Figure 11.12 to pop up.

Figure 11.12 Project Popup Menu



Additionally, the File popup menu shown in Figure 11.13 is displayed when you click on a file node.

Figure 11.13 File Popup Menu



11.5.3 Main Menu

The Main menu provides access to major functions of the ZSP IDE such as opening, closing, and maintaining workspaces, projects and files, as well as building and debugging projects

11.5.3.1 Operating the Main Menu

Main menu items may be selected either by left-clicking with the mouse or by typing the menu accelerator key (underlined character in the menu name). To open the menus from the keyboard, depress the ALT key and the desired accelerator key concurrently. 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 open a menu or the Escape key to cancel your selection.

11.5.3.2 Main Menu Functions

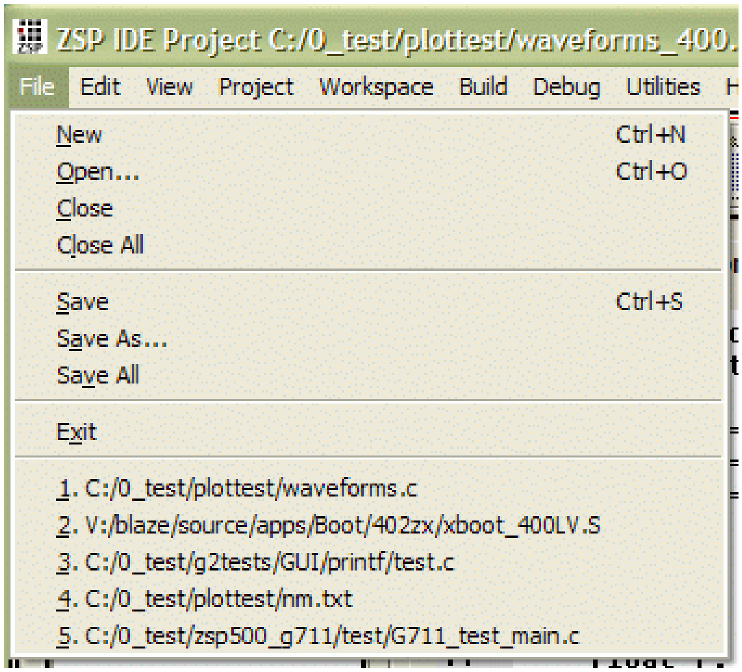
The ZSP IDE Main menu provides the following submenus:

- File menu
- Edit menu
- View menu
- Project menu
- Workspace menu
- Build menu
- Debug menu
- Utilities menu

11.5.3.3 File Menu

The File menu, shown in Figure 11.14, is used for operations on text files such as source files, include files, batch files, or any other text files. It opens new or existing files and saves and closes active files.

Figure 11.14 File Menu



A file opened using the File menu does not automatically belong to the active project. You need to explicitly add it to a project as described in Section 11.3.2, "Working With Projects." You can open and edit a file even if no workspace or project is active.

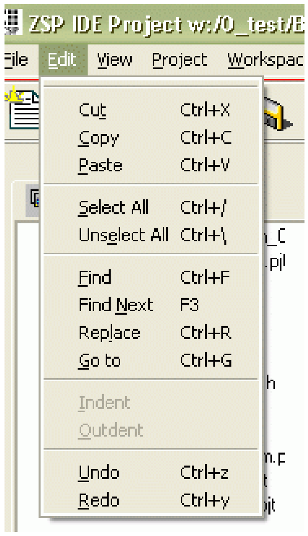
11.5.3.4 Edit Menu

A simple editor is included in the IDE. The Edit menu, Figure 11.15, 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 active for a file you are editing in the Edit window. They are not active for projects, workspaces, and directories, and they will cause errors if used for anything but file editing.

Shortcut keys are also available for common edit functions.

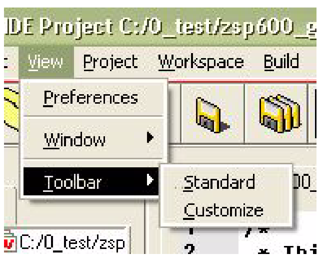
Figure 11.15 Edit Menu



11.5.3.5 View Menu

The View menu is available to selectively display and customize ZSP IDE screen components. See Figure 11.16.

Figure 11.16 View Menu



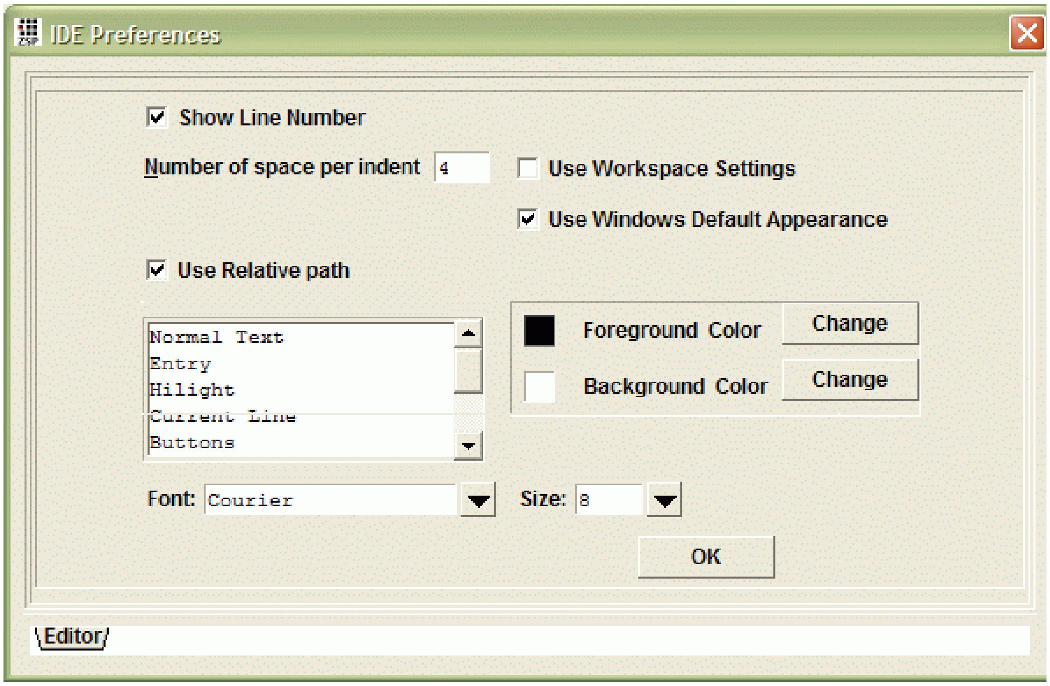
View Preferences – You may set IDE enviromnent preferences by selecting View -> Preferences. The Preferences dialog box, shown in Figure 11.17, offers options to alter editor settings in a tab labeled Editor. You can set colors, text style, line number, and other preferences in this window.

The checkbox labeled “Use Workspace Settings” controls the default project settings when a new project is created. If it is checked, then the project is created with the default workspace settings, otherwise the project is created with generic defaults.

The checkbox 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 are used for workspace components (projects, files, include directories, etc.) Otherwise, relative paths are used. Relative path hierarchy 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.

For interoperation of projects and workspaces between Windows and Solaris platforms, always use relative paths.

Figure 11.17 View Preferences Dialog Box



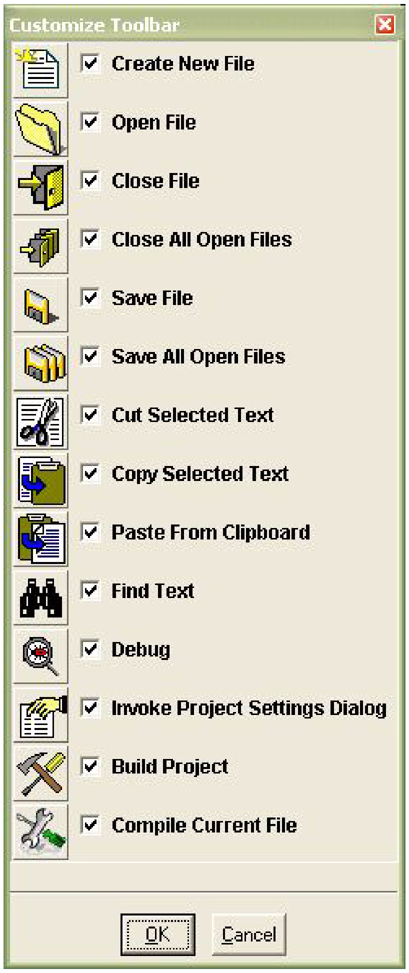
After you set the preferences, click OK to save the settings.

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 button icons at the top of the IDE window can be customized to your liking. Select View -> Toolbar -> Customize to display the Customize Toolbar dialog box shown in Figure 11.18. The dialog box shows the current icon assignments.

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. When you click OK, the toolbar is altered to display the customized settings.

Figure 11.18 Customize Toolbar Dialog Box



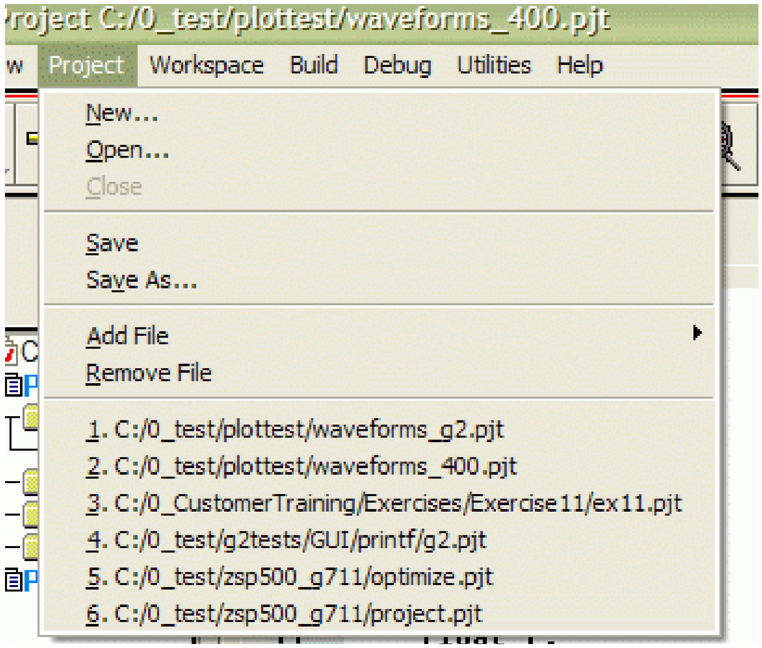
You can switch back to the standard settings by selecting View->Toolbar->Standard.

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.

11.5.3.6 Project Menu

The Project menu, shown in Figure 11.19, allows you to create and maintain projects.

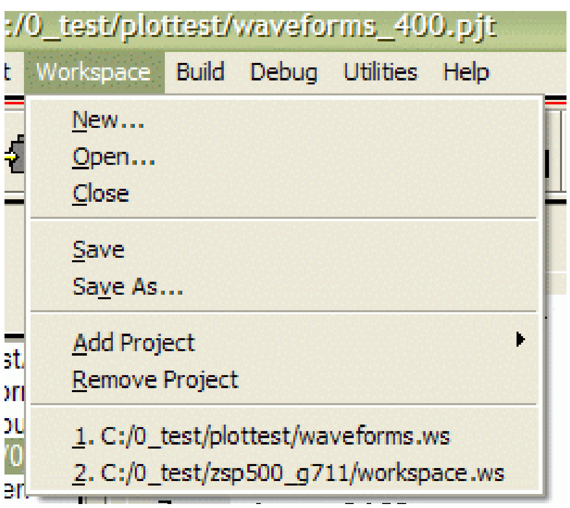
Figure 11.19 Project Menu



11.5.3.7 Workspace Menu

The Workspace Menu, shown in Figure 11.20, allows you to create and maintain workspaces.

Figure 11.20 Workspace Menu



11.5.3.8 Build Menu

The Build menu invokes the ZSP IDE Build process and allows you to customize project build parameters. The Build menu is shown in Figure 11.21.

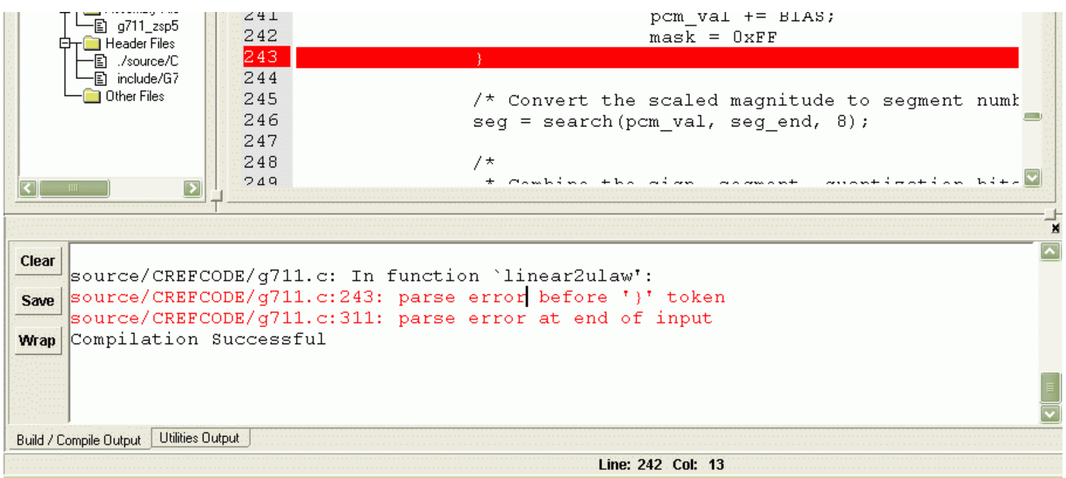
Figure 11.21 Build Menu

Build Project – Once a project is created and the constituent files are 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 builds 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 are displayed in the Output window in the Build/Compile Output tab, if enabled. See Figure 11.22.

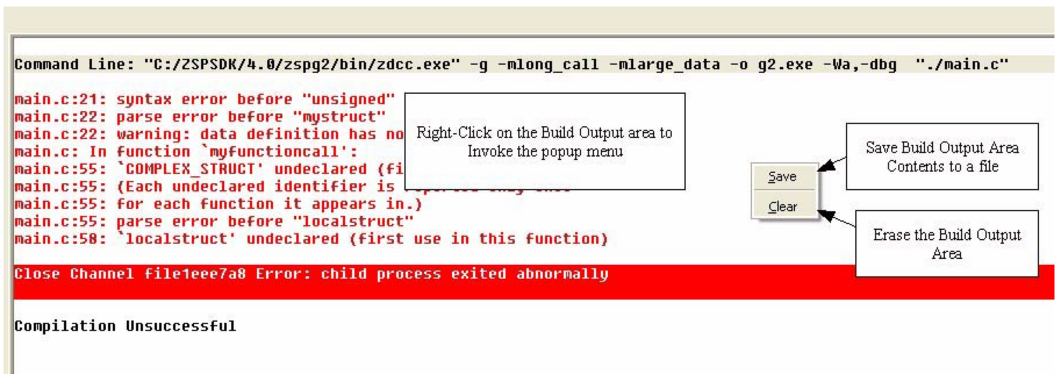
Figure 11.22 Build/Compile Output Window



The Build Output window displays the 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 (see Figure 11.23) to save or clear the window contents.

Figure 11.23 Build Output Window Popup Menu



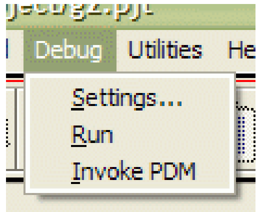
Settings – Select Settings in the Build menu to customize the parameters to be used for building your project. This functionality is also available from the popup menu on the Project Tree when the project file is the selected node.

Compile Current – Select Compile Current in the Build menu to compile the currently selected source file. The ZSP compiler is invoked with the `-c` option and an object file is 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.

11.5.3.9 Debug Menu

The Debug menu, shown in Figure 11.24, provides configuration and control of project debugging.

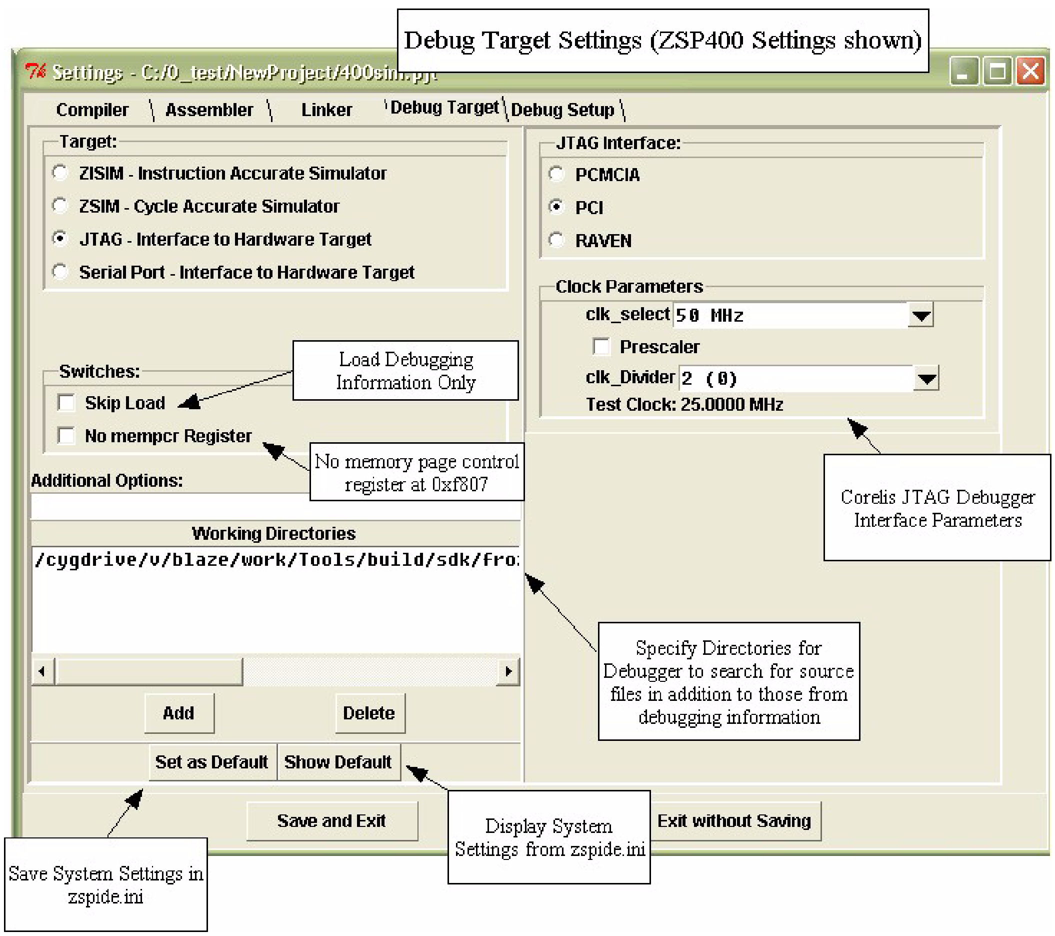
Figure 11.24 Debug Menu



Settings – Select Settings in the Debug menu to display the project Settings dialog box. In the project Settings dialog box, select Debug Target or Debug Setup tabs to display debugger settings.

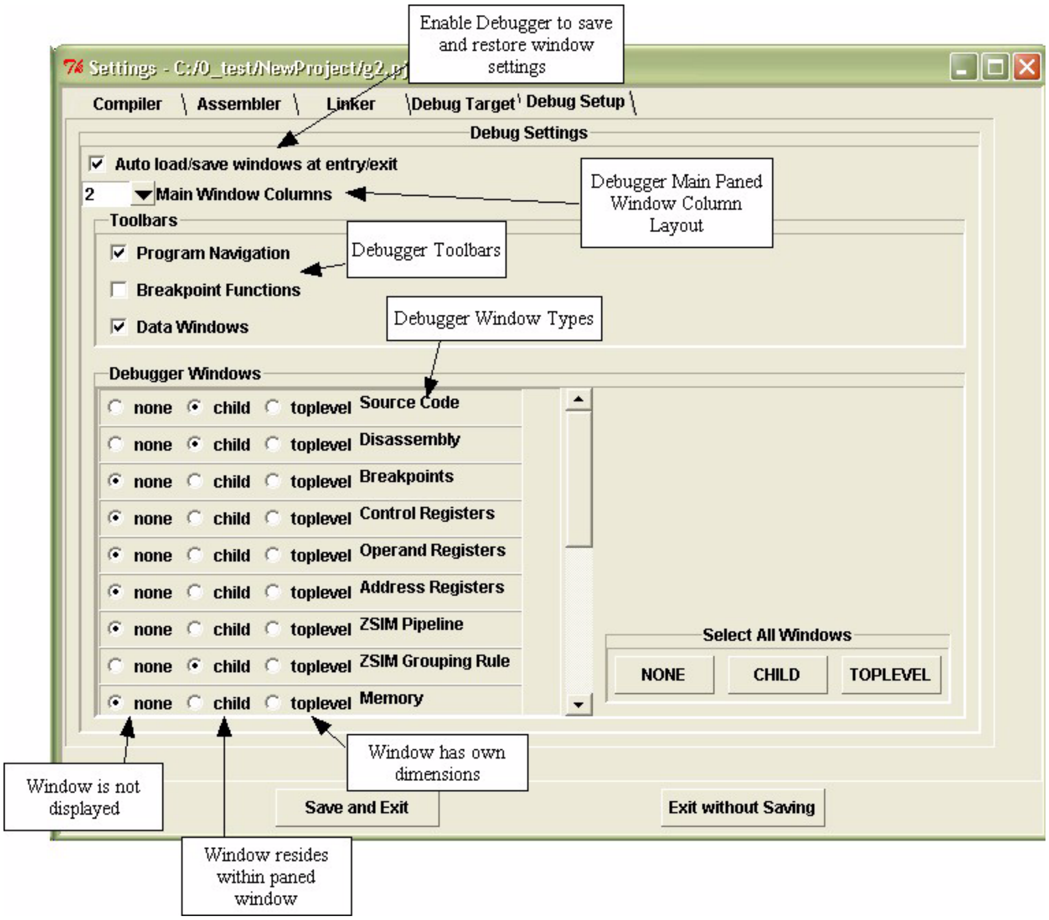
As shown in Figure 11.25, Debug Target displays the valid target types for the processor type that is specified in your project's compiler settings.

Figure 11.25 Debug Target Dialog Box



The Debug Setup dialog box is shown in Figure 11.26.

Figure 11.26 Debug Setup Dialog Box



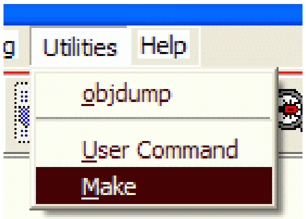
Run – Select Run in the Debug menu to launch the ZSP IDE Debugger using the selected processor and debug target settings.

Invoke PDM – Select Invoke PDM in the Debug menu to run the Parallel Debug Manager (PDM) 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 11.6, “Parallel Debug Manager,” page 11-36, for more information on this feature.

11.5.3.10 Utilities Menu

The Utilities menu, shown in Figure 11.27, provides the ability to examine object files, execute custom commands and work with a makefile to create custom targets.

Figure 11.27 Utilities Menu



objdump – Select objdump in the Utilities menu to display the Object File Utility dialog box shown in Figure 11.28.

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 button.

Figure 11.28 Object File Utility Dialog Box

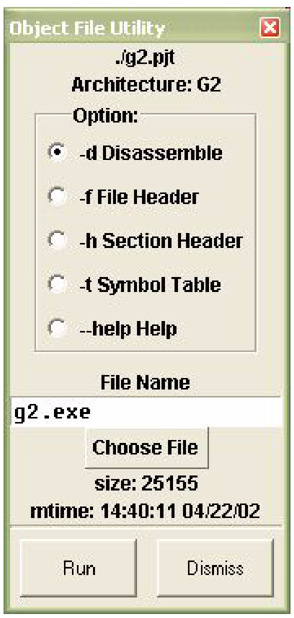
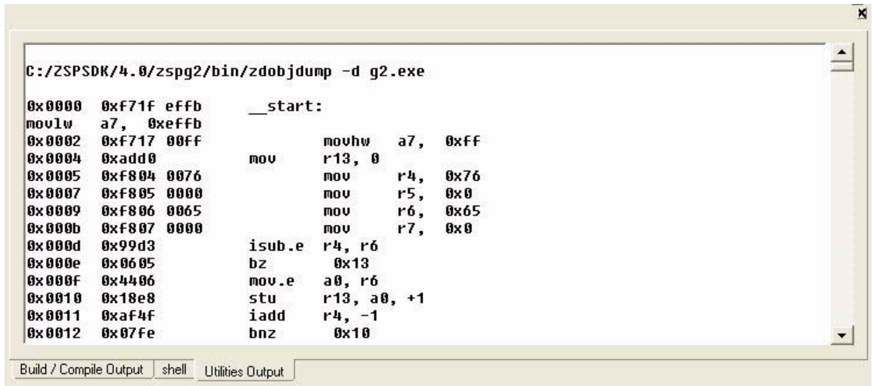
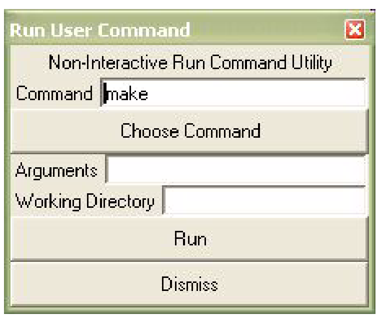


Figure 11.29 Utility Output Window Showing Disassembled Code



User Command – Select User Command in the Utilities menu to display a dialog box (see Figure 11.30) that allows execution of a custom command to be executed.

Figure 11.30 Run User Command Dialog Box

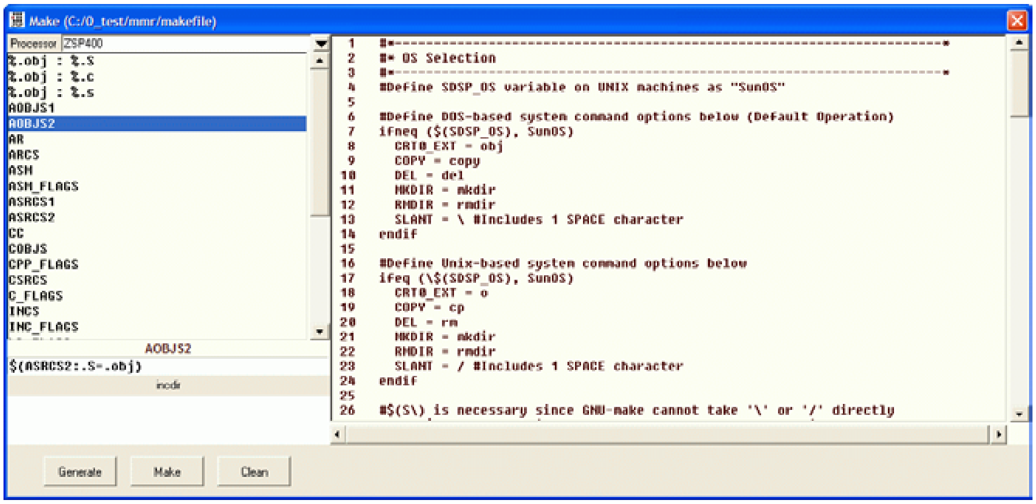


Make – Select Make from the Utilities menu to display the Make Utility shown in Figure 11.31.

The Make Utility creates a makefile named ‘makefile’ in the project directory of the current project. This filename is reserved for use by the Make Utility and is overwritten each time the Make Utility is invoked.

A set of makefile variables is maintained by the Make Utility and stored in the ZSPIDE project file for your current project.

Figure 11.31 Make Utility



The Make Utility includes

- Command buttons to generate a makefile and make common targets.
- makefile variable editing functionality.
- An output text area for viewing the generated makefile and the output of the make process.

Processor Selection – The Make Utility configures the makefile to build with the appropriate tools for the selected processor.

Make Variables Selection List – A listing of make variables is generated and saved in the ZSPIDE `<project>.pjt` file. To view and edit each of the variables, select it from the list.

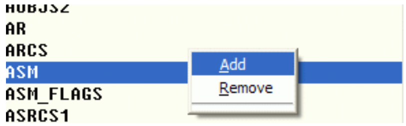
Make Variable Editing Area – The variable definition is displayed below the Make Variables Selection List for editing.

Text Output Area – View the makefile after generation and the output of the make process in the text output area.

When the Make Utility is invoked, the list of files in the ZSPIDE project are imported. New files may be added to the project and the Make Utility detects and adds them to the makefile variables. C source files are added to the CSRS variable. Assembly source files, `.s`, are added to the ASRCS1 variable and assembly source files, `.S`, are added to the ASRCS2 variable. These three source file lists are required for the Make

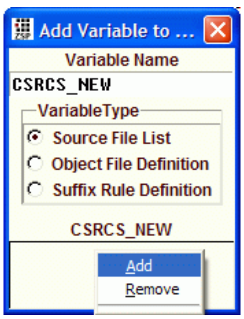
Utility to function and should not be deleted. Additional source file lists may be created so that alternate procedures may be performed on the additional lists. To add a source file list to the make configuration, right-click on the makefile variable list and select Add from the popup menu shown in Figure 11.32.

Figure 11.32 Makefile Variable Popup Menu



The Add Variable to dialog box shown in Figure 11.33 is displayed.

Figure 11.33 Add Variable to Dialog Box



Select Source File List in the Variable Type selection list and edit the variable name in the Variable Name entry box. Files may be added by right-clicking on the variable editing area and selecting Add. This invokes a file selection dialog box from which new files may be selected. Of course you may also manage the makefile variables by editing the project file with a text editor. Make sure that spied is not using the project file though, as zspide may overwrite the project file at any time.

Generate - When the Generate button in the Make Utility is selected, the the makefile is generated and displayed in the text area. This is not an editing area; changes made directly to the makefile view are not saved. If you need to make custom changes, copy the makefile to a new file for editing. You may invoke zdmake on the new filename by specifying `-f <filename>` on the zdmake command line.

Make - When the Make button in the Make Utility is selected, the Make Utility invokes zdmake to build the makefile project.

Clean - Select the Clean button in the Make Utility to delete the object files to force rebuilding all objects on a subsequent invocation of zdmake.

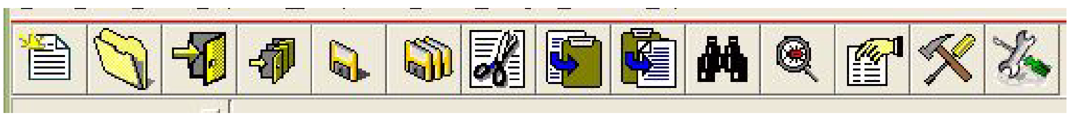
Note: While it is possible to create suffix rules for suffixes other than the supported `.c`, `.s`, `.S`, `.obj`, and `.o` extensions, such usage is not recommended.

11.5.4 Toolbar

The Toolbar, shown in Figure 11.34, provides easy access to commonly used functions of ZSP IDE.

Note: The icons shown here are the standard or default icons for the tools. The View Toolbar section on page 11-22 describes how to customize the icon assignments.

Figure 11.34 ZSP IDE Toolbar

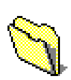


The following functions are available through the toolbar.

New File

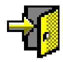
Select the New File toolbar button to create a new text file in the IDE editor window. The new file is not automatically included in the current working project. If you want the new file to be a project component, use the Add File option either from the Main menu's Project menu or from the Project Tree 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 want the opened file to be a project component, use the Add File option either from the Main menu's 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 being edited in the IDE edit window.

Close All

Select the Close All toolbar button to close all of the text files that are 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.

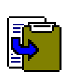
Save All

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

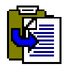
Select the Cut toolbar button to cut selected (highlighted) text from the editor window.

Copy



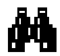
Select the Copy toolbar button to copy the selected text from the editor window into the clipboard buffer.

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

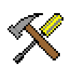


Select the Find toolbar button to display the Find dialog box, which allows searching the current source file for the desired text.

Settings

Select the Settings toolbar button to display the Settings window for the currently selected project or workspace.

Build

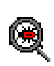


Select the Build toolbar button to display the build tools using the settings from the currently selected project.

Compile

Select the Compile toolbar button to compile the currently selected source file.

Debug

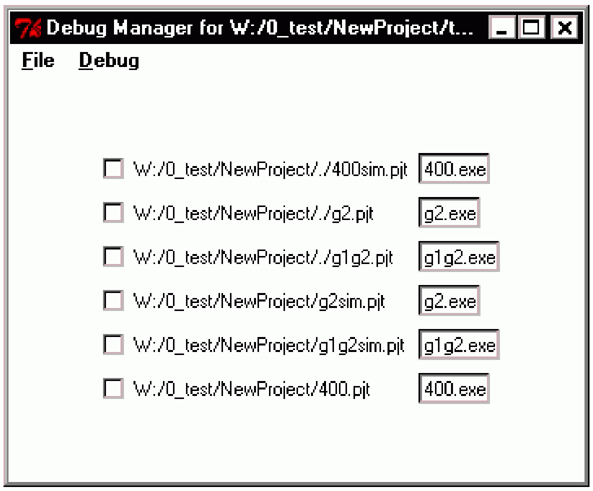


Select the Debug toolbar button to invoke the GUI debugger for the currently selected project.

11.6 Parallel Debug Manager

The Parallel Debug Manager is invoked by selecting Debug -> Invoke PDM in the Main menu. When PDM starts, the Debug Manager dialog box, shown in Figure 11.35, is displayed in which you may select the projects from within your workspace that you want to debug. Click in the check boxes to select projects.

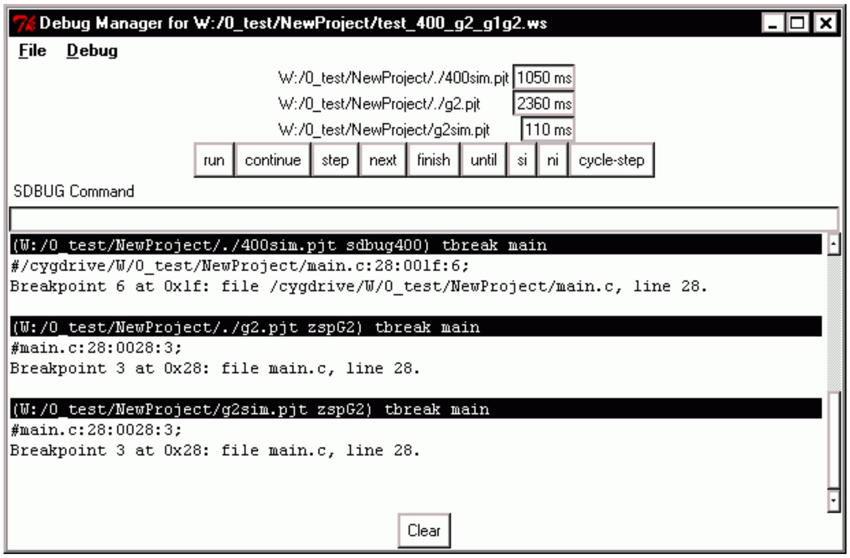
Figure 11.35 Debug Manager Dialog Box



Select Run from the Debug menu to start debugging. The Debug Manager dialog box changes to debugging mode, as shown in Figure 11.36, 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 have output displayed in the PDM output window for each of the projects being debugged.

Figure 11.36 Debug Manager Control Window

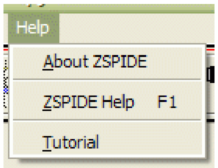


11.7 Help Menu

The Help menu, shown in Figure 11.37, provides three choices:

- About ZSPIDE displays the About dialog box. It contains system information including version numbers for various components of the SDK tools.
- ZSPIDE Help displays the online help for ZSPIDE.
- Tutorial runs the tutorial demonstration.

Figure 11.37 Help Menu



11.8 Editor

The ZSP IDE Editor is a window where you can write your code. It allows basic editing functionality.

Chapter 12
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.

ZSP IDE Debugger is a menu-driven user interface to the ZSP Command-Line Debugger. It provides a user-friendly graphical interface that allows navigation through application 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, and 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.

12.1 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 projects created with previous versions of SDK Tools.
- Windows and UNIX Debugger platforms
- Support for multiple targets
- Processor Register Windows - Operand, Control, Address Registers (G2)

- Displays cycle-accurate simulator information, code statistics, code profile, instruction grouping rules, core pipeline.
- 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 the GNU Debugger (sdbug400, zdbug, zdxbug) for the ZSP processor.

Table 12.1 Command Line Debugger Executables

Target	Command Line Debugger
ZSP400	sdbug400.exe
G2	zdbug.exe
G1G2	zdxbug.exe

Target Interfaces –

Table 12.2 Debugger Targets

Simulator targets
Cycle accurate simulator (zsim for ZSP400 and G2)
Instruction level simulator (zisim) for ZSP400 and G2
Hardware Targets
Corelis PCMCIA based JTAG for ZSP400 and G2
Corelis PCI based JTAG for ZSP400 and G2
Greenhills Slingshot JTAG for ZSP400
FS2 JTAG for ZSP G2
UART (Serial Port) for ZSP400

ZSP IDE Debugger supports JTAG hardware targets, UART (Serial Port) hardware target, ZISIM instruction-accurate simulators, and ZSIM cycle-accurate simulators.

12.2 GUI Debugger Overview

12.2.1 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.

12.2.2 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.

12.2.3 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 12.2.7.3, "Top Level Window Presentation," page 12-8).

12.2.4 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.

12.2.5 Main Menu

The Main Menu provides access to major functions of the debugger such as controlling breakpoints, executing navigation commands, and displaying Debugging Windows.

12.2.5.1 Operating the Main Menu

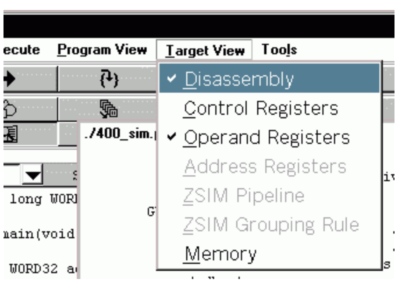
Main menu items may be selected either by left-clicking with the mouse or by typing a menu accelerator key (underlined character in the menu name). To invoke the menus from the keyboard, depress the ALT key and the accelerator key for the Main Menu item concurrently. This displays the pull-down subitem 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.

12.2.5.2 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 12.1 Menu Checkmarks For Debugging Windows



12.2.6 Main Toolbars

Toolbars are available as menu shortcuts to provide access to commonly used debugging features.

12.2.6.1 Available Toolbars

Toolbars exist for the following areas:

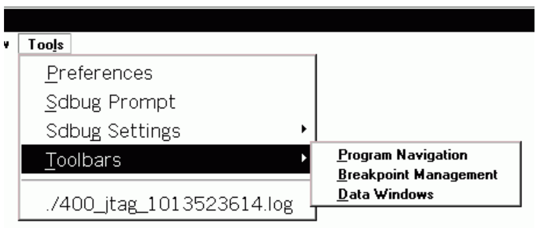
- Program navigation (Execute Menu shortcuts)

- Breakpoint management (Breakpoint Menu shortcuts)
- Data windows (Debugging Window menu shortcuts)

12.2.6.2 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 12.2 Tools Menu - Invoke Toolbars



12.2.6.3 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, shown in Figure 12.3, then deselect the “use images” checkbutton.

Figure 12.3 Preferences - Use Images For Toolbar Buttons

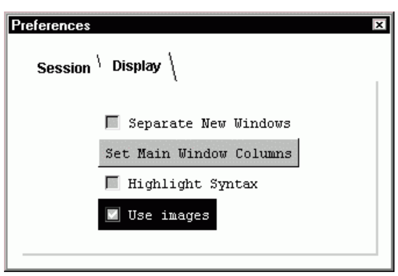


Figure 12.4 and Figure 12.5 show the appearance of the toolbar for each of these annotation modes. Each Toolbar Button has a text description that appears when the mouse cursor is moved over the button.

Figure 12.4 Toolbar Buttons with Text Annotation

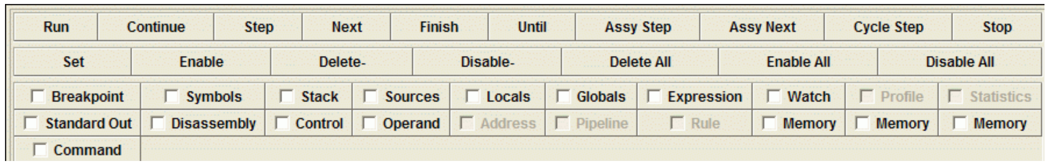
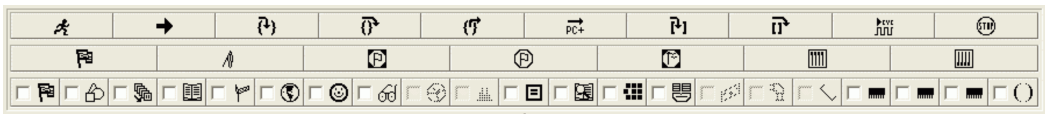


Figure 12.5 Toolbar Buttons with Image Annotation



12.2.7 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

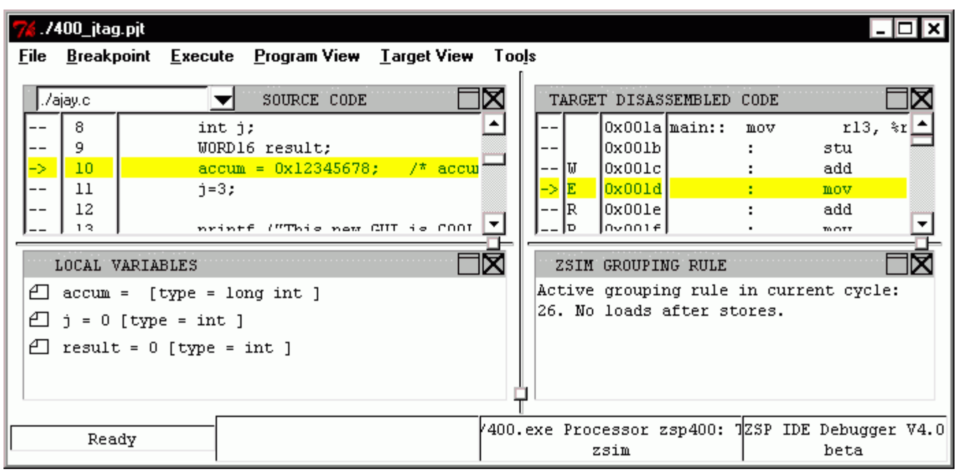
12.2.7.1 Debugging Window Operation

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.

12.2.7.2 Debugging Windows Paned Window Presentation

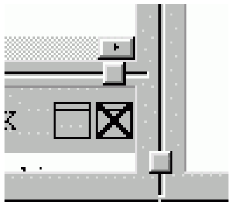
Debugging Windows appear by default in a Paned Window view as child windows within the Main Window. In this configuration, all windows appear at the same level—i.e., no separate Debugging Windows. Each Debugging Window may be separated from the Paned Window (see Debugging Window Top Level Preference on page 12-9 and Changing Debugging Window View Mode on page 12-10).

Figure 12.6 Debugger Paned Window



Paned Window Operation – Windows displayed in the Paned Window may be resized by dragging the handles of the paned window controls that separate the rows and columns of the Debugging Window area.

Figure 12.7 Paned Window Handles



Resizing columns affects all windows in that column while resizing rows only modifies one window plus its vertical neighbor.

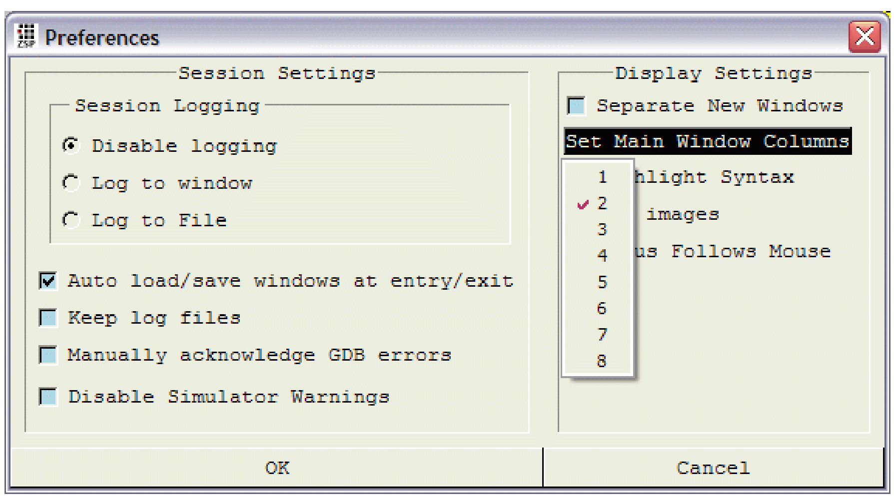
Paned Window Configuration – The presentation of windows in the Paned Window may be configured in 1-4 columns by selecting Preferences from the Tools Menu and “Main Window Columns” from the Preferences Window Display Tab. To change the number of columns displayed during a session,

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)

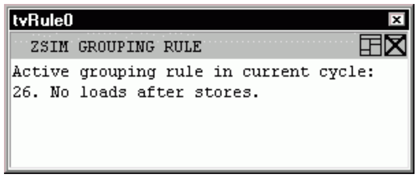
Figure 12.8 Preferences - Set Main Window Columns



12.2.7.3 Top Level Window Presentation

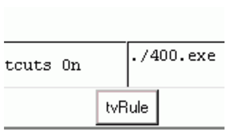
Top Level presentation of a Debugging Window displays that window as a separate Top Level window.

Figure 12.9 Top Level Debugging Window



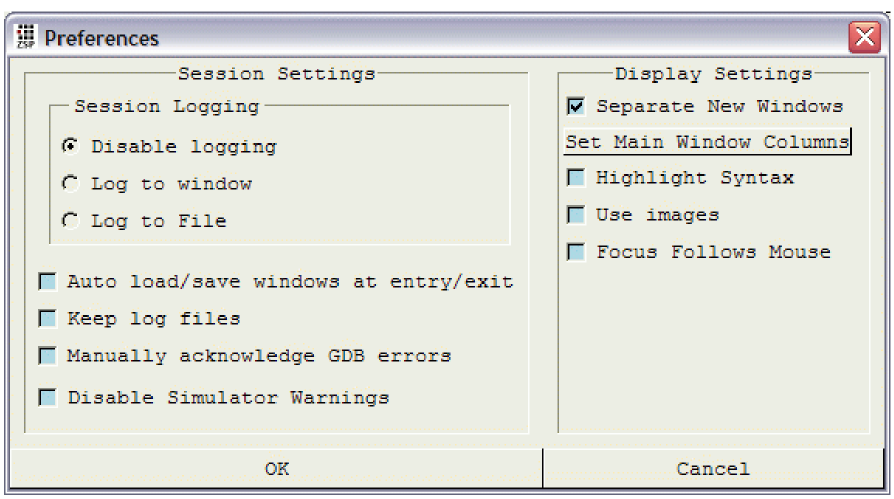
Top Level Focus Control – Top Level Debugging Windows that are obscured by other graphics on the screen may be brought into focus for viewing by selecting the corresponding Window Button on the toolbar at the bottom of the Paned Window.

Figure 12.10 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 12.11 Preferences - Separate New Window

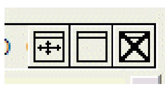


12.2.7.4 Changing Debugging Window View Mode

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.

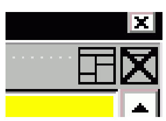
Click the left mouse button on the Window icon to separate the window into a Top Level Window. Click the left mouse button on the "X" icon to close the window. To relocate the window within the main paned window, depress and hold the left mouse button on the arrow window icon, drag the mouse cursor to the desired new position and then release the left mouse button.

Figure 12.12 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.

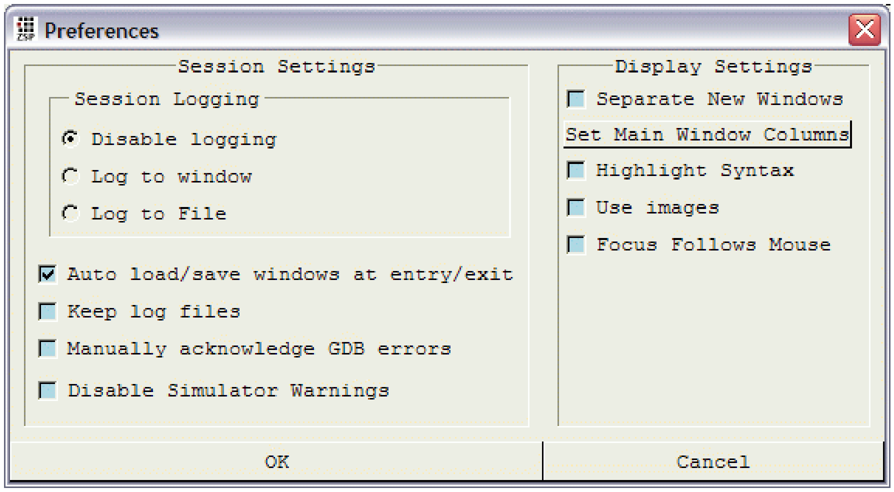
Figure 12.13 Display Controls for Top Level Window



12.2.7.5 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 12.14 Preferences - Autoload Windows



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.

12.3 Detailed Descriptions

12.3.1 Main Menu

12.3.1.1 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

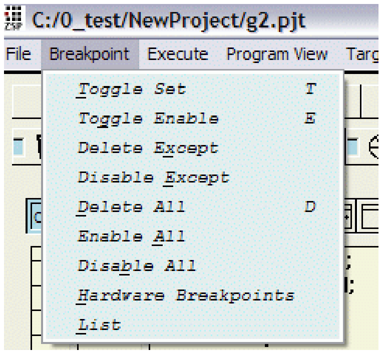
12.3.1.2 Breakpoint Menu

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 is 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 12.15 Breakpoint Menu



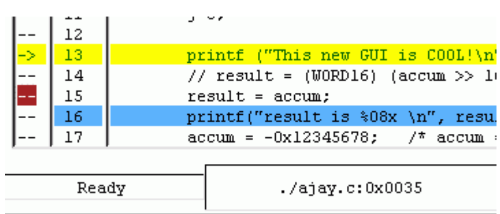
Current Selection Line – When setting a breakpoint from the breakpoint menu, the breakpoint is set at the Current Selection Line.

At the completion of each program navigation step (e.g. 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 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 are applied to that line.

Figure 12.16 Source Code Window Current Selection Line



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 Functions –

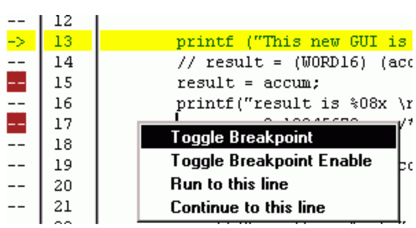
Toggle Set
When ‘Toggle Set’ is selected from the breakpoint menu, the debugger checks for the existence of a breakpoint at the current line. If a breakpoint exists, it is deleted. If a breakpoint does not exist, one is created at the current line.

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"

Figure 12.17 shows an example of Source Code Window breakpoint controls and displays:

Figure 12.17 Source Code Window Breakpoints



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, 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, 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, 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, a breakpoint at the current line is created.

Delete All
Selecting "Delete All" from the Breakpoint Menu causes all breakpoints to be deleted.

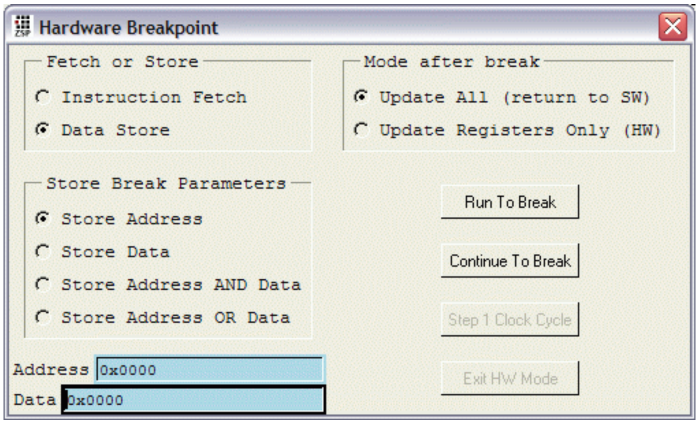
Enable All
Selecting "Enable All" from the Breakpoint Menu causes all existing breakpoints to be enabled.

Disable All
Selecting "Disable All" from the Breakpoint Menu causes all existing breakpoints to be disabled.

Hardware Breakpoints –

The JTAG hardware targets for ZSP400 and ZSP G2 allow hardware breakpoints to be maintained within the device emulation unit (DEU) of the processor.

Figure 12.18 ZSP400 Hardware Breakpoints Window



The ZSP400 hardware breakpoint window provides the capability to break on either of the following conditions:

- Instruction Fetch - Select Instruction Fetch to cause a breakpoint when the core fetches the instruction at the address specified in the Address entry box.
- Data Store - Select Data Store to cause a breakpoint when the core executes a store instruction.

Select “Store Address” to cause a breakpoint when any data is stored to the address specified in the Address entry box.

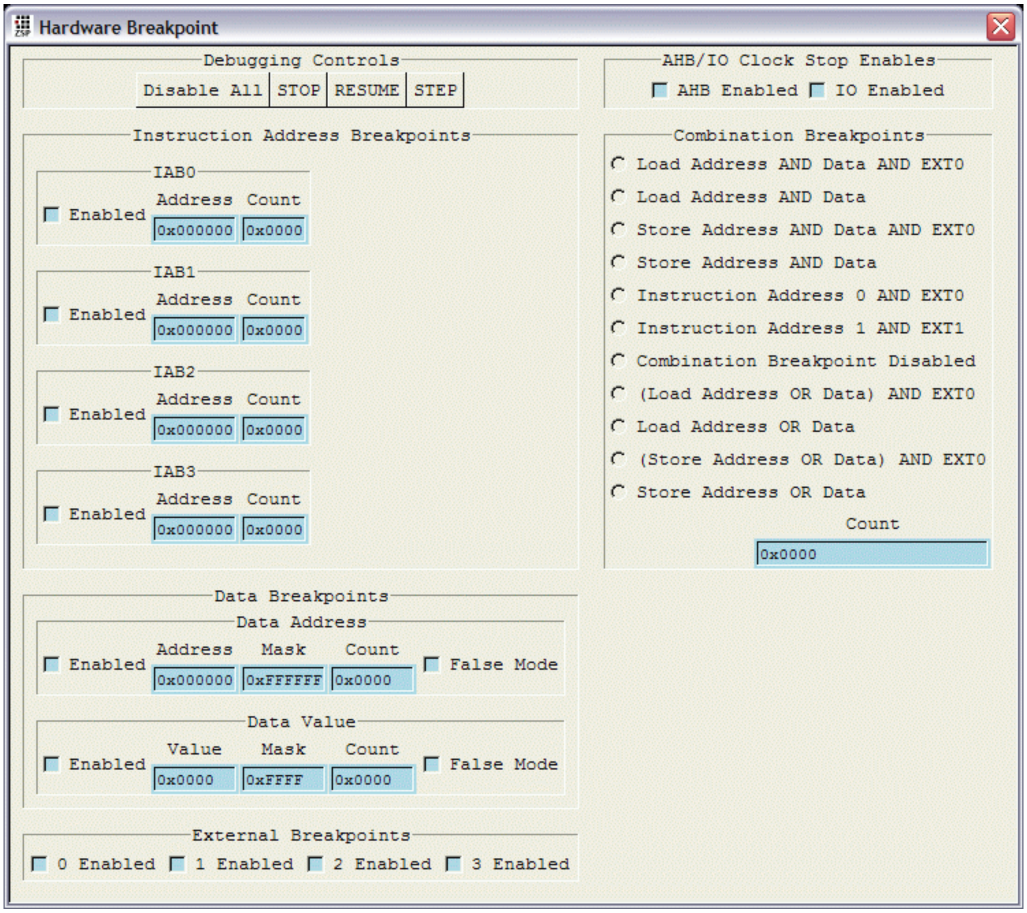
Select “Store Data” to cause a breakpoint when a the data specified in the Data entry box is stored to any address.

Select “Store Address AND Data” to cause a breakpoint when both address and data conditions are met within a single store instruction.

Select “Store Address OR Data” to cause a breakpoint when either address or data condition is met within a single store instruction.

“Mode After Break” determines the behavior of the GUI Debugger after the breakpoint is encountered. When the hardware breakpoint is encountered, the core clock is stopped and the only debugging information available to the debugger through a scan operation is the register contents. Select “Update All” to restart the core clock and perform a software debugging refresh operation after the hardware breakpoint is encountered. Select “Update Registers Only” to leave the debugger in hardware break mode. Register contents will be updated but other windows will not. When in hardware break mode, single stepping is allowed by selecting “Step 1 clock cycle”. After each single step operation, the register contents only will be refreshed. To return to normal software debugging, select “Exit HW Mode”. This sends the hw return_to_sw_dbg command to the command line debugger.

Figure 12.19 ZSP G2 Hardware Breakpoint Window



ZSP G2 Hardware breakpoints are used to stop the clocks to the G2 core at a designated execution point. In order to resume execution, the debugger sends a DEU RESTART command to the DEU.

Instruction Address Breakpoints

The ZSP500 DEU provides four 24-bit instruction address breakpoints. Each instruction address breakpoint comprises an address, enable, and 16-bit counter. The breakpoint activates when the counter is zero. The counter (when non-zero) decrements each time the breakpoint address is encountered until it reaches zero. The breakpoint counter register within the DEU is write-only, so the value displayed in the GUI panel is always the initial value.

Data Address Breakpoint

A single data address breakpoint enables you to stop the core clock when the data address is issued to load data from memory or store data to memory. The data address breakpoint comprises an address value, enable, mask, counter, and false mode directive. The 24-bit address value is compared against the load address and the store address for both the load/store parts of the core. The counter for the data address breakpoint behaves the same way as the instruction address breakpoint counter. The 24-bit mask register causes the comparison logic within the DEU to ignore certain bits in the address. When a bit is set in the data address mask register, the corresponding bit in the data address comparison is ignored. The false mode is used to stop the core clock when the condition indicated by the address and mask fields are false.

Data Value Breakpoint

A single data value breakpoint enables you to stop the core clock when a specified data value is loaded from memory or stored to memory. When loads or stores that have a size greater than 16 bits are issued, the 16-bit data value is compared against every valid 16-bit portion of the load / store value. The data value breakpoint comprises a data value, enable, mask, counter, and false mode directive that behave in the same manner as the corresponding elements of the Data Address Breakpoint.

External Breakpoints

Four external pins are provided that can cause breakpoints. These controls allow enabling each of the breakpoints based on the input at these pins.

Combination Breakpoint

The logical combination breakpoint is a logical AND or OR of other breakpoints. The logical combinations are

- Store Address Value AND Store Data Value
- Store Address Value OR Store Data Value
- Load Address Value AND Load Data Value
- Load Address Value OR Load Data Value
- Store Address Value AND Store Data Value AND External BP0

- (Store Address Value OR Store Data Value) AND External BP0
- Load Address Value AND Load Data Value AND External BP0
- (Load Address Value OR Load Data Value) AND External BP0
- Instruction Address 0 AND External BP0
- Instruction Address 1 AND External BP1

For each logical combination breakpoint, care must be taken. Each of the terms in the logical combination may contain an individual breakpoint counter. These counters must be set to 0 for the logical combination breakpoint to operate. All breakpoints involved in the combination are enabled by the debugger when the combination is enabled.The logical combination breakpoint also contains a 16-bit counter. The breakpoint can only activate when this counter reaches zero. The counter, when non-zero, decrements every time that the logical combination is hit.

List

Selecting "List" from the Breakpoint Menu displays a Debugging Window showing details of breakpoints that are currently set.

Note: For details about the Breakpoint List Window, see Breakpoint List Window on page 12-27.

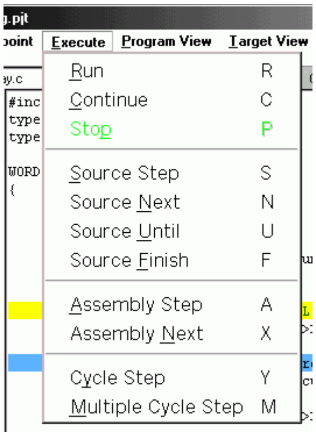
12.3.1.3 Execute Menu

The Execute Menu, shown in Figure 12.20, 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 12.20 Execute Menu



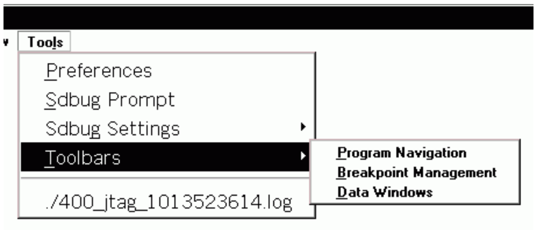
Alternative to execute menu for execute functions – Additional means of navigation are:

- 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 Program Navigation from the Toolbar submenu of the Tools Menu, as shown in Figure 12.21.

Figure 12.21 Toolbar Submenu



Keyboard Shortcut Keys

The keyboard shortcut keys listed in Table 12.3 allow single-keystroke navigation through program execution.

Table 12.3 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
P p	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 is invoked for navigation through the debug execution steps. If the current source is assembly code then Step advances by one assembly instruction, stepping into 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 follows calls to step into functions.

Assembly Next

Assembly next advances program execution by an assembly-level instruction. Assembly next steps over calls and does 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.

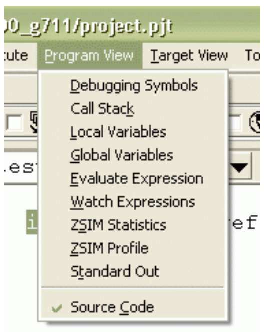
Multiple Cycle-Step

Multiple Cycle-step advances program execution by a user-selected number of processor clock cycles. Multiple Cycle-step is available for the ZSIM simulator target only.

12.3.1.4 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 12.3.2.1, "C/Assembly Program Windows," page 12-24 for detailed window information.

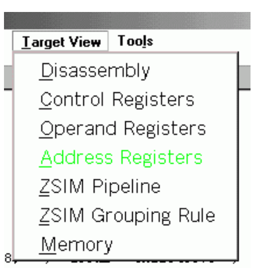
Figure 12.22 Program View Menu



12.3.1.5 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 12.3.2.2, "Target Windows," page 12-33 for detailed window information.

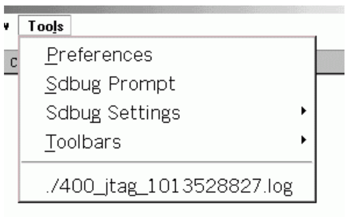
Figure 12.23 Target View Menu



12.3.1.6 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 12.3.2.3, “Tools Windows and Functions,” page 12-40 for more information on the Tools Menu items.

Figure 12.24 Tools Menu



12.3.1.7 Help Menu

The Help Menu provides help.

12.3.2 Debugging Window Detailed Descriptions

12.3.2.1 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
- Standard Output Window

Source Code Window – The Source Code window displays the program source files for debugging. The locations 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”.

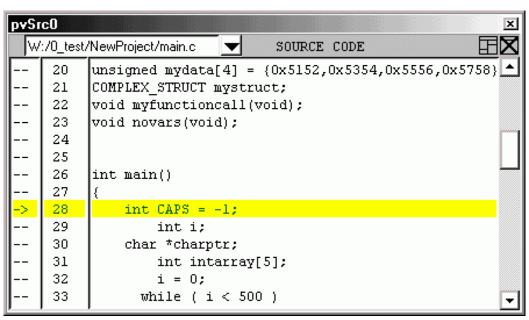
Program Execution Tracking

Tracking of program execution is visible through the Source and Disassembly Windows. The Current Line is highlighted as the next instruction to be executed.

Source Code Window Display

The Source Code Window displays 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 12.25 Source Code Window



Source Code Window Syntax Highlighting

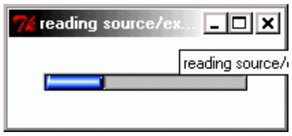
If the project preferences indicate that syntax highlighting is desired, each file is highlighted at creation.

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, the progress bar is also displayed as a Top

Level window. Otherwise, the progress bar is displayed in the Main Window status area.

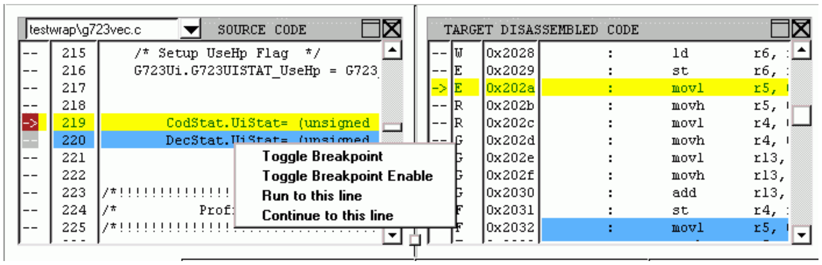
Figure 12.26 Progress Bar Window



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 12.27 Source Code Window (Shown with Disassembly Window)

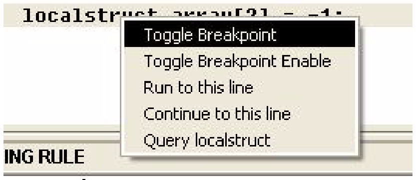


- Source Code Window Breakpoint Area
  The breakpoint area shows enabled breakpoints in red and disabled breakpoints in gray. The current line is indicated by an ASCII arrow.
- Source Code Window Line Highlighting
  The Source Code Window has two important items highlighted for user information: the Current Program Execution Line and the User-Selected Line.
  - Source Code Window Current Program Execution Line
    This is indicated by a highlighted background on the code and line number areas.
  - Source Code Window User-Selection Line
    This is 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.

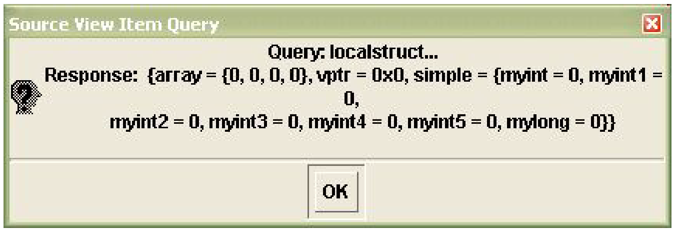
- Source Code Window Popup Menu
  The popup menu for the source code or Disassembly Window is invoked by right-clicking the mouse over the code area. The popup menu allows you to toggle a breakpoint or breakpoint enable at the selected line, run from start to the selected line, or continue from the current 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.

Figure 12.28 Source Code Window Popup Menu



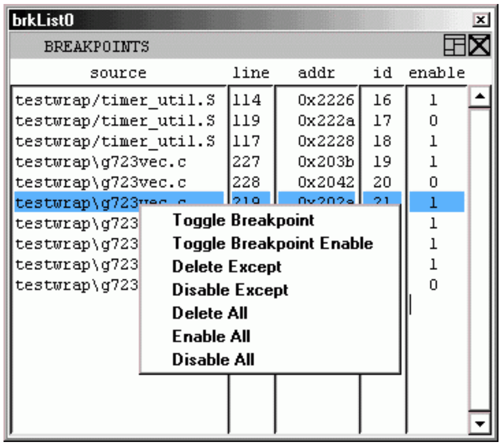
The source code window popup menu also allows a command-line debugger query to be performed using the word beneath the mouse pointer as the query expression. Figure 12.29 shows a sample query result.

Figure 12.29 Example Source Code Popup Query Result



Breakpoint List Window – Selecting "Breakpoint -> List" from the Main Window causes a Debugging Window window to be displayed showing details of breakpoints currently set.

Figure 12.30 Breakpoint List Window



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

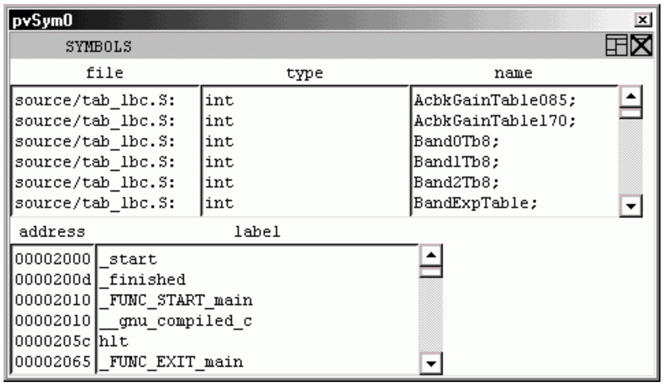
Right-click on a line in the breakpoint list to display a popup menu of breakpoint operations that may be applied to the selected Breakpoint.

Saving Breakpoints

Breakpoints are saved and restored with the project session when Autoload is selected 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 12.31 Debugging Symbols Window

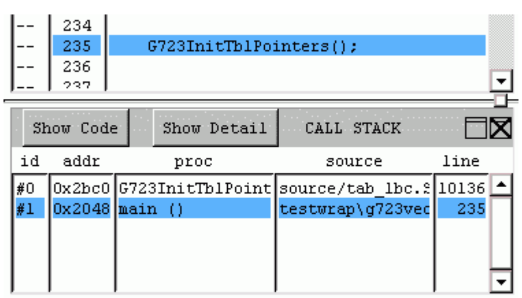


- Program Data Symbols
  The Symbols Window lists variables that are global, indicates the source file in which they are defined, and lists the data type associated with the variable.
- Program Instruction Symbols
  The Symbols Window lists instruction labels for the program being debugged and the associated addresses.

The Debugging Symbols Window is only populated when it is invoked, since it does not change within the debugging session.

Call Stack Window – To display a program's Call Stack, select Call Stack from the Program View Menu.

Figure 12.32 Call Stack Window



Call Stack Code Viewing

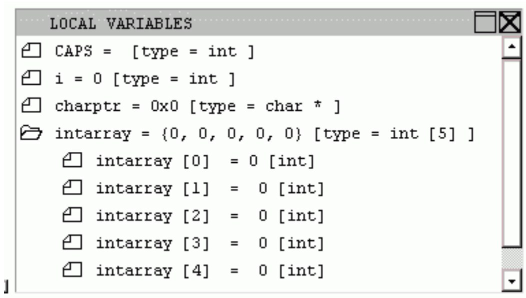
To view the code associated with one of the stack levels displayed, select that line in the Stack Window and select the Show Code button. The Source and Disassembly Windows will display the associated code.

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.

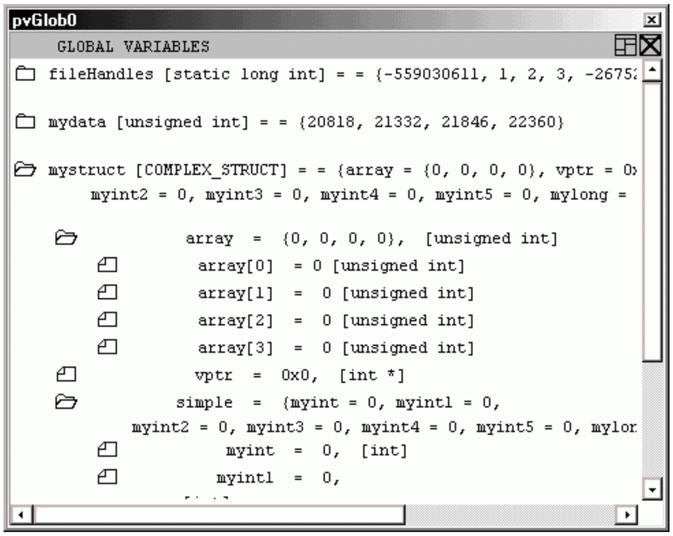
Local Variables Window – To display local variables, select Local Variables from the Program View Menu. The Local Variables Window shows all variables that are in the local scope.

Figure 12.33 Local Variables Window



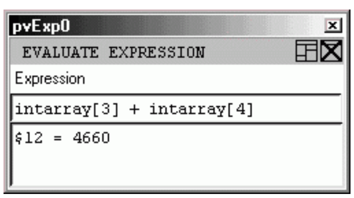
Global Variables Window – A view of global variables is available from the Main Menu by selecting ’Program View > Globals’. The Global Variables Window shows all variables that are global in scope.

Figure 12.34 Global Variables Window



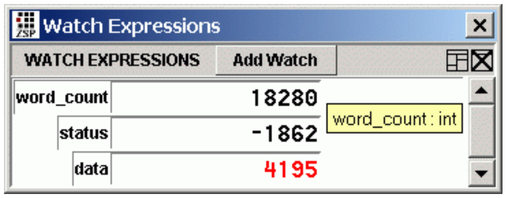
Expression Window – To have the debugger evaluate and display a single expression at each display refresh, use the Evaluate Expression Window. To invoke the Evaluate Expression Window, select Evaluate Expression from the Program View Menu. Type the expression you want to evaluate into the entry area and the expression will be evaluated and displayed after each execution step

Figure 12.35 Expression Window



Watch Expression Window – 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 right-clicking on the watch expression and selecting Remove from the popup menu.

Figure 12.36 Watch Expressions Window

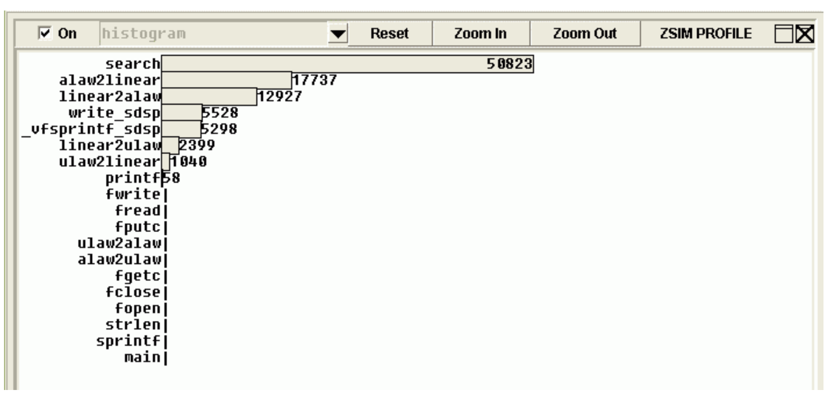


ZSIM Target Windows – ZSIM Debugging windows are available when ZSIM is selected as the target in the IDE Debug>Setup window.

- ZSIM Profile Window
  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. A reset button is provided on the profile view submenu to reset the collection of profile information to the current execution point.

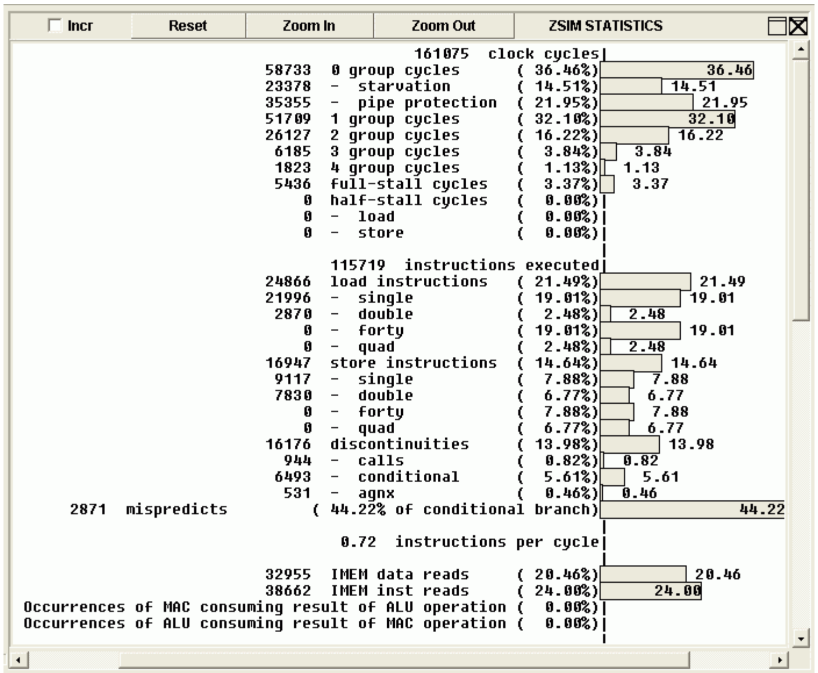
  The Profile Window shows each function name that is available for profiling, the histogram, cumulative and calls information reported by ZSIM. A bargraph chart is displayed with data type selectable from a drop-down selection box.

Figure 12.37 ZSIM Profile Window



- ZSIM Statistics Window
  A view of code execution statistics is available for the ZSIM target by selecting Statistics from the Program View Menu.

Figure 12.38 ZSIM Statistics Window



Standard Output Window –

A view of the output produced by the program being debugged (by invoking printf etc.) is available through the standard output window.

12.3.2.2 Target Windows

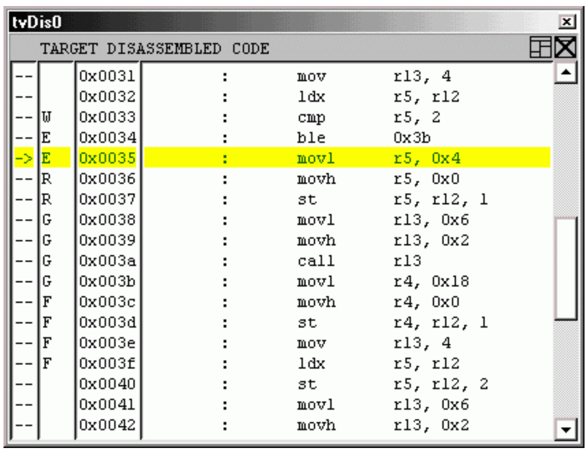
Available from the Target View Menu or from the Window Toolbar, the Target Windows display data pertinent to the state of the processor after each navigational step in the debug session. Available Target Windows include

- Disassembly Window
- Control Registers Window
- Operand Registers Window
- Address Registers Window (G2 only)
- Memory Window
- ZSIM Grouping Rule Window
- ZSIM Pipeline 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.

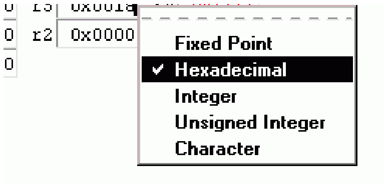
Figure 12.39 Disassembly Window



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.

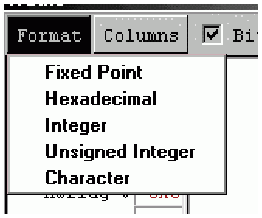
Figure 12.40 Register Element Popup Format Menu



Register Windows each contain a subwindow menu that includes the following functions.

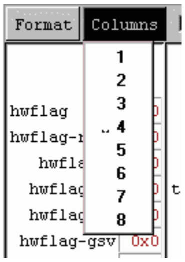
- Format
  The Format Menu in a Register Window, shown in Figure 12.41, allows reformatting of data for all of the visible registers in one of the following formats:
  - Fixed Point (for 16-, 32-, or 40-bit numbers)
  - Hexadecimal
  - Integer
  - Unsigned Integer
  - ASCII Character

Figure 12.41 Register Window Format Menu



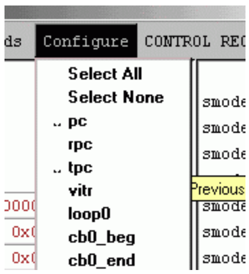
- Columns
  The Columns Menu in the Register Window allows arrangement of the individual registers in the Window into 1-8 columns.

Figure 12.42 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.

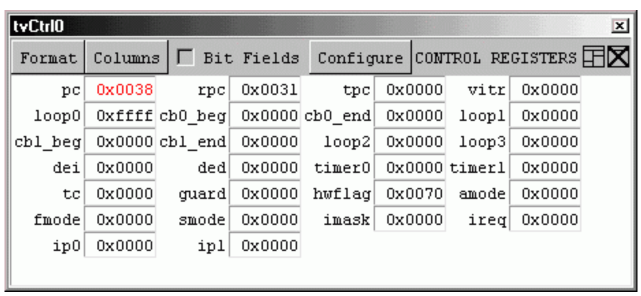
Figure 12.43 Register Window Configure Menu



Control Registers Window –

The Control Registers Window provides access to the target processor's control registers.

Figure 12.44 Control Register Window - Standard Mode



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 12.45 Control Register Bitfield Entry Annotation

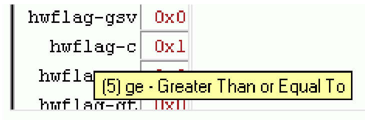
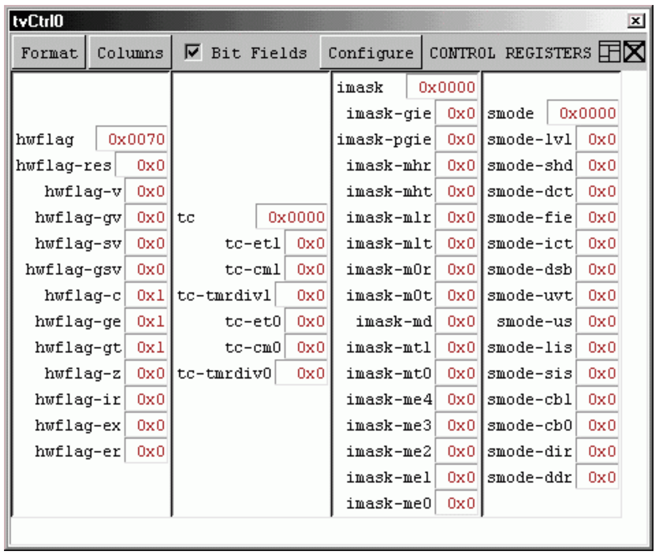
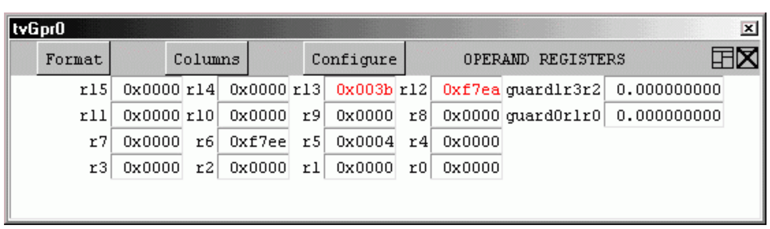


Figure 12.46 Control Register Window - Bitfield Mode



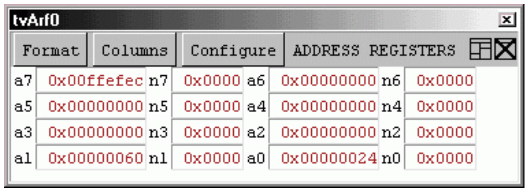
Operand Registers Window – The Operand Registers window provides access to the target processor's operand registers. Menu items in the operand register Window include Format, Columns, and Configure functionality described above in the general Register Window description.

Figure 12.47 Operand Register Window



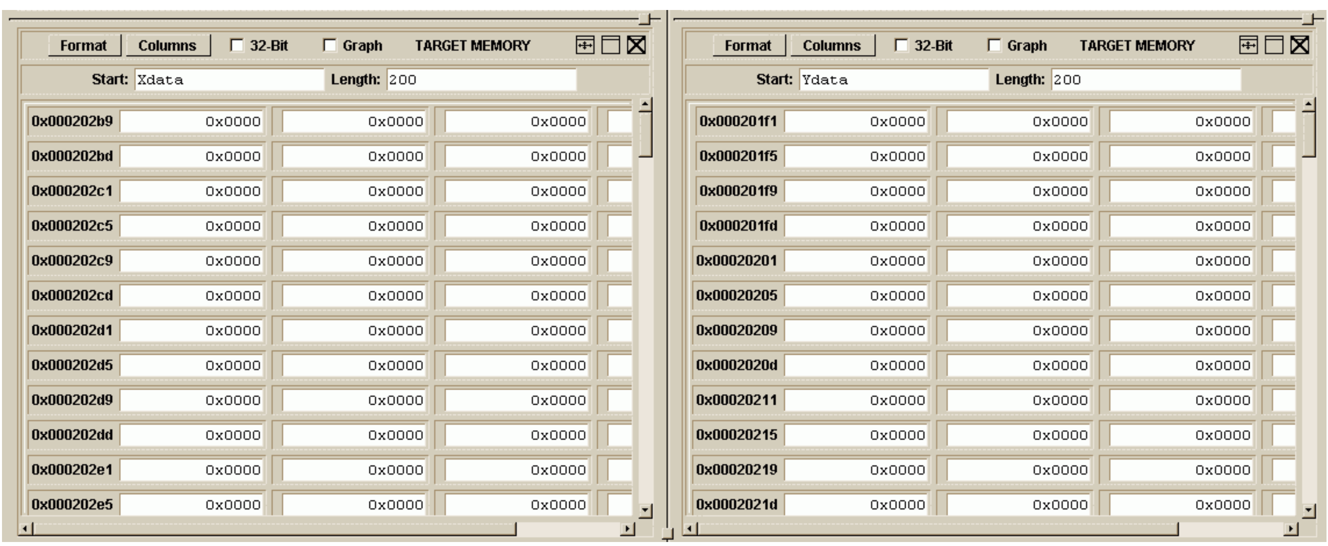
Address Registers Window (G2) – The Address Registers window provides access to the target processor's address and index registers. Menu items in the operand register Window include Format, Columns, and Configure functionality described above in the general Register Window description.

Figure 12.48 Address Registers Window



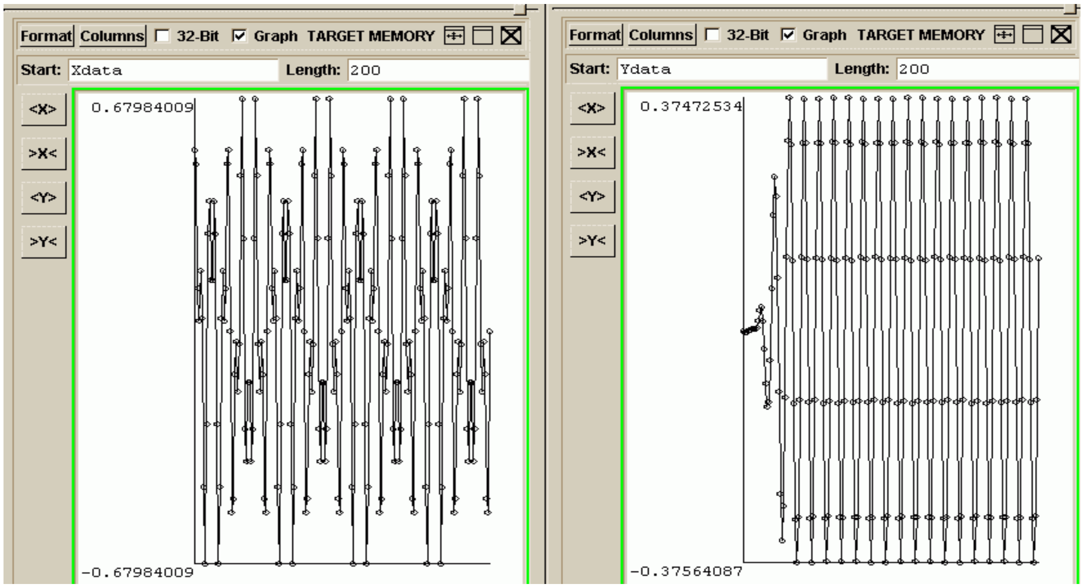
Memory Window – The Memory Window provides access to the target processor's memory. Menu items in the memory Window include Format and Columns functionality described above in the general Register Window description. Memory may displayed in up to 16 columns.

Figure 12.49 Memory Window



Checkboxes for 32-Bit and Graph cause memory to be displayed in those formats. Figure 12.50 shows a Graph display of the memory.

Figure 12.50 Graph Display of Memory

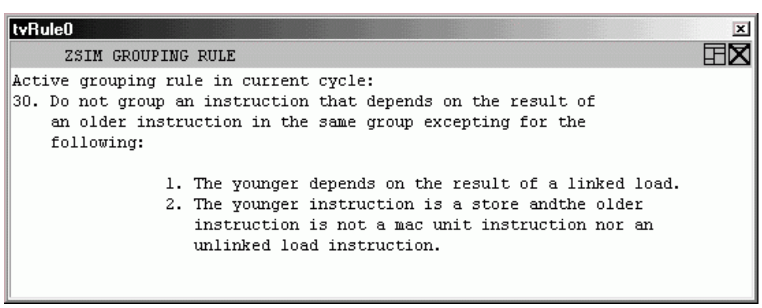


The start address for the memory Window may be an address or debugging symbol.

ZSIM Target Windows – ZSIM Debugging Windows are available when ZSIM is the current debugging target.

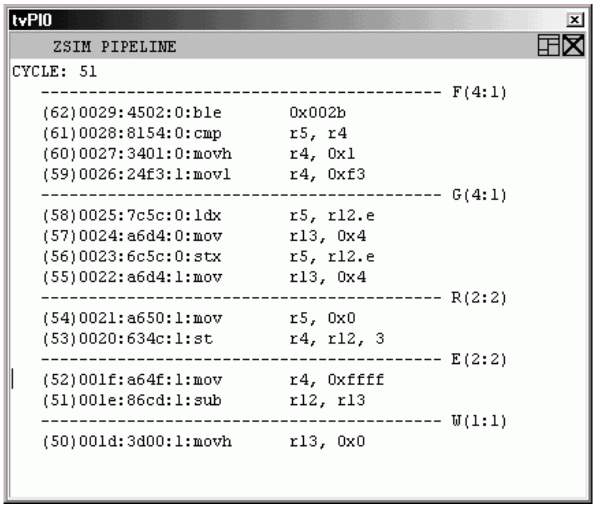
- ZSIM Grouping Rule Window
  The Grouping Rule Window displays ZSIM instruction grouping information. The rule displayed applies to instructions currently in the grouping stage in the pipeline.

Figure 12.51 ZSIM Grouping Rule Window



- ZSIM Pipeline Window
  The ZSIM Pipeline Window displays ZSIM pipeline information.

Figure 12.52 ZSIM Pipeline Window

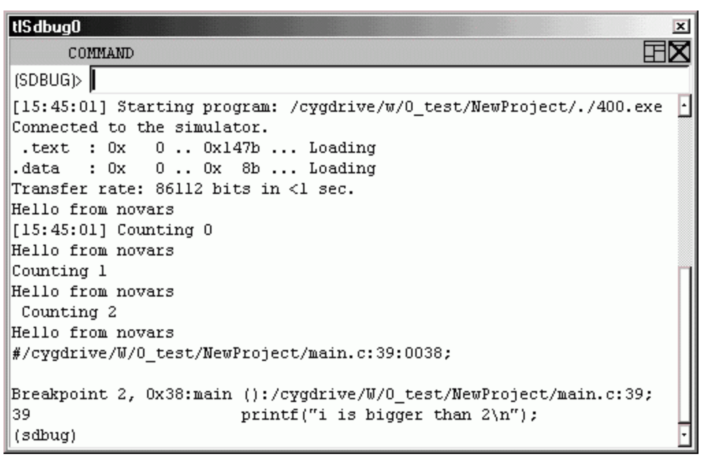


12.3.2.3 Tools Windows and Functions

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 12.53 Command Line Window



12.3.2.4 Data Graphing

The LSI Debugger IDE has an integrated Data graphing (DG) module, which can plot real time data. You can plot both "C" variables and memory in one or more independent DG windows. You can generate 2-dimensional plots for "C' and Memory variables and 3-dimensional plots for memory vectors (a set of values from consecutive memory locations).

The following section describes how to use a DG module for plotting.

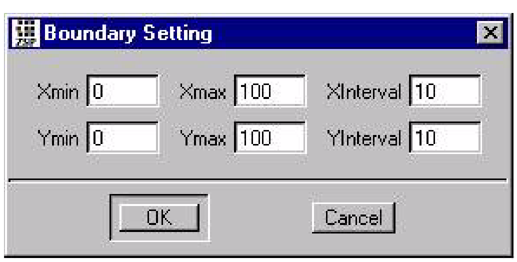
Launching a DG Window –

2D Plotting

To plot a 2D graph, launch a new DG window by clicking on Tools > DataGraph Plotter > 2-Dimensional Plot in the Debugger IDE. On doing this:

- The Boundary Setting dialog box appears, as shown in Figure 12.54. This is where you enter the minimum and maximum values to be plotted on the X and Y axes: Xmin, Xmax, Ymin, and Ymax, respectively, and the scales to be used on x-axis (X scale) and y-axis (Y scale).
- When all required information is entered, click on OK. A new DG window (top level) is opened with the entered values and with the respective scales on the X and Y axes.
- At this point, this DG window is ready for 2D plotting.

Figure 12.54 2D Boundary Setting Dialog Box

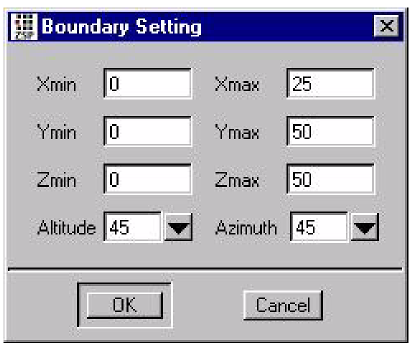


3D Plotting

To plot a 3D graph, launch a new DG window by clicking on Tools > DataGraph Plotter > 3-Dimensional Plot in the Debugger IDE. On doing this:

- The Boundary Setting dialog box appears, as shown in Figure 12.55. This is where you enter the minimum and maximum values to be plotted on the X, Y and Z axes: Xmin, Xmax, Ymin, Ymax, Zmin, and Zmax, respectively.
- Additional values must be entered for Altitude and Azimuth (in degrees) to set the viewing angles. The Altitude represents the viewing angle above the XY plane.
- The Azimuth is defined so that when it is 0, the observer sees the XZ plane face-on. As the angle is increased, the plot is rotated counter-clockwise as viewed from above the XY plane.
- When all required information is entered, click on OK. A new DG window for 3D (top level) is opened with the entered values of Xmin, Xmax, Ymin, Ymax, Zmin, and Zmax.
- At this point, this DG window is ready for 3D plotting.

Figure 12.55 3D Boundary Setting Dialog Box

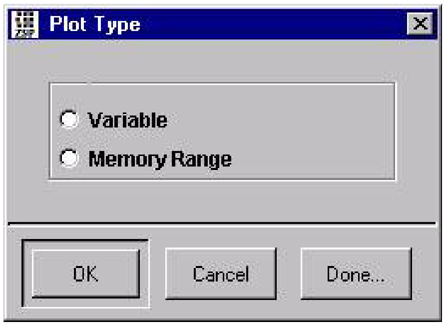


Setting up an Update Point –

An Update point is a marker in a source window. Whenever the application control flow crosses this point, all the plots related to this update point are updated. Here is the procedure for setting an Update point:

- Left-click on the line number column in the "C" source window in the Debugger IDE. The **Plot Type** dialog box appears, as shown in Figure 12.56). In this dialog box, you can choose to plot a "C" variable or a memory range.

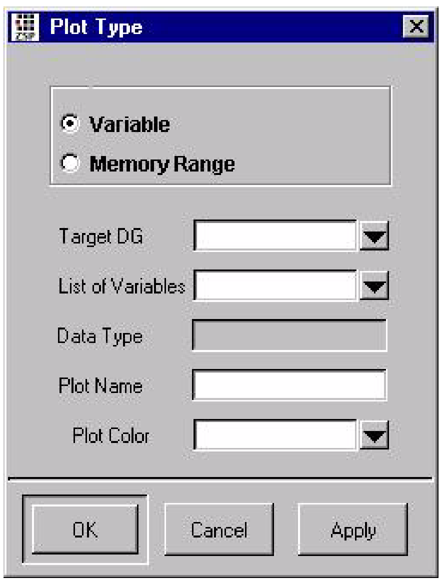
Figure 12.56 Plot Type Dialog Box



Plotting a Variable –

If you select Variable in the dialog box, the dialog box changes as shown in Figure 12.57 to accept various configurable options:

Figure 12.57 Plot Type Dialog Box - Variable Option



Here is an explanation of the options in this dialog box:

- **Target DG**: This shows a list of all DG windows where 2D plotting can be done. Select any one of these DG windows, where the variable will be plotted.
- **List of Variables**: This shows a list of all local and global "C" variables available in the current context for plotting. Select a variable to plot from this list.
- **Data Type:** This shows the data type of the selected variables.

- **Plot Name**: Specify a name to be displayed as a plot label in the DG window. The default name of the plot is the name of the C variable chosen from the list of variables.
- **Plot Color**: From this list of available colors you can select a color for plotting the graph.

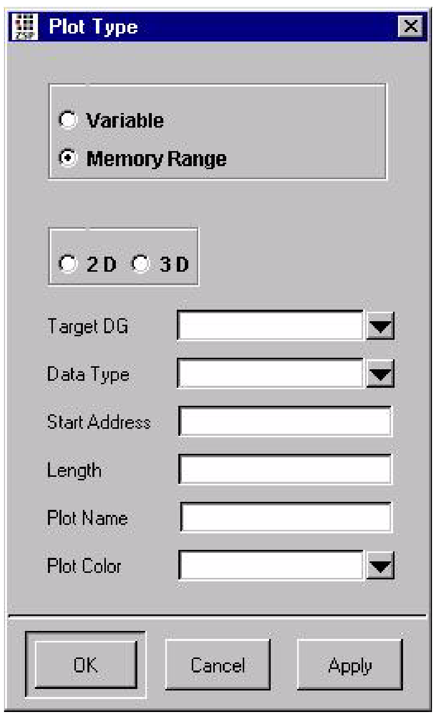
The Plot Name set as above will also be shown with the same color as that of Plot Color in the DG window.

Click the Apply button to save the current configuration. You can select as many plots as you want. When you are finished configuring all the plots, click the **OK** button to save the configuration and close the dialog box. You can click **Cancel** to cancel the current selection and close the dialog box. All previously configured plots will still be enabled.

Plotting Memory –

If you select Memory in the Plot Type dialog box, the dialog box changes as shown in Figure 12.58.

Figure 12.58 Plot Type Dialog Box - Memory Range Option



Here is an explanation of the options in this dialog box:

- **2D**: When you select this option, a 2-dimensional graph will be plotted. Selecting this option also sets the Length option to 1.
- **3D**: When you select this option, a 3-dimensional graph will be plotted.
- **Target DG**: This shows a list of all DG windows where plotting can be done. If you select 2D, this list shows only the DG window names where 2D plotting can be done. If you select 3D, the list shows only the DG windows where 3D plotting can be done.
- **Data Type**: This shows a list of available data formats to represent the values read from the memory. The following data formats are supported:
  - 16-bit integer
  - 32-bit integer
  - 32-bit floating
- **Start Address**: Specify the starting address in the memory from where the values should be read.
- **Length**: Specify the number of values that should be read and plotted from the Start Address.
- **Plot name:** Specify a name to be displayed as a plot label in the DG window. The default name of the plot is the starting address you have selected for plotting.
- **Plot Color**: From this list of available colors you can select a color for plotting the graph.

The Plot Name set as above will also be shown with the same color as that of Plot Color in the DG window.

The OK, Cancel, and Apply buttons have the same functionality as described above.

Removing an Update Point –

You can remove the existing Update Point by clicking the left mouse button over the Update Point.

Changing the Properties of an Update Point –

You can change the properties of an existing Update Point by clicking the right mouse button on the Update Point and following the same steps as described above.

When you run the debugger, whenever the application control crosses the Update Point, all the requisite data is channeled to DG windows, and the respective plots are updated.

DG Window Functionality –

The tables in this section describes the various menu options available in a DG window.

Table 12.4 DG Window - File Menu

Option	Functionality
Load Dataset Format …	Opens the Open file dialog box, where user can load a previously saved data set into DG Window.
Save Dataset Format …	Opens the Save dialog box, where user can save the plots of the current DG window data set format for future viewing.
Save as …	Opens a Save As dialog box to save the plots in the image format selected in the Save Image Format option.
Save Image Format	Allows the user to set the image format in which the plots are saved. Supported format options are PostScript (default), JPEG, and GIF. This option should be set before saving the plot in image format.
Exit	Closes the DG window.

Table 12.5 DG Window - Orient Menu

Option	Functionality
0 Degrees	The plot is shown as it is, with no rotation.
90 Degrees	The plot is shown rotated by 90^o counter-clockwise
180 Degrees	The plot is shown rotated by 180^o counter-clockwise
270 Degrees	The plot is shown rotated by 270^o counter-clockwise

Table 12.6 DG Window - Zoom Menu

Option	Functionality
Select	Enables the user to select any rectangular area in the DG Window by clicking and dragging the mouse pointer. When the mouse button is released, the selected area is zoomed in.
Back	Returns the DG window to the previous zoomed state.
Forward	This option works as complementary to the Back option.
Reset	Returns the DG Window to its original state.
Fit In Window	Causes the graphs to fit in the visible window.

Table 12.7 DG Window - Options Menu

Option	Functionality
Crosshairs	Cause the mouse pointer to take the shape of a plus (+). The user can see the coordinates on the x-axis and y-axis with the help of plus (+) shaped mouse pointer.
Appearance…	Opens the Appearance dialog box, shown in Figure 12.59. This is used to change the background color and axes color of the DG window.
Remove Plot…	Opens the Remove Plot dialog box, shown in Figure 12.60. This is used to remove plots related to that DG window.

Figure 12.59 Appearance... Dialog Box

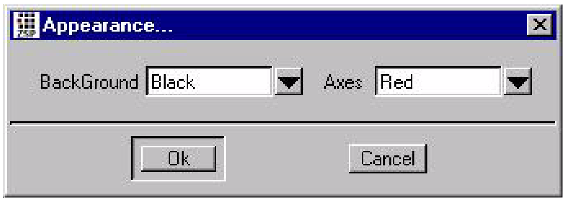
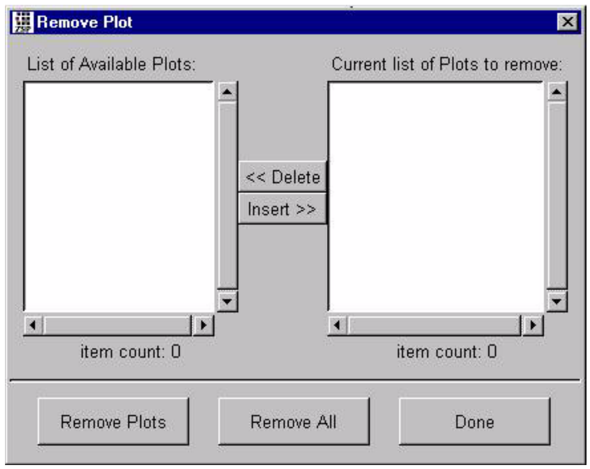


Figure 12.60 Remove Plot Dialog Box



Here is an explanation of the options in the Remove Plot dialog box:

- **List of Available Plots**: This show a list of all the currently available plots.
- **Current List of Plots to remove**: This shows a list of all the plots selected for removal.
- **Insert**: To move a plot from "List of Available Plots" to "Current List of Plots to remove", you select the plot and click the Insert button.
- **Delete:** To move a plot from "Current List of Plots to remove" back to "List of Available Plots", you select the plot and click the Delete button.
- **Remove Plots**: When you click this button, all selected plots in "**Current List of Plots to remove"** are removed from the DG window.
- **Remove All**: When you click this button, all plots currently drawn in the DG window are removed.
- **Done**: Click this button to close the **Remove Plot** dialog box.

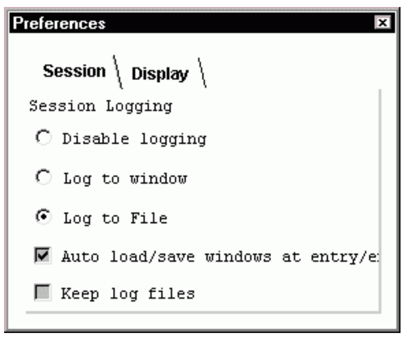
Help

Select the Help menu option in the DG window to display the About dialog box for the DG window.

12.3.2.5 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 12.61 Preferences Window - Logging



Session Log Types – The Session Log may be disabled by selecting the radio button labeled "Disable logging" in the Preferences Window. This setting is recommended for the best speed performance of the debugging environment.

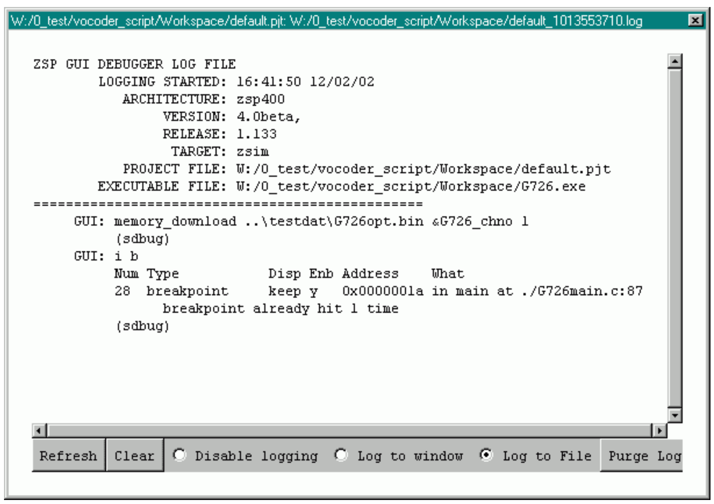
The Session Log may be directed to a window by selecting the radio button labeled "Log to Window" in the Preferences Window. Logging to a window provides continuous non-interactive update throughout the debugging session. Logging to a window is faster than logging to a file. There is no permanent Session 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 radio button 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 12.62). 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 12.62 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, as shown in Figure 12.63.

Figure 12.63 Session Log Window



Here is an explanation of the options in the Log Window:

- **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 Radio Buttons** - The “Disable Logging”, “Log To Window”, and “Log To File” radio buttons have the same functionality as their counterparts in the Preferences Window. These radio buttons allow 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 (that is, files with a .log extension) from your project directory.

Appendix A
Example Programs

This appendix contains three example programs, demo.c, hw_dbg.s, and pie.exe, that are referred to in previous chapters of this document, and a collection of files and scripts that demonstrate various aspects of the tools in more depth. 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. The third demonstrates the use of zdcc, zdas, zdopt, and zdar in producing an example executable that shows how in-line assembly and intrinsic functions are coded. It also shows how to relocate sections of an executable and how inter-section calls are performed.

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;
      }
    }

    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;
  }
```

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 **/
```

```
    .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 **/
```

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
```

```
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
```

```
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
```

```
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
```

```
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
```

A.3 Example Program pie.exe

This example illustrates compiling, assembling, and linking a program for the G2 architecture using the command-line tools under Solaris. The program itself performs two inter-section calls, shows how to use the N_vv_mac intrinsic, then calculates and prints 100 digits of π using two separate routines and calculates and prints 100 digits of *e*.

The files included are:

- build - A script that builds the pie.exe executable
- main.c - The entry function that calls the various demonstration routines.
- N_Intrinsic.c - A C routine that demonstrates the use of the N_vv_mac intrinsic.
- fast_e.c - The routine that calculates 100 digits of the constant *e*.
- fast_pi.c - A quick routine to calculate 100 digits of π.
- sections.s - An assembly routine that demonstrates how to call between sections.
- slow_pi.s - A somewhat slower routine to calculate 100 digits of π.

To build the executable, execute the build script, shown below:

```
#!/bin/csh -x

# This build script demonstrates how to build the example program.
#
# This example shows the following:
#   - How to use several compiler switches for zdcc
#   - How to call assembly from C
#   - How to re-locate sections of code
#   - How to build an archive library and link it with a program
#   - How to optimize assembly code using zdopt

# -O3 turns on maximum optimizations.  zdopt is run on the assembly
# output from compilation to optimize for the G2 architecture.
# -mlong_call is required since we are going to re-locate this func
#   very far from the main function.  Also, look in the file at the
#   in-lined assembly directive changing the relative start of this
#   file within the text segment.
# -mlarge_data is required since we are going to re-locate the data
#   and bss segments far from the text segment.
zdcc -O3 -mlong_call -mlarge_data -c fast_pi.c


# Same as before, except no optimizations, no relocation in-line.
zdcc -c -mlong_call -mlarge_data fast_e.c N_Intrinsic.c

# Optimize the slow pi calculation routine
zdopt -asm slow_pi.s > slow_opt_pi.s

# Assemble the slow version of pi computation.
zdas -o slow_pi.o slow_opt_pi.s

# Assemble 'foo' which calls between sections
zdas -o sections.o sections.s

# Now, build an archive library containing the fast versions of the
#   objects used for computing pi and e.  The library is fast.a.
zdar r fast.a fast_pi.o fast_e.o N_Intrinsic.o sections.o

# Multiple steps are done here.  main.c is compiled with long calls
#   and large data revferences and linked with the objects created
#   previously. The bss segment is relocated to address 0x03f000,
#   the data segment is relocated to address 0x07e700, and the text
#   segment (code) is relocated to 0x455.  A map file is also
#   produced and printed on stdout.
# The resulting executable, pie.exe, can be simulated with zisimg2
#   or zsimg2.
zdcc -mlong_call -mlarge_data -o pie.exe main.c fast.a slow_pi.o
-Tbss 0x03f000 -Tdata 0x07e700 -Ttext 0x455 -Wl,-M
```

When the resulting executable is executed using the command:

```
zisimg2 -exec pie.exe
```

the following output is produced:

```
***(info) Starting address: 0x0455
.text   : Loading to INT-INST memory... 0x0455 -> 0x1d5d0 (0x1d17c)
text2   : Loading to INT-INST memory... 0x1d5d2 -> 0x1d5d5 (0x0004)
text1   : Loading to INT-INST memory... 0x1d5d6 -> 0x1d5dd (0x0008)
.data   : Loading to INT-DATA memory... 0x7e700 -> 0x7e7ac (0x00ad)
Loading "pie.exe" successfully.
*************************************************
            ZISIM     1.233 (4.2)
                  ZSP500
           Instruction Set Simulator

                  LSI Logic
*************************************************
Call across sections PASSED
N_Intrinsics PASSED
3.1415926535897932384626433832795028841971693993751058209749445923O
7816406286208998628034825342117067
3.1415926535897932384626433832795028841971693993751058209749445923O
7816406286208998628034825342117067
2.71828182845904523536028747135266249775724709369995957496696762772
40766303535475945713821785251664274
(SYSTEM HALT).................. Instructions=23328715 PC=0x00000475
 g0=0x00  r1=0xffd0  r0=0x0000 g1=0x00  r3=0x0000  r2=0x0000
 g2=0x00  r5=0x0000  r4=0x0065 g3=0x00  r7=0x0007  r6=0x0001
 g4=0x00  r9=0x0000  r8=0x0000 g5=0x00 r11=0x0000 r10=0x0000
 g6=0x00 r13=0x0480 r12=0x0000 g7=0x00 r15=0x0000 r14=0x0000
   a0=0x00000001   n0=0x0003   a1=0x0007e76b   n1=0x0000
   a2=0x00000000   n2=0x0000   a3=0x00000000   n3=0x0000
   a4=0x00000000   n4=0x0000   a5=0x00000000   n5=0x0000
   a6=0x00000000   n6=0x0000   a7=0x00ffeffe   n7=0x0000
   fmode=0x0000  hwflag=0x0068 shwflag=0x0000   imask=0x0000
     ip0=0x0000     ip1=0x0000    ireq=0x0000     dei=0x0000
   loop0=0xffff   loop1=0x0000   loop2=0x0000   loop3=0x0000
   smode=0x8000  psmode=0x0000   amode=0x0000      tc=0x0000
  timer0=0x0000  timer1=0x0000 vitr=0x00000000 cb0_beg=0x00000000
 cb0_end=0x00000000 cb1_beg=0x00000000 cb1_end=0x00000000
cb2_beg=0x00000000
 cb2_end=0x00000000 cb3_beg=0x00000000 cb3_end=0x00000000
pc=0x00000475
     rpc=0x00000474     tpc=0x00000000     ded=0x00000000
```

The key demonstration point in `N_Intrinsic.c` is that the `N_vv_mac` intrinsic ‘`N_vv_mac(acc, a, 1, &b[9], -1, i);`’ can be used in place of the following loop:

```
for(j = 0; j < i; j++)
{
       acc2 += a[j] * b[9-j];
}
```

The `fast_pi.c` file shows how to use assembler directives within a C program and also how to write in-line assembly code. The line:

```
asm(".org 0x8888");
```

relocates the code within this module up by 0x8888 words. The in-line assembly line:

```
asm(“iadd %0, %2”: “=r” (ndx): “0” (ndx), “r” (i));
```

is equivalent to the C statement ‘`ndx += i;`’.

The `sections.s` file is shown below.

```
.section “text1”, “ax”

.global _foo
_foo:
   mov    r13, %rpc
   nop
   call      test_label_1
   call      test_label_2
   mov    %rpc, r13
   ret

.section “text2”, “ax”

.global   test_label_1
.global   test_label_2

.walign 2
test_label_1:
   add    r4, 3
   ret

.walign 2
test_label_2:
   add    r4, -2
   ret
```

This file declares an external function entry point ‘`_foo`’ within the ‘`text1`’ section and two other function entry points ‘`test_label_1`’ and ‘`test_label_2`’ within the ‘`text2`’ section. Since the function ‘`int foo(void)`’ is to be called from C, its name must be prefixed with an underscore.

The `slow_pi.s` file was generated by the `zdcc` compiler without optimizations from a C program. It shows the kind of assembly file produced by the compiler and gives us the opportunity for demonstrating the kinds of optimizations `zdopt` will do. The build script creates the `slow_opt_pi.s` file, which is the optimized version of `slow_pi.s`.

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
16	`%pc`	Program Counter

Table B.1 ZSP400 Control Registers (Cont.)

Register Reference Number	Control Register	Register Description
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

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	%psmode	Previous System Mode Register
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
16	%pc	Program Counter

Table C.1 G2 Control Registers (Cont.)

Register Reference Number	Control Register	Register Description
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

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 3.6, "N-Intrinsics," page 3-19.

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.

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);
```

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.

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.

Appendix E
Signal Processing Library

The libraries, libalg_zsp500.a and libalg_zsp600.a, contain some basic functionality commonly used in signal processing. They are only available for the ZSPG2 architecture. The interface to libalg*.a is contained in alg.h, which can be accessed with:

```
#include <alg.h>
```

To use either library, they must be linked in with either the -lalg_zsp500 or -lalg_zsp600 switch on the link line.

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.

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 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.

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.

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.

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".

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).

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).

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

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.

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.

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.

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.

Index

Symbols

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z

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 Software Development Kit User's Guide.* 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 ____________________