zilog®
Embedded in Life
An IXYS Company

Zilog Developer Studio II – ZNEO™

User Manual

UM017105-0511

This publication is subject to replacement by a later edition. To determine whether a later edition exists, or to request copies of publications, visit www.zilog.com.

Warning: DO NOT USE THIS PRODUCT IN LIFE SUPPORT SYSTEMS.

LIFE SUPPORT POLICY

ZILOG'S PRODUCTS ARE NOT AUTHORIZED FOR USE AS CRITICAL COMPONENTS IN LIFE SUPPORT DEVICES OR SYSTEMS WITHOUT THE EXPRESS PRIOR WRITTEN APPROVAL OF THE PRESIDENT AND GENERAL COUNSEL OF ZILOG CORPORATION.

As used herein

Life support devices or systems are devices which (a) are intended for surgical implant into the body, or (b) support or sustain life and whose failure to perform when properly used in accordance with instructions for use provided in the labeling can be reasonably expected to result in a significant injury to the user. A critical component is any component in a life support device or system whose failure to perform can be reasonably expected to cause the failure of the life support device or system or to affect its safety or effectiveness.

Document Disclaimer

Revision History

Each instance in the Revision History table below reflects a change to this document from its previous version. For more details, click the appropriate links in the table.

Date	Revision Level	Description	Page
May 2011	05	Updated Using the Integrated Development Environment.	15
		Updated Using the Editor.	109
		Updated Structures and Unions in Assembly Code section of Using the Macro Assembler.	239
		Updated Warning and Error Messages section of Using the Macro Assembler.	255
		Updated Disassembly Window section of Using the Debugger.	341
Dec 2010	04	Updated ZDS II System Requirements section.	v
Feb 2007	03	Changed the description of the Project Settings dialog box.	8
		Changed *Select* Active *Configuration* to *Select Build Configuration*. Changed *File Verify* button to *Verify Download* button.	15
		Removed PL and PW for CR 3684.	211
		Added Table 47 on page 346. Added the `checksum`, `fillmem`, `loadmem`, and `savemem` commands. Updated the sample command script file.	359
		Updated.	v, 2, 32, 37, 40, 43, 74, 74, 84, 91, 96, 97, 106, 197, 217, 217, 235, 343, 349, 363, 365, 366, 366, 387
		Added new sections:	40,40, 41, 83, 239, and 251.
		Added note for CR 5661.	197, 297
		Added new shortcuts.	24

Date	Revision Level	Description	Page
Jun 2006	02	Updated ZDS II System Requirements section.	v
		Changed FAQ.txt to FAQ.html.	viii
		Updated screenshots.	1
		Updated various sections, including the description of the memory map for CR 7124.	15
		Added messages.	197
		Updated Label Field section.	217
		Updated text and figures for CRs 7123 and 7124.	301
		Added new Cyclic Redundancy Check sections.	336, 367
Jan 2006	01	Initial release.	All

Preface

The following sections provide an introduction to Zilog Developer Studio II:

- ZDS II System Requirements – see page v
- Zilog Technical Support – see page vii

ZDS II System Requirements

To effectively use Zilog Developer System II, you need a basic understanding of the C and assembly languages, the device architecture, and Microsoft Windows.

The following sections describe the ZDS II system requirements:

- Supported Operating Systems – see page v
- Recommended Host System Configuration – see page vi
- Minimum Host System Configuration – see page vi
- When Using the Serial Smart Cable – see page vi
- When Using the USB Smart Cable – see page vii
- When Using the Opto-Isolated USB Smart Cable – see page vii
- When Using the Ethernet Smart Cable – see page vii

Supported Operating Systems

- Windows 7 64-bit
- Windows 7 32-bit
- Windows Vista 64-bit
- Windows Vista 32-bit
- Windows XP Professional 32-bit

Note: The USB Smart Cable is not supported on 64-bit Windows Vista and Windows XP for ZDS II – Z8 Encore! versions 4.10.1 or earlier.

- Windows 2000 SP4

Recommended Host System Configuration

- Windows XP Professional SP3 or later
- Pentium IV 2.2 GHz processor or higher
- 1024 MB RAM or more
- 135 MB hard disk space (includes application and documentation)
- Super VGA video adapter
- CD-ROM drive for installation
- USB high-speed port (when using the USB Smart Cable)
- RS-232 communication port with hardware flow control
- Internet browser (Internet Explorer or Netscape)

Minimum Host System Configuration

- Windows XP Professional SP2
- Pentium IV 2.2 GHz processor
- 512 MB RAM
- 50 MB hard disk space (application only)
- Super VGA video adapter
- CD-ROM drive for installation
- USB full-speed port (when using the USB Smart Cable)
- RS-232 communication port with hardware flow control
- Internet browser (Internet Explorer or Netscape)

When Using the Serial Smart Cable

- RS-232 communication port with hardware flow and modem control signals

Note: Some USB to RS-232 devices are not compatible because they lack the necessary hardware signals and/or they use proprietary auto-sensing mechanisms which prevent the Smart Cable from connecting.

When Using the USB Smart Cable

- High-speed USB (fully compatible with original USB)
- Root (direct) or self-powered hub connection

Note: The USB Smart Cable is a high-power USB device. Windows NT is not supported.

When Using the Opto-Isolated USB Smart Cable

- High-speed USB (fully compatible with original USB)
- Root (direct) or self-powered hub connection

Note: The USB Smart Cable is a high-power USB device. Windows NT is not supported.

When Using the Ethernet Smart Cable

- Ethernet 10Base-T compatible connection

Zilog Technical Support

For technical questions about our products and tools or for design assistance, please visit the Zilog website at http://www.zilog.com. You must provide the following information in your support ticket:

- Product release number (located in the heading of the toolbar)
- Product serial number
- Type of hardware you are using
- Exact wording of any error or warning messages
- Any applicable files attached to the email

To receive Zilog Developer Studio II (ZDS II) product updates and notifications, register at the Technical Support web page at http://support.zilog.com.

Before Contacting Technical Support

Before you contact Zilog Technical Support, consult the following documentation:

Readme.txt File. Refer to the `readme.txt` file in the following ZDS II directory for last-minute tips and information about problems that could occur while installing or running ZDS II:

<ZILOGINSTALL>\ZDSII_product_version\

where:

- *ZILOGINSTALL* is the ZDS II installation directory. For example, the default installation directory is `C:\Program Files\Zilog`.
- *product* is the specific Zilog product. For example, *product* can be ZNEO, Z8Encore! or eZ80Acclaim!.
- *version* is the ZDS II version number. For example, *version* might be 4.11.0 or 5.0.0.

FAQ.html File. The `FAQ.html` file contains answers to frequently-asked questions and other information about getting the best results from ZDS II. The information in this file does not generally go out of date from release to release as quickly as the information in the `readme.txt` file. You can find the `FAQ.html` file in the following directory:

<ZILOGINSTALL>\ZDSII_product_version\

where:

- *<ZILOGINSTALL>* is the ZDS II installation directory. For example, the default installation directory is `C:\Program Files\Zilog`.
- *product* is the specific Zilog product family. For example, *product* can be ZNEO, Z8Encore! or eZ80Acclaim!.
- *version* is the ZDS II version number. For example, *version* could be 4.11.0 or 5.0.0.

Troubleshooting Section. See the Troubleshooting the Linker section on page 295.

Table of Contents

List of Figures

List of Tables

Chapter 1. Getting Started

This section provides a tutorial of the developer's environment, so you can be working with the ZDS II graphical user interface (GUI) in a short time. This section covers the following topics:

- Installing ZDS II on page 1
- Developer's Environment Tutorial on page 1

Note: You can use this tutorial to install and start using ZDS II without any attached hardware. If you have a development kit, use the included Quick Start Guide to set up your hardware and install ZDS II. For steps to create a new project using target hardware, see New Project on page 32.

Installing ZDS II

If you have not already installed ZDS II, perform the following procedure:

1. Insert the CD in your CD-ROM drive.
2. Follow the setup instructions on your screen.
3. Install the application in an appropriate folder location on your PC.

Developer's Environment Tutorial

This tutorial shows you how to use the basic features of Zilog Developer Studio. To begin this tour, you need a basic understanding of Microsoft Windows. Estimated time for completing this exercise is 15 minutes.

In this tour, you perform the following brief procedure.

- Create a New Project on page 2
- Add a File to the Project on page 6
- Set Up the Project on page 8

When you complete this tour and save your project, you will have a `sample.lod` file that can be used for debugging.

Note: Be sure to read Menu Bar on page 31 to learn more about all of the dialog boxes and their options discussed in this tour.

For the purpose of this tutorial, your ZNEO developer's environment directory will be referred to as <*ZDS Installation Directory*>, which equates to the following nomenclature:

`<ZILOGINSTALL>\ZDSII_ZNEO_<`*version*`>\`

where:

- *ZILOGINSTALL* is the ZDS II installation directory. For example, the default installation directory is `C:\Program Files\Zilog`.
- *version* is the ZDS II version number. For example, *version* might be `4.11.0` or `5.0.0`.

Create a New Project. Start the ZDS II program if it is not already running.

1. To create a new project, select **New Project** from the **File** menu. The **New Project** dialog box is displayed.
2. From the **New Project** dialog box, click the **Browse** button (...) to navigate to a directory in which to save your project. The **Select Project Name** dialog box is displayed; see Figure 1.

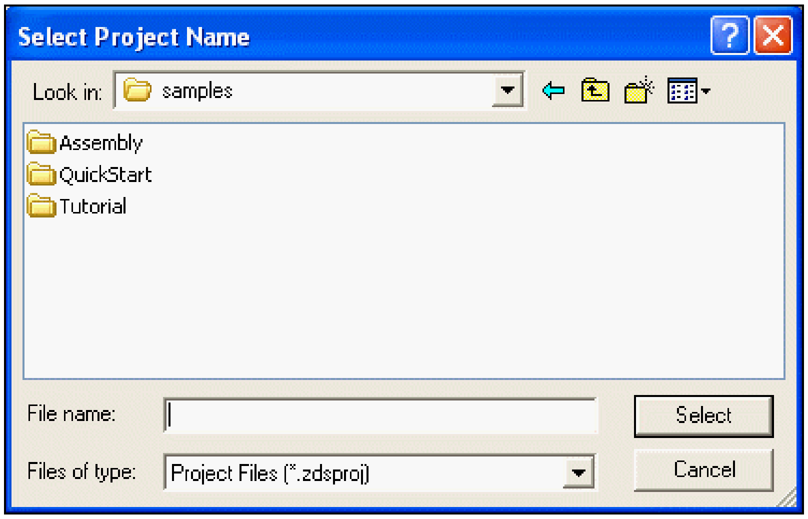


Figure 1. Select Project Name Dialog Box

3. Use the **Look In** drop-down menu to navigate to the directory in which you'll save your project. For this tutorial, Zilog recommends you place your project in the following directory:

   <*ZDS Installation Directory*>`\samples\Tutorial`

   If Zilog Developer Studio was installed in the default directory, the actual filepath would be:

   `C:\Program Files\Zilog\ZDSII_ZNEO_4.10.1\samples\Tutorial`

4. In the **File Name:** field, enter `sample` as the name of your project.

   The ZNEO developer's environment creates a project file. By default, project files have the `.zdsproj` extension (for example, <*project name*>`.zdsproj`). You do not have to enter the extension `.zdsproj` in this field. It is added automatically.

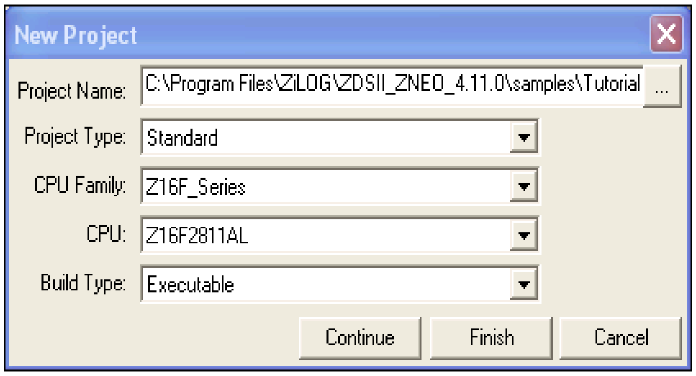
5. Click **Select** to return to the **New Project** dialog box.

6. Because the `sample` project uses `.c` files, select **Standard** from the **Project Type** drop-down menu.

7. In the **CPU Family** drop-down menu, select **Z16F_Series**.

8. In the **CPU** drop-down menu, select **Z16F2811AL**.

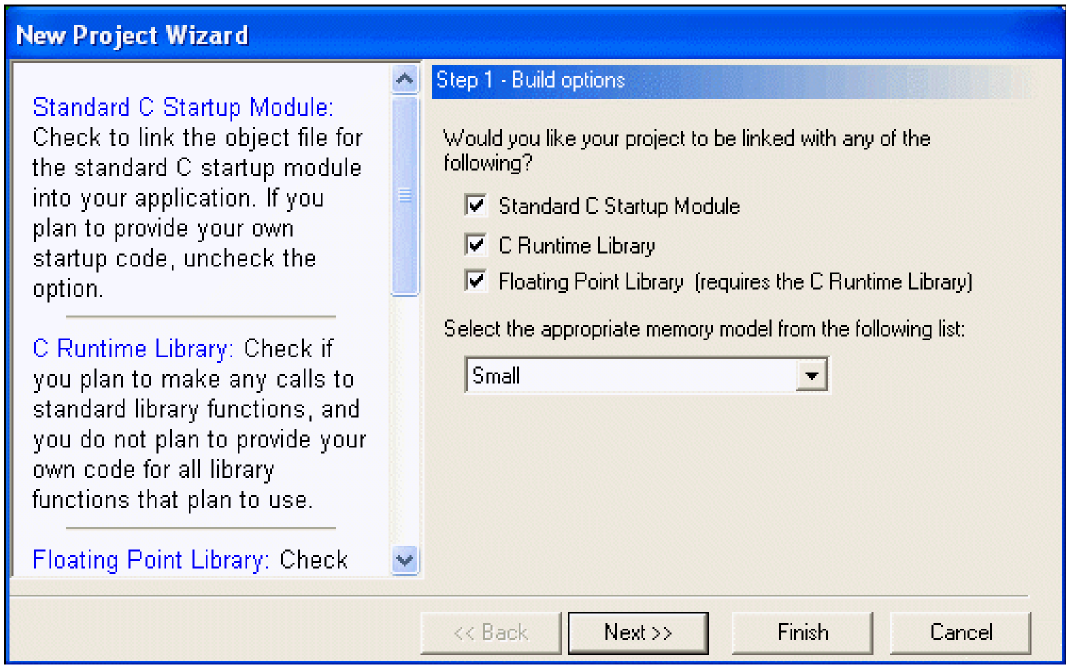
9. In the **Build Type** drop-down menu, select **Executable** to build an application (see Figure 2).



Project Name: C:\Program Files\ZiLOG\ZDSII_ZNEO_4.11.0\samples\Tutorial
Project Type: Standard
CPU Family: Z16F_Series
CPU: Z16F2811AL
Build Type: Executable

Figure 2. New Project Dialog Box

10. Click **Continue**. The **New Project Wizard** dialog box is displayed (see Figure 3). It allows you to modify the initial values for some of the project settings during the project creation process.


New Project Wizard

Standard C Startup Module: Check to link the object file for the standard C startup module into your application. If you plan to provide your own startup code, uncheck the option.

C Runtime Library: Check if you plan to make any calls to standard library functions, and you do not plan to provide your own code for all library functions that plan to use.

Floating Point Library: Check

Step 1 - Build options

Would you like your project to be linked with any of the following?

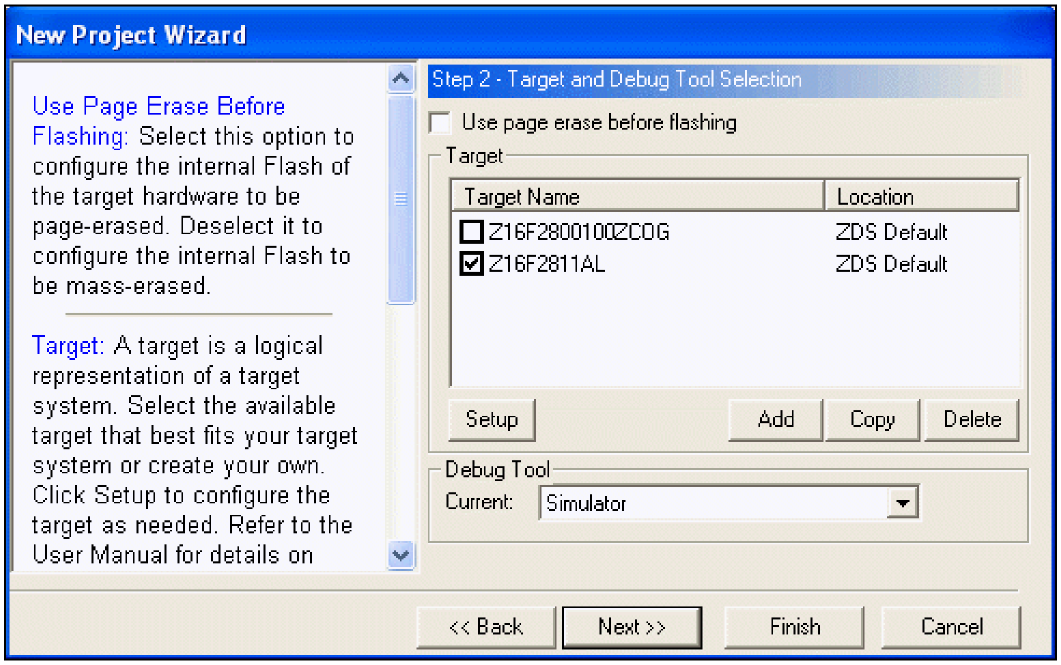
- [✓] Standard C Startup Module
- [✓] C Runtime Library
- [✓] Floating Point Library (requires the C Runtime Library)

Select the appropriate memory model from the following list:

Small

Figure 3. New Project Wizard Dialog Box—Build Options Step

11. Accept the defaults by clicking **Next**. The **Target and Debug Tool Selection** step of the **New Project Wizard** dialog box is displayed; see Figure 4.



Use Page Erase Before Flashing: Select this option to configure the internal Flash of the target hardware to be page-erased. Deselect it to configure the internal Flash to be mass-erased.

Target: A target is a logical representation of a target system. Select the available target that best fits your target system or create your own. Click Setup to configure the target as needed. Refer to the User Manual for details on

Step 2 - Target and Debug Tool Selection

☐ Use page erase before flashing

Target

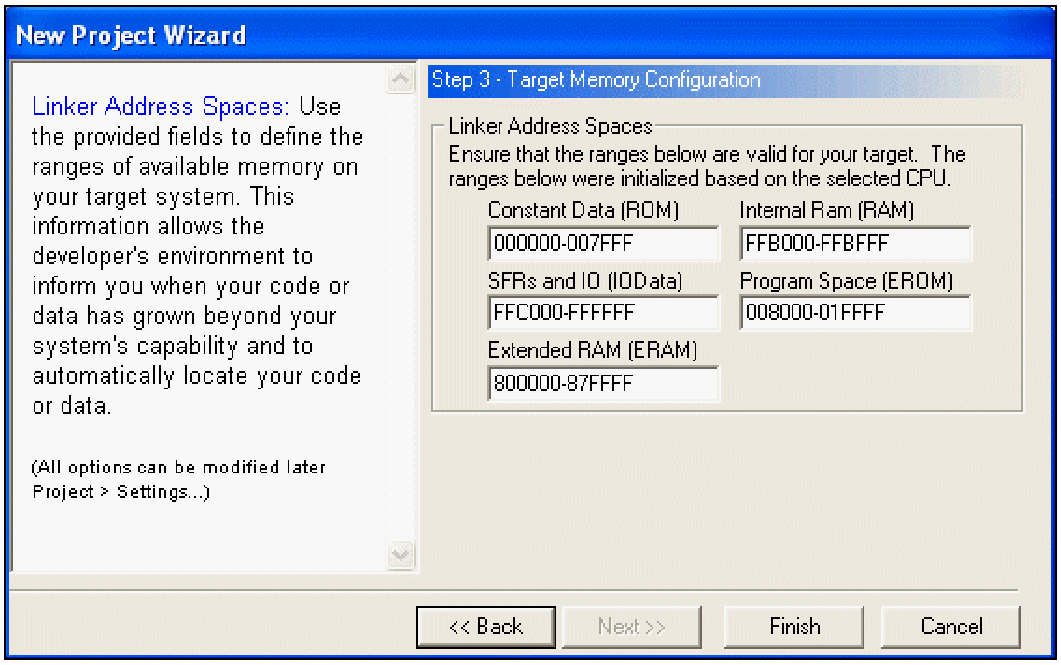
Target Name	Location
☐ Z16F2800100ZCOG	ZDS Default
☑ Z16F2811AL	ZDS Default

Debug Tool

Current: Simulator

Figure 4. New Project Wizard Dialog Box—Target and Debug Tool Selection Step

12. Click **Next** to accept the defaults. The **Target Memory Configuration** step of the **New Project Wizard** dialog box is displayed; see Figure 5.



Linker Address Spaces: Use the provided fields to define the ranges of available memory on your target system. This information allows the developer's environment to inform you when your code or data has grown beyond your system's capability and to automatically locate your code or data.

(All options can be modified later Project > Settings...)

Linker Address Spaces

Ensure that the ranges below are valid for your target. The ranges below were initialized based on the selected CPU.

Constant Data (ROM): 000000-007FFF
Internal Ram (RAM): FFB000-FFBFFF
SFRs and IO (IOData): FFC000-FFFFFF
Program Space (EROM): 008000-01FFFF
Extended RAM (ERAM): 800000-87FFFF

Figure 5. New Project Wizard Dialog Box—Target Memory Configuration Step

13. Click **Finish**. ZDS II creates a new project named `sample`. Two empty folders, Standard **Project Files** and **External Dependencies**, are displayed in the **Project Workspace** window on the left side of the integrated development environment (IDE).

Add a File to the Project

In this section, you add the `main.c` source file (provided) to the `sample` project.

1. From the **Project** menu, select **Add Files**. The **Add Files to Project** dialog box is displayed; see Figure 6.

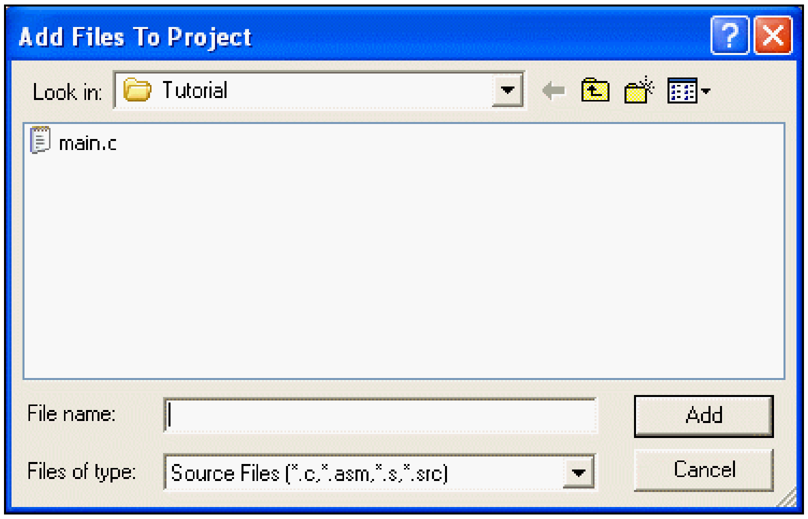


Figure 6. Add Files to Project Dialog Box

2. In the **Add Files to Project** dialog box, return to the tutorial directory by navigating to

   <ZDS Installation Directory>`\samples\Tutorial`

3. Select the `main.c` file and click **Add**. The `main.c` file is displayed under the **Standard Project Files** folder in the **Project Workspace** window on the left side of the IDE, as shown in Figure 7.

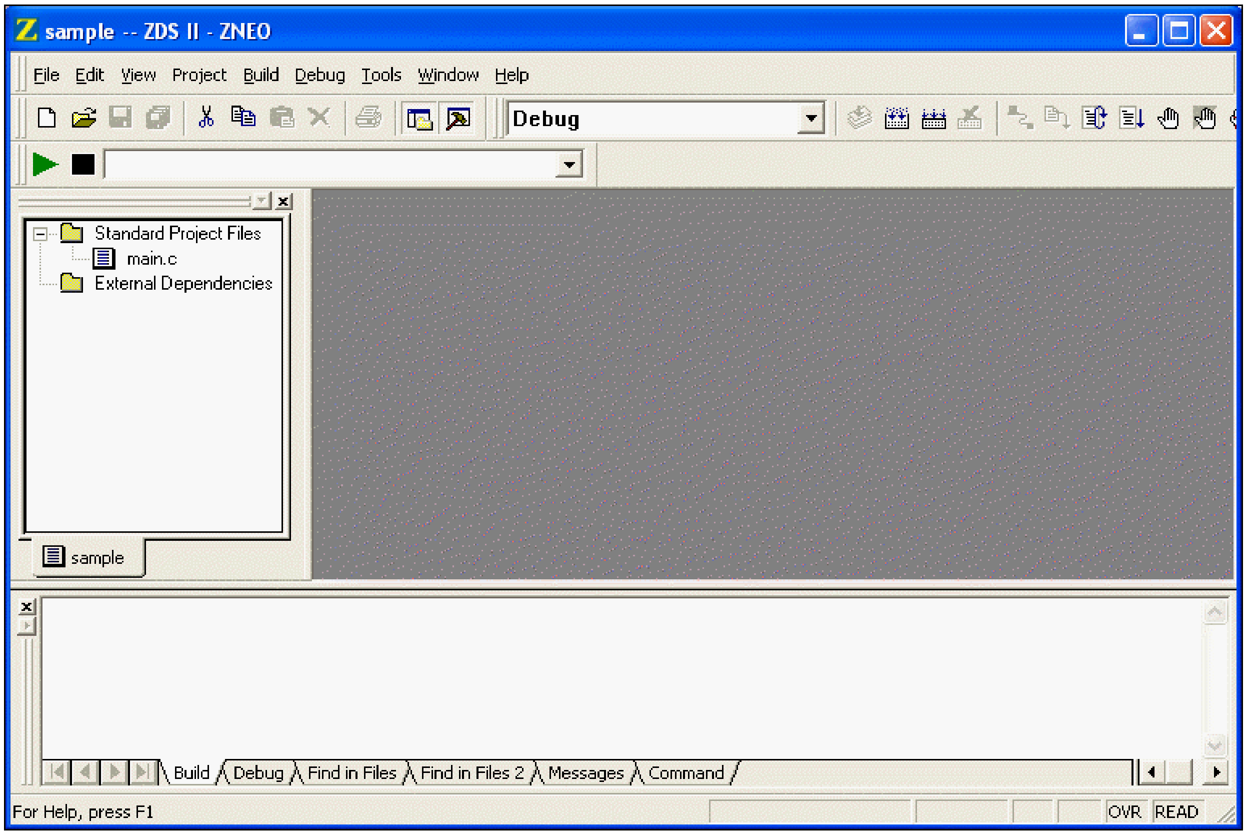


Figure 7. Sample Project

Note: To view a file in the Edit window during the tutorial, double-click the file in the **Project Workspace** pane.

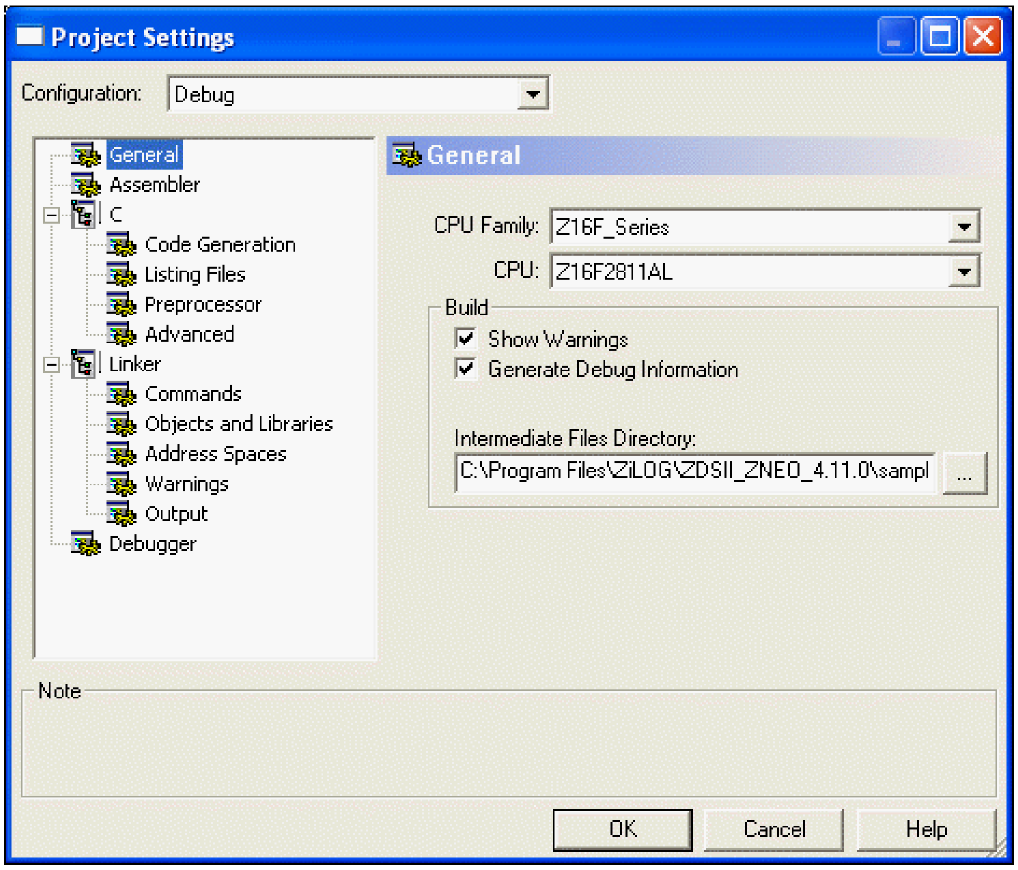
Set Up the Project

Before you save and build the `sample` project, check the settings in the **Project Settings** dialog box.

1. From the **Project** menu, select **Settings**. The **Project Settings** dialog box is displayed. It provides various project configuration pages that can be accessed by selecting the page name in the pane on the left side of the dialog box. There are several pages grouped together for the C compiler and Linker that allow you to set up subsettings for that tool. For more information about this topic, see Settings on page 46.

2. In the **Configuration** drop-down menu in the upper left corner of the **Project Settings** dialog box, make sure the **Debug** build configuration is selected, as shown in Figure 8.

   For your convenience, the Debug configuration is a predefined configuration of defaults set to enable the debugging of program code. For more information about project configurations such as adding your own configuration, see Set Active Configuration on page 83.



Configuration: Debug

CPU Family: Z16F_Series

CPU: Z16F2811AL

☑ Show Warnings

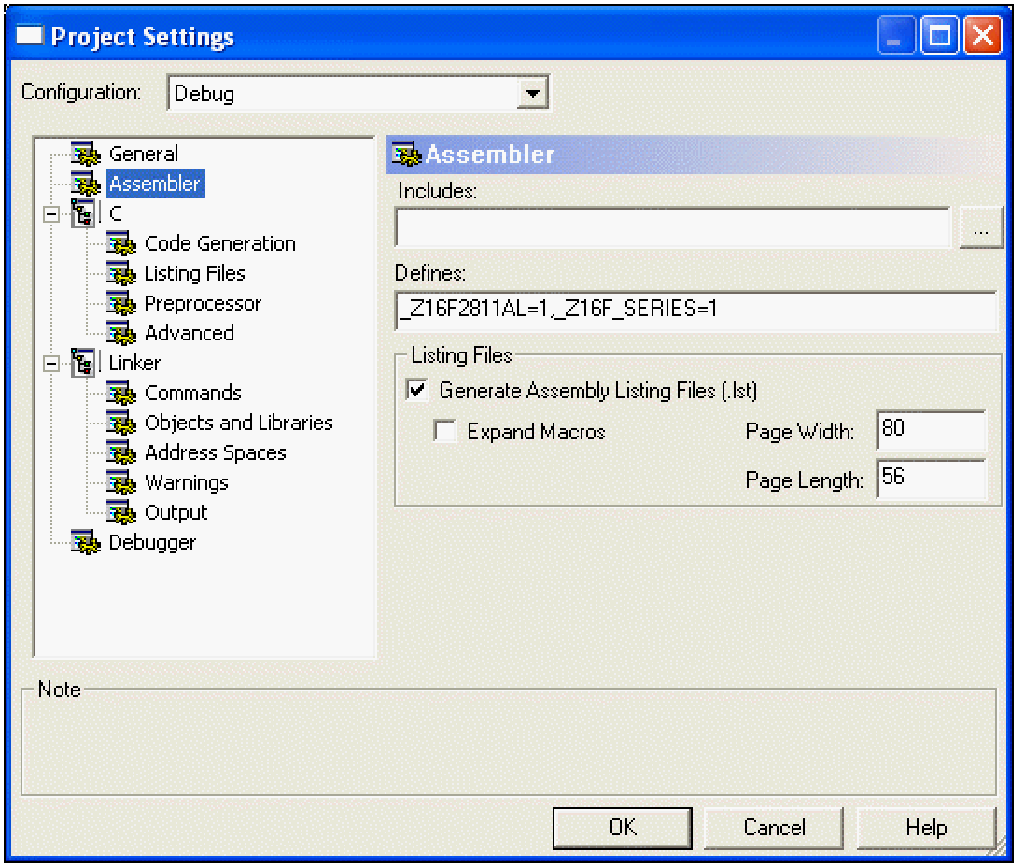
☑ Generate Debug Information

Intermediate Files Directory:
C:\Program Files\ZiLOG\ZDSII_ZNEO_4.11.0\sampl

Figure 8. General Page of the Project Settings Dialog Box

3. Click the **Assembler** page.

4. Make sure that the **Generate Assembly Listing Files (.lst)** checkbox is selected, as shown in Figure 9.



Configuration: Debug

Includes:

Defines: _Z16F2811AL=1,_Z16F_SERIES=1

Listing Files
- [✓] Generate Assembly Listing Files (.lst)
- [ ] Expand Macros

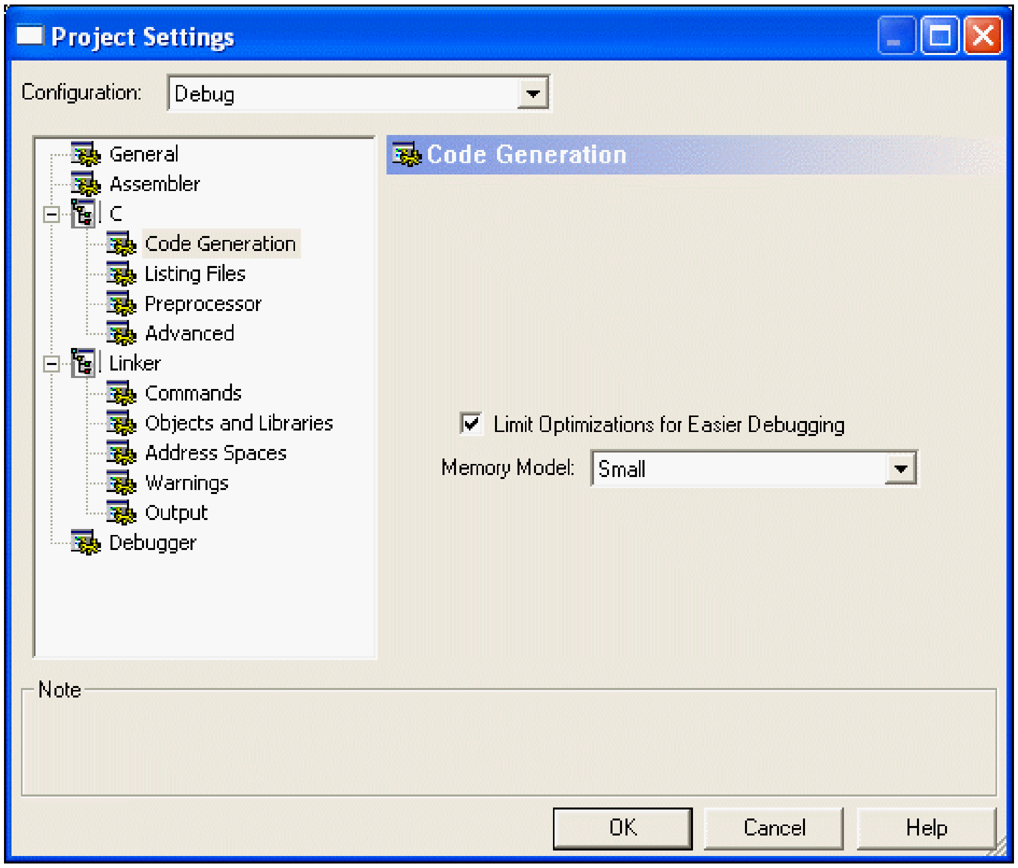
Page Width: 80

Page Length: 56

Figure 9. Assembler Page of the Project Settings Dialog Box

5. Click the **Code Generation** page.

6. Select the **Limit Optimizations for Easier Debugging** checkbox, as shown in Figure 10.



Configuration: Debug

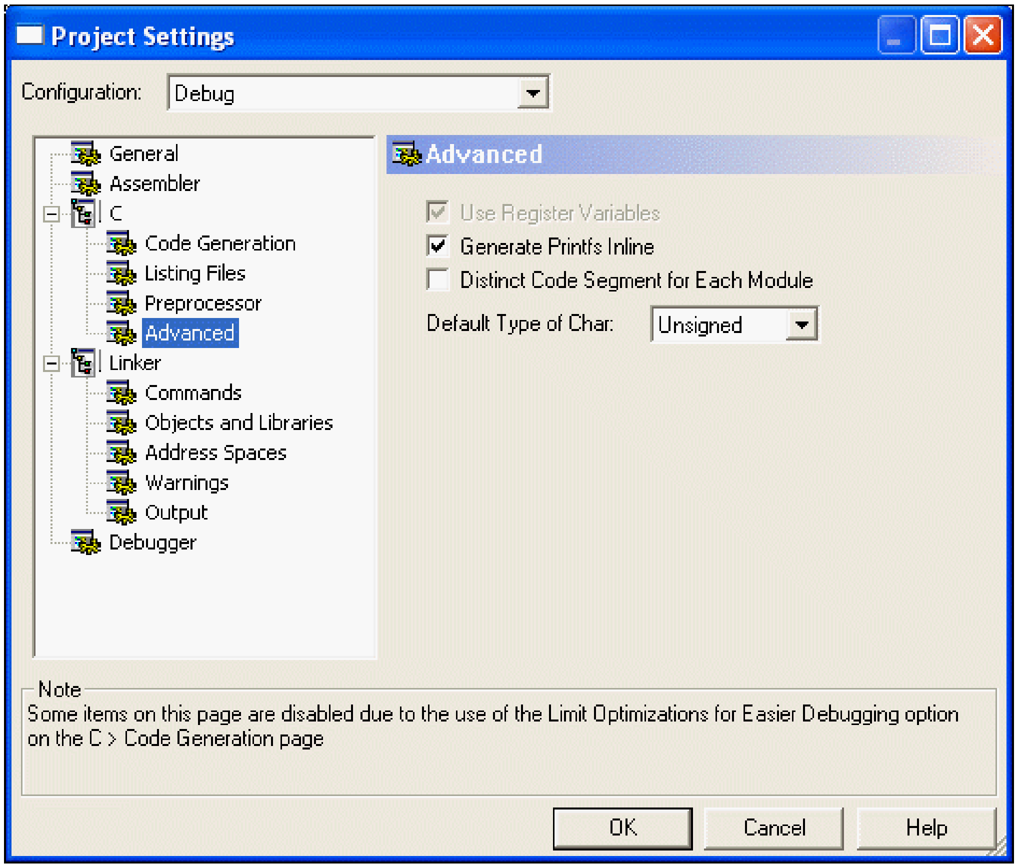
[✓] Limit Optimizations for Easier Debugging

Memory Model: Small

Figure 10. Code Generation Page of the Project Settings Dialog Box

7. Click the **Advanced** page.

8. Make certain the **Generate Printfs Inline** checkbox is selected, as shown in Figure 11.



Configuration: Debug

[✓] Use Register Variables
[✓] Generate Printfs Inline
[ ] Distinct Code Segment for Each Module

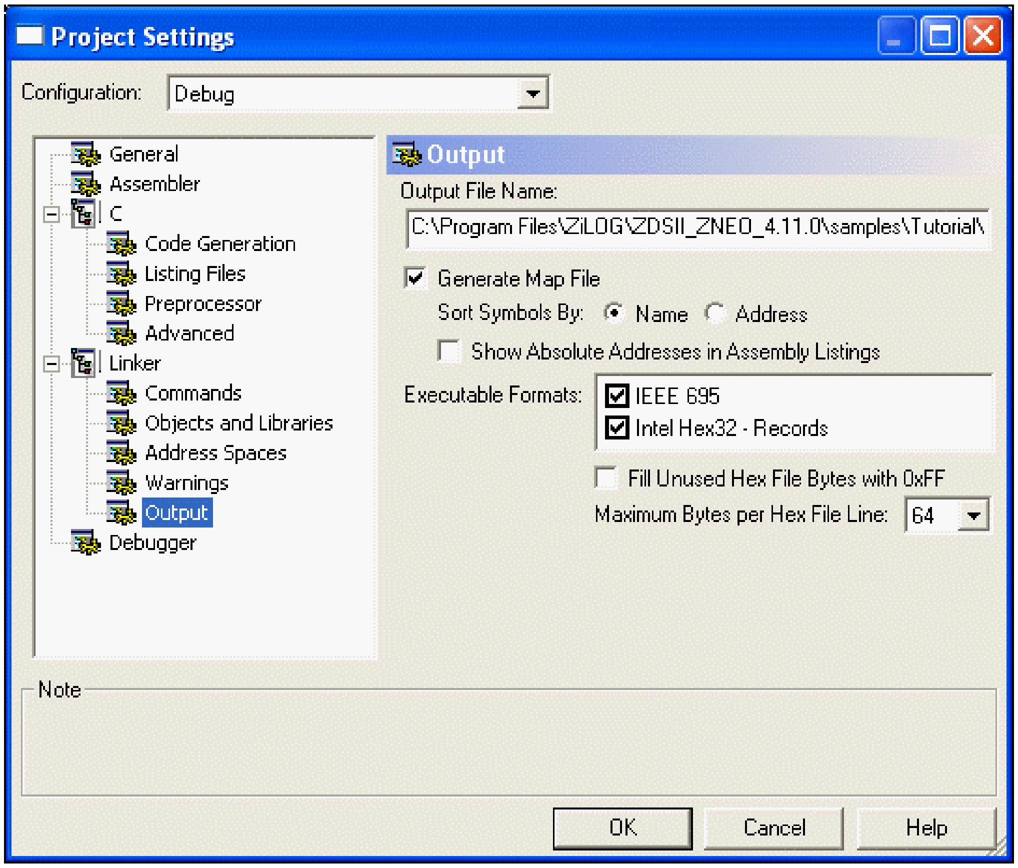
Default Type of Char: Unsigned

Note
Some items on this page are disabled due to the use of the Limit Optimizations for Easier Debugging option on the C > Code Generation page

Figure 11. Advanced Page of the Project Settings Dialog Box

9. Click the **Output** page.

10. Make sure that the **IEEE 695** and **Intel Hex32 - Records** checkboxes are both selected, as shown in Figure 12.



Configuration: Debug

Output

Output File Name: C:\Program Files\ZiLOG\ZDSII_ZNEO_4.11.0\samples\Tutorial\

[✓] Generate Map File

Sort Symbols By: [✓ Name] Address

[ ] Show Absolute Addresses in Assembly Listings

Executable Formats: [✓] IEEE 695 [✓] Intel Hex32 - Records

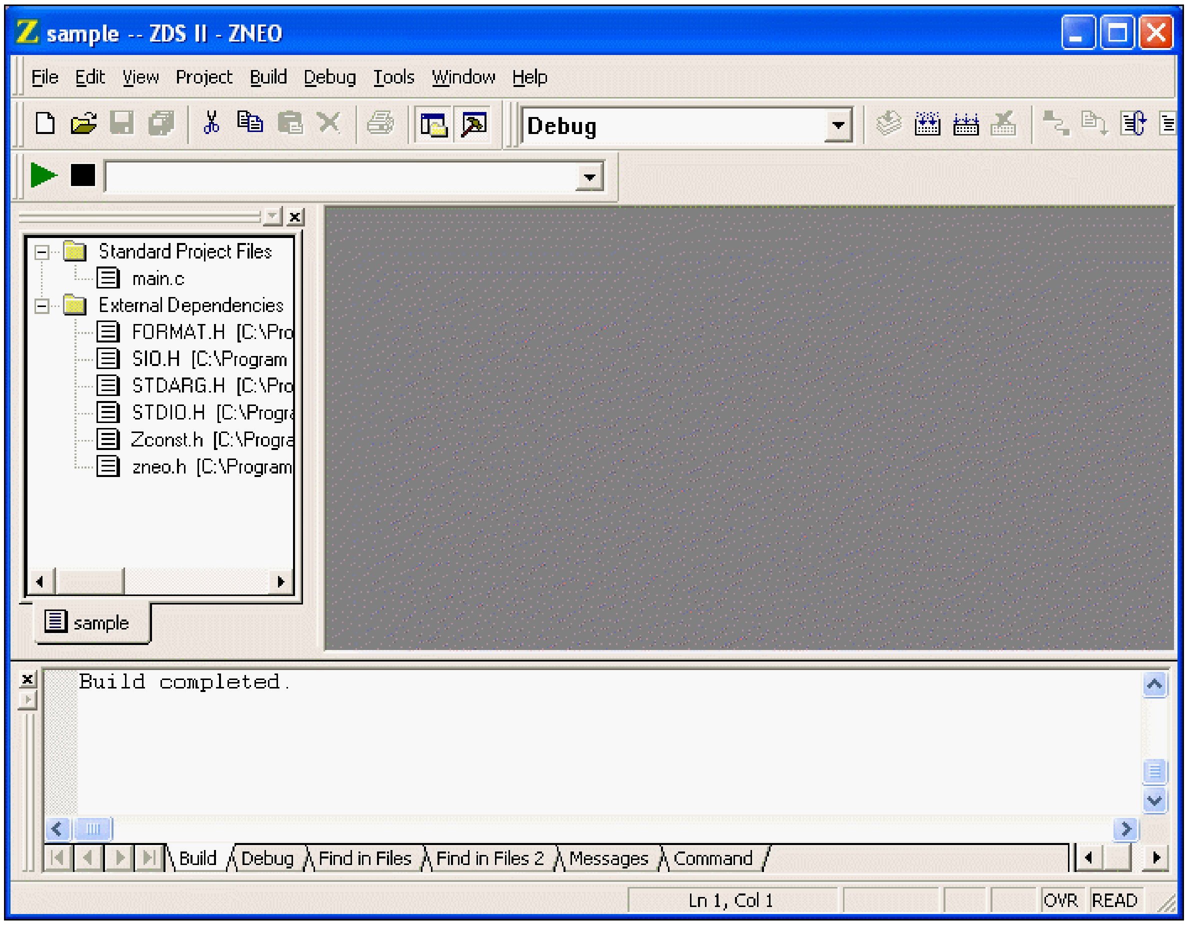
[ ] Fill Unused Hex File Bytes with 0xFF

Maximum Bytes per Hex File Line: 64

Figure 12. Output Page of the Project Settings Dialog Box

11. Click **OK** to save all project settings. The Development Environment will prompt you to build the project when changes are made to the project settings that would effect the resulting build program. The message displays: “`The project settings have changed since the last build. Would you like to rebuild the affected files?`”
12. Click **Yes** to build the project. The developer’s environment builds the `sample` project.

13. Observe the compilation process in the **Build Output** window, as shown in Figure 13. When the `Build completed` message is displayed in the **Build Output** window, you have successfully built the sample project and created a `sample.lod` file to debug.



```
Build completed.
```

Figure 13. Build Output Window

14. From the **File** menu, select **Save Project**.

Chapter 2. Using the Integrated Development Environment

This section discusses how to use the following integrated development environment (IDE) elements.

- Toolbars – see page 16
- Windows – see page 23
- Menu Bar – see page 31
- Shortcut Keys – see page 105

To effectively understand how to use the developer's environment, be sure to go through the Developer's Environment Tutorial on page 1.

After the discussion of the toolbars and windows, this section discusses the menu bar, shown in Figure 14, from left to right—File, Edit, View, Project, Build, Debug, Tools, Window, and Help—and the dialog boxes accessed from the menus. For example, the **Project Settings** dialog box is discussed as a part of the Project menu section.

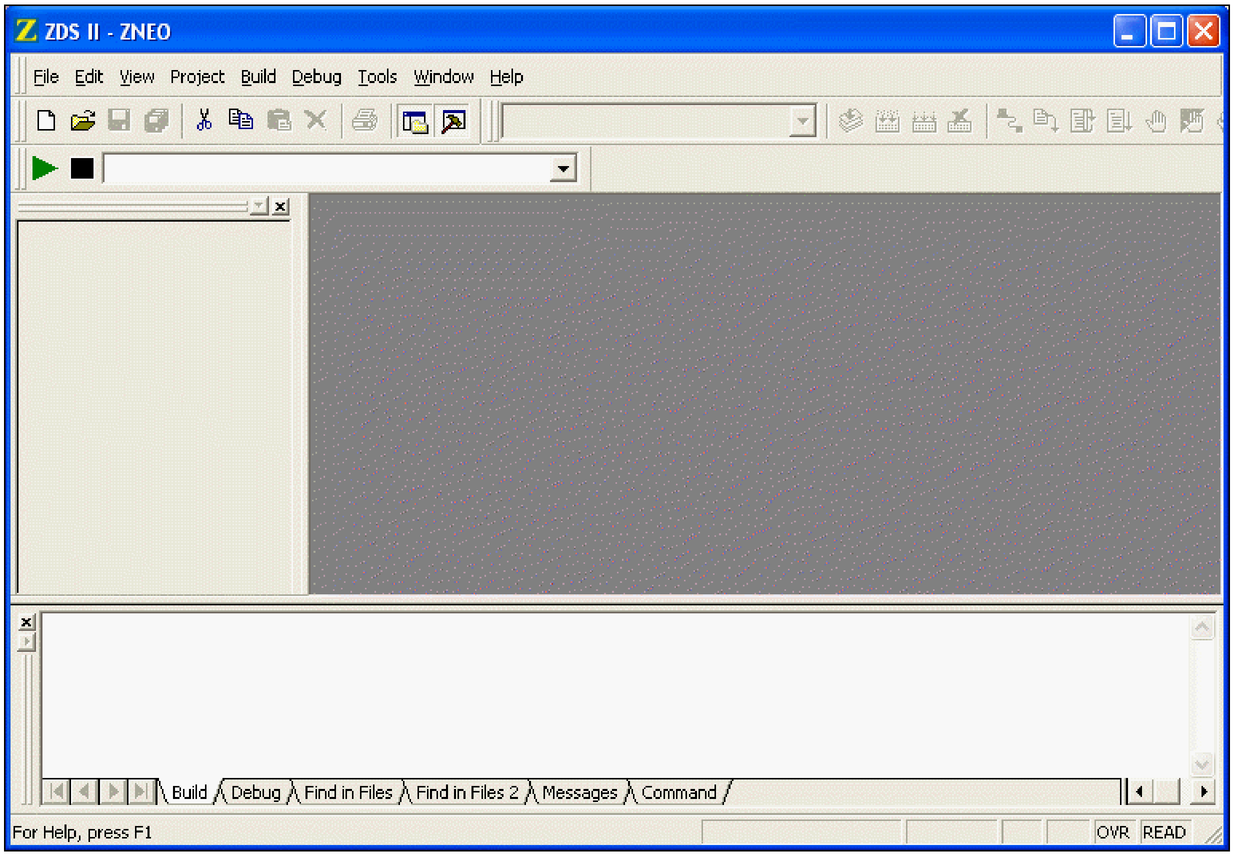

ZDS II - ZNEO
File Edit View Project Build Debug Tools Window Help
Build Debug Find in Files Find in Files 2 Messages Command
For Help, press F1
OVR READ

Figure 14. ZNEO Integrated Development Environment (IDE) Window

For a table of all of the shortcuts used in the ZNEO developer's environment, see Shortcut Keys on page 105.

Toolbars

The toolbars provide quick access to most features of the ZNEO developer's environment. You can use these buttons – even cue cards – to perform any task. As you move the mouse pointer across the toolbars, the main function of each button is displayed in a pop-up dialog. Additionally, you can drag and move the toolbars to different areas on the screen.

The following toolbars are described in this section.

- File Toolbar – see page 17
- Build Toolbar – see page 18

- Find Toolbar – see page 19
- Command Processor Toolbar – see page 19
- Debug Toolbar – see page 20
- Debug Windows Toolbar – see page 22

Note: For more information about debugging, see the Using the Debugger chapter on page 327.

File Toolbar

The File toolbar, shown in Figure 15, allows you to perform basic functions with your files using a number of buttons, each of which is briefly described below.

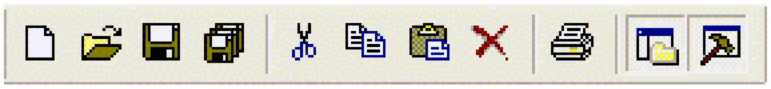


Figure 15. The File Toolbar

New. Creates a new file.

Open. Allows you to open an existing file.

Save. Saves the active file.

Save All. Saves all open files and the currently loaded project.

Cut. Deletes selected text from the active file and puts it on the Windows clipboard.

Copy. Copies selected text from the active file and puts it on the Windows clipboard.

Paste. Pastes the current contents of the clipboard into the active file at the current cursor position.

Delete. Deletes selected text from the active file.

Print. Prints the active file.

Workspace Window. Shows or hides the Project Workspace window.

Output Window. Shows or hides the Output window.

Build Toolbar

The Build toolbar, shown in Figure 16, allows you to build your project, set breakpoints, and select a project configuration with the following controls and buttons; a description of each follows.

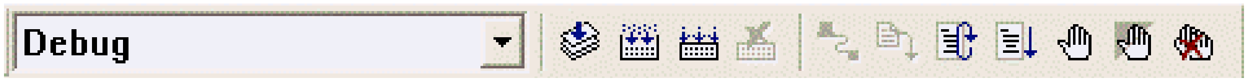


Figure 16. The Build Toolbar

Select Build Configuration List Box. Lets you activate the build configuration for your project. See the Set Active Configuration section on page 83 for more information.

Compile/Assemble File Button. Compiles or assembles the active source file.

Build Button. Builds your project by compiling and/or assembling any files that have changed since the last build and then links the project.

Rebuild All Button. Rebuilds all files and links the project.

Stop Build Button. Stops a build in progress.

Connect to Target Button. Starts a debug session and initializes the communication to the target hardware. Clicking this button does not download the software or reset to main. Use this button to access target registers, memory, and so on, without loading new code or to avoid overwriting the target's code with the same code. This button is not enabled when the target is the simulator.

Download Code Button. Downloads the executable file for the currently open project to the target for debugging. The button also initializes the communication to the target hardware if it has not been done yet. Use this button anytime during a debug session.

Note: Using the **Download Code** button overwrites the current code on the target.

Reset Button. The **Reset** button resets the program counter to the beginning the program. If not in Debug mode, a debug session is started. By default and if possible, clicking the **Reset** button resets the program counter to symbol 'main'. If you deselect the **Reset to Symbol 'main' (Where Applicable)** checkbox on the **Debugger** tab of the **Options** dialog box (see page 102), the program counter resets to the first line of the program.

Go Button. The **Go** button executes project code from the current program counter. If not in Debug mode when the button is clicked, a debug session is started.

Insert/Remove Breakpoint Button. The **Insert/Remove Breakpoint** button sets a new breakpoint or removes an existing breakpoint in the active file at the line in which the cursor is. You can set a breakpoint in any line with a blue dot displayed to the left of the line (shown in Debug mode only).

Enable/Disable Breakpoint Button. The **Enable/Disable Breakpoint** button activates or deactivates an existing breakpoint at the line in which the cursor is. A red octagon indicates an enabled breakpoint; a white octagon indicates a disabled breakpoint.

Remove All Breakpoints Button. The **Remove All Breakpoints** button deletes all breakpoints in the currently loaded project.

Find Toolbar

The Find toolbar, shown in Figure 17, provides access to text search functions using either of two entry points, each of which is briefly described below.

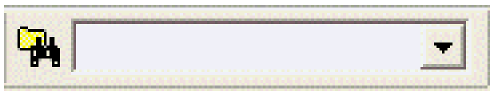


Figure 17. The Find Toolbar

Find in Files Button. Opens the Find in Files dialog box, allowing you to search for text in multiple files.

Find Field. To locate text in the active file, enter the text in the Find field and press the **Enter** key. The search term is highlighted in the file. To search again, press the **Enter** key again.

Command Processor Toolbar

The Command Processor toolbar, shown in Figure 18, allows you to execute IDE and debugger commands using either of two entry points, each of which is briefly described below.

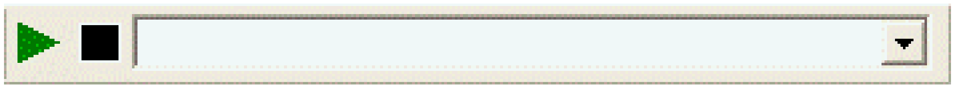


Figure 18. The Command Processor Toolbar

See Supported Script File Commands on page 364 for a list of supported commands.

Run Command Button. Executes the command in the Command field. Output from the execution of the command is displayed in the Command tab of the Output window.

Stop Command Button. Stops any commands currently running.

Command Field. The Command field allows you to enter a new command. Click the **Run Command** button or press the **Enter** key to execute the command. Output from the execution of the command is displayed in the **Command** tab of the **Output** window.

To modify the width of the Command field, perform the following brief procedure:

1. Select **Customize** from the **Tools** menu.
2. Click to set your cursor in the **Command** field. A hatched rectangle highlights the **Command** field.
3. Use your mouse to select and drag the side of the hatched rectangle. The new size of the **Command** field is saved as a new project setting.

Debug Toolbar

The Debug toolbar, shown in Figure 19, allows you to perform debugging functions with the following buttons:

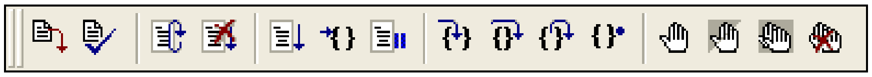


Figure 19. The Debug Toolbar

Download Code Button. Downloads the executable file for the currently open project to the target for debugging. The button also initializes the communication to the target hardware if it has not been done yet. Use this button anytime during a debug session.

Note: The current code on the target is overwritten.

Verify Download Button. Determines download correctness by comparing executable file contents to target memory.

Reset Button. Resets the program counter to the beginning the program. If not in Debug mode, a debug session is started. By default and if possible, clicking the **Reset** button resets the program counter to symbol 'main'. If you deselect the **Reset to Symbol 'main' (Where Applicable)** checkbox on the **Debugger** tab of the **Options** dialog box (see page 102), the program counter resets to the first line of the program.

Stop Debugging Button. Ends the current debug session. To stop program execution, click the **Break** button.

Go Button. Executes project code from the current program counter. If not in Debug mode when the button is clicked, a debug session is started.

Run to Cursor Button. Executes the program code from the current program counter to the line containing the cursor in the active file or the Disassembly window. The cursor must be placed on a valid code line (a C source line with a blue dot displayed in the gutter or any instruction line in the Disassembly window).

Break Button. Stops program execution at the current program counter.

Step Into Button. Executes one statement or instruction from the current program counter, following the execution into function calls. When complete, the program counter resides at the next program statement or instruction unless a function was entered, in which case the program counter resides at the first statement or instruction in the function.

Step Over Button. Executes one statement or instruction from the current program counter without following the execution into function calls. When complete, the program counter resides at the next program statement or instruction.

Step Out Button. Executes the remaining statements or instructions in the current function and returns to the statement or instruction following the call to the current function.

Set Next Instruction Button. Sets the program counter to the line containing the cursor in the active file or the Disassembly window.

Insert/Remove Breakpoint Button. Sets a new breakpoint or removes an existing breakpoint at the line containing the cursor in the active file or the Disassembly window. A breakpoint must be placed on a valid code line (a C source line with a blue dot displayed in the gutter or any instruction line in the Disassembly window). For more information about breakpoints, see the Using Breakpoints section on page 343.

Enable/Disable Breakpoint Button. Activates or deactivates the existing breakpoint at the line containing the cursor in the active file or the Disassembly window. A red octagon indicates an enabled breakpoint; a white octagon indicates a disabled breakpoint. For more information about breakpoints, see the Using Breakpoints section on page 343.

Disable All Breakpoints Button. Deactivates all breakpoints in the currently loaded project. To remove breakpoints from your program, use the **Remove All Breakpoints** button.

Remove All Breakpoints Button. Deletes all breakpoints in the currently loaded project. To deactivate breakpoints in your project, use the **Disable All Breakpoints** button.

Debug Windows Toolbar

The Debug Windows toolbar, shown in Figure 20, allows you to display a number of Debug windows using a number of buttons, each of which is briefly described below.

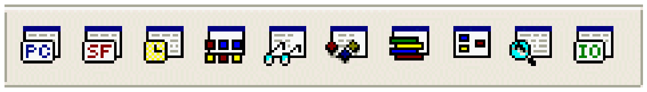


Figure 20. The Debug Windows Toolbar

Registers Window Button. Displays or hides the Registers window. This window is described in the Registers Window section on page 329.

Special Function Registers Window Button. Opens one of ten Special Function Registers windows. This window is described in the Special Function Registers Window section on page 330.

Clock Window Button. Displays or hides the Clock window. This window is described in the Clock Window section on page 331.

Memory Window Button. Opens one of ten Memory windows. This window is described in the Memory Window section on page 331.

Watch Window Button. Displays or hides the Watch window. This window is described in the Watch Window section on page 337.

Locals Window Button. Displays or hides the Locals window. This window is described in the Locals Window section on page 339.

Call Stack Window Button. Displays or hides the Call Stack window. This window is described in the Call Stack Window section on page 340.

Symbols Window Button. Displays or hides the Symbols window. This window is described in the Symbols Window section on page 340.

Disassembly Window Button. Displays or hides the Disassembly window. This window is described in the Disassembly Window section on page 341.

Simulated UART Output Window Button. Displays or hides the Simulated UART Output window. This window is described in the Simulated UART Output Window section on page 342.

Windows

Four ZDS II windows allow you to see various aspects of the tools while working with your project. The first three of these windows, listed below, are described in this section; the fourth is described in the chapter titled *Using the Debugger.*

- Project Workspace Window – see page 23
- Edit Window – see page 24
- Output Windows – see page 29
- Debug Windows – see page 329

Project Workspace Window

The Project Workspace window, located on the left side of the developer's environment (and shown in Figures 21 and 22), allows you to view your project files.

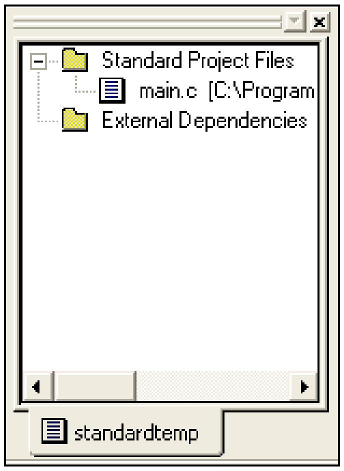


Figure 21. Project Workspace Window for Standard Projects

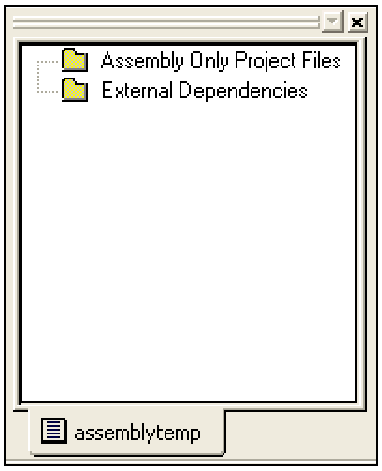


Figure 22. Project Workspace Window for Assembly Only Projects

The Project Workspace window provides access to related functions using context menus. To access context menus, right-click a file or folder in the window. Depending on which file or folder is highlighted, the context menu provides some or all of the following functions:

- Dock the Project Workspace window
- Hide the Project Workspace window
- Add files to the project
- Remove the highlighted file(s) from the project
- Build project files or external dependencies
- Build or compile the highlighted file
- Undock the Project Workspace window, allowing it to float in the Edit window

Edit Window

This section covers the following topics:

- Using the Context Menus – see page 25
- Using Bookmarks – see page 26

The Edit window area, located on the right side of the developer's environment (and shown in Figure 23), allows you to edit the files in your project.

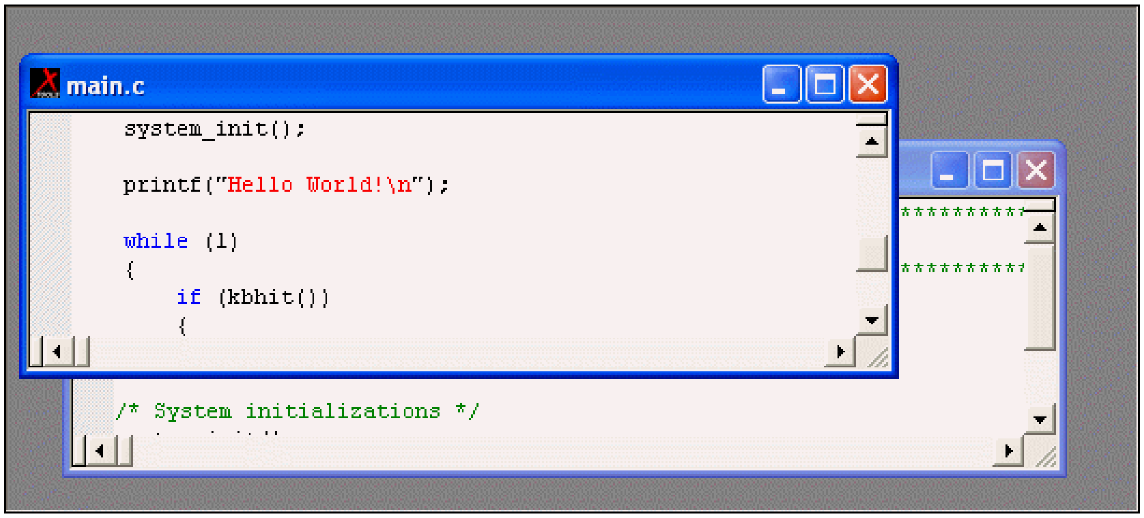

```
system_init();

printf("Hello World!\n");

while (1)
{
    if (kbhit())
    {
/* System initializations */
```

Figure 23. Edit Window

The Edit window supports the shortcuts listed in Table 1.

Table 1. Edit Shortcuts

Shortcuts	Function
Ctrl + Z	Undo
Ctrl + Y	Redo
Ctrl + X	Cut
Ctrl + C	Copy
Ctrl + V	Paste
Ctrl + F	Find
F3	Repeat the previous search
Ctrl + G	Go to
Ctrl + E Ctrl +]	Go to the matching { or } symbol. Place your cursor at the right or left of an opening or closing brace and press Ctrl + E or Ctrl +] to move the cursor to the matching opening or closing brace.

Using the Context Menus

There are two context menus in the Edit window, depending on where you click. When you right-click in a file, the context menu allows you to do the following (depending on whether any text is selected or you are running in Debug mode):

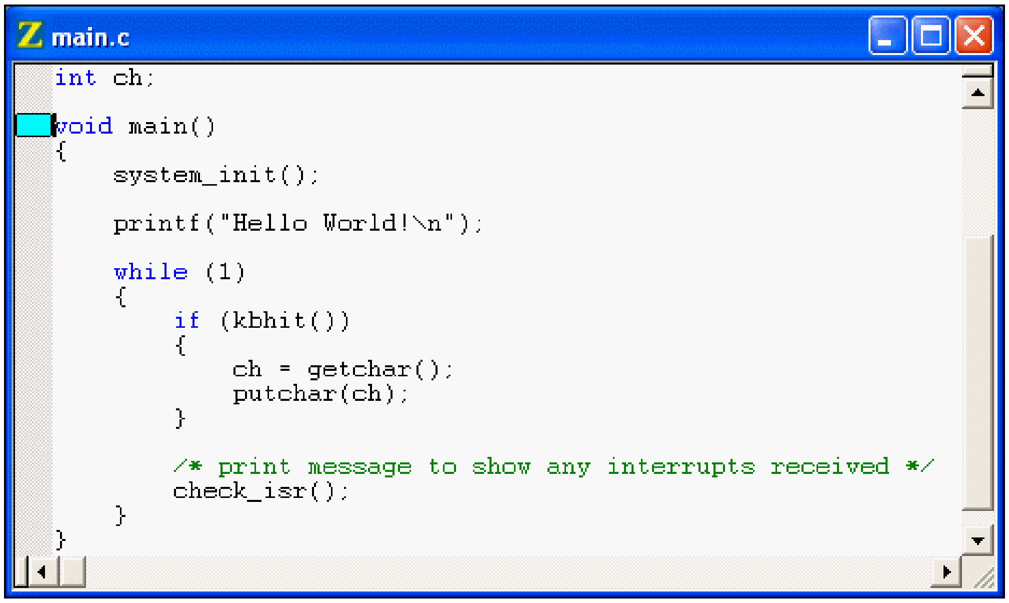
- Cut, copy, and paste text
- Go to the Disassembly window
- Show the program counter
- Insert, edit, enable, disable, or remove breakpoints
- Reset the debugger
- Stop debugging
- Start or continue running the program (Go)
- Run to the cursor
- Pause the debugging (Break)
- Step into, over, or out of program instructions
- Set the next instruction at the current line
- Insert or remove bookmarks (for more, see the Using Bookmarks section on page 26)

When you right-click outside of all files, the context menu allows you to perform the following tasks:

- Show or hide the Output windows, Project Workspace window, status bar, File toolbar, Build toolbar, Find toolbar, Command Processor toolbar, Debug toolbar, Debug Windows toolbar
- When in Workbook Mode, each open file features an associated tab along the bottom of the Edit Windows area that allows users to toggle in and out of Workbook Mode.
- Customize the buttons and toolbars

Using Bookmarks

A bookmark is a marker that identifies a position within a file. Bookmarks appear as cyan boxes in the gutter portion (left) of the file window, as shown in Figure 24. The cursor can be quickly positioned on a lines containing bookmarks.

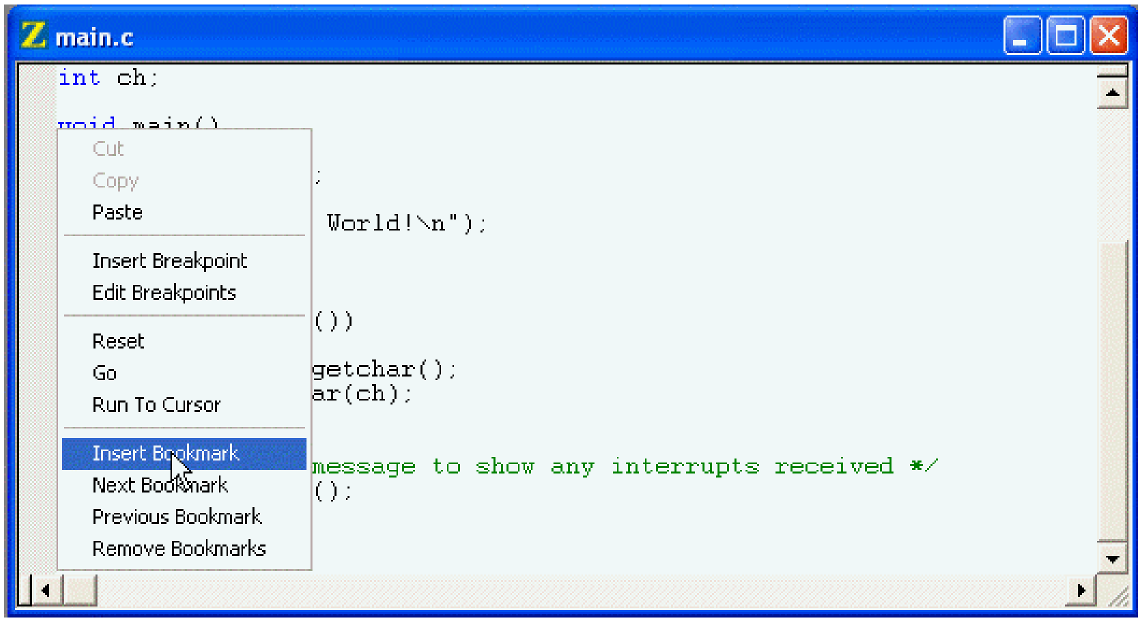


```
int ch;

void main()
{
    system_init();

    printf("Hello World!\n");

    while (1)
    {
        if (kbhit())
        {
            ch = getchar();
            putchar(ch);
        }

        /* print message to show any interrupts received */
        check_isr();
    }
}
```

Figure 24. Bookmark Example

To insert a bookmark, position the cursor on the desired line of the active file and perform one of the following actions:

- Right-click in the Edit window and select **Insert Bookmark** from the resulting context menu, shown in Figure 25.
- Select **Toggle Bookmark** from the **Edit** menu.
- Using your keyboard, enter `Ctrl+M`.



```
int ch;

void main()
                         ;
                          World!\n");

                         ())

                         getchar();
                         ar(ch);

                         message to show any interrupts received */
                         ();
```

Figure 25. Inserting a Bookmark

To remove a bookmark, position the cursor on the line of the active file containing the bookmark to be removed and perform one of the following actions:

- Right-click in the Edit window and select **Remove Bookmark** from the resulting context menu
- Select **Toggle Bookmark** from the **Edit** menu
- Using your keyboard, enter `Ctrl+M`

To remove all bookmarks in the active file, right-click in the Edit window and select **Remove Bookmarks** from the resulting context menu.

To remove all bookmarks in the current project, select **Remove All Bookmarks** from the **Edit** menu.

To position the cursor at the next bookmark in the active file, perform one of the following actions:

- Right-click in the Edit window and select **Next Bookmark** from the resulting context menu.
- Select **Next Bookmark** from the **Edit** menu.
- Press the F2 key. The cursor moves forward through the file, starting at its current position and beginning again when the end of file is reached, until a bookmark is

encountered. If no bookmarks are set in the active file, this function has no effect.

To position the cursor at the previous bookmark in the active file, perform one of the following actions:

- Right-click in the Edit window and select **Previous Bookmark** from the resulting context menu.
- Select **Previous Bookmark** from the **Edit** menu.
- Using your keyboard, enter `Shift+F2`. The cursor moves backwards through the file, starting at its current position and starting again at the end of the file when the file beginning is reached, until a bookmark is encountered. If no bookmarks are set in the active file, this function has no effect.

Output Windows

The Output windows display output, errors, and other feedback from various components of the Integrated Development Environment.

Select one of the tabs at the bottom of the Output window to select one of the Output windows, each of which is listed below and is described in this section.

- Build Output Window – see page 29
- Debug Output Window – see page 30
- Find in Files Output Windows – see page 30
- Messages Output Window – see page 31
- Command Output Window – see page 31

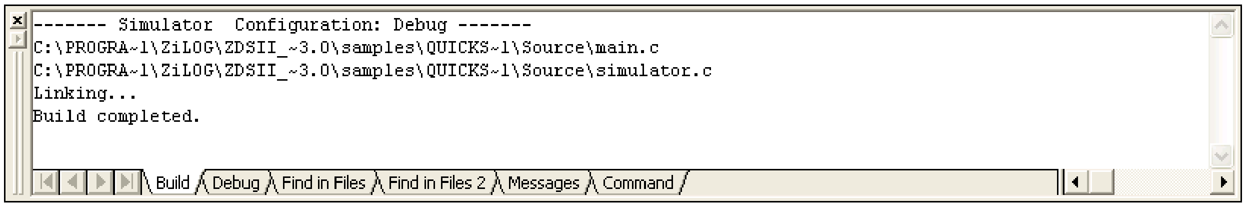
To dock the Output window with another window, click and hold the window’s grip bar, then move the window.

Double-click the window’s grip bar to cause it to become a floating window.

Double-click the floating window's title bar to change it to a dockable window.

Use the context menu to copy text from or to delete all text in the Output window.

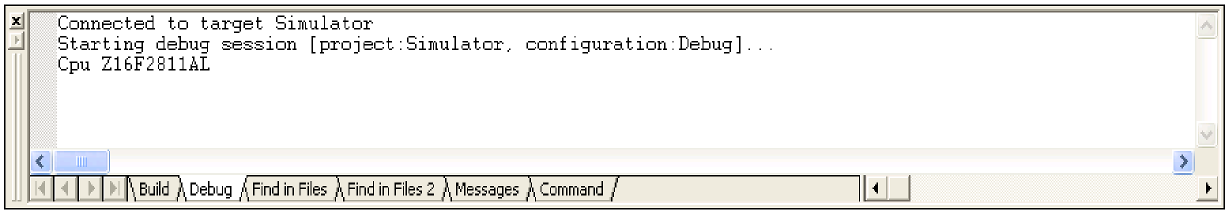
Build Output Window. Holds all text messages generated by the compiler, assembler, librarian, and linker, including error and warning messages; see Figure 26.



```
------- Simulator  Configuration: Debug -------
C:\PROGRA~1\ZiLOG\ZDSII_~3.0\samples\QUICKS~1\Source\main.c
C:\PROGRA~1\ZiLOG\ZDSII_~3.0\samples\QUICKS~1\Source\simulator.c
Linking...
Build completed.
```

Figure 26. Build Output Window

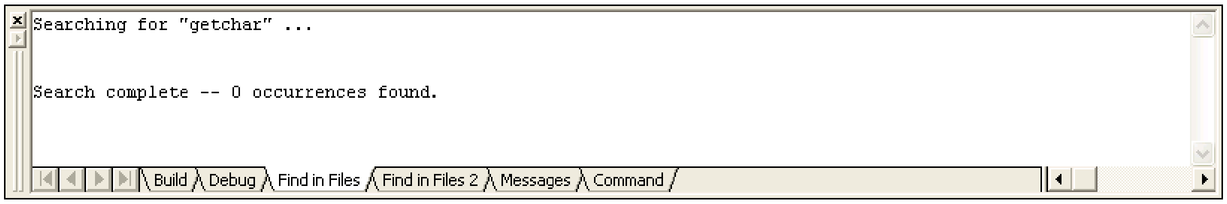
Debug Output Window. Holds all text messages generated by the debugger while you are in Debug Mode; see Figure 27.



```
Connected to target Simulator
Starting debug session [project:Simulator, configuration:Debug]...
Cpu Z16F2811AL
```

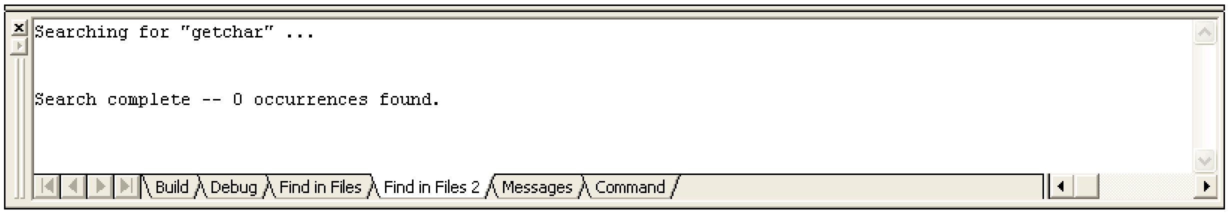
Figure 27. Debug Output Window

Find in Files Output Windows. The two Find in Files Output windows, shown in Figures 28 and 29, display the results of the Find in Files command (available from the **Edit** menu and the **Find** toolbar). The **File in Files 2** window is used when the **Output to Pane 2** checkbox is selected in the **Find in File** dialog box (for more, see Find in Files on page 41).



```
Searching for "getchar" ...


Search complete -- 0 occurrences found.
```

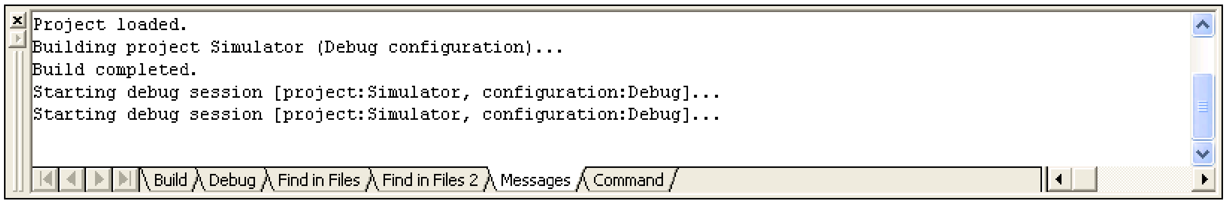
Figure 28. Find in Files Output Window



```
Searching for "getchar" ...


Search complete -- 0 occurrences found.
```

Figure 29. Find in Files 2 Output Window

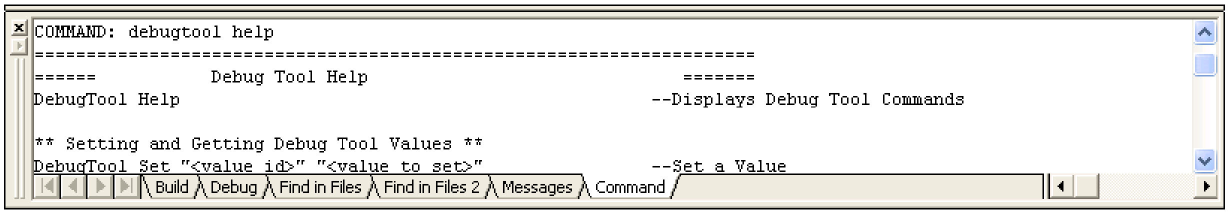
Messages Output Window. Holds informational messages intended for the user. The Messages Output window also displays the chip revision identifier (always `0x0800` for ZNEO) and the Smart Cable firmware version; see Figure 30.



```
Project loaded.
Building project Simulator (Debug configuration)...
Build completed.
Starting debug session [project:Simulator, configuration:Debug]...
Starting debug session [project:Simulator, configuration:Debug]...
```

Figure 30. Messages Output Window

Command Output Window. Holds output from the execution of commands; see Figure 31.



```
COMMAND: debugtool help
=====================================================================
======           Debug Tool Help                              =======
DebugTool Help                                             --Displays Debug Tool Commands

** Setting and Getting Debug Tool Values **
DebugTool Set "<value id>" "<value to set>"                --Set a Value
```

Figure 31. Command Output Window

Menu Bar

The menu bar lists menu items used in the ZNEO developer's environment. Clicking a menu bar item displays a list of selection items. If an option on a menu item ends with an ellipsis (...), selecting the option displays a dialog box. A number of items are displayed on the menu bar; each is listed below and described in this section.

- File Menu – see page 32

- Edit Menu – see page 40
- View Menu – see page 44
- Project Menu – see page 45
- Build Menu – see page 83
- Debug Menu – see page 86
- Tools Menu – see page 87
- Window Menu – see page 104
- Help Menu – see page 105

File Menu

The **File** menu enables you to perform basic commands in the developer's environment; each of these file commands is briefly described in this section.

New File. Select **New File** from the **File** menu to create a new file in the Edit window.

Open File. Select **Open File** from the **File** menu to display the **Open** dialog box (see Figure 32) which allows you to open the files for viewing and editing.

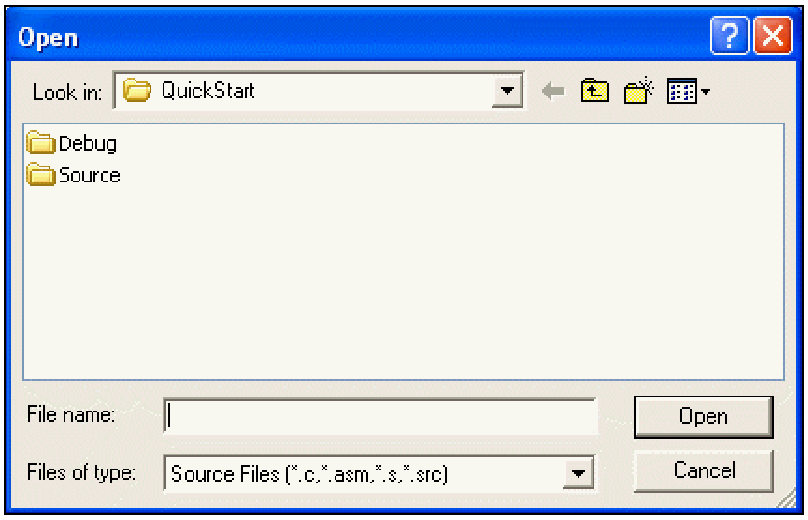
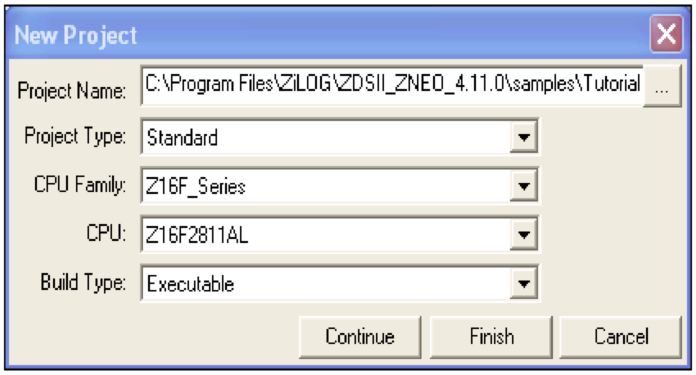


Figure 32. Open Dialog Box

Close File. Select **Close File** from the **File** menu to close the selected file.

New Project. To create a new project, perform the following brief procedure:

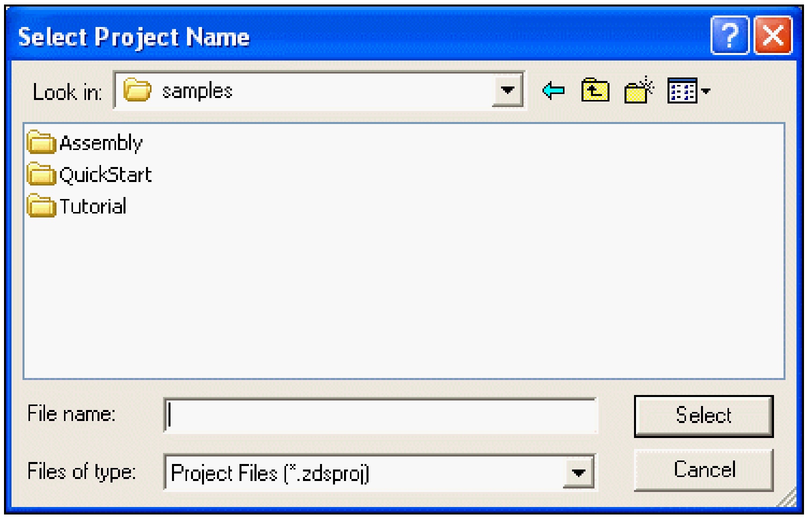
1. Select **New Project** from the **File** menu. The **New Project** dialog box is displayed; see Figure 33.



Project Name: C:\Program Files\ZiLOG\ZDSII_ZNEO_4.11.0\samples\Tutorial
Project Type: Standard
CPU Family: Z16F_Series
CPU: Z16F2811AL
Build Type: Executable

Figure 33. New Project Dialog Box

2. From the **New Project** dialog box, click the **Browse** button (...) to navigate to the directory in which you'll save your project. The **Select Project Name** dialog box is displayed; see Figure 34.



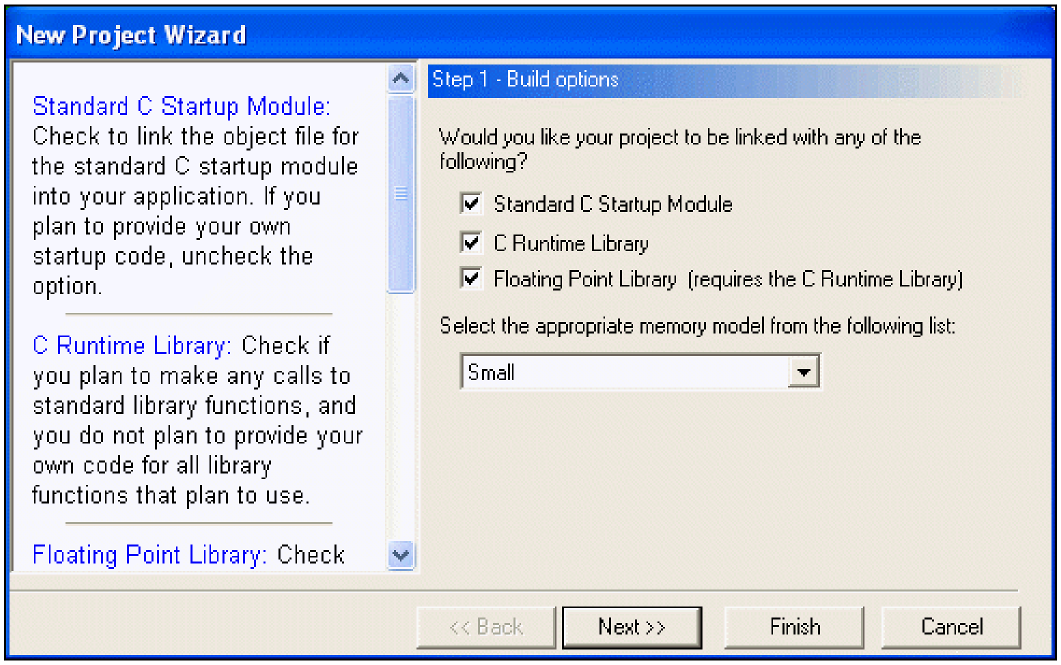
Look in: samples
Assembly
QuickStart
Tutorial
File name:
Files of type: Project Files (*.zdsproj)

Figure 34. Select Project Name Dialog Box

3. In the **File Name** field, enter the name of your project. You do not have to enter the `.zdsproj` extension; it is added automatically.

Note: The following characters cannot be used in a project name: ( ) $, . - + [ ] ' &

4. Click **Select** to return to the **New Project** dialog box.
5. In the **Project Type** field, select **Standard** for a project that uses `.c` files. Select **Assembly Only** for a project that will include only assembly source code.
6. In the **CPU Family** drop-down menu, select **Z16F_Series**.
7. In the **CPU** drop-down menu, select a CPU.
8. In the **Build Type** drop-down menu, select **Executable** to build an application or select **Static Library** to build a static library. The default is **Executable**, which creates an IEEE 695 executable format (`.lod`). For more information, see the Project Settings—Output Page section on page 72.
9. Click **Continue** to change the default project settings using the New Project Wizard. To accept all default settings, or to create static libraries, click **Finish**. For Standard projects, the **New Project Wizard** dialog box is displayed (see Figure 35). For Assembly-Only executable projects, continue to Step 11.


New Project Wizard

Standard C Startup Module: Check to link the object file for the standard C startup module into your application. If you plan to provide your own startup code, uncheck the option.

C Runtime Library: Check if you plan to make any calls to standard library functions, and you do not plan to provide your own code for all library functions that plan to use.

Floating Point Library: Check

Step 1 - Build options

Would you like your project to be linked with any of the following?

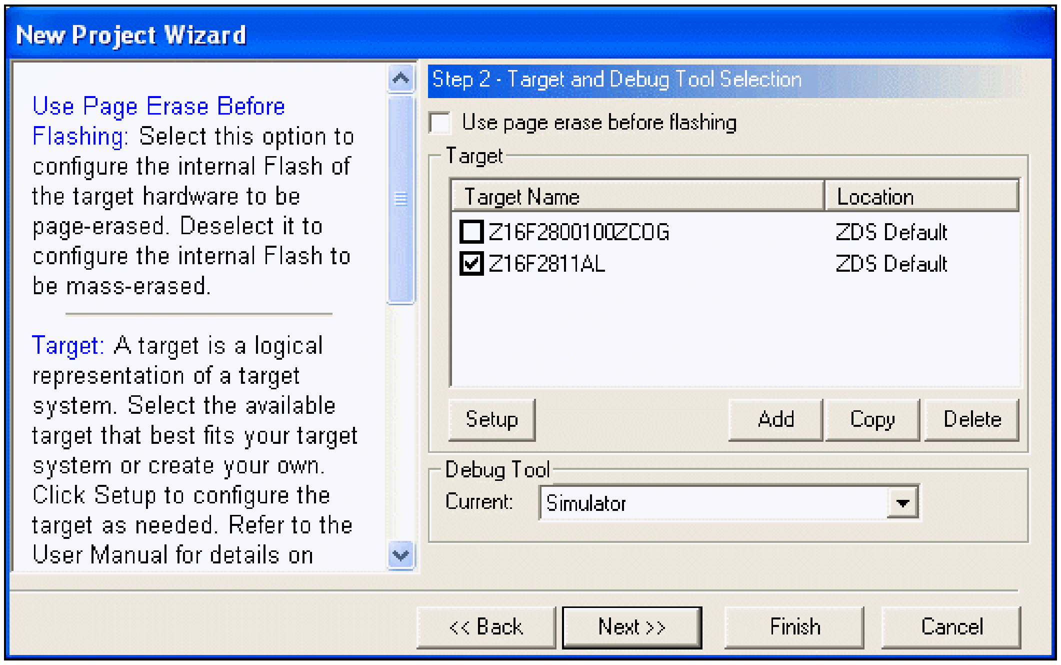
[✓] Standard C Startup Module
[✓] C Runtime Library
[✓] Floating Point Library (requires the C Runtime Library)

Select the appropriate memory model from the following list:

Small

Figure 35. New Project Wizard Dialog Box—Build Options

10. Select whether your project is linked to the standard C start-up module, C run-time library, and floating-point library; select a small or large memory model (see the Memory Models section on page 161); and click **Next**. For executable projects, the **Target and Debug Tool Selection** step of the **New Project Wizard** dialog box is displayed; see Figure 36.



Use Page Erase Before Flashing: Select this option to configure the internal Flash of the target hardware to be page-erased. Deselect it to configure the internal Flash to be mass-erased.

Target: A target is a logical representation of a target system. Select the available target that best fits your target system or create your own. Click Setup to configure the target as needed. Refer to the User Manual for details on

☐ Use page erase before flashing

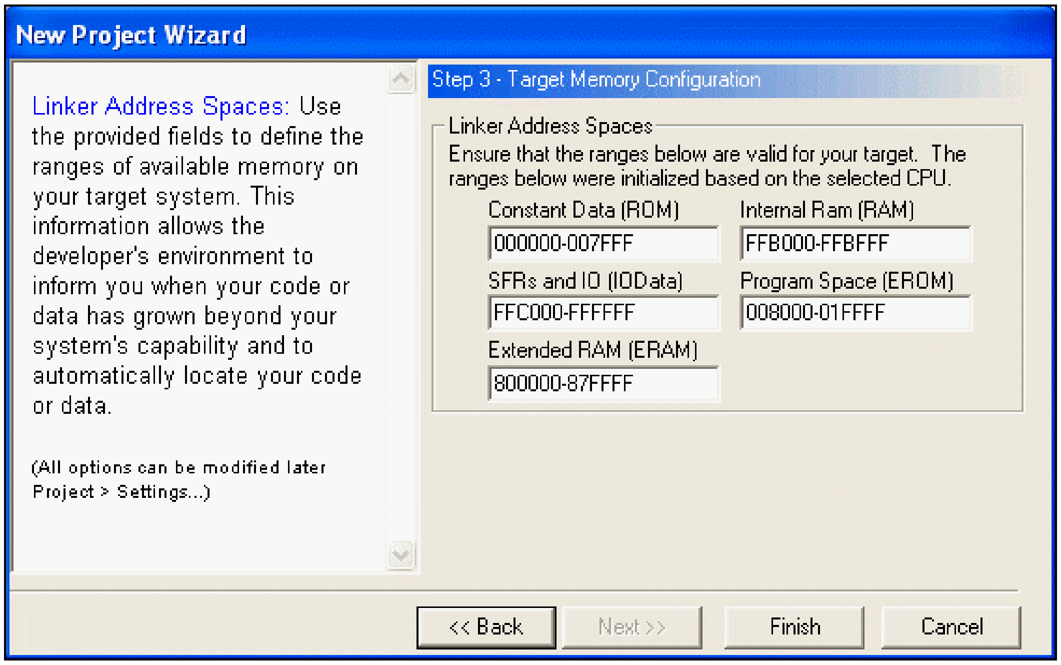
Target

Target Name	Location
☐ Z16F2800100ZCOG	ZDS Default
☑ Z16F2811AL	ZDS Default

Debug Tool
Current: Simulator

Figure 36. New Project Wizard Dialog Box—Target and Debug Tool Selection

11. Select the **Use Page Erase Before Flashing** checkbox to configure the internal Flash memory of the target hardware to be page-erased. If this checkbox is not selected, the internal Flash is configured to be mass-erased.
12. Select the appropriate target from the **Target** list box.
13. Click **Setup** in the Target area. Refer to the Setup section on page 76 for details on configuring a target.
14. Click **Add** to create a new target (see Add on page 78) or click **Copy** to copy an existing target (see Copy on page 79).
15. Select the appropriate debug tool and (if you have not selected the Simulator) click **Setup** in the Debug Tool area. Refer to the Debug Tool section on page 80 for details about the available debug tools and how to configure them.
16. Click **Next**. The **Target Memory Configuration** step of the **New Project Wizard** dialog box is displayed; see Figure 37.


New Project Wizard

Linker Address Spaces: Use the provided fields to define the ranges of available memory on your target system. This information allows the developer's environment to inform you when your code or data has grown beyond your system's capability and to automatically locate your code or data.

(All options can be modified later Project > Settings...)

Step 3 - Target Memory Configuration

Linker Address Spaces

Ensure that the ranges below are valid for your target. The ranges below were initialized based on the selected CPU.

Constant Data (ROM): 000000-007FFF
Internal Ram (RAM): FFB000-FFBFFF
SFRs and IO (IOData): FFC000-FFFFFF
Program Space (EROM): 008000-01FFFF
Extended RAM (ERAM): 800000-87FFFF

Figure 37. New Project Wizard Dialog Box—Target Memory Configuration

17. Enter the memory ranges appropriate for the target CPU.
18. Click **Finish**.

Open Project. To open an existing project, perform the following procedure:

1. Select **Open Project** from the **File** menu. The **Open Project** dialog box is displayed; see Figure 38.

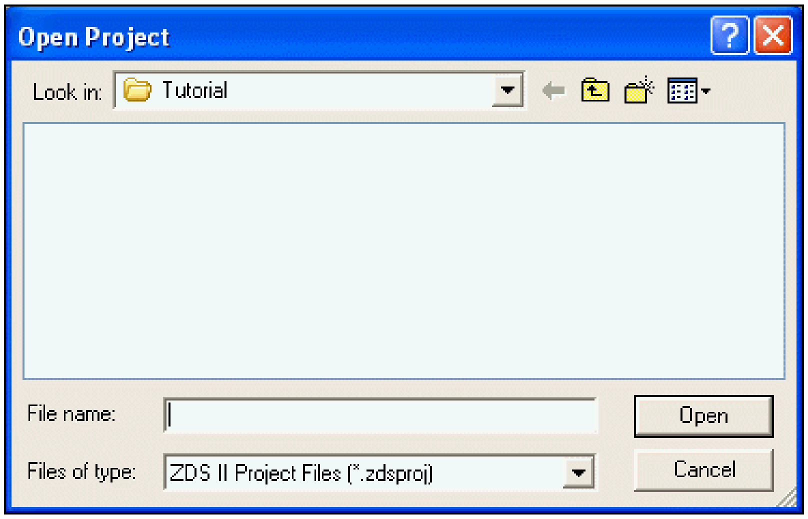


Figure 38. Open Project Dialog Box

2. Use the **Look In** drop-down menu to navigate to the directory in which your project is located.
3. Select the project to be opened, and click **Open** to open your project.

Note: To quickly open a project you were working in recently, see the Recent Projects section on page 39.

Save Project. Select **Save Project** from the **File** menu to save the currently active project. By default, project files and configuration information are saved in a file named <*project name*>`.zdsproj`. An alternate file extension is used if provided when the project is created.

Note: The <*project name*>`.zdsproj`.file contains all project data. If deleted, the project is no longer available.

If the **Save/Restore Project Workspace** checkbox is selected (see the Options—General Tab section on page 96), a file named <*project name*>`.wsp` is also created or updated with workspace information such as window locations and bookmark details. The`.wsp` file supplements the project information. If it is deleted, the last known workspace data is lost, but this does not affect or harm the project.

Close Project. Select **Close Project** from the **File** menu to close the currently active project.

Save. Select **Save** from the **File** menu to save the active file.

Save As

To save the active file with a new name, perform the following steps:

1. Select **Save As** from the **File** menu. The **Save As** dialog box is displayed; see Figure 39.

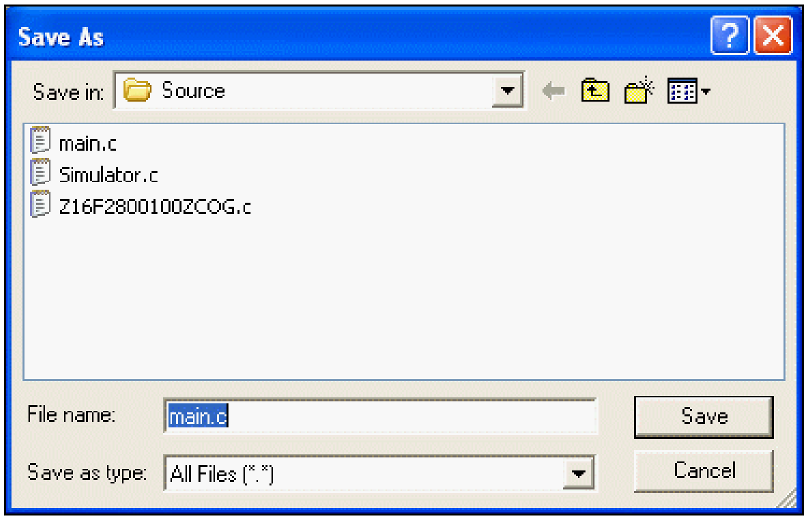


Figure 39. Save As Dialog Box

2. Use the **Save In** drop-down menu to navigate to the appropriate directory.
3. Enter the new file name in the **File Name** field.
4. Use the **Save as Type** drop-down menu to select the file type.
5. Click **Save**. A copy of the file is saved with the name you entered.

Save All. Select **Save All** from the **File** menu to save all open files and the currently loaded project.

Print. Select **Print** from the **File** menu to print the active file.

Print Preview. Select **Print Preview** from the **File** menu to display the file you want to print in Preview mode in a new window, as shown in the following brief procedure.

1. In the Edit window, highlight the file you want to show a Print Preview.

2. From the **File** menu, select **Print Preview**. The file is displayed in a new window. As shown in Figure 40, `main.c` is in Print Preview mode.

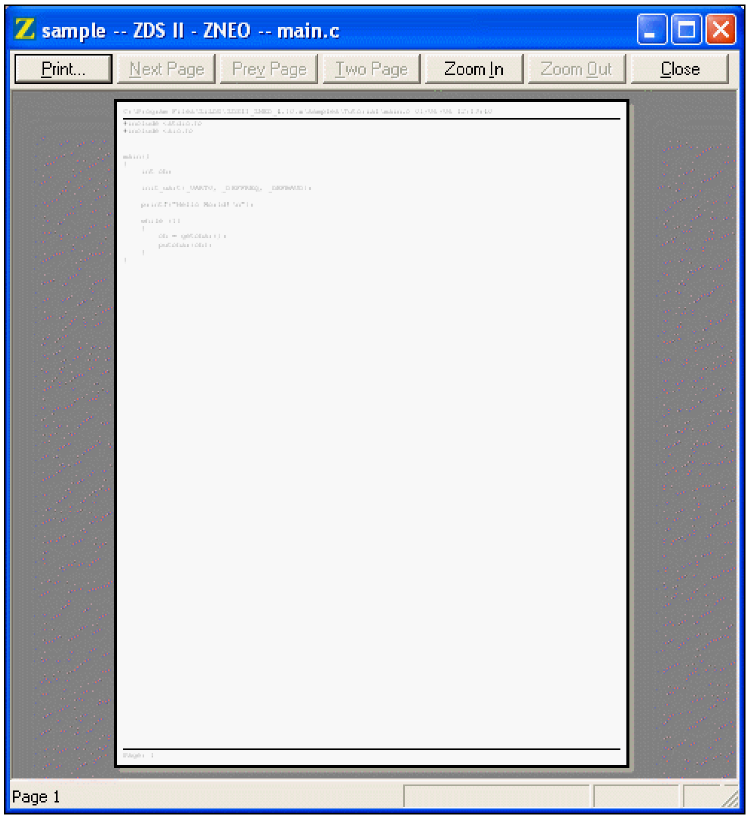


Figure 40. Print Preview Window

3. To print the file, click **Print**. To cancel the print preview, click **Close**. The file returns to its edit mode in the Edit window.

Print Setup. Select **Print Setup** from the **File** menu to display the **Print Setup** dialog box, which allows you to modify the printer's default configuration, if desired, before you print the file.

Recent Files. Select **Recent Files** from the **File** menu and then select a file from the resulting submenu to open a recently opened file.

Recent Projects. Select **Recent Projects** from the **File** menu and then select a project file from the resulting submenu to quickly open a recently opened project.

Exit. Select **Exit** from the **File** menu to exit the application.

Edit Menu

The **Edit** menu provides access to basic editing, text search, and breakpoint and bookmark manipulation features. A number of edit menu options are available; each is described in this section.

Undo. Undo the last edit made to the active file.

Redo. Redo the last edit made to the active file.

Cut. Delete selected text from the active file and place it on the Windows clipboard.

Copy. Copy selected text from the active file and put it on the Windows clipboard.

Paste. Paste the current contents of the clipboard into the active file at the current cursor position.

Delete. Delete selected text from the active file.

Select All. Highlight all text in the active file.

Show Whitespaces. Select **Show Whitespaces** from the **Edit** menu to display all whitespace characters such as spaces and tabs in the active file.

Find. To find text in the active file, observe the following procedure:

1. Select **Find** from the **Edit** menu. The **Find** dialog box is displayed; see Figure 41.

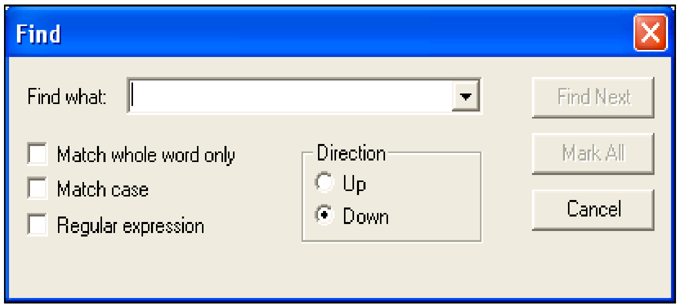


Figure 41. Find Dialog Box

2. Enter a search string in the **Find What** field or select a recent entry from the **Find What** drop-down menu. (By default, the currently selected text in a source file or the text where your cursor is located in a source file is displayed in the **Find What** field.)
3. Select the **Match Whole Word Only** checkbox if you want to ignore the search text when it occurs as part of longer words.

4. Select the **Match Case** checkbox if you want the search to be case-sensitive.
5. Select the **Regular Expression** checkbox if you want to use regular expressions.
6. Select the direction of the search with the **Up** or **Down** button.
7. Click **Find Next** to jump to the next occurrence of the search text or click **Mark All** to insert a bookmark on each line containing the search text.

Note: After clicking **Find Next**, the dialog box closes. You can press the F3 key or use the **Find Again** command to find the next occurrence of the search term without displaying the Find dialog box again.

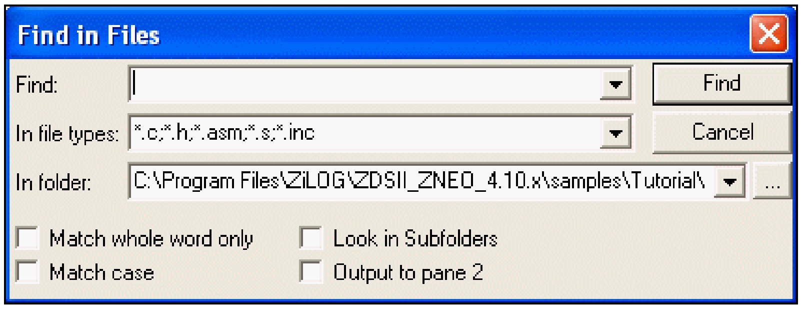
Find Again. Select **Find Again** from the **Edit** menu to continue searching in the active file for text previously entered in the Find dialog box.

Find in Files. This function searches the contents of the files on disk; therefore, unsaved data in open files is not searched.

To find text in multiple files, observe the following procedure:

1. Select **Find in Files** from the **Edit** menu.

   The **Find in Files** dialog box is displayed; see Figure 42.



.c;.h;*.asm;*.s;*.inc

C:\Program Files\ZiLOG\ZDSII_ZNEO_4.10.x\samples\Tutorial\

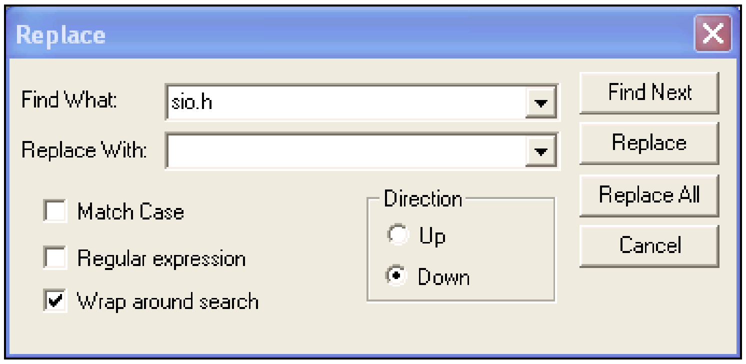
Figure 42. Find in Files Dialog Box

2. Enter a search string in the **Find** field or select a recent entry from the **Find** drop-down menu. (If you select text in a source file before displaying the **Find** dialog box, the text is displayed in the **Find** field.)
3. Select or enter the file type(s) to search for in the **In File Types** drop-down menu. Separate multiple file types with semicolons.

4. Use the **Browse** button (...) or the **In Folder** drop-down menu to select the location of the files in which you plan to search.
5. Select the **Match Whole Word Only** checkbox if you want to ignore the search text when it occurs as part of longer words.
6. Select the **Match Case** checkbox if you want the search to be case-sensitive.
7. Select the **Look in Subfolders** checkbox if you want to search within subfolders.
8. Select the **Output to Pane 2** checkbox if you want the search results displayed in the Find in Files 2 Output window. If this button is not selected, the search results are displayed in the Find in Files Output window.
9. Click **Find** to perform the search.

Replace. To find and replace text in the active file, observe the following procedure:

1. Select **Replace** from the **Edit** menu. The **Replace** dialog box is displayed; see Figure 43.



Find What: sio.h
Replace With:
☐ Match Case
☐ Regular expression
☑ Wrap around search
Direction: Up [✓ Down]

Figure 43. Replace Dialog Box

2. Enter a search string in the **Find What** field or select a recent entry from the **Find What** drop-down menu. (By default, the currently selected text in a source file or the text where your cursor is located in a source file is displayed in the **Find What** field.)
3. Enter the replacement text in the **Replace With** field or select a recent entry from the **Replace With** drop-down menu.
4. Select the **Match Case** checkbox if you want the search to be case-sensitive.
5. Select the **Regular Expression** checkbox if you want to use regular expressions.
6. Select the **Wrap Around Search** checkbox to continue the search past the end (or beginning) of the file until the current cursor position is reached.
7. Select the direction of the search with the **Up** or **Down** button.

8. Click **Find Next** to jump to the next occurrence of the search text, click **Replace** to replace the highlighted text, or click **Replace All** to automatically replace all instances of the search text.

Go to Line. To position the cursor at a specific line in the active file, select **Go to Line** from the **Edit** menu to display the **Go to Line Number** dialog box, as shown in Figure 44. Enter the desired line number in the edit field and click **Go To**.

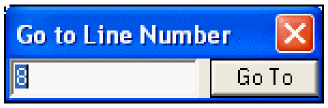
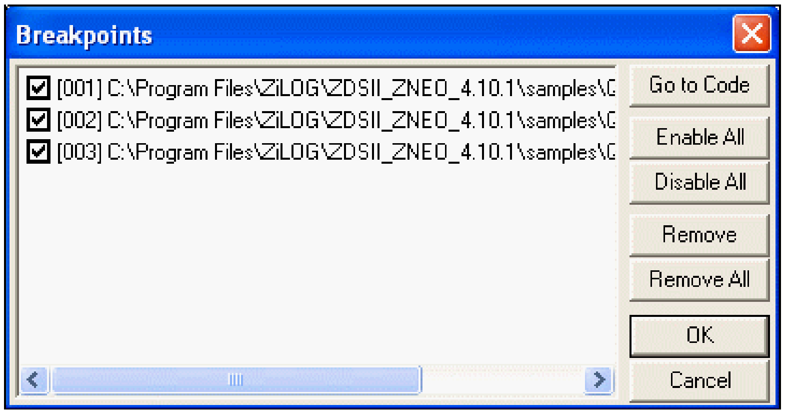


Figure 44. Go to Line Number Dialog Box

Manage Breakpoints

To view, go to, enable, disable, or remove breakpoints in an active project, select **Manage Breakpoints** from the **Edit** menu. You can access the **Breakpoints** dialog box, shown in Figure 45, during Debug mode and Edit mode.



[001] C:\Program Files\ZiLOG\ZDSII_ZNEO_4.10.1\samples\
[002] C:\Program Files\ZiLOG\ZDSII_ZNEO_4.10.1\samples\
[003] C:\Program Files\ZiLOG\ZDSII_ZNEO_4.10.1\samples\

Figure 45. Breakpoints Dialog Box

The **Breakpoints** dialog box lists all existing breakpoints for the currently loaded project. A check mark in the box to the left of the breakpoint description indicates that the breakpoint is enabled. Each of the buttons in this dialog box is described in this section.

Go to Code. To move the cursor to a particular breakpoint you have set in a file, highlight the breakpoint in the **Breakpoints** dialog box and click **Go to Code**.

Enable All. To make all listed breakpoints active, click **Enable All**. Individual breakpoints can be enabled by clicking in the box to the left of the breakpoint description. Enabled breakpoints are indicated by a check mark in the box to the left of the breakpoint description.

Disable All. To make all listed breakpoints inactive, click **Disable All**. Individual breakpoints can be disabled by clicking in the box to the left of the breakpoint description. Disabled breakpoints are indicated by an empty box to the left of the breakpoint description.

Remove. To delete a particular breakpoint, highlight the breakpoint in the **Breakpoints** dialog box and click **Remove**.

Remove All. To delete all of the listed breakpoints, click **Remove All**.

Note: For more information about breakpoints, see the Using Breakpoints section on page 343.

Toggle Bookmark. Select **Toggle Bookmark** from the **Edit** menu to insert a bookmark in the active file for the line in which your cursor is located or to remove the bookmark for the line in which your cursor is located.

Next Bookmark. Select **Next Bookmark** from the **Edit** menu to position the cursor at the line in which the next bookmark in the active file is located.

Note: The search for the next bookmark does not stop at the end of the file; the next bookmark might be the first bookmark in the file.

Previous Bookmark. Select **Previous Bookmark** from the **Edit** menu to position the cursor at the line in which the previous bookmark in the active file is located.

Note: The search for the previous bookmark does not stop at the beginning of the file; the previous bookmark might be the last bookmark in the file.

Remove All Bookmarks. Select **Remove All Bookmarks** from the **Edit** menu to delete all of the bookmarks in the currently loaded project.

View Menu

The **View** menu allows you to select the windows you want to display in the ZNEO developer's environment. The **View** menu contains four options, each of which is described in this section.

Debug Windows. When you are in Debug mode (running the debugger), you can select any of the Debug windows. From the **View** menu, select **Debug Windows** and then the appropriate Debug window. For more information about the Debug windows, see the Debug Windows section on page 329.

The **Debug Windows** submenu contains the following elements, each of which is described in the chapter titled *Using the Debugger.*

- Registers Window – see page 329
- Special Function Registers Window – see page 330
- Clock Window – see page 331
- Memory Window – see page 331
- Watch Window – see page 337
- Locals Window – see page 339
- Call Stack Window – see page 340
- Symbols Window – see page 340
- Disassembly Window – see page 341
- Simulated UART Output Window – see page 342

Workspace. Display or hide the Project Workspace window.

Output. Display or hide the Output windows.

Status Bar. Display or hide the status bar, which resides beneath the Build Output window.

Project Menu

The Project menu allows you to add files to your project, set configurations for your project, and export a make file.

The Project menu contains the following options; the first three are described in this section.

- Add Files – see page 46
- Remove Selected File(s) – see page 46
- Settings – see page 46
- Export Makefile – see page 82

Add Files

To add files to your project, observe the following procedure:

1. From the **Project** menu, select **Add Files**.The **Add Files to Project** dialog box is displayed; see Figure 46.
2. Use the **Look In** drop-down menu to navigate to the appropriate directory in which the files you want to add are saved.

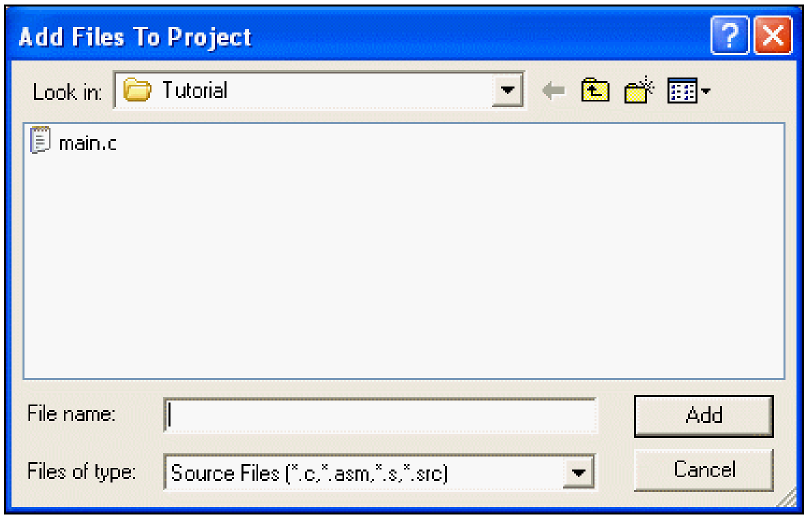


Figure 46. Add Files to Project Dialog Box

3. Click the file you want to add or highlight multiple files by clicking on each file while holding down the Shift key.
4. Click **Add** to add these files to your project.

Remove Selected File(s)

Select this option from the **Project** menu to delete highlighted files in the Project Workspace window.

Settings

Select **Settings** from the **Project** menu to display the **Project Settings** dialog box, which allows you to change your active configuration as well as set up your project.

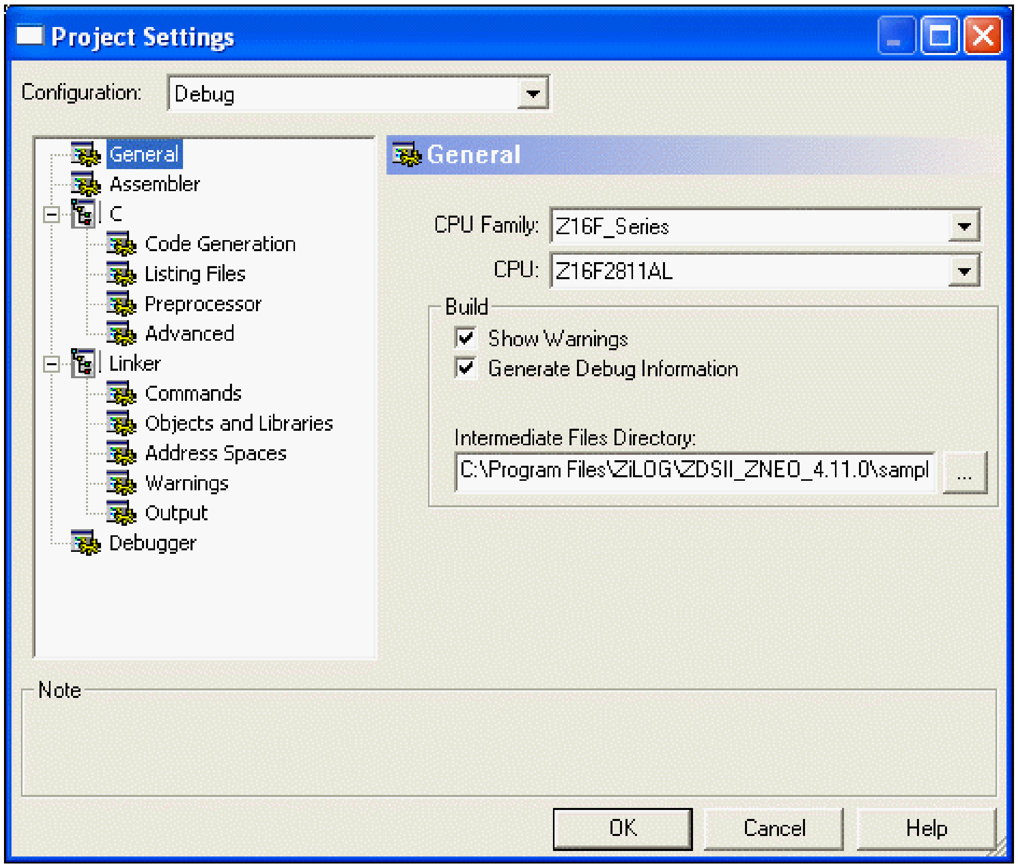
Select the active configuration for the project in the **Configuration** drop-down menu in the upper left corner of the **Project Settings** dialog box. For your convenience, the Debug and Release configurations are predefined. For more information about project configurations, such as adding your own configuration, see the Set Active Configuration section on page 83.

The Project Settings dialog box provides various project configuration pages that can be accessed by selecting the page name in the pane on the left side of the dialog box. There are several pages grouped together for the C (Compiler) and Linker that allow you to set up subsettings for that tool. The pages for the C (Compiler) are Code Generation, Listing Files, Preprocessor, and Advanced. The pages for the Linker are Commands, Objects and Libraries, Address Spaces, Warnings, and Output.

Note: If you change project settings that affect the build, the following message is displayed when you click **OK** to exit the **Project Settings** dialog box: "`The project settings have changed since the last build. Would you like to rebuild the affected files?`" Click **Yes** to save and then rebuild the project.

Each of the Project Settings pages is described in this section.

Project Settings—General Page. From the **Project Settings** dialog box, select the **General** page. The options on the **General** page, shown in Figure 47, are described in this section.



CPU Family: Z16F_Series
CPU: Z16F2811AL
[✓] Show Warnings
[✓] Generate Debug Information
Intermediate Files Directory: C:\Program Files\ZiLOG\ZDSII_ZNEO_4.11.0\sampl

Figure 47. General Page of the Project Settings Dialog Box

CPU Family. Allows you to select the appropriate ZNEO family.

CPU. Defines which CPU you want to define for the ZNEO target. To change the CPU for your project, select the appropriate CPU in the **CPU** drop-down menu.

Note: Selecting a CPU does not automatically select include files for your C or assembly source code. Include files must be manually included in your code. Selecting a new CPU automatically updates the compiler preprocessor defines, assembler defines, and, where necessary, the linker address space ranges and selected debugger target based on the selected CPU.

Show Warnings. This checkbox controls the display of warning messages during all phases of the build. If the checkbox is enabled, warning messages from the assembler, compiler, librarian, and linker are displayed during the build. If the checkbox is disabled, all these warnings are suppressed.

Generate Debug Information. This checkbox makes the build generate debug information that can be used by the debugger to allow symbolic debugging. Enable this option if you are planning to debug your code using the debugger. The checkbox enables debug information in the assembler, compiler, and linker.

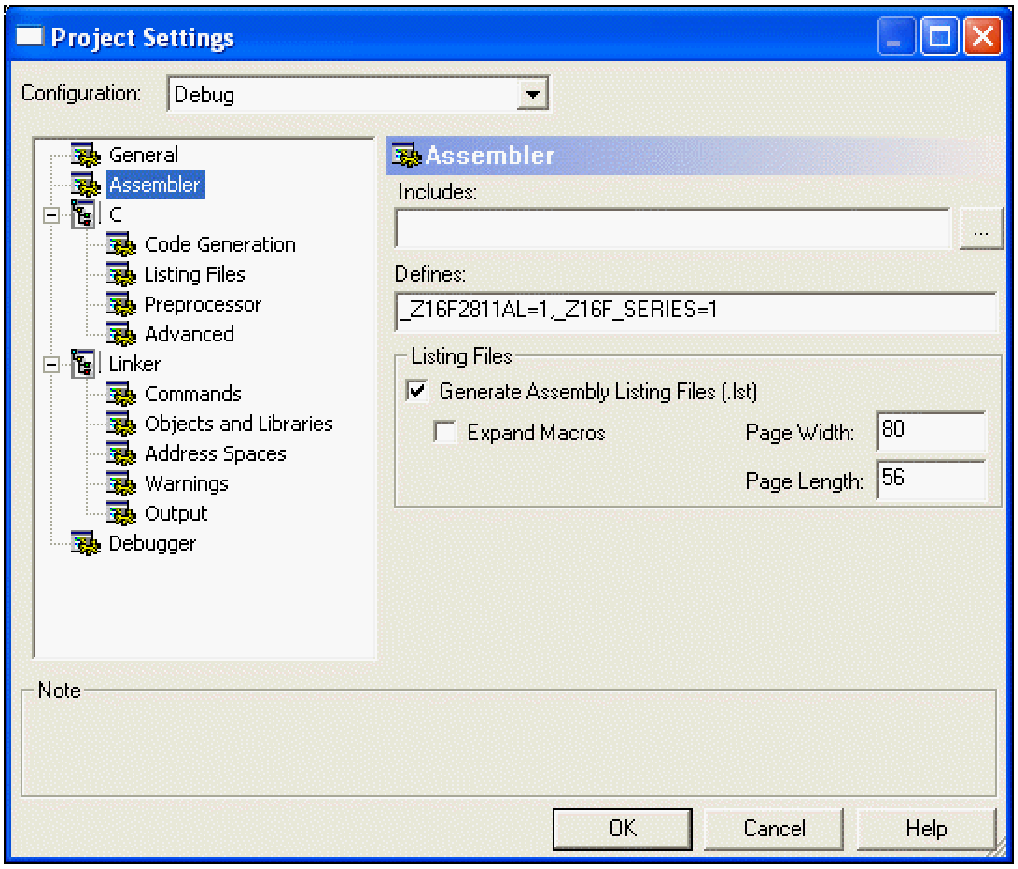
Enabling this option usually increases your overall code size by a moderate amount for two reasons. First, if your code makes any calls to the C run-time libraries, the library version used is the one that was built using the **Limit Optimizations for Easier Debugging** setting (see the Limit Optimizations for Easier Debugging section on page 52). Second, the generated code sets up the stack frame for every function in your own program. Many functions (those whose parameters and local variables are not too numerous and do not have their addresses taken in your code) would not otherwise require a stack frame in the ZNEO architecture, so the code for these functions is slightly smaller if this checkbox is disabled.

Note: The **Generate Debug Information** checkbox interacts with the **Limit Optimizations for Easier Debugging** checkbox on the Code Generation page (see the Limit Optimizations for Easier Debugging section on page 52). When the **Limit Optimizations for Easier Debugging** checkbox is selected, debug information is always generated so that debugging can be performed. The **Generate Debug Information** checkbox is grayed out (disabled) when the **Limit Optimizations for Easier Debugging** checkbox is selected. If the **Limit Optimizations for Easier Debugging** checkbox is later deselected (even in a later ZDS II session), the **Generate Debug Information** checkbox returns to the setting it had before the **Limit Optimizations for Easier Debugging** checkbox was selected.

Ignore Case of Symbols. When the **Ignore Case of Symbols** checkbox is enabled, the assembler and linker ignore the case of symbols when generating and linking code. This checkbox is occasionally required when a project contains source files with case-insensitive labels. This checkbox is only available for Assembly Only projects with no C code.

Intermediate Files Directory. This directory specifies the location where all intermediate files produced during the build will be located. These files include make files, object files, and generated assembly source files and listings that are generated from C source code. This field is provided primarily for the convenience of users who might want to delete these files after building a project, while retaining the built executable and other, more permanent files. Those files are placed into a separate directory specified in the **Output** page (see the Project Settings—Output Page section on page 72).

Project Settings—Assembler Page. In the **Project Settings** dialog box, select the **Assembler** page. The assembler uses the contents of the **Assembler** page to determine which options are to be applied to the files assembled. The options on the Assembler page, shown in Figure 48, are each described in this section.



Configuration: Debug

Assembler

Includes:

Defines: _Z16F2811AL=1,_Z16F_SERIES=1

Listing Files

[✓] Generate Assembly Listing Files (.lst)

[ ] Expand Macros

Page Width: 80

Page Length: 56

Figure 48. Assembler Page of the Project Settings Dialog Box

Includes. The Includes field allows you to specify the series of paths for the assembler to use when searching for include files. The assembler first checks the current directory, then the paths in the Includes field, and finally the default ZDS II include directories.

The ZDS II default include directory is:

<*ZDS Installation Directory*>`\include\std`

where <*ZDS Installation Directory*> is the directory in which Zilog Developer Studio was installed. By default, this would be `C:\Program Files\Zilog\ZDSII_ZNEO_<version>`, where <*version*> might be `4.11.0` or `5.0.0`.

Defines. The Defines field is equivalent to placing <*symbol*> `EQU` <*value*> in your assembly source code. It is useful for conditionally built code. Each defined symbol must have a corresponding value (<*name*>=<*value*>). Multiple symbols can be defined and must be separated by commas.

Generate Assembly Listing Files (.lst). When selected, the **Generate Assembly Listing Files (.lst)** checkbox tells the assembler to create an assembly listing file for each assembly source code module. This file displays the assembly code and directives, as well as the hexadecimal addresses and op codes of the generated machine code. The assembly listing files are saved in the directory specified by the Intermediate Files Directory field on the **General** page (see the Intermediate Files Directory section on page 49). By default, this checkbox is selected.

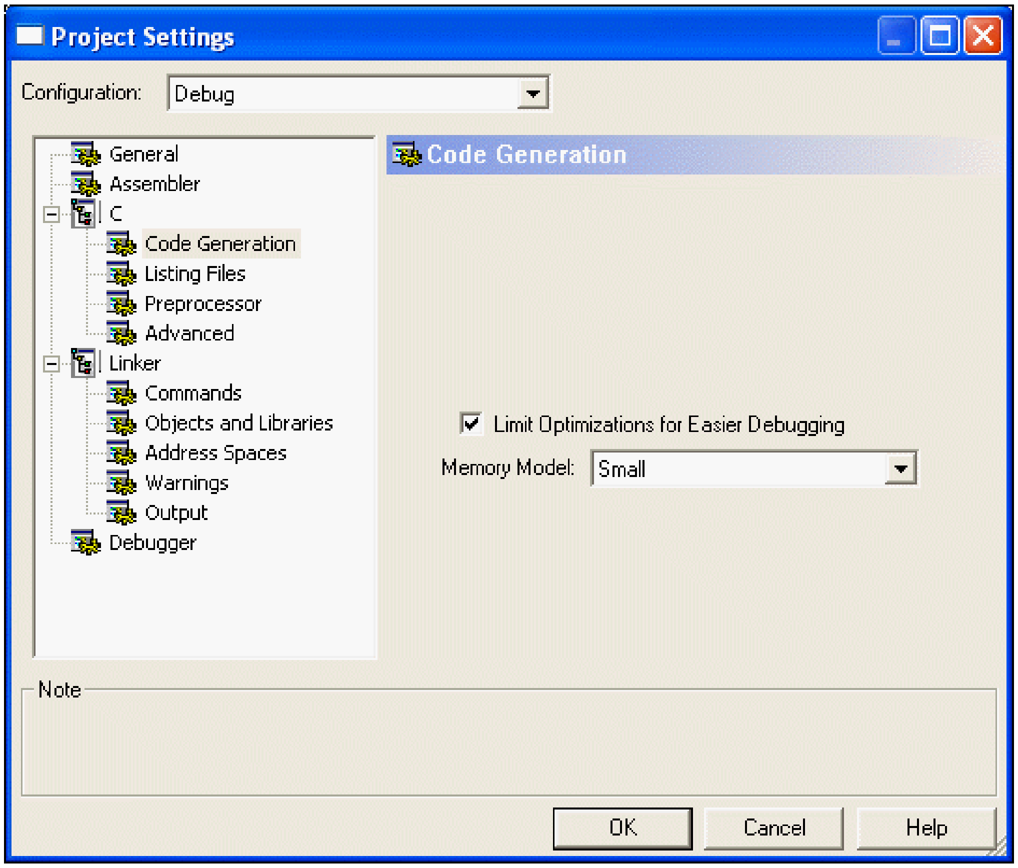
Expand Macros. When selected, the **Expand Macros** checkbox tells the assembler to expand macros in the assembly listing files.

Page Length. When the assembler generates the listing files, the Page Length field sets the maximum number of lines between page breaks. The default is 56.

Page Width. When the assembler generates the listing files, the Page Width field sets the maximum number of characters on a line. The default is 80; the maximum width is 132.

Project Settings—Code Generation Page. For Assembly Only projects, the Code Generation page is not available.

Figure 49 shows the Code Generation page.



[✓] Limit Optimizations for Easier Debugging

Memory Model: Small

Figure 49. Code Generation Page of the Project Settings Dialog Box

Limit Optimizations for Easier Debugging. Selecting this checkbox causes the compiler to generate code in which certain optimizations are turned off. These optimizations can cause confusion when debugging. For example, they might rearrange the order of instructions so that they are no longer exactly correlated with the order of source code statements or remove code or variables that are not used. You can still use the debugger to debug your code without selecting this checkbox, but it might difficult because of the changes that these optimizations make in the assembly code generated by the compiler.

Selecting this checkbox makes it more straightforward to debug your code and interpret what you see in the various Debug windows. However, selecting this checkbox also causes a moderate increase in code size. Many users select this checkbox until they are ready to go to production code and then deselect it.

Selecting this checkbox can also increase the data size required by your application. This happens because this option turns off the use of register variables (see the Use Register Variables section on page 57). The variables that are no longer stored in registers must instead be stored in memory (and on the stack if dynamic frames are in use), thereby increasing the overall data storage requirements of your application. Usually this increase is fairly small.

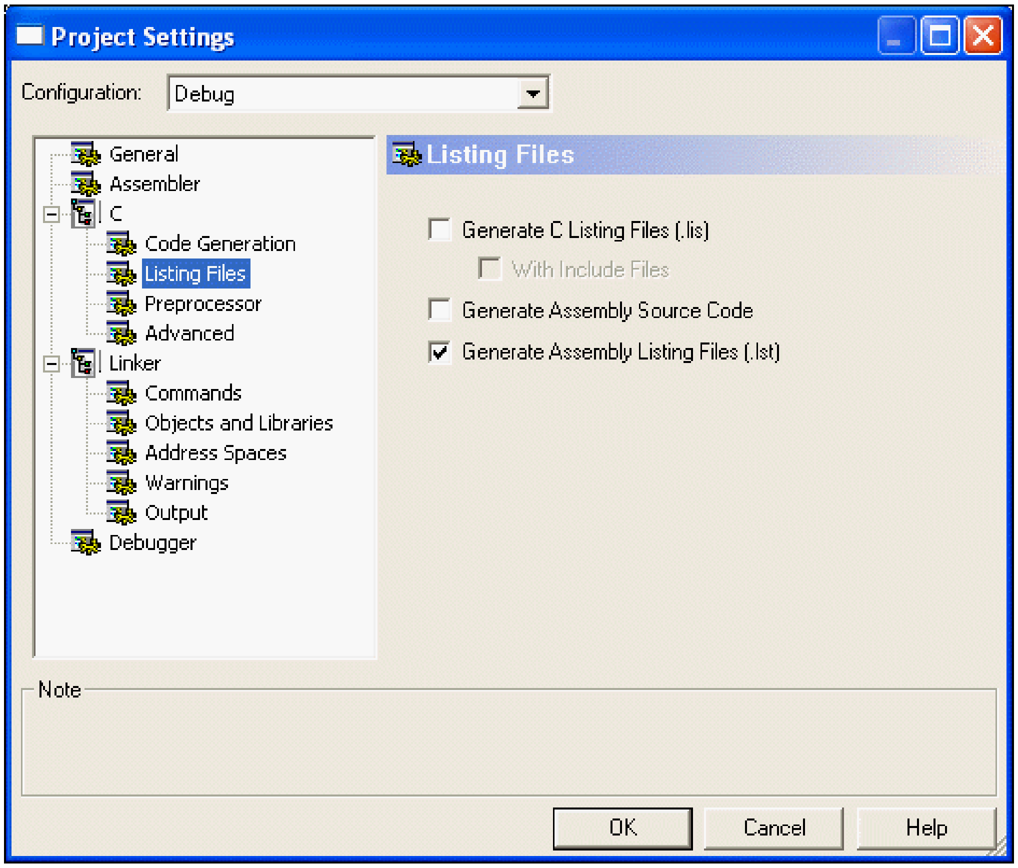
You *can* debug your application when this checkbox is deselected. The debugger continues to function normally, but debugging might be more confusing due to the factors described earlier.

Note: The **Limit Optimizations for Easier Debugging** checkbox interacts with the **Generate Debug Information** checkbox (see the Generate Debug Information section on page 49).

Memory Model. The Memory Model drop-down list allows you to choose between the two memory models supported by the ZNEO C-Compiler, **Small** or **Large**. One fundamental difference between these models is that the small model can be implemented using only ZNEO CPU's internal Flash and RAM memory, but the large model requires the presence of external RAM. Using the small model also results in more compact code and often reduces the RAM requirements as well. However, the small model places constraints on the data space size (not on the code space size) of your application. Some applications might not be able to fit into the small model's data space size; the large model is provided to support such applications. See the Memory Models section on page 161 for full details of the memory models.

Project Settings—Listing Files Page. Figure 50 shows the Listing Files page.

Note: For Assembly Only projects, the Listing Files page is not available.



[ ] Generate C Listing Files (.lis)
[ ] With Include Files
[ ] Generate Assembly Source Code
[✓] Generate Assembly Listing Files (.lst)

Figure 50. (Listing Files Page of the Project Settings Dialog Box)

Generate C Listing Files (.lis). When selected, the **Generate C Listing Files (.lis)** checkbox tells the compiler to create a listing file for each C source code file in your project. All source lines are duplicated in this file, as are any errors encountered by the compiler.

With Include Files. When this checkbox is selected, the compiler duplicates the contents of all files included using the `#include` preprocessor directive in the compiler listing file. This can be helpful if there are errors in included files.

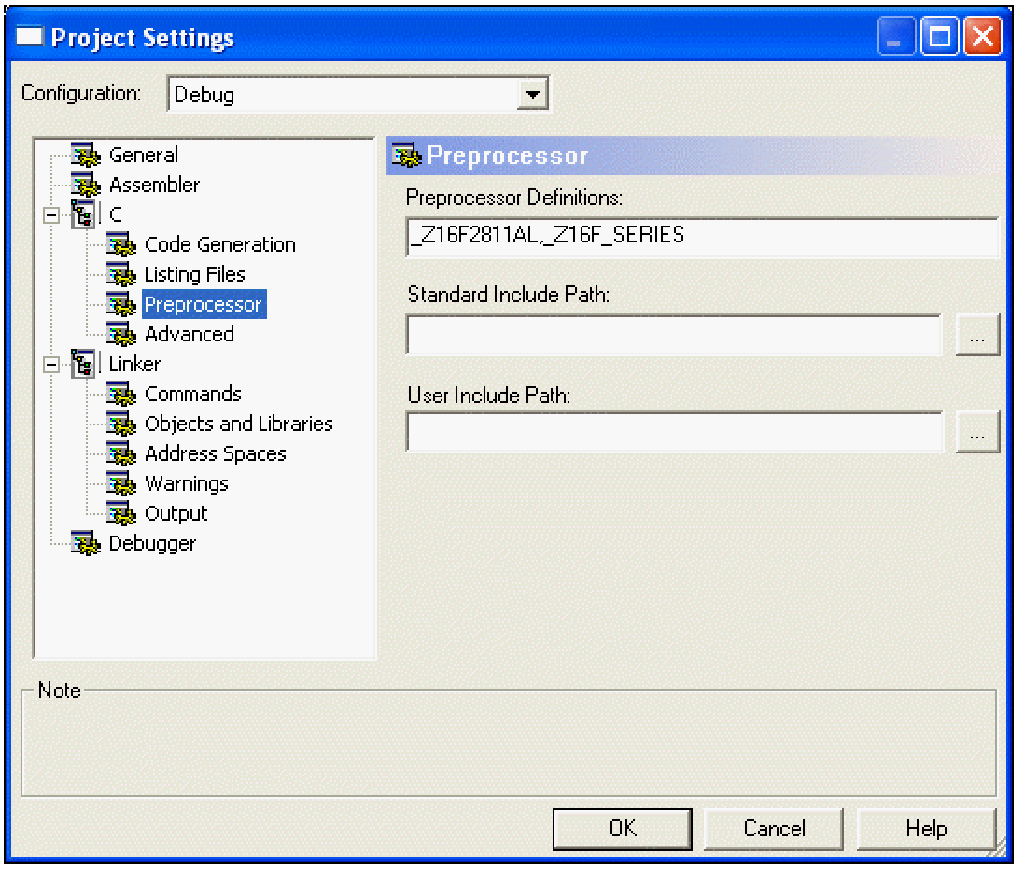
Generate Assembly Source Code. When this checkbox is selected, the compiler generates, for each C source code file, a corresponding file of assembler source code. In this file (which is a legal assembly file that the assembler will accept), the C source code (commented out) is interleaved with the generated assembly code and the compiler-generated assembly directives. This file is placed in the directory specified by the **Intermediate**

Files Directory checkbox in the **General** page. See the Intermediate Files Directory section on page 49.

Generate Assembly Listing Files (.lst). When this checkbox is selected, the compiler generates, for each C source code file, a corresponding assembly listing file. In this file, the C source code is displayed, interleaved with the generated assembly code and the compiler-generated assembly directives. This file also displays the hexadecimal addresses and op codes of the generated machine code. This file is placed in the directory specified by the Intermediate Files Directory field in the **General** page. See Intermediate Files Directory on page 49.

Project Settings—Preprocessor Page. Figure 51 shows the Preprocessor page.

Note: For Assembly Only projects, the Preprocessor page is not available.



Configuration: Debug

Preprocessor Definitions: _Z16F2811AL,_Z16F_SERIES

Standard Include Path:

User Include Path:

Figure 51. Preprocessor Page of the Project Settings Dialog Box

Preprocessor Definitions. The **Preprocessor Definitions** field is equivalent to placing `#define` preprocessor directives before any lines of code in your program. It is useful for conditionally compiling code. Do *not* put a space between the *symbol\name* and equal sign; however, multiple symbols can be defined and must be separated by commas.

Standard Include Path. The Standard Include Path field allows you to specify the series of paths for the compiler to use when searching for standard include files. Standard include files are those included with the `#include <`*file*`.h>` preprocessor directive. If more than one path is used, the paths are separated by semicolons (;). The compiler first checks the current directory, then the paths in the Standard Include Path field. The default standard includes are located in the following directories:

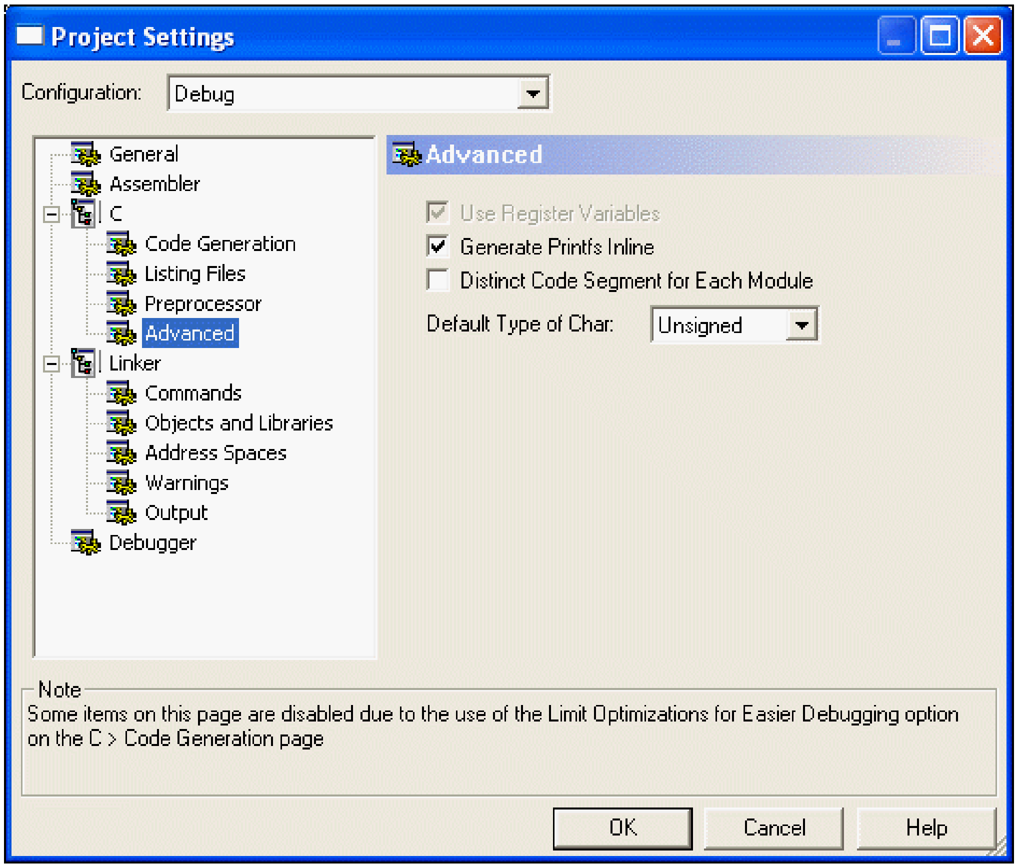
<*ZDS Installation Directory*>`\include\std`
<*ZDS Installation Directory*>`\include\zilog`

where <*ZDS Installation Directory*> is the directory in which Zilog Developer Studio was installed. By default, this would be `C:\Program Files\Zilog\ZDSII_ZNEO_<version>`, where <*version*> might be `4.11.0` or `5.0.0.`

User Include Path. The User Include Path field allows you to specify the series of paths for the compiler to use when searching for user include files. User include files are those included with the `#include` “`file.h`” in the compiler. If more than one path is used, the paths are separated by semicolons (;). The compiler first checks the current directory, then the paths in the User Include Path field.

Project Settings—Advanced Page. Figure 52 shows the Advanced page.

Note: For Assembly Only projects, the Advanced page is not available.



[✓] Use Register Variables
[✓] Generate Printfs Inline
[ ] Distinct Code Segment for Each Module
Default Type of Char: Unsigned

Note
Some items on this page are disabled due to the use of the Limit Optimizations for Easier Debugging option on the C > Code Generation page

Figure 52. Advanced Page of the Project Settings Dialog Box

Use Register Variables. Selecting this checkbox allows the compiler to allocate local variables in registers, rather than on the stack, when possible. This usually makes the resulting code smaller and faster and, therefore, the default is that this checkbox is enabled. However, in some applications, this checkbox might produce larger and slower code when a function contains a large number of local variables.

The effect of this checkbox on overall program size and speed can only be assessed globally across the entire application, which the compiler cannot do automatically. Usually the overall application size is smaller but there can be exceptions to that rule. For example, in an application that contains 50 functions, this checkbox might make 2 functions larger and the other 48 functions smaller. Also, if those two functions run slower with the checkbox enabled but the others run faster, then whether the overall program speed is improved or worsened depends on how much time the application spends in each function.

Because the effect of applying this option must be evaluated across an application as a whole, user experimentation is required to test this for an individual application. Only a small fraction of applications benefit from deselecting the **Use Register Variables** checkbox.

Notes: The **Use Register Variables** checkbox interacts with the **Limit Optimizations for Easier Debugging** checkbox on the C page (see the Limit Optimizations for Easier Debugging section on page 52). When the **Limit Optimizations for Easier Debugging** checkbox is selected, register variables are not used because they can cause confusion when debugging. The **Use Register Variables** checkbox is disabled (grayed out) when the **Limit Optimizations for Easier Debugging** checkbox is selected. If the **Limit Optimizations for Easier Debugging** checkbox is later deselected (even in a later ZDS II session), the **Use Register Variables** checkbox returns to the setting it had before the **Limit Optimizations for Easier Debugging** checkbox was selected.

Using register variables can complicate debugging in at least two ways. One way is that register variables are more likely to be optimized away by the compiler. If variables you want to observe while debugging are being optimized away, you can usually prevent this by any of the following actions:

- Select the **Limit Optimizations for Easier Debugging** checkbox (see Limit Optimizations for Easier Debugging on page 52)
- Deselect the **Use Register Variables** checkbox
- Rewrite your code so that the variables in question become global rather than local

The other way in which register variables can lead to confusing behavior when debugging is when the same register is used to store different variables or temporary results at different times in the execution of your code. Because the debugger is not always aware of these multiple uses, sometimes a value for a register variable might be shown in the Watch window that is not actually related to that variable at all.

Generate Printfs Inline. Normally, a call to `printf()` or `sprintf()` parses the format string at run time to generate the required output. When the **Generate Printfs Inline** checkbox is selected, the format string is parsed at compile time, and direct inline calls to the lower level helper functions are generated. This results in significantly smaller overall code size because the top-level routines to parse a format string are not linked into the project, and only those lower level routines that are actually used are linked in, rather than every routine that could be used by a call to `printf`. The code size of each routine that calls `printf()` or `sprintf()` is slightly larger than if the **Generate Printfs Inline** checkbox is deselected, but this is more than offset by the significant reduction in the size of library functions that are linked to your application.

To reduce overall code size by selecting this checkbox, the following conditions are necessary:

- All calls to `printf()` and `sprintf()` must use string literals, rather than `char*` variables, as parameters. For example, the following code allows the compiler to reduce the code size:

  ```
  sprintf ("Timer will be reset in %d seconds", reset_time);
  ```

  But code such as the following results in larger code:

  ```
  char * timerWarningMessage;
  ...
  sprintf (timerWarningMessage, reset_time);
  ```

- The functions `vprintf()` and `vsprintf()` cannot be used, even if the format string is a string literal.

If the **Generate Printfs Inline** checkbox is selected and these conditions are not met, the compiler warns you that the code size cannot be reduced. In this case, the compiler generates correct code, and the execution is significantly faster than with normal `printf` calls. However, there is a net increase in code size because the generated inline calls to lower level functions require more space with no compensating savings from removing the top-level functions.

In addition, an application that makes over 100 separate calls of `printf` or `sprintf` might result in larger code size with the **Generate Printfs Inline** checkbox selected because of the cumulative effect of all of the inline calls. The compiler cannot warn about this situation. If in doubt, simply compile the application both ways and compare the resulting code sizes.

The **Generate Printfs Inline** checkbox is selected by default.

Distinct Code Segment for Each Module. For most applications, the code segment for each module compiled by the ZNEO compiler is named CODE. Later, in the linker step of the build process, the linker gathers all these small CODE segments into a single large CODE segment and then places that segment in the appropriate address space, thus ensuring that all of the executable code is kept in a single contiguous block within a single address space. However, some users might need a more complex configuration in which particular code modules are put in different address spaces.

Such users can select the **Distinct Code Segment for Each Module** checkbox to accomplish this purpose. When this checkbox is selected, the code segment for every module receives a distinct name; for example, the code segment generated for the `myModule.c` module is given the name `myModule_TEXT`. You can then add linker directives to the linker command file to place selected modules in the appropriate address spaces. This checkbox is deselected by default.

An example of the use of this feature is to place most of the application's code in the usual EROM address space (see the Project Settings—Address Spaces Page section on page 68 for a discussion of the ZNEO address spaces) except for a particular module that is to be run from the RAM (16-bit addressable RAM) space. See the Special Case: Partial Download to RAM section on page 317 for an example of how to configure the linker command file for this type of application.

Note: It is the user's responsibility to configure the linker command file properly when the **Distinct Code Segment for Each Module** checkbox is selected.

Default Type of Char. The ANSI C Standard permits the compiler to regard `char` variables that are not otherwise qualified as either `signed` or `unsigned`, at the compiler's discretion; the compiler is only required to consistently apply the choice to all such variables. So in the following declarations:

```
signed char sc;
unsigned char uc;
char cc;
```

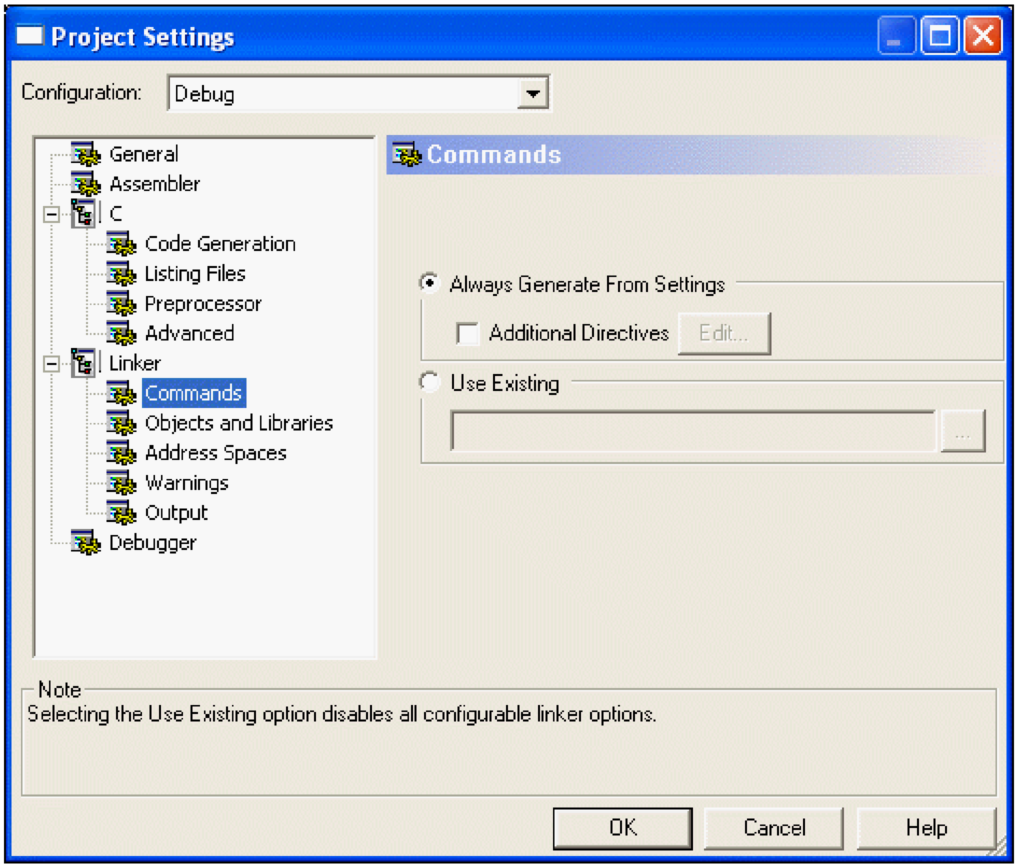
the signedness of `cc` is left to the compiler. The Default Type of Char drop-down menu allows you to make this decision. The selection, **Signed** or **Unsigned**, is applied to all `char` variables whose signedness is not explicitly declared. The default value for ZNEO is **Unsigned**.

Project Settings—Librarian Page. This page is available for Static Library projects only.

To configure the librarian, observe the following procedure:

1. Select **Settings** from the **Project** menu. The **Project Settings** dialog box is displayed.
2. Click the **Librarian** page.
3. Use the **Output File Name** field to specify where your static library file is saved.

Project Settings—Commands Page. Figure 56 shows the Commands page.



[● Always Generate From Settings]
☐ Additional Directives Edit...
○ Use Existing

Note
Selecting the Use Existing option disables all configurable linker options.

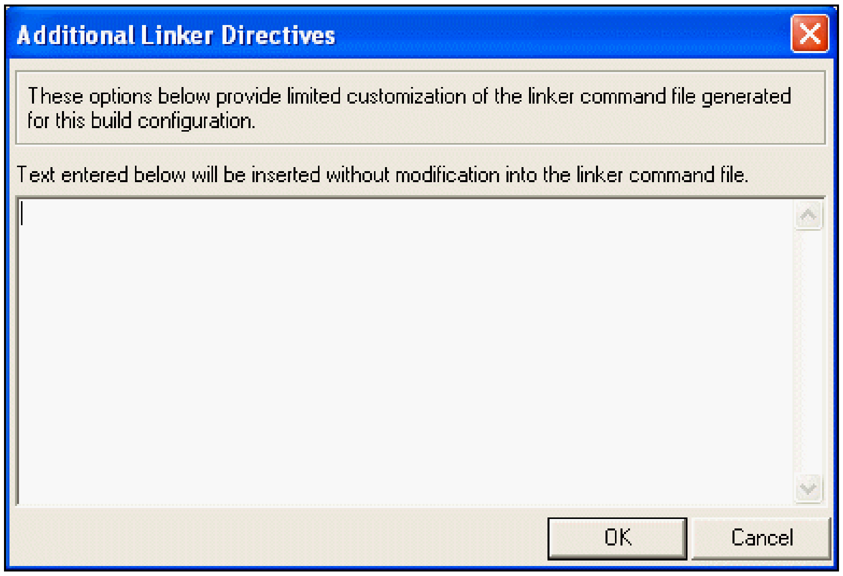
Figure 53. (Commands Page of the Project Settings Dialog Box

Always Generate from Settings. When this button is selected, the linker command file is generated afresh each time you build your project; the linker command file uses the project settings that are in effect at the time. This button is selected by default, which is the preferred setting for most users. Selecting this button means that all changes you make in your project, such as adding more files to the project or changing project settings, are automatically reflected in the linker command file that controls the final linking stage of the build. If you do not want the linker command file generated each time your project builds, select the **Use Existing** button (see the Use Existing section on page 63).

Note: Even though selecting **Always Generate from Settings** causes a new linker command file to be generated when you build your project, any directives that you have specified in the **Additional Linker Directives** dialog box are not erased or overridden.

Additional Directives. To specify additional linker directives that are to be added to those that the linker generates from your settings when the **Always Generate from Settings** button is selected, perform the following brief procedure.

1. Select the **Additional Directives** checkbox.
2. Click **Edit**. The **Additional Linker Directives** dialog box is displayed; see Figure 54.



These options below provide limited customization of the linker command file generated for this build configuration.

Text entered below will be inserted without modification into the linker command file.

Figure 54. Additional Linker Directives Dialog Box

3. Add new directives or edit existing directives.
4. Click **OK**.

You can use the **Additional Directives** checkbox if you must make some modifications or additions to the settings that are automatically generated from your project settings, but you still want all your project settings and newly added project files to take effect automatically on each new build.

You can add or edit your additional directives in the **Additional Linker Directives** dialog box. The manually inserted directives are always placed in the same place in your linker command file: after most of the automatically generated directives and just before the final directive that gives the name of the executable to be built and the modules to be included in the build. This position makes your manually inserted directives override any conflicting directives that occur earlier in the file, so it allows you to override particular directives that are autogenerated from the project settings. (The RANGE and ORDER linker directives are exceptions to this rule; they do not override earlier RANGE and ORDER directives but combine with them.) Use caution with this override capability because some of the autogenerated directives might interact with other directives and because there is no visual indication to remind you that some of your project settings might not be fully taking effect on later builds. If you need to create a complex linker command file, contact Zilog Technical Support for assistance. See Zilog Technical Support on page vii.

If you have selected the **Additional Directives** checkbox, your manually inserted directives are not erased when you build your project. They are retained and re-inserted into the same location in each newly created linker command file every time you build your project.

Note: In earlier releases of ZDS II, it was necessary to manually insert a number of directives if you had a C project and did not select the Standard C Start-up Module. This is no longer necessary. The directives required to support a C start-up module are now always added to the linker command file. The only time these directives are not added is if the project is an Assembly Only project.

Use Existing. Observe the following procedure if you do not want a new linker command file to be generated when you build your project:

1. Select the **Use Existing** button.
2. Click the **Browse** button (...). The **Select Linker Command File** dialog box is displayed; see Figure 55.

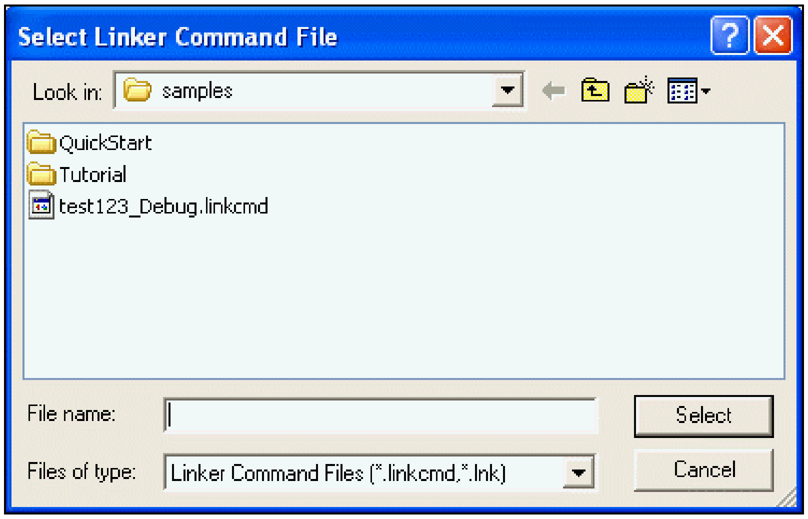


Figure 55. Select Linker Command File Dialog Box

3. Use the **Look In** drop-down menu to navigate to the linker command file that you want to use.

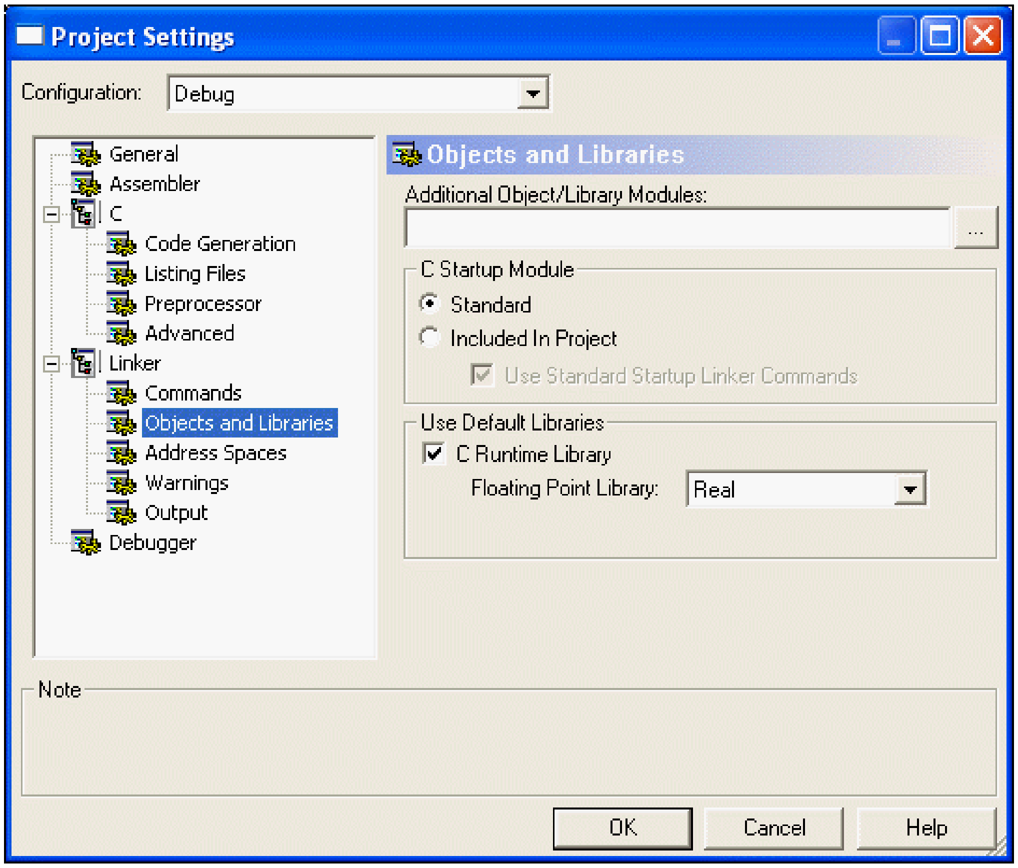
4. Click **Select**.

The **Use Existing** button is the alternative to the **Always Generate from Settings** button (see the Always Generate from Settings section on page 61). When this button is selected, a new linker command file is not generated when you build your project. Instead, the linker command file that you specify in this field is applied every time.

When the **Use Existing** button is selected, many project settings are grayed out, including all of the settings on the **Objects and Libraries** page, **Warnings** page, and **Output** page. These settings are disabled because when you have specified that an existing linker command file is to be used, those settings have no effect.

Note: When the **Use Existing** button is selected, some other changes that you make in your project such as adding new files to the project also do not automatically take effect. To add new files to the project, you must not only add them to the Project Workspace window (see the Project Workspace Window section on page 23), but you must also edit your linker command file to add the corresponding object modules to the list of linked modules at the end of the linker command file.

Project Settings—Objects and Libraries Page. Figure 56 shows the **Objects and Libraries** page.



Configuration: Debug

Additional Object/Library Modules:

C Startup Module
- (●) Standard
- ( ) Included In Project
- [✓] Use Standard Startup Linker Commands

Use Default Libraries
- [✓] C Runtime Library
- Floating Point Library: Real

Figure 56. Objects and Libraries Page of the Project Settings Dialog Box

Additional Object/Library Modules. Use the **Additional Object/Library Modules** field to list additional object files and modules that you want linked with your application. You do not need to list modules that are otherwise specified in your project, such as the object modules of your source code files that appear in the Project Workspace window, the C start-up module, and the Zilog default libraries listed in the **Objects and Libraries** page. Separate multiple module names with commas.

Note: Modules listed in this field are linked before the Zilog default libraries. Therefore, if there is a name conflict between symbols in one of these user-specified additional modules and in a Zilog default library, the user-specified module takes precedence and its version of the symbol is the one used in linking. You can take advantage of this to provide your own replacement for one or more functions (for example, C run-time library functions) by compiling the function and then including the object module name in this field. This is an alternative to including the source code for the revised function explicitly in your project, which would also override the function in the default run-time library.

C Start-up Module. The buttons and checkbox in this area (which are not available for Assembly Only projects) control which start-up module is linked to your application. All C programs require some initialization before the main function is called, which is typically done in a start-up module.

Standard. If the **Standard** button is selected, the precompiled start-up module shipped with ZDS II is used. This standard start-up module performs a minimum amount of initialization to prepare the run-time environment as required by the ANSI C Standard and also does some ZNEO-specific configuration such as interrupt vector table initialization. See the Language Extensions section on page 156 for details about the operations performed in the standard start-up module.

Some of these steps carried out in the standard start-up module might not be required for every application; therefore if code space is extremely tight, you might want to make some judicious modifications to the start-up code. The source code for the start-up module is located in the following file:

<*ZDS Installation Directory*>`\src\boot\common\startupX.asm`

Here, <*ZDS Installation Directory*> is the directory in which Zilog Developer Studio was installed. By default, this is `C:\Program Files\Zilog\ZDSII_ZNEO_<version>`, where <*version*> might be `4.11.0` or `5.0.0.` The `X` in `startupX.asm` is `s` for the small model or `l` for the large model.

Included in Project. If the **Included in Project** button is selected, then the standard start-up module is not linked to your application. In this case, you are responsible for including suitable start-up code, either by including the source code in the Project Workspace window or by including a precompiled object module in the **Additional Object/Library Modules** field. If you modify the standard start-up module to tailor it to your project, you must select the **Included in Project** button for your changes to take effect.

Use Standard Startup Linker Commands. If you select this checkbox, the same linker commands that support the standard start-up module are inserted into your linker command file, even though you have chosen to include your own, nonstandard start-up module in the project. This option is usually helpful in getting your project properly configured and initialized because all C start-up modules have to do most of the same tasks. Formerly, these linker commands had to be inserted manually when you were not using the standard startup.

The standard startup commands define a number of linker symbols that are used in the standard start-up module for initializing the C run-time environment. You do not have to refer to those symbols in your own start-up module, but many users will find it useful to do so, especially since user-customized start-up modules are often derived from modifying the standard start-up module. There are also a few linker commands (such as `CHANGE`, `COPY`, `ORDER`, and `GROUP`) that are used to configure your memory map. See the Linker Commands section on page 260 for a description of these commands.

This option is only available when the **Included in Project** button has been selected. The default for newly created projects is that this checkbox, if available, is selected.

Use Default Libraries. These controls determine whether the available default libraries that are shipped with Zilog Developer Studio II are to be linked with your application. For ZNEO, there is essentially one available library, the C run-time library. The subset of the run-time library dedicated to floating-point operations also has a separate control to allow for special handling, as explained in the Floating Point Library section on page 67.

Use C Runtime Library. The C run-time library included with ZDS II provides selected functions and macros from the Standard C Library. Zilog's version of the C run-time library supports a subset of the Standard Library adapted for embedded applications, as described more fully in the Using the ANSI C-Compiler chapter on page 155. If your project makes any calls to standard library functions, you must select the **Use C Runtime Library** checkbox unless you prefer to provide your own code for all library functions that you call. As noted in the Additional Object/Library Modules section on page 65, you can also set up your application to call a mixture of Zilog-provided functions and your own customized library functions. To do so, select the **Use C Runtime Library** checkbox. Calls to standard library functions will then call the functions in the Zilog default library except when your own customized versions exist.

Zilog's version of the C run-time library is organized with a separate module for each function or, in a few cases, for a few closely related functions. Therefore, the linker links only those functions that you actually call in your code. This means that there is no unnecessary code size penalty when you select the **Use C Runtime Library** checkbox; only functions you call in your application are linked into your application.

Floating Point Library. The **Floating Point Library** drop-down menu allows you to choose which version of the subset of the C run-time library that deals with the floating-point operations will be linked to your application:

- Real

  If you select **Real**, the true floating-point functions are linked in, and you can perform any floating-point operations you want in your code.

- Dummy

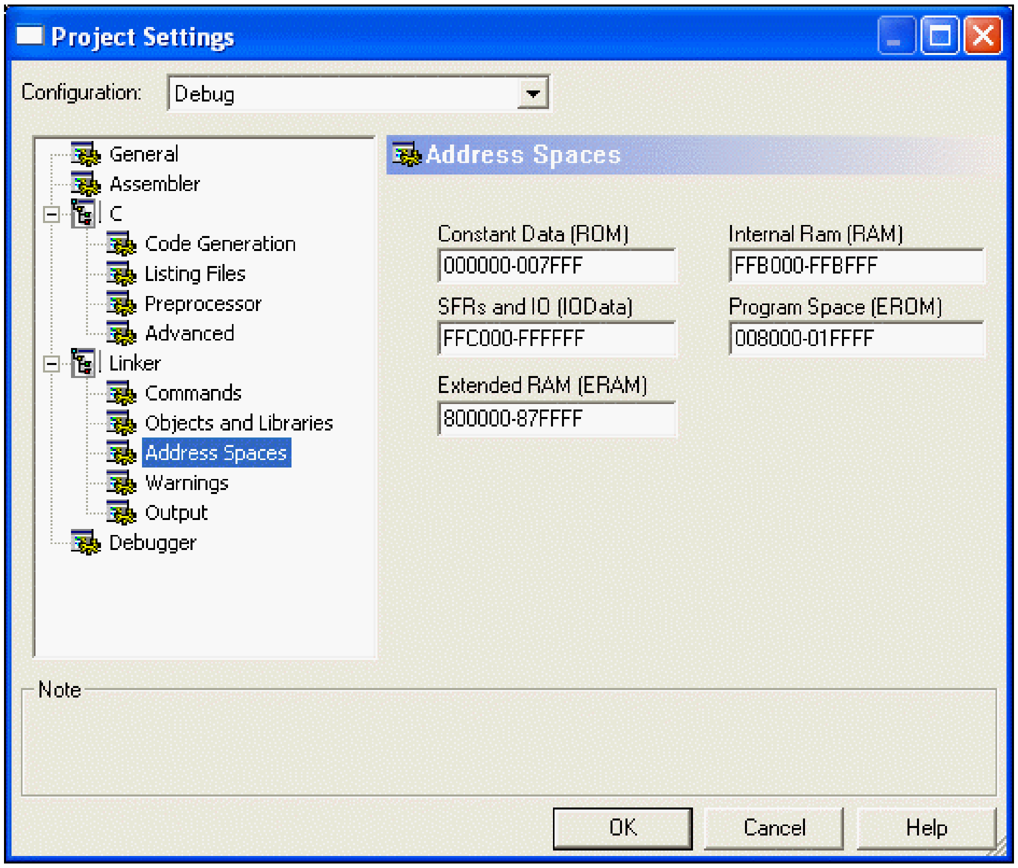
  If you select **Dummy**, your application is linked with alternate versions that are stubbed out and do not actually carry out any floating-point operations. This dummy floating-point library has been developed to reduce code bloat caused by including calls to `printf()` and related functions such as `sprintf()`. Those functions in turn make calls to floating-point functions for help with formatting floating-point expressions, but those calls are unnecessary unless you actually need to format floating-point values. For most users, this problem has now been resolved by the **Generate Printfs Inline** checkbox (see the Generate Printfs Inline section on page 58 for a full discussion). You only need to select the dummy floating-point library if you have to disable the **Generate Printfs Inline** checkbox and your application uses no floating-point operations. In that case, selecting **Dummy** keeps your code size from bloating unnecessarily.

- None

  If you can select **None**, no floating-point functions are linked to your application at all. This can be a way of ensuring that your code does not inadvertently make any floating-point calls, because, if it does and this option is selected, you receive a warning message about an undefined symbol.

Note: None of the libraries mentioned here are available for Assembly Only projects.

Project Settings—Address Spaces Page. Figure 57 shows the Address Spaces page.



Constant Data (ROM): 000000-007FFF
Internal Ram (RAM): FFB000-FFBFFF
SFRs and IO (IOData): FFC000-FFFFFF
Program Space (EROM): 008000-01FFFF
Extended RAM (ERAM): 800000-87FFFF

Figure 57. Address Spaces Page of the Project Settings Dialog Box

The memory range fields in the Address Spaces page allow you to inform ZDS II about the amount and location of memory and I/O on your target system. The appropriate settings for these fields depend on the CPU selection and target system design. ZDS II uses the memory range settings to let you know when your code or data has grown beyond your system's capability. The system also uses memory ranges to automatically locate your code or data.

See the Programmer's Model of ZNEO Memory section on page 303 for details about address range functions. The ZDS II address ranges are:

Constant data (ROM). This range is typically `000000-001FFF` for devices with 32 KB of internal Flash, `000000-003FFF` for devices with 64 KB of internal Flash, and `000000-007FFF` for devices with 128 KB of internal Flash. The lower boundary must be `00_000H`. The upper boundary can be lower than `007FFF`, but no higher.

Program space (EROM). This range is typically `002000-007FFF` for devices with 32 KB of internal Flash, `004000-00FFFF` for devices with 64 KB of internal Flash, or `008000-01FFFF` for devices with 128 KB of internal Flash. Specify a larger range only if the target system provides external nonvolatile memory.

Note: To use any external memory provided on the target system, you must configure the memory's chip select in the Configure Target dialog box. See the Project Settings—Debugger Page section on page 74.

Extended RAM (ERAM). Specify an ERAM range only if the target system provides external random access memory below `FF8000`. The ERAM field does not accept a starting address below `800000`.

Internal RAM (RAM). This range is typically `FFB700-FFBFFF` for devices with 2 KB of internal RAM or `FFB000-FFBFFF` for devices with 4 KB of internal RAM. Despite its name, this range can be expanded up to `FF8000-FFBFFF` if the target system provides external random access memory to fill out this address range. This field does not allow a high RAM address boundary above `FFBFFF`.

Special Function Registers and IO (IODATA). Typically `FFC000-FFFFFF`. The microcontroller reserves addresses `FFE000` and above for its special function registers, on-chip peripherals, and I/O ports. The ZDS II GUI expects addresses `FFC000` to `FFDFFF` to be used for external I/O (if any) on the target system.

Address range settings must not overlap. The following example presents the syntax used in the address range fields:

<*low address*> – <*high address*> [,<*low address*> – <*high address*>] ...

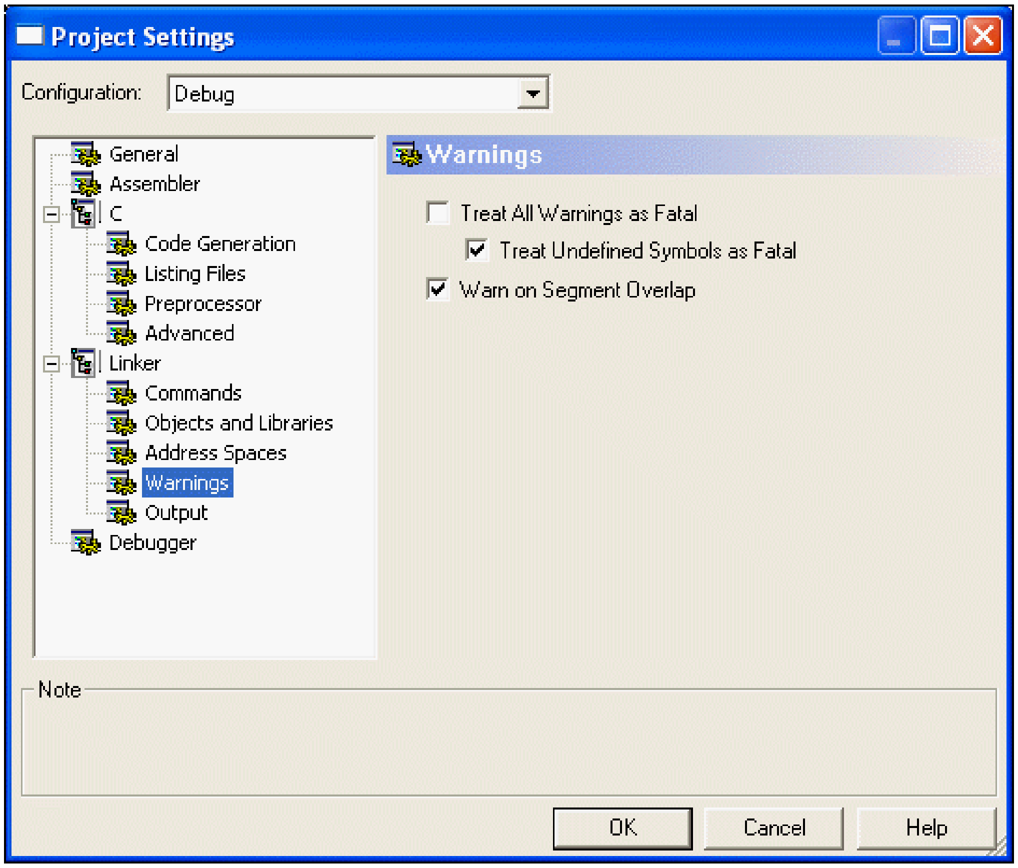
where <*low address*> is the hexadecimal lower boundary of a range and <*high address*> is the hexadecimal higher boundary of the range. The following are legal memory ranges:

```
0000-7fff
ffb000-ffbfff
008000-01ffff,050000-07ffff
```

The last example line shows how a comma is used to define *holes* in a memory range for the linker. The linker does not place any code or data outside of the ranges specified here. If your code or data cannot be placed within the ranges, a range error is generated.

The C-Compiler does not support gaps (holes) within the ERAM or RAM ranges.

Project Settings—Warnings Page. Figure 58 shows the Warnings page.



Configuration: Debug

Warnings

[ ] Treat All Warnings as Fatal

[✓] Treat Undefined Symbols as Fatal

[✓] Warn on Segment Overlap

Figure 58. Warnings Page of the Project Settings Dialog Box

Treat All Warnings as Fatal. When selected, this checkbox causes the linker to treat all warning messages as fatal errors. When the checkbox is selected, the linker does not generate output file(s) if there are any warnings while linking. By default, this checkbox is deselected, and the linker proceeds with generating output files even if there are warnings.

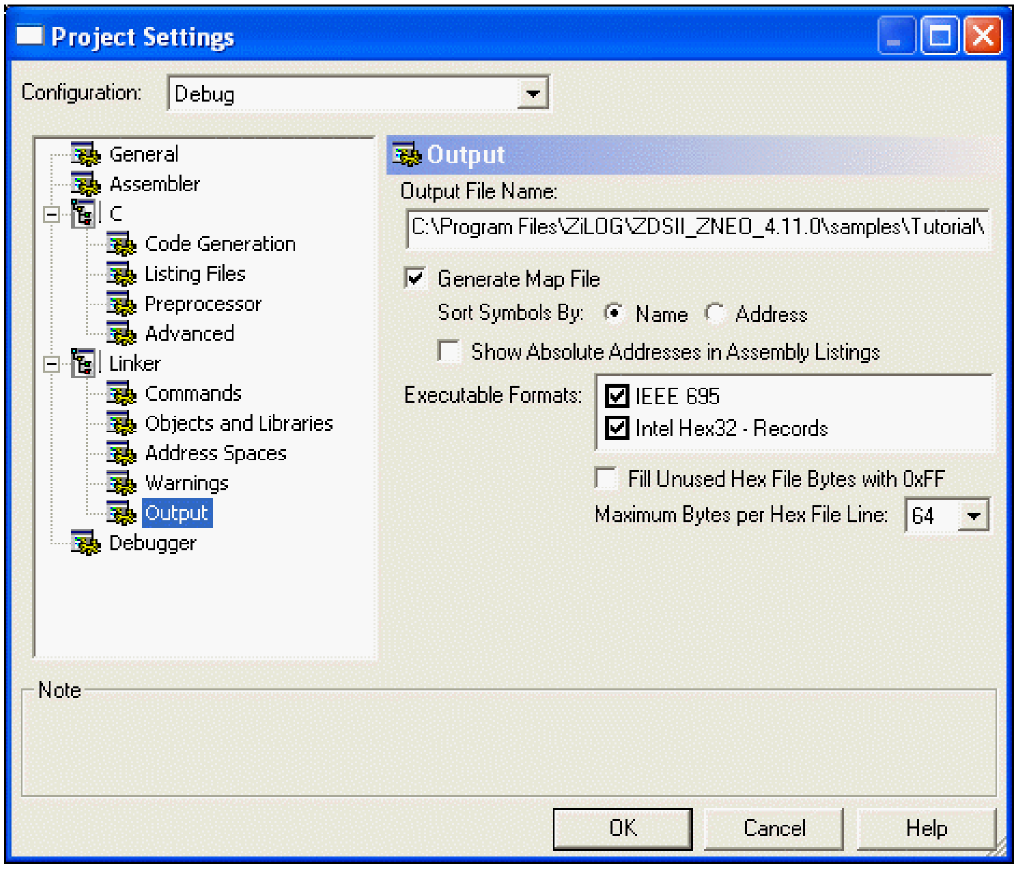
Note: Selecting this checkbox displays any warning as an error, regardless of the state of the **Show Warnings** checkbox in the **General** page (see the Show Warnings section on page 48).

Treat Undefined Symbols as Fatal. When selected, this checkbox causes the linker to treat *undefined external symbol* warnings as fatal errors. If this checkbox is selected, the linker quits generating output files and terminates with an error message immediately if the linker cannot resolve any undefined symbol. By default, this checkbox is selected because a completely valid executable cannot be built when the program contains references to undefined external symbols. If this checkbox is deselected, the linker proceeds with generating output files even if there are undefined symbols.

Note: Selecting this checkbox displays any **undefined external symbol** warning as an error, regardless of the state of the **Show Warnings** checkbox in the **General** page (see the Show Warnings section on page 48).

Warn on Segment Overlap. This checkbox enables or disables warnings when overlap occurs while binding segments. By default, the checkbox is selected, which is the recommended setting for ZNEO. For some Zilog processors, benign segment overlaps can occur, but, for the ZNEO, an overlap condition usually indicates an error in project configuration that must be corrected. These errors in ZNEO can be caused either by user assembly code that erroneously assigns two or more segments to overlapping address ranges or by user code defining the same interrupt vector segment in two or more places.

Project Settings—Output Page. Figure 59 shows the Output page.



Output

Output File Name: C:\Program Files\ZiLOG\ZDSII_ZNEO_4.11.0\samples\Tutorial\

[✓] Generate Map File

Sort Symbols By: [● Name] ○ Address

[ ] Show Absolute Addresses in Assembly Listings

Executable Formats: [✓] IEEE 695 [✓] Intel Hex32 - Records

[ ] Fill Unused Hex File Bytes with 0xFF

Maximum Bytes per Hex File Line: 64

Figure 59. Output Page of the Project Settings Dialog Box

Output File Name. You can change the name (including the full path name) of your executable in the Output File Name field. After your program is linked, the appropriate extension is added.

Generate Map File. This checkbox determines whether the linker generates a link map file each time it is run. The link map file is named with your project's name with the `.map` extension and is placed in the same directory as the executable output file. See the MAP command on page 267 and How much memory is my program using? on page 295. Inside the map file, symbols are listed in the order specified by the Sort Symbols By area (see Sort Symbols By on page 74).

Note: The link map is an important place to look for memory restriction or layout problems.

Sort Symbols By. You can choose whether to have symbols in the link map file sorted by name or address.

Show Absolute Addresses in Assembly Listings. When this checkbox is selected, all assembly listing files that are generated in your build are adjusted to show the absolute addresses of the assembly code statements. If this checkbox is deselected, assembly listing files use relative addresses beginning at zero.

For this option to be applied to listing files generated from assembly source files, the **Generate Assembly Listing Files (.lst)** checkbox in the **Assembler** page of the **Project Settings** dialog box must be selected.

For this option to be applied to listing files generated from C source files, both the **Generate Assembly Source Code** and **Generate Assembly Listing Files (.lst)** checkboxes in the Listing Files page of the **Project Settings** dialog box must be selected.

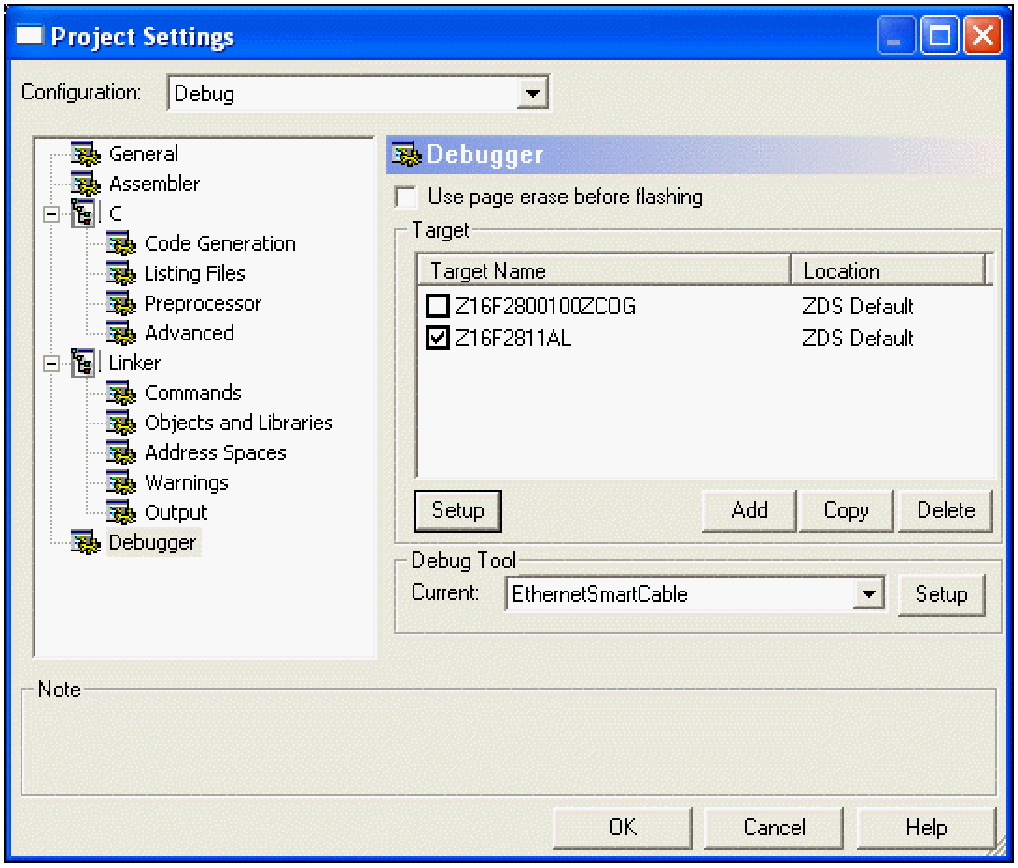
Executable Formats. These checkboxes determine which object format is used when the linker generates an executable file. The linker supports the following formats: IEEE 695 (`.lod`) and Intel Hex32 Records (`.hex`). IEEE 695 is the default format for debugging. Selecting Intel Hex32 - Records generates a hex file in the Intel Hex32 format, which is a backward-compatible superset of the Intel Hex16 format. You can also select both checkboxes, which produces executable files in both formats.

Fill Unused Hex File Bytes with 0xFF. This checkbox is available only when the Intel Hex32 Records executable format is selected. When the **Fill Unused Hex File Bytes with 0xFF** checkbox is selected, all unused bytes of the hex file are filled with the value `0xFF`. This option is sometimes required so that when interoperating with other tools that set otherwise uninitialized bytes to 0xFF, the hex file checksum calculated in ZDS II will match that in the other tools.

Note: Use caution when selecting this option. The resulting hex file begins at the first hex address (`0x0000`) and ends at the last page address that the program requires. This significantly increases the programming time when using the resulting output hex file. The hex file might try to fill nonexistent external memory locations with `0xFF`.

Maximum Bytes per Hex File Line. This drop-down menu sets the maximum length of a hex file record. This option is provided for compatibility with third-party or other tools that might have restrictions on the length of hex file records. This option is available only when the Intel Hex32 Records executable format is selected.

Project Settings—Debugger Page. In the **Project Settings** dialog box, select the Debugger page; see Figure 60.



[ ] Use page erase before flashing

Target Name	Location
[ ] Z16F2800100ZCOG	ZDS Default
[✓] Z16F2811AL	ZDS Default

Debug Tool Current: EthernetSmartCable

Figure 60. Debugger Page of the Project Settings Dialog Box

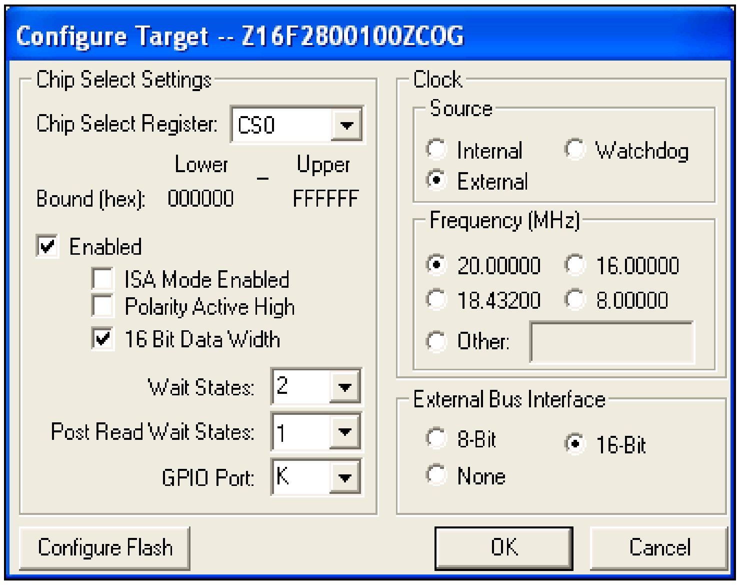
The source-level debugger is a program that allows you to find problems in your code at the C or assembly level. The Windows interface is quick and easy to use. You can also write batch files to automate debugger tasks.

Your understanding of the debugger design can improve your productivity because it affects your view of how things work. The debugger requires target and debug tool settings that correspond to the physical hardware being used during the debug session. A target is a logical representation of a target board. A debug tool represents debug communication hardware such as the USB Smart Cable or an emulator. A simulator is a software debug tool that does not require the existence of physical hardware. Currently, the debugger supports debug tools for the ZNEO simulator and the USB Smart Cable.

Use Page Erase Before Flashing. Select the **Use Page Erase Before Flashing** checkbox to configure the internal Flash memory of the target hardware to be page-erased. If this checkbox is not selected, the internal Flash is configured to be mass-erased.

Target. Select the appropriate target from the **Target** list box.

Setup. Click **Setup** in the Target area to display the **Configure Target** dialog box; see Figure 61.



Configure Target -- Z16F2800100ZCOG

Chip Select Settings

Chip Select Register: CS0

Bound (hex): Lower 000000 – Upper FFFFFF

[✓] Enabled

[ ] ISA Mode Enabled

[ ] Polarity Active High

[✓] 16 Bit Data Width

Wait States: 2

Post Read Wait States: 1

GPIO Port: K

Clock

Source: Internal, Watchdog, [✓ External]

Frequency (MHz): [✓ 20.00000], 16.00000, 18.43200, 8.00000, Other:

External Bus Interface: 8-Bit, [✓ 16-Bit], None

Figure 61. Configure Target Dialog Box

Note: The options displayed in the **Configure Target** dialog box depend on the CPU you selected in the **New Project** dialog box (see the New Project section on page 32) or the **General** page of the **Project Settings** dialog box (see the Project Settings—General Page section on page 47). Chip select and external bus interface settings are only available for CPUs that support an external bus.

---

1. Select an 8-bit, 16-bit, or no external bus interface. Selecting an external bus interface is appropriate only for target designs that use an external bus.
2. If an external bus interface is selected, do the following steps for each chip select that is used by the target system for external memory or I/O. The settings appropriate for each chip select depend on the target system design.
   a. Choose the chip select register (CS0–CS5) from the **Chip Select Registers** drop-down menu.
   b. Select the **Enabled** checkbox to enable the chip select. Do not enable chip selects that the target does not use.

   c. To use ISA-compatible mode, select the **ISA Mode Enabled** checkbox.
   d. To use Active High polarity, select the **Polarity Active High** checkbox.
   e. To use 16-bit data width, select the **16 Bit Data Width** checkbox.
   f. Select the number of wait states from the **Wait States** drop-down menu.
   g. Select the number of post-read wait states from the **Post Read Wait States** drop-down menu.
   h. Select the appropriate GPIO port from the **GPIO Port** drop-down menu. This list box is only available if the chip select is an alternate function on more than one GPIO port.

3. Select the internal, watchdog, or external clock source in the Source area.
4. Select the appropriate clock frequency in the Clock Frequency (MHz) area or enter the clock frequency in the **Other** field. For the emulator, this frequency must match the clock oscillator on Y4. For the development kit, this frequency must match the clock oscillator on Y1. The emulator clock cannot be supplied from the target application board.

Note: The Clock Frequency value is used even when the Simulator is selected as the Debug Tool. The frequency is used when converting clock cycles to elapsed times in seconds, which can be viewed in the Debug Clock window when running the simulator.

5. Click **Configure Flash**. The **Target Flash Settings** dialog box is displayed; see Figure 62.

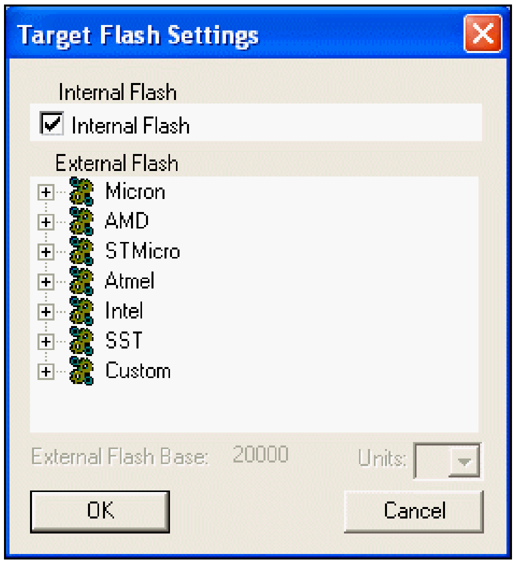


Figure 62. Target Flash Settings Dialog Box

a. Select the **Internal Flash** checkbox if you want to use internal Flash. The internal Flash memory configuration is defined in the `CpuFlashDevice.xml` file. The device is the currently selected microcontroller or microprocessor.

b. If you want to use external Flash, select which Flash devices you want to program. The Flash devices are defined in the `FlashDevice.xml` file.

c. The device is the current external Flash device's memory arrangement. The external Flash device options are predefined Flash memory arrangements for specific Flash devices such as the Micron MT28F008B3. The Flash Loader uses the external Flash device option arrangements as a guide for erasing and loading data to the appropriate blocks of Flash memory.

d. In the External Flash Base field, enter the location in which you want external Flash to start.

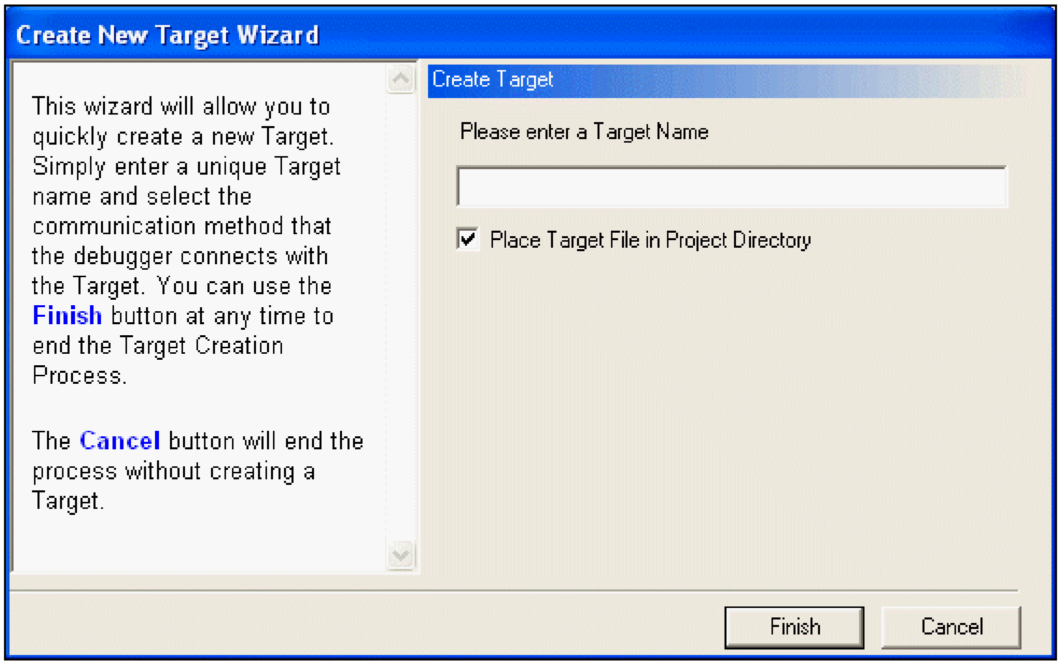
e. In the **Units** drop-down menu, select the number of Flash devices present.

For example, if you have two devices stacked on top of each other, select **2** in the Units list box.

f. Click **OK** to return to the **Configure Target** dialog box.

6. Click **OK**.

Add. Click **Add** to display the **Create New Target Wizard** dialog box; see Figure 63.



This wizard will allow you to quickly create a new Target. Simply enter a unique Target name and select the communication method that the debugger connects with the Target. You can use the **Finish** button at any time to end the Target Creation Process.

The **Cancel** button will end the process without creating a Target.

Please enter a Target Name

☑ Place Target File in Project Directory

Figure 63. Create New Target Wizard Dialog Box

Type a unique target name in the field, select the **Place Target File in Project Directory** checkbox if you want your new target file to be saved in the same directory as the currently active project, and click **Finish**.

Copy. Click **Copy** to display the **Target Copy or Move** dialog box; see Figure 64.

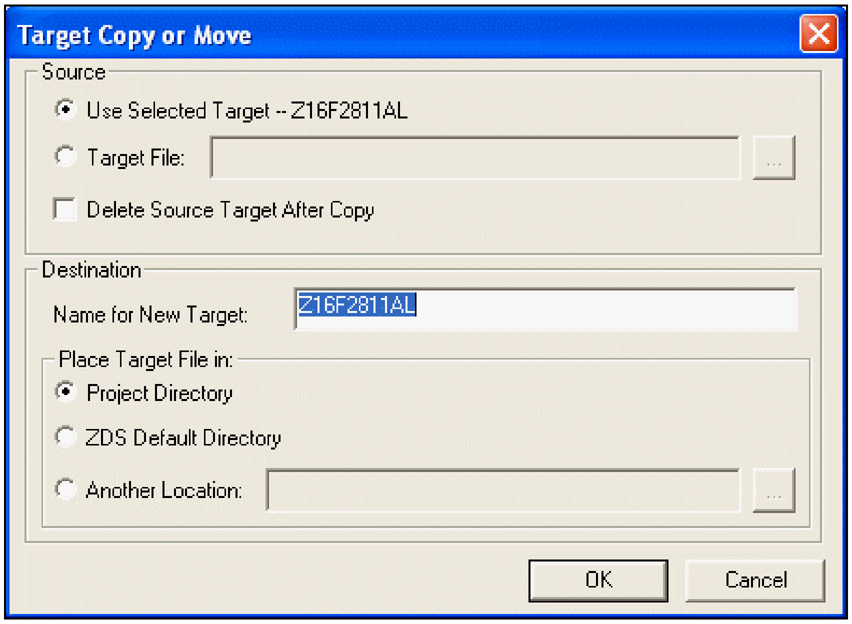


Figure 64. Target Copy or Move Dialog Box

1. Select the **Use Selected Target** button if you want to use the target listed to the right of this button description or select the **Target File** button to use the **Browse** button (...) to navigate to an existing target file.

   If you select the **Use Selected Target** button, enter the name of the new target in the **Name for New Target** field.

2. Select the **Delete Source Target After Copy** checkbox if you do not want to keep the original target.

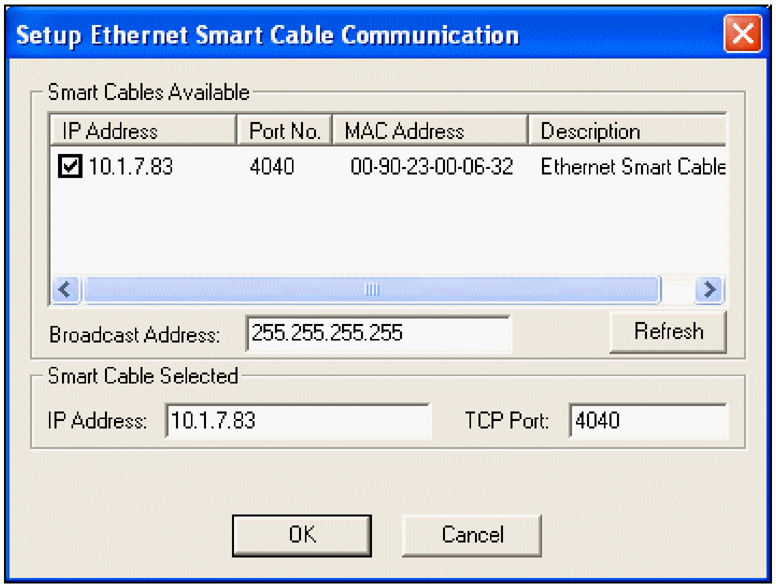
3. In the **Place Target File In** area, select the location in which you want the new target file saved, whether in the project directory, the ZDS default directory, or another location.

4. Click **OK**.

Delete. Click **Delete** to remove the currently-highlighted target. The following message is displayed: “`Delete` *target_name* `Target?`”. Click **Yes** to delete the target or **No** to cancel the command.

Debug Tool. Select the appropriate debug tool in the **Current** drop-down menu.

- If you select **EthernetSmartCable** and click **Setup** in the **Debug Tool** area, the **Setup Ethernet Smart Cable Communication** dialog box is displayed; see Figure 65.



IP Address	Port No.	MAC Address	Description
☑ 10.1.7.83	4040	00-90-23-00-06-32	Ethernet Smart Cable

Broadcast Address: 255.255.255.255

IP Address: 10.1.7.83 TCP Port: 4040

Figure 65. Setup Ethernet Smart Cable Communication Dialog Box

Note: If a Windows Security Alert is displayed with the following message: “`Do you want to keep blocking this program?`”, click **Unblock.**

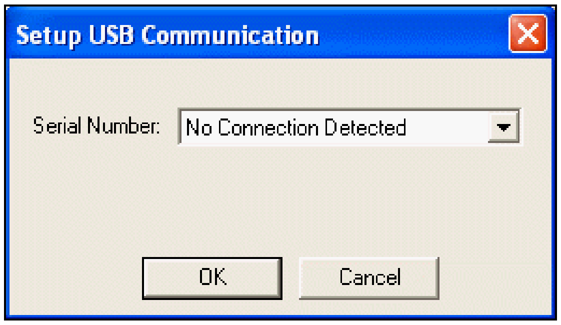
a. Click **Refresh** to search the network and update the list of available Ethernet Smart Cables. The number in the **Broadcast Address** field is the destination address to which ZDS II sends the scan message to determine which Ethernet Smart Cables are accessible. The default value of `255.255.255.255` can be used if the Ethernet Smart Cable is connected to your local network. Other values such as `192.168.1.255` or `192.168.1.50` can be used to direct or focus the search. ZDS II uses the default broadcast address if the **Broadcast Address** field is empty.

b. Select an Ethernet Smart Cable from the list of available Ethernet Smart Cables by checking the box next to the Smart Cable you want to use. Alternately, select the Ethernet Smart Cable by entering a known Ethernet Smart Cable IP address in the **IP Address** field.

c. Enter the port number in the **TCP Port** field.

d. Click **OK**.

- If you select **USBSmartCable** and click **Setup** in the **Debug Tool** area, the **Setup USB Communication** dialog box is displayed; see Figure 66.



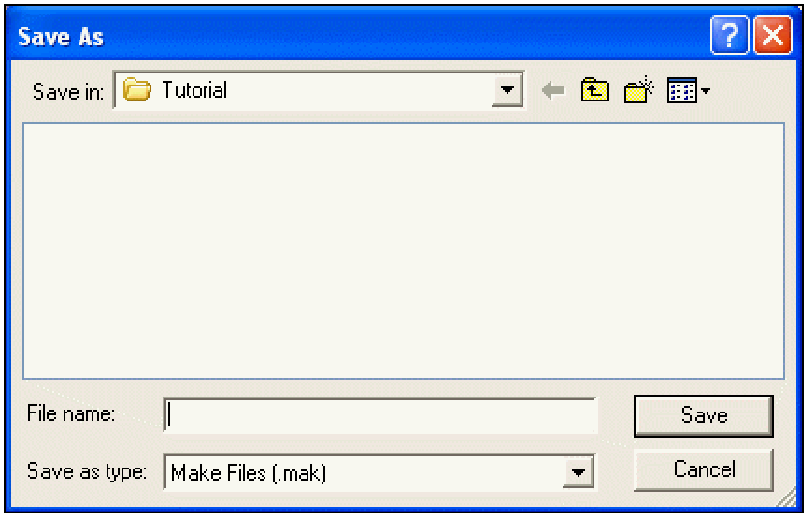
Serial Number: No Connection Detected

Figure 66. Setup USB Communication Dialog Box

a. Use the **Serial Number** drop-down menu to select the appropriate serial number.
b. Click **OK**.

Export Makefile. Export Makefile exports a buildable project in an external make file format via the following procedure.

1. From the **Project** menu, select **Export Makefile**. The **Save As** dialog box is displayed; see Figure 67.



Save in: Tutorial

File name:

Save as type: Make Files (.mak)

Figure 67. Save As Dialog Box

2. Use the **Save In** drop-down menu to navigate to the directory in which you want to save your project. The default location is in your project directory.
3. Enter the makefile name in the **File Name** field and click **Save**. The project is now available as an external make file.

Note: You do not have to enter the `.mak` extension; it is added automatically.

Build Menu

With the Build menu, you can build individual files as well as your project. You can also use this menu to select or add configurations for your project.

The Build menu contains a number of elements, each of which is described in this section.

Compile. Compile or assemble the active file in the Edit window.

Build. Compiles and/or assembles any files that have changed since the last build and then links the project.

Rebuild All. Rebuild *all* of the files in your project. This option also links the project.

Stop Build. Stop a build in progress.

Clean. Remove intermediate build files.

Update All Dependencies. Update your source file dependencies.

Set Active Configuration

You can use the Select Configuration dialog box to select the active build configuration you want:

1. From the **Build** menu, select **Set Active Configuration** to display the **Select Configuration** dialog box, shown in Figure 68.

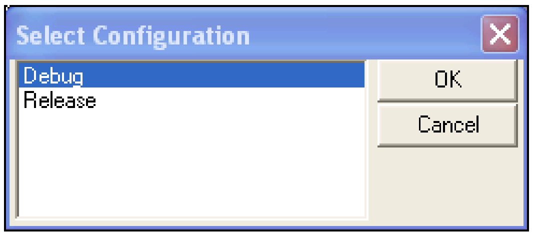


Figure 68. Select Configuration Dialog Box

2. Highlight the configuration that you want to use and click **OK**.

There are two standard configuration build configurations:

Debug. This configuration contains all of the project settings for running the project in Debug mode.

Release. This configuration contains all of the project settings for creating a Release version of the project.

For each project, you can modify the settings, or you can create your own configurations. These configurations allow you to easily switch between project setting types without having to remember all of the setting changes that must be made for each type of build that might be necessary during the creation of a project. All changes to project settings are stored in the current configuration setting.

Note: To add your own configuration(s), see the Manage Configurations section on page 84.

Use one of the following methods to activate a build configuration:

- Use the **Select Configuration** dialog box. See the Set Active Configuration section on page 83.
- Use the Build toolbar. See the Select Build Configuration List Box section on page 18.
- Use the **Project Settings** dialog box to modify build configuration settings. See the Settings section on page 46
.

Manage Configurations

For your specific needs, you can add different configurations for your projects. To add a customized configuration, perform the following brief procedure.

1. From the **Build** menu in ZDS II, select **Manage Configurations**. The **Manage Configurations** dialog box is displayed; see Figure 69.

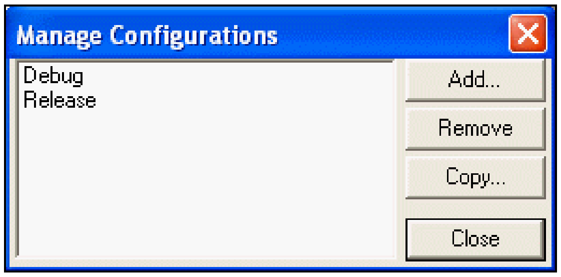
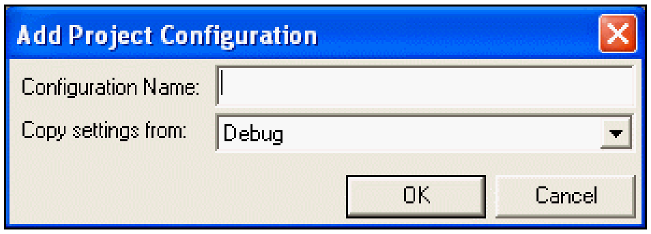


Figure 69. Manage Configurations Dialog Box

2. Click the **Add** button. The **Add Project Configuration** dialog box appears, as shown in Figure 70.



Configuration Name:

Copy settings from: Debug

Figure 70. Add Project Configuration Dialog Box

3. In the **Configuration Name** field, enter the name of the new configuration.
4. Select a similar configuration from the **Copy Settings From** drop-down menu.
5. Click **OK**. Your new configuration is displayed in the configurations list in the **Manage Configurations** dialog box.
6. Click **Close**. The new configuration is the current configuration as shown in the **Select Build Configuration** drop-down menu on the Build toolbar.
7. Now that you have created a blank template, you are ready to select the settings for this new configuration. From the **Project** menu, select **Settings**. The **Project Settings** dialog box is displayed.
8. Select the settings for the new configuration and click **OK**.
9. From the **File** menu, select **Save All**.

Debug Menu

Use the Debug menu to access a number of functions in the ZDS II debugger. Each of these functions is described in this section. For more information about the Debugger, see the chapter titled Using the Debugger on page 327.

Connect to Target. The Connect to Target command starts a debug session and initializes the communication to the target hardware. This command does not download the software or reset to main. Use this button to access target registers, memory, and so on, without loading new code or to avoid overwriting the target's code with the same code. This command is not enabled when the target is the simulator.

Download Code. The Download Code command downloads the executable file for the currently open project to the target for debugging. The command also initializes the communication to the target hardware if it has not been done yet. Use this command anytime during a debug session. This command is not enabled when the target is the simulator.

Note: When using the Download Code command, the current code on the target will be overwritten.

Verify Download. Select **Verify Download** from the **Debug** menu to determine download correctness by comparing executable file contents to target memory.

Stop Debugging. Select **Stop Debugging** from the **Debug** menu to end the current debug session. To stop program execution, select the Break command.

Reset. Select **Reset** from the **Debug** menu to reset the program counter to the beginning of the program. If not in Debug mode, a debug session is started. By default and if possible, the Reset command resets the program counter to **Symbol 'main'**. If you deselect the Reset to **Symbol 'main' (Where Applicable)** checkbox on the **Debugger** tab of the **Options** dialog box (see page 102), the program counter resets to the first line of the program.

Go. Select **Go** from the **Debug** menu to execute project code from the current program counter. If not in Debug mode when the command is selected, a debug session is started.

Run to Cursor. Select **Run to Cursor** from the **Debug** menu to execute the program code from the current program counter to the line containing the cursor in the active file or the Disassembly window. The cursor must be placed on a valid code line (a C source line with a blue dot displayed in the gutter or any instruction line in the Disassembly window).

Break. Select **Break** from the **Debug** menu to stop program execution at the current program counter.

Step Into. Select **Step Into** from the **Debug** menu to execute one statement or instruction from the current program counter, following execution into function calls. When complete, the program counter resides at the next program statement or instruction unless a function was entered, in which case the program counter resides at the first statement or instruction in the function.

Step Over. Select **Step Over** from the **Debug** menu to execute one statement or instruction from the current program counter without following execution into function calls. When complete, the program counter resides at the next program statement or instruction.

Step Out. Select **Step Out** from the **Debug** menu to execute the remaining statements or instructions in the current function and returns to the statement or instruction following the call to the current function.

Set Next Instruction. Select **Set Next Instruction** from the **Debug** menu to set the program counter to the line containing the cursor in the active file or the **Disassembly** window.

Tools Menu

The Tools menu lets you set up the Flash Loader, customize the appearance of the ZNEO developer's environment, update your firmware, and perform a cyclic redundancy check.

The Tools menu features the following selections, each of which is described in this section.

- Flash Loader – see page 87
- Firmware Upgrade – see page 91
- Show CRC – see page 91
- Calculate File Checksum – see page 92
- Customize – see page 93
- Options – see page 96

Flash Loader

Observe the following procedure to program internal and external Flash for the ZNEO processors:

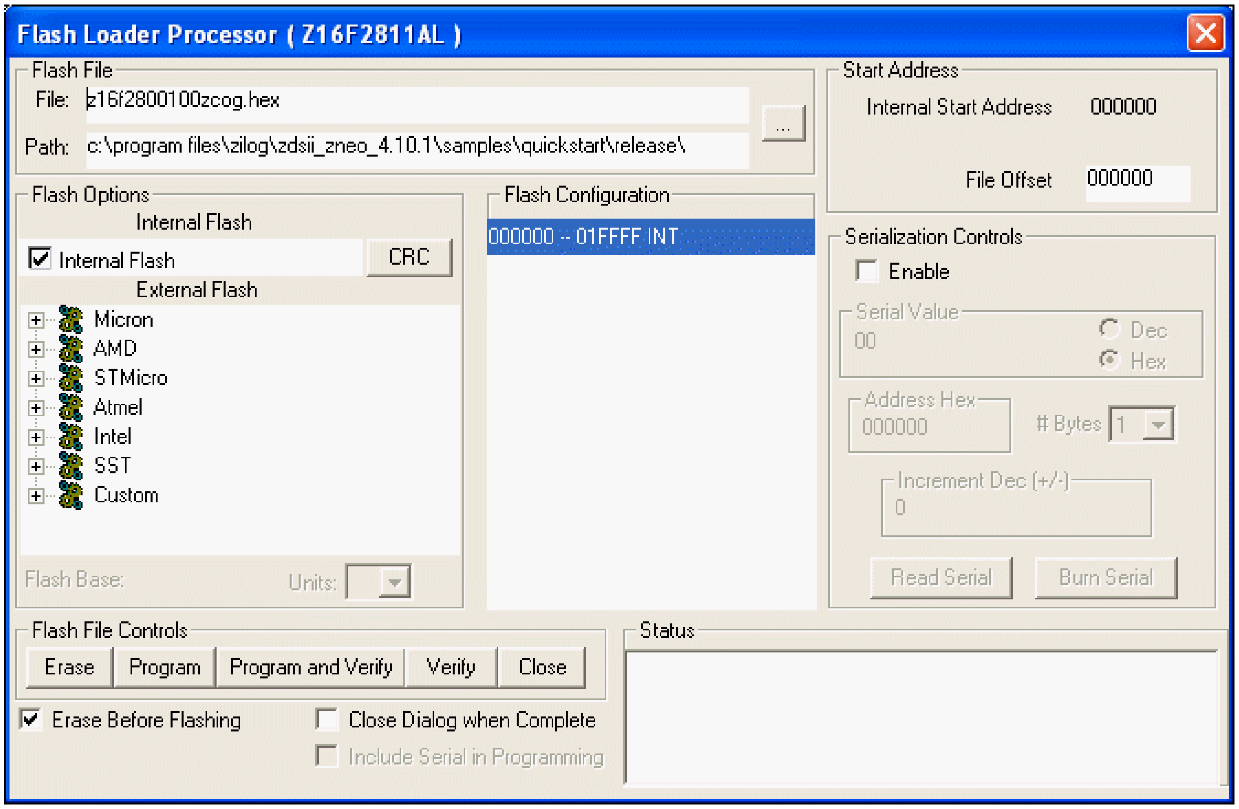
1. Ensure that the target board is powered up and the emulator is connected and operating properly.
2. In the **Configure Target** dialog box (see page 76), perform the following brief procedure:

a. If external memory is used on the target, ensure that the appropriate external bus interface is selected and that each chip select register used on the target is enabled and configured properly.

b. Configure the **Clock Source** and **Frequency** settings to match the clock source and frequency used on the target.

3. In the **Address Spaces** page (see page 68), configure the address range for each type of memory that is present on the target.

4. Select **Flash Loader** from the **Tools** menu. The Flash Loader connects to the target and sets up communication. The **Flash Loader Processor** dialog box is displayed (see Figure 71) with the appropriate Flash target options for the selected CPU.


Flash Loader Processor (Z16F2811AL)

Flash File
File: z16f2800100zcog.hex
Path: c:\program files\zilog\zdsii_zneo_4.10.1\samples\quickstart\release\

Start Address
Internal Start Address 000000
File Offset 000000

Flash Options
Internal Flash
☑ Internal Flash
External Flash
Micron
AMD
STMicro
Atmel
Intel
SST
Custom
Flash Base:
Units:

Flash Configuration
000000 -- 01FFFF INT

Serialization Controls
☐ Enable
Serial Value 00
Dec [✓ Hex]
Address Hex 000000
Bytes 1
Increment Dec (+/-) 0

Flash File Controls

☑ Erase Before Flashing
☐ Close Dialog when Complete
☐ Include Serial in Programming

Status

Figure 71. Flash Loader Processor Dialog Box

5. Click the **Browse** button (...) to navigate to the hex file to be flashed.

6. Select the Flash targets in the **Flash Options** area. The Flash Options are displayed in the **Flash Loader Processor** dialog box depend on the CPU you selected in the **Gen-**

eral page of the **Project Settings** dialog box (see the Project Settings—General Page section on page 47).

7. Select at least one of the following checkboxes in the **Flash Options** area before erasing or flashing a target:
   - **Internal Flash**: The internal Flash memory configuration is defined in the `CpuFlashDevice.xml` file. The device is the currently selected microcontroller or microprocessor. When internal Flash is selected, the address range is displayed in the **Flash Configuration** area with an INT extension.
   - **External Flash**: If you select the **External Flash** checkbox, select which Flash devices you want to program. The Flash devices are defined in the `FlashDevice.xml` file.

   The device is the current external Flash device's memory arrangement. When an external Flash device is selected, the Flash Loader uses the address specified in the **Flash Base** field to begin searching for the selected Flash device. The Flash Loader reads each page of memory from the `FlashDevice.xml` file, checking if the page is enabled by the chip select register settings. It then queries the actual address to verify that the correct Flash device is found. If the correct Flash device is found, the page's range with an EXT extension and chip select register are displayed in the **Flash Configuration** area.

   The external Flash device options are predefined Flash memory arrangements for specific Flash devices such as the Micron MT28F008B3. The Flash Loader uses the external Flash device option arrangements as a guide for erasing and loading the Intel hexadecimal file in the appropriate blocks of memory.

Note: The Flash Loader is unable to identify, erase or write to a page of Flash that is protected through hardware. For example, a target might have a write enable jumper to protect the boot block. In this case, the write enable jumper must be set before flashing the area of Flash. The Flash Loader displays this page as disabled.

8. To perform a cyclic redundancy check on the entire range of internal Flash memory, click **CRC**. The checksum is displayed in the **Status** area of the **Flash Loader Processor** dialog box.
9. In the **Flash Base** field, enter where you want the Flash programming to start. The Flash base defines the start of external Flash.
10. In the **Units** drop-down menu, select the number of Flash devices to program. For example, if you have two devices stacked on top of each other, select **2** in the **Units** list box.
11. Select the pages to erase before flashing in the **Flash Configuration** area. Pages that are grayed out are not available.

12. Enter the appropriate offset values in the **File Offset** field to offset the base address of the hex file. The hex file address is shifted by the offset defined in the **Start Address** area; however, you must allow for the shift in any defined jump table index. This offset value also shifts the erase range for the Flash.
13. Select the **Erase Before Flashing** checkbox to erase all Flash memory before writing the hex file to Flash memory.

Caution: Be careful when selecting pages from which to delete Flash memory. Clicking **ERASE** deletes only the pages that are selected.

14. If appropriate, select the **Close Dialog When Complete** checkbox to close the dialog box after writing the hex file to Flash memory.
15. To use the serialization feature, observe the following procedure.
    a. Select the **Include Serial in Programming** checkbox. This option programs the serial number after the selected hex file has been written to Flash.
    b. Select the **Enable** checkbox.
    c. Enter the start value for the serial number in the **Serial Value** field and select the **Dec** button for a decimal serial number or the **Hex** button for a hexadecimal serial number.
    d. Enter the location in which you want the serial number to be located in the **Address Hex** field.
    e. Select the number of bytes that you want the serial number to occupy in the **# Bytes** drop-down menu.
    f. Enter the decimal interval that you want the serial number incremented by in the **Increment Dec** (+/–) field. If you want the serial number to be decremented, enter a negative number. After the current serial number is programmed, the serial number is then incremented or decremented by the amount in the Increment Dec (+/–) field.
    g. Select the **Erase Before Flashing** checkbox. This option erases the Flash before writing the serial number.
    h. Click **Burn Serial** to write the serial number to the current device or click **Program** or **Program and Verify** to program the Flash memory with the specified hex file and write the serial number.
16. If you want to check a serial number that has already been programmed at an address, perform the following brief procedure.
    a. Select the **Enable** checkbox.
    b. Enter the address that you want to read in the **Address Hex** field.
    c. Select the number of bytes to read from **# Bytes** drop-down menu.

d. Click **Read Serial** to check the serial number for the current device.

17. Program the Flash memory by clicking one of the following buttons:
    - Click **Program** to write the hex file to Flash memory and perform no checking while writing.
    - Click **Program and Verify** to write the hex file to Flash memory by writing a segment of data and then reading back the segment and verifying that it has been written correctly.

18. Click **Verify** to verify Flash memory. The Flash Loader reads and compares the hex file contents with the current contents of Flash memory. This function does not change target Flash memory.

Firmware Upgrade

Note: This command is available only when a supporting debug tool is selected (see the Debug Tool section on page 80).

- USB Smart Cable

  <*ZDS Installation Directory*>`\bin\firmware\USBSmartCable\USBSmartCable upgrade information.txt`

- Serial Smart Cable

  This product is not available for this release.

- Ethernet Smart Cable

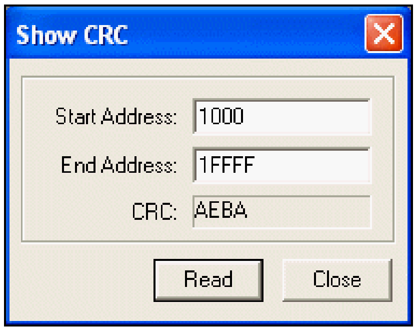
  <*ZDS Installation Directory*>`\bin\firmware\EthernetSmartCable\EthernetSmartCable upgrade information.txt`

Show CRC

Note: This command is only available when the target is not a simulator.

Observe the following procedure to perform a cyclic redundancy check (CRC):

1. Select **Show CRC** from the **Tools** menu. The **Show CRC** dialog box is displayed; see Figure 72.



Start Address: 1000
End Address: 1FFFF
CRC: AEBA

Figure 72. Show CRC Dialog Box

2. Enter the start address in the **Start Address** field. The start address must be on a 4K boundary. If the address is not on a 4K boundary, ZDS II produces an error message.
3. Enter the end address in the **End Address** field. If the end address is not a 4K increment, it is rounded up to a 4K increment.
4. Click **Read**. The checksum is displayed in the **CRC** field.

Calculate File Checksum

Observe the following procedure to calculate the file checksum:

1. Select **Calculate File Checksum** from the **Tools** menu. The **Calculate Checksum** dialog box is displayed; see Figure 73.

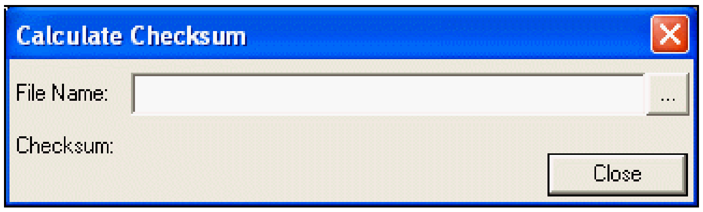


Figure 73. Calculate Checksum Dialog Box

2. Click the **Browse** button (...) to select the `.hex` file for which you want to calculate the checksum. The IDE adds the bytes in the files and displays the result in the checksum field; see Figure 74.

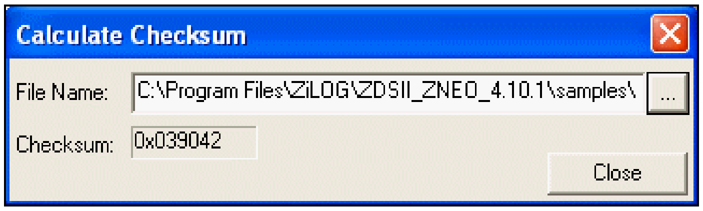


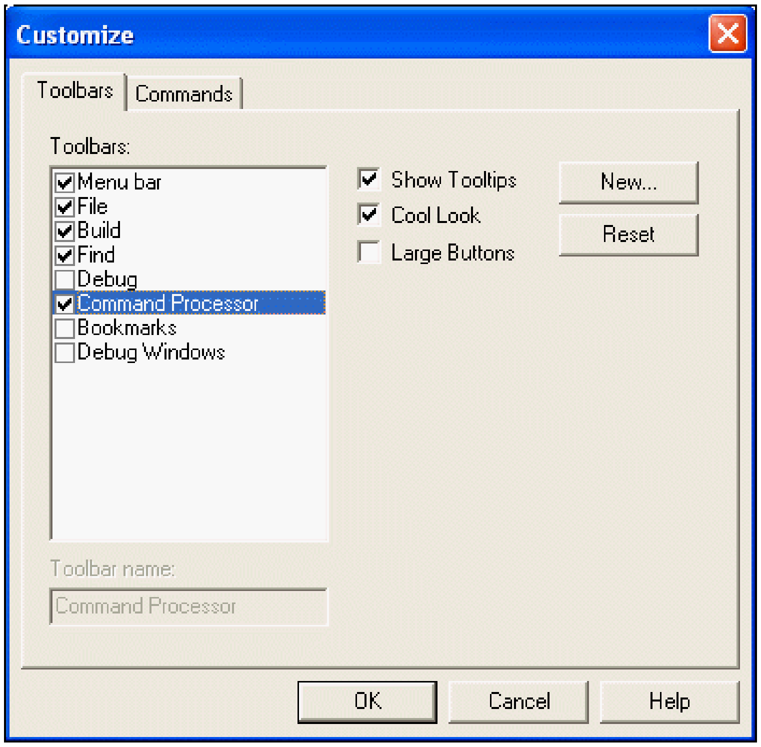
Figure 74. Calculate Checksum Dialog Box

3. Click **Close**.

Customize

The **Customize** dialog box contains two tabs; a description of each follows.

Customize—Toolbars Tab. The **Toolbars** tab, shown in Figure 75, lets you select the toolbars you want to display in the ZNEO developer's environment, change the way the toolbars are displayed, or create a new toolbar. You cannot delete, customize, or change the names of the default toolbars.



Toolbars:

- [✓] Menu bar
- [✓] File
- [✓] Build
- [✓] Find
- [ ] Debug
- [✓] Command Processor
- [ ] Bookmarks
- [ ] Debug Windows

- [✓] Show Tooltips
- [✓] Cool Look
- [ ] Large Buttons

Toolbar name: Command Processor

Figure 75. Customize Dialog Box–Toolbars Tab

To create a new toolbar, observe the following procedure:

1. Select **Customize** from the **Tools** menu. The **Customize** dialog box is displayed.
2. Click the **Toolbars** tab.
3. Click **New**. The **New Toolbar** dialog box is displayed; see Figure 76.

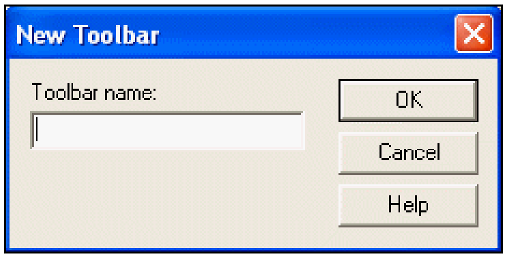


Figure 76. New Toolbar Dialog Box

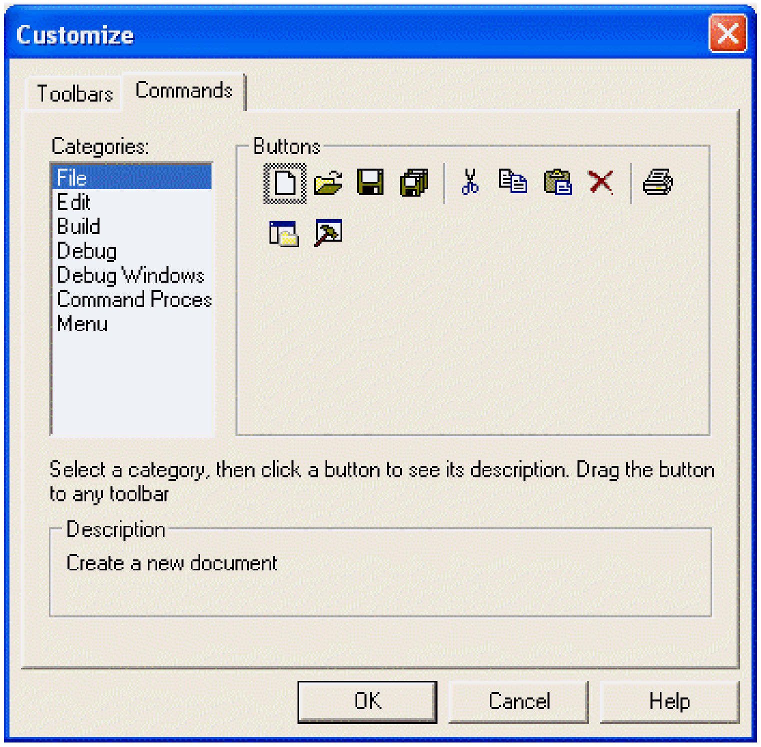
4. In the **Toolbar Name** field, enter the name of the new toolbar.

5. Click **OK**. The new toolbar is displayed as a gray box. You can change the name by selecting the new toolbar in the **Toolbars** list box, entering a new name in the **Toolbar Name** field, and pressing the **Enter** key.
6. Click the **Commands** tab.
7. Drag buttons from any category to your new toolbar. To delete the new toolbar, select the new toolbar in the **Toolbars** list box and click **Delete**.
8. Click **OK** to apply your changes or click **Cancel** to close the dialog box without making any changes.

Customize—Commands Tab. The **Commands** tab lets you modify the following functions by selecting the appropriate categories.

- File Toolbar – see page 17
- Find Toolbar – see page 19
- Build Toolbar – see page 18
- Debug Toolbar – see page 20
- Debug Windows Toolbar – see page 22
- Command Processor Toolbar – see page 19
- Menu Bar – see page 31

To see a description of each toolbar button, highlight the icon as shown in Figure 77.



Categories:
File
Edit
Build
Debug
Debug Windows
Command Proces
Menu

Buttons

Select a category, then click a button to see its description. Drag the button to any toolbar

Description
Create a new document

Figure 77. Customize Dialog Box–Commands Tab

Options

The **Options** dialog box contains the following tabs:

- Options—General Tab – see page 96
- Options—Editor Tab – see page 97
- Options—Editor—Advance Editor Options – see page 100
- Options—Debugger Tab – see page 102

Options—General Tab

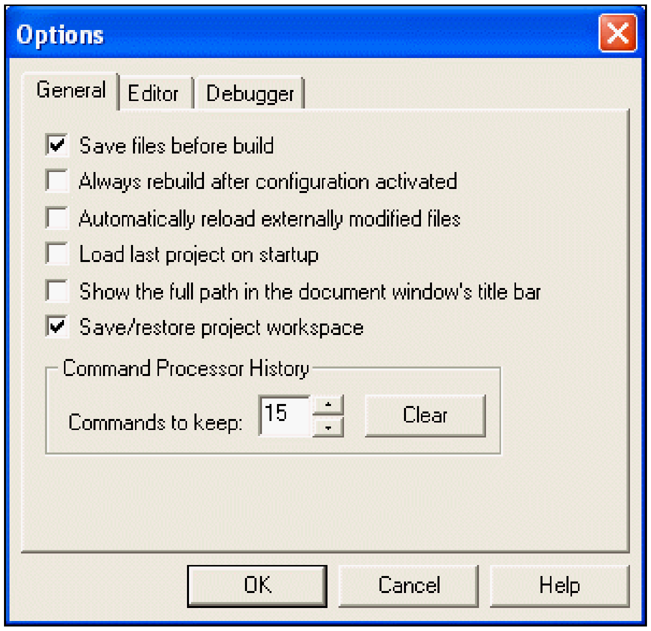
The **General** tab has the following checkboxes:

- Select the **Save Files Before Build** checkbox to save files before you build. This option is selected by default.
- Select the **Always Rebuild After Configuration Activated** checkbox to ensure that the first build after a project configuration (such as Debug or Release) is activated results in the reprocessing of all of the active project's source files. A project configu-

ration is activated by being selected (using the **Select Configuration** dialog box or the **Select Build Configuration** drop-down list box) or created (using the **Manage Configurations** dialog box). This option is not selected by default.

- Select the **Automatically Reload Externally Modified Files** checkbox to automatically reload externally modified files. This option is not selected by default.
- Select the **Load Last Project on Startup** checkbox to load the most recently active project when you start ZDS II. This option is not selected by default.
- Select the **Show the Full Path in the Document Window's Title Bar** checkbox to add the complete path to the name of each file open in the **Edit** window.
- Select the **Save/Restore Project Workspace** checkbox to save the project workspace settings each time you exit from ZDS II. This option is selected by default.

Select a number of commands to save in the **Commands to Keep** field or click **Clear** to delete the saved commands.



[✓] Save files before build
[ ] Always rebuild after configuration activated
[ ] Automatically reload externally modified files
[ ] Load last project on startup
[ ] Show the full path in the document window's title bar
[✓] Save/restore project workspace
Commands to keep: 15

Figure 78. Options Dialog Box—General Tab

Options—Editor Tab

Use the **Editor** tab to change the default settings of the editor for your assembly, C, and default files. The *syntax style* of each file can be configured individually.

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed; see Figure 79.

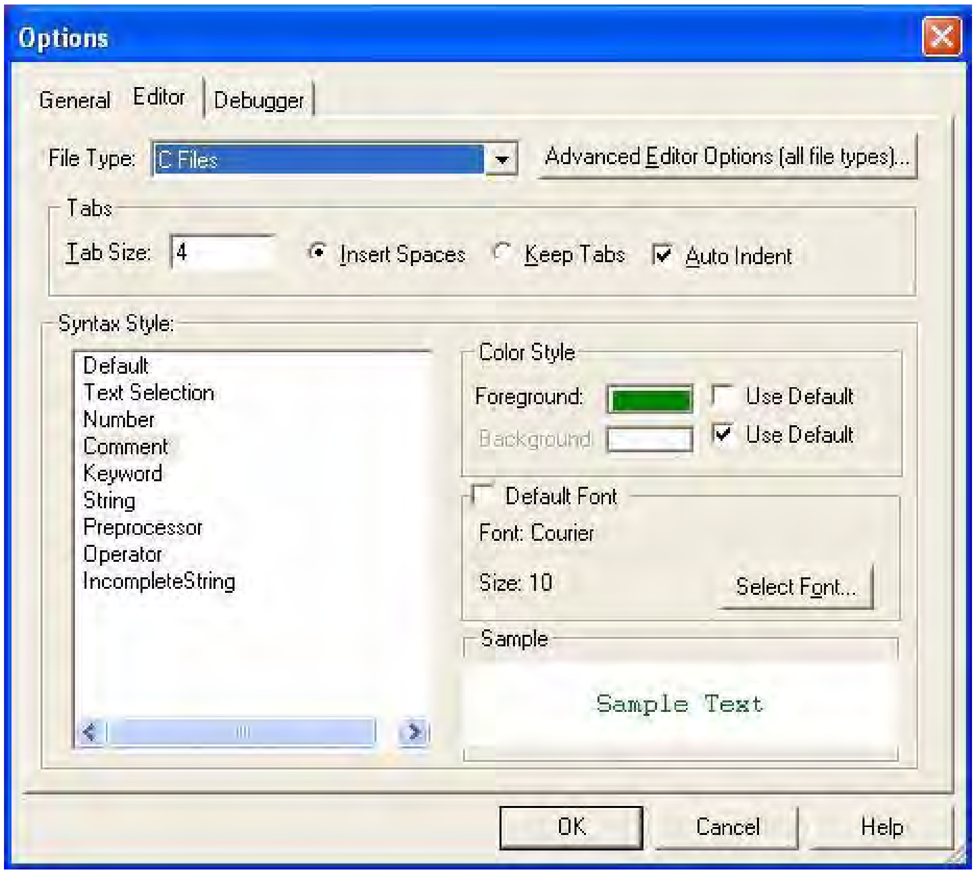


Figure 79. Options Dialog Box—Editor Tab

2. Click the **Editor** tab.
3. Select a file type from the **File Type** drop-down list box, in which you can select C files, assembly files, or other files and windows.
4. In the **Tabs** area, perform the following tasks:
   - Use the **Tab Size** field to change the number of spaces that a tab indents code.
   - Select the **Insert Spaces** button or the **Keep Tabs** button to indicate how to format indented lines.
   - Select the **Auto Indent** checkbox if you want the IDE to automatically add indentation to your files.

5. The syntax style of each file type can have its own configuration for background, foreground and font. Select an item in the **Syntax Style:** drop-down list box.

6. To configure the background or foreground color of the selected item, make sure that the **Use Default** checkboxes are not selected, then select the color of your choice in the **Foreground** or **Background** fields to display its respective **Color** dialog box (see Figure 80).

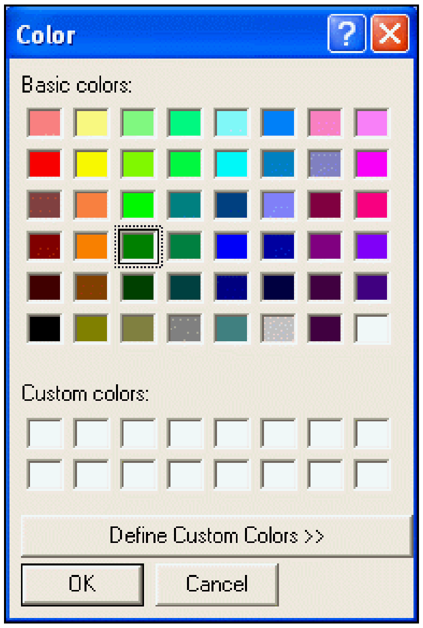


Figure 80. Color Dialog Box

7. If you want to use the default foreground or background color for the selected syntax style, enable the **Use Default** checkbox next to the **Foreground** or **Background** checkbox (see Figure 79). The default color configuration can be changed by selecting **Default** from the **Syntax Style** drop-down list box.

8. Click **OK** to close the **Color** dialog box.

9. To change the font of the selected syntax style, make sure that the **Default Font** checkboxes are not selected in the **Options** dialog box, then click the **Select Font** button to display the **Font** dialog box, in which you can change the font, font style and font size; see Figure 81.

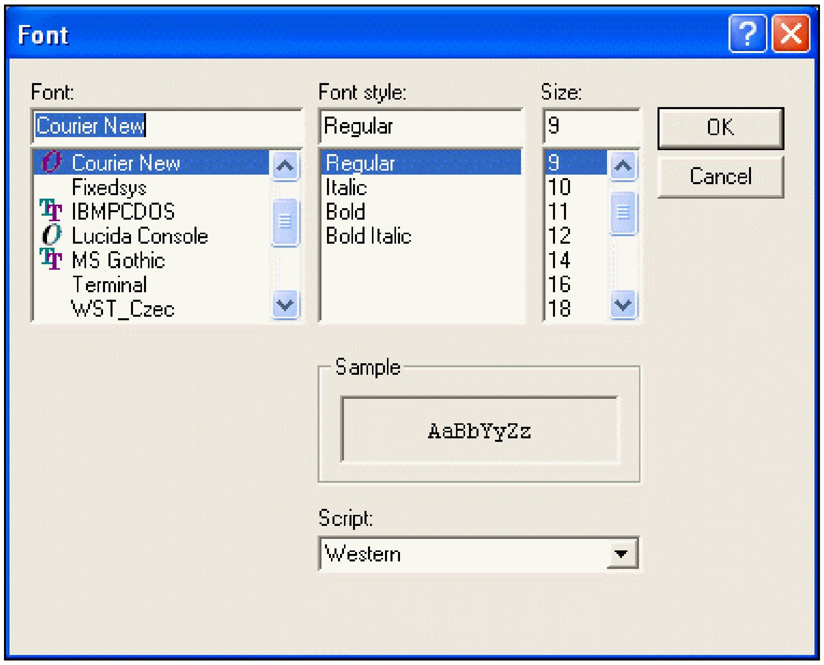


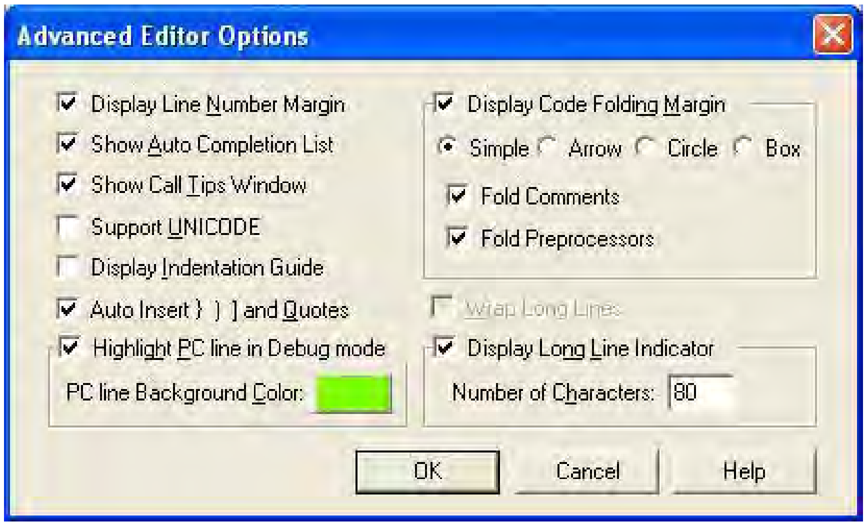
Figure 81. Font Dialog Box

10. Click **OK** to close the **Font** dialog box.
11. Click **OK** to close the **Options** dialog box.

Options—Editor—Advance Editor Options

You can enable or disable some of the intelligent editor behavior using the Advanced Editor options. To open the **Advanced Editor Options** dialog from the **Options** dialog box, click the **Editor** tab, then click the button labeled **Advanced Editor Options (all file types)**.

A description of each of the Advanced Editor Options follows Figure 82.



- [✓] Display Line Number Margin
- [✓] Show Auto Completion List
- [✓] Show Call Tips Window
- [ ] Support UNICODE
- [ ] Display Indentation Guide
- [✓] Auto Insert })] and Quotes
- [✓] Highlight PC line in Debug mode
  - PC line Background Color: 🟩
- [✓] Display Code Folding Margin
  - [✓ Simple] Arrow Circle Box
  - [✓] Fold Comments
  - [✓] Fold Preprocessors
- [ ] Wrap Long Lines
- [✓] Display Long Line Indicator
  - Number of Characters: 80

Figure 82. Options Dialog Box—Editor Tab—Advanced Editor Options Dialog Box

Display Line Number Margin

The **Display Line Number Margin** option allows you to show or hide the line number margin in the **Editor** window. To learn more, see the Line Number Margin section on page 137.

Show Auto Completion List

The **Show Auto Completion List** option allows you to enable or disable automatic completion of keyboarded elements. It launches a pop-up window that lists all of the relevant choices as you enter characters from your keyboard and allows you to choose the appropriate one. To learn more, see the Auto Completion section on page 111.

Show Call Tips Window

The **Show Call Tips Window** option allows you to enable or disable the **Call Tips** window. Call Tips is a hovering and short-lived small window that displays the prototype of a function whenever you use your keyboard to type the function followed by a left parenthesis, or "(". To learn more, see the Call Tips section on page 115.

Support UNICODE

The **Support Unicode** option allows you to enable or disable UNICODE support. Enabling UNICODE support allows you to use non-English language scripts as part of a comment section and string in your source code. To learn more, see the UNICODE Support section on page 131.

Display Indentation Guide

The **Display Indentation Guide** option allows you to enable or disable the indentation guides in the **Editor** window. Indentation guides allow you easily identify the boundaries of a block of code. To learn more, see the Indentation Guides section on page 143.

Auto Insert })] and Quotation Marks

The **Auto Insert })] and Quotes** option allows you to enable or disable the automatic insertion of the `}`, `)`, `]`, `'`, and `"` closing characters. To learn more, see the Auto Insertion of Braces and Quotes section on page 127.

Highlight PC Line in Debug Mode

The **Highlight PC line in Debug Mode** option allows you to enable or disable Program Counter line highlighting in the **Editor** window. To learn more, see the Highlighting a Program Counter Line section on page 149.

Display Code Folding Margin

The **Display Code Folding Margin** option allows you to show or hide the code folding margin in the **Editor** window. To learn more, see the Code Folding Margin section on page 135.

Wrap Long Lines

The **Wrap Long Lines** option allows you to enable or disable the wrapping of long lines of characters in the **Editor** window. To learn more, see the Wrap Long Lines section on page 142.

Display Long Line Indicator

The **Display Long Line Indicator** option allows you to show or hide the long line indicator in the E**ditor** window. To learn more, see the Long Line Indicator section on page 129.

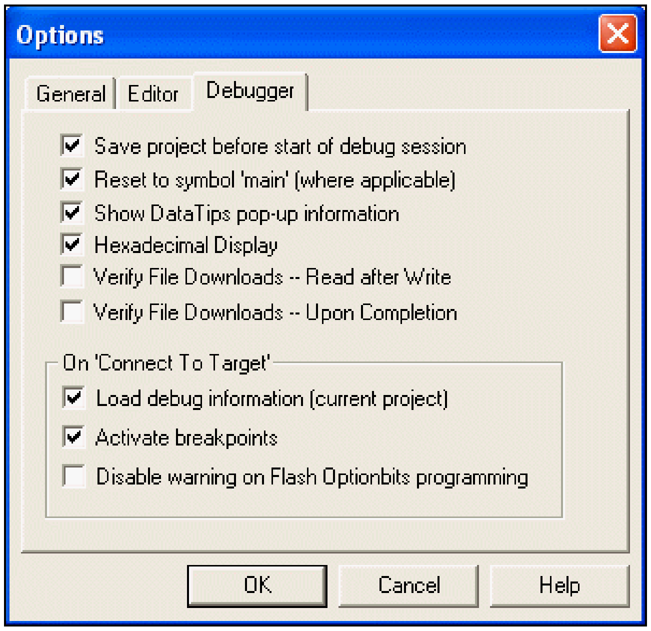
Options—Debugger Tab

The **Debugger** tab contains the following checkboxes:

- Select the **Save Project Before Start of Debug Session** checkbox to save the current project before entering **Debug** mode. This option is selected by default.
- Select the **Reset to Symbol 'main' (Where Applicable)** checkbox to skip the start-up (boot) code and start debugging at the main function for a project that includes a C language main function. When this checkbox is selected, a user reset (clicking the **Reset** button on the **Build** and **Debug** toolbars, selecting **Reset** from the **Debug** menu, or using the `reset` script command) results in the program counter (PC) pointing to the beginning of the main function. When this checkbox is not selected, a user

reset results in the PC pointing to the first line of the program (the first line of the start-up code).

- When the **Show DataTips Pop-Up Information** checkbox is selected, holding the cursor over a variable in a C file in the **Edit** window in **Debug** mode displays the value.
- Select the **Hexadecimal Display** checkbox to change the values in the **Watch** and **Locals** windows to hexadecimal format. Deselect the checkbox to change the values in the **Watch** and **Locals** windows to decimal format.
- Select the **Verify File Downloads—Read After Write** checkbox to perform a read after write verify of the Code Download function. Selecting this checkbox increases the time taken for the code download to complete.
- Select the **Verify File Downloads—Upon Completion** checkbox to verify the code that you downloaded after it has downloaded.
- Select the **Load Debug Information (Current Project)** checkbox to load the debug information for the currently open project when the **Connect to Target** command is executed (from the **Debug** menu or from the **Connect to Target** button). This option is selected by default.
- Select the **Activate Breakpoints** checkbox for the breakpoints in the current project to be active when the **Connect to Target** command is executed (from the **Debug** menu or from the **Connect to Target** button). This option is selected by default.
- Select the **Disable Warning on Flash Optionbits Programming** checkbox to prevent messages from being displayed before programming Flash option bits.


Options

General | Editor | Debugger

- [✓] Save project before start of debug session
- [✓] Reset to symbol 'main' (where applicable)
- [✓] Show DataTips pop-up information
- [✓] Hexadecimal Display
- [ ] Verify File Downloads -- Read after Write
- [ ] Verify File Downloads -- Upon Completion

On 'Connect To Target'

- [✓] Load debug information (current project)
- [✓] Activate breakpoints
- [ ] Disable warning on Flash Optionbits programming

Figure 83. Options Dialog Box—Debugger Tab

Window Menu

The Window menu allows you to select the way you want to arrange your files in the Edit window and allows you to activate the Project Workspace window or the Output window.

This section describes the six Windows menu options.

New Window

Select **New Window** to create a copy of the file you have active in the **Edit** window.

Close

Select **Close** to close the active file in the **Edit** window.

Close All

Select **Close All** to close all of the files in the **Edit** window.

Cascade

Select **Cascade** to cascade the files in the **Edit** window. Use this option to display all open windows whenever you cannot locate a window.

Tile

Select **Tile** to tile the files in the **Edit** window so that you can see all of them simultaneously.

Arrange Icons

Select **Arrange Icons** to arrange files alphabetically in the **Edit** window.

Help Menu

The Help menu contains three options, each described here.

Help Topics

Select **Help Topics** to display the ZDS II online help.

Technical Support

Select **Technical Support** to access the Technical Support page on the Zilog website.

About

Select **About** to display installed product and component version information.

Shortcut Keys

This section describes keyboard shortcuts to alternatively access the Zilog Developer Studio II menus.

File Menu Shortcuts

The five File menu options and their shortcuts are described below.

Table 2. File Menu Shortcuts

Option	Shortcut	Description
New File	Ctrl+N	To create a new file in the Edit window.
Open File	Ctrl+O	To display the Open dialog box for you to find the appropriate file.
Save	Ctrl+S	To save the file.
Save All	Ctrl+L	To save all files in the project.
Print	Ctrl+P	To print a file.

Edit Menu Shortcuts

Sixteen **Edit** menu options and their shortcuts are described below.

Table 3. Edit Menu Shortcuts

Option	Shortcut	Description
Undo	Ctrl+Z	To undo the last edit made to the active file.
Redo	Ctrl+Y	To redo the last edit made to the active file.
Cut	Ctrl+X	To delete selected text from a file and put it on the clipboard.
Copy	Ctrl+C	To copy selected text from a file and put it on the clipboard.
Paste	Ctrl+V	To paste the current contents of the clipboard into a file.
Delete	Ctrl+D	To remove a file from the current project.
Select All	Ctrl+A	To highlight all text in the active file.
Show Whitespaces	Ctrl+Shift+8	To display all whitespace characters such as spaces and tabs.
Find	Ctrl+F	To find a specific value in the designated file.
Find Again	F3	To repeat the previous search.
Replace	Ctrl+H	To replace a specific value to the designated file.
Go to Line	Ctrl+G	To jump to a specified line in the current file.
Toggle Bookmark	Ctrl+F2	To insert a bookmark in the active file for the line in which your cursor is located or to remove the bookmark for the line in which your cursor is located.
Next Bookmark	F2	To position the cursor at the line in which the next bookmark in the active file is located. The search for the next bookmark does not stop at the end of the file; the next bookmark might be the first bookmark in the file.
Previous Bookmark	Shift+F2	To position the cursor at the line in which the previous bookmark in the active file is located. The search for the previous bookmark does not stop at the beginning of the file; the previous bookmark might be the last bookmark in the file.
Remove All Bookmarks	Ctrl+Shift+F2	To delete all of the bookmarks in the currently loaded project.

Project Menu Shortcuts

The Project menu option and its shortcut is described below.

Table 4. Project Menu Shortcuts

Option	Shortcut	Description
Settings	Alt+F7	To display the Project Settings dialog box.

Build Menu Shortcuts

The Build menu options and their shortcuts are described below.

Table 5. Build Menu Shortcuts

Option	Shortcut	Description
Build	F7	To build your file and/or project.
Stop Build	Ctrl+Break	To stop the build of your file and/or project.

Debug Menu Shortcuts

These are the shortcuts for the options on the Debug menu.

Table 6. Debug Menu Shortcuts

Option	Shortcut	Description
Stop Debugging	Shift+F5	To stop debugging of your program.
Reset	Ctrl+Shift+F5	To reset the debugger.
Go	F5	To invoke the debugger (go into Debug mode).
Run to Cursor	Ctrl+F10	To make the debugger run to the line containing the cursor.
Break	Ctrl+F5	To break the program execution.
Step Into	F11	To execute the code one statement at a time.
Step Over	F10	To step to the next statement regardless of whether the current statement is a call to another function.
Step Out	Shift+F11	To execute the remaining lines in the current function and return to execute the next statement in the caller function.
Set Next Instruction	Shift+F10	To set the next instruction at the current line.

Chapter 3. Using the Editor

ZDS II provides an intelligent editor that comprises a number of features to shorten your application development time. The editor allows you to read and write code faster, navigate intelligently and identify and correct mistakes.

The editor offers the following key features.

Write Code Faster

- Auto Completion – see page 111
- Call Tips – see page 115
- Auto Indentation – see page 117
- Multiple Clipboards – see page 119
- Line and Block Comments – see page 121
- Abbreviations and Expansions – see page 121
- Auto Insertion of Braces and Quotes – see page 127
- Long Line Indicator – see page 129
- UNICODE Support – see page 131

Read Code Faster

- Auto Syntax Styler – see page 132
- Code Folding Margin – see page 135
- Line Number Margin – see page 137
- Type Info Tips – see page 139
- Highlighting and Finding Matched Braces – see page 140
- Matching Preprocessor Conditional Macros – see page 141
- Wrap Long Lines – see page 142
- Indentation Guides – see page 143
- Zoom In/Out – see page 145

Navigate Intelligently

- Bookmarks – see page 145
- Opening an Include File – see page 148

- Opening an Include File – see page 148

Identify and Correct Mistakes

- Mismatched Brace Highlighting – see page 151
- Auto Conversion of "." to "→" – see page 153

In addition to the above feature set, the editor supports many useful hot keys to help improve your productivity. The hot keys can save you valuable time by allowing you to keep your hands near the keyboard rather than having to repeatedly reach for the mouse.

A complete reference of the hot keys supported by ZDS II, as well as other supported tools (including the editor), can be found in the ZDS II help files. Simply navigate via the ZDS II **Help** menu to **Hotkeys**.

Tables 7 through 10 list a number of useful hot keys.

Table 7. Working with Words

Command name	Hotkey	Description
Word Left	Ctrl+Left Arrow	Moves back one word.
Word Right	Ctrl+Right Arrow	Moves forward one word.
Word Left Select	Ctrl+Shift+Left Arrow	Extends the selection back one word.
Word Right Select	Ctrl+Shift+Right Arrow	Extends the selection forward one word.
Word Backward Delete	Ctrl+BackSpace	Deletes a word to the left.
Word Forward Delete	Ctrl+Delete	Deletes a word to the right.

Table 8. Working with Lines

Command name	Hotkey	Description
Line Join	Ctrl+J	Joins the selected lines.
Line Split	Ctrl+Shift+J	Splits the selected line that is not fit within the visible window area.
Line Cut	Ctrl+L	Deletes the cursor line or the selected lines and puts them on the Clipboard.
Line Delete	Ctrl+Shift+L	Deletes the cursor line or selected lines.
Line Copy	Ctrl+T	Copies the current line or selected lines and put them in the Clipboard.
Line Transpose	Ctrl+Shift+T	Swaps the current and previous line.

Table 8. Working with Lines (Continued)

Line or Block Duplicate	Ctrl+D	Duplicates the cursor line or the selected lines.
Line Start Delete	Ctrl+Shift+BackSpace	Deletes the line contents to its start.
Line End Delete	Ctrl+Shift+Delete	Deletes the line contents to its end.
Indent	Tab	Indents the cursor line or selected lines.
Un-indent	Shift+Tab	Un-indents the cursor line or selected lines.

Table 9. Working with Paragraphs

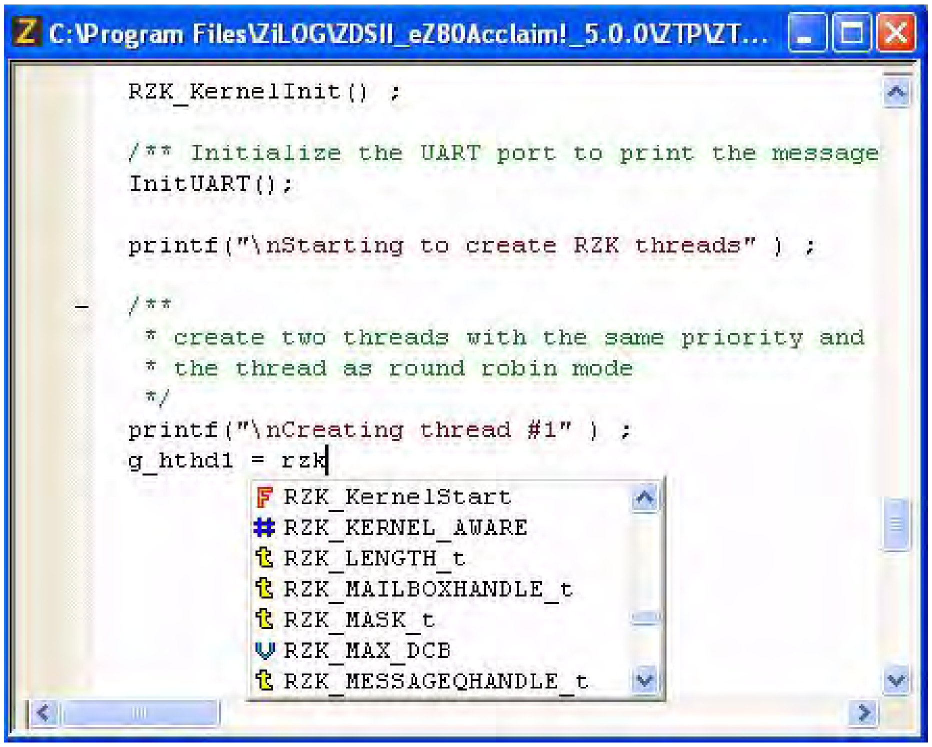
Command name	Hotkey	Description
Paragraph Previous	Ctrl+[	Moves to the start of the previous paragraph.
Paragraph Next	Ctrl+]	Moves to the start of the next paragraph.
Paragraph Previous Select	Ctrl+Shift+[	Extends the selection to the start of the previous paragraph.
Paragraph Next Select	Ctrl+Shift+]	Extends the selection to the start of the next paragraph.

Table 10. Working with Files

Command name	Hotkey	Description
File Forward Navigate	Ctrl+Tab	Navigates to the next opened file.
File Backward Navigate	Ctrl+Shift+Tab	Navigates to the previous opened file.
File Close	Ctrl+F4	Closes the active file.

Auto Completion

You can accelerate your keyboarding with an autocompletion list that appears as you type. Essentially, when you begin typing the first few characters of a word, a window will pop up to display a list of all relevant choices and allow you to choose the appropriate one; see Figure 84 for an example.



```
RZK_KernelInit() ;

/** Initialize the UART port to print the message
InitUART();

printf("\nStarting to create RZK threads" ) ;

/**
 * create two threads with the same priority and
 * the thread as round robin mode
 */
printf("\nCreating thread #1" ) ;
g_hthd1 = rzk
```

```
RZK_KernelStart
RZK_KERNEL_AWARE
RZK_LENGTH_t
RZK_MAILBOXHANDLE_t
RZK_MASK_t
RZK_MAX_DCB
RZK_MESSAGEQHANDLE_t
```

Figure 84. Auto Completion

Typing one or two characters is typically enough for the editor to show the autocompletion list; simply enter more characters to refine your choices. Typing within commented lines or in a string does not launch the autocompletion list box.

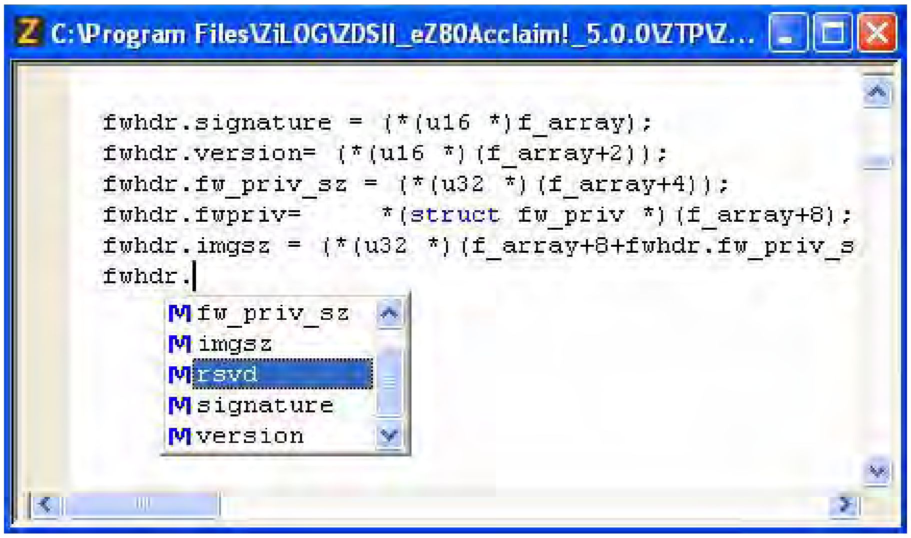
Use your arrow keys to scroll through the list; press the **Tab** or **Enter** key to insert a currently-selected item into your document, or press the **Esc** key to cancel a pop-up list.

Press **Ctrl+Enter** to open the autocompletion list anytime, provided that the text caret is positioned anywhere on a word or at the end of a word.

Data Structure Member List box

The C data structure construct members, `struct` and `union`, are listed upon entering "." or "→" after a variable or pointer of either type. With your keyboard, enter a few characters of the member to refine the choices and select the relevant one.

Upon typing the period character "." following the structure variable name, the editor shows the autocompletion list of all of the members of that structure. See the code snapshot in Figure 85.

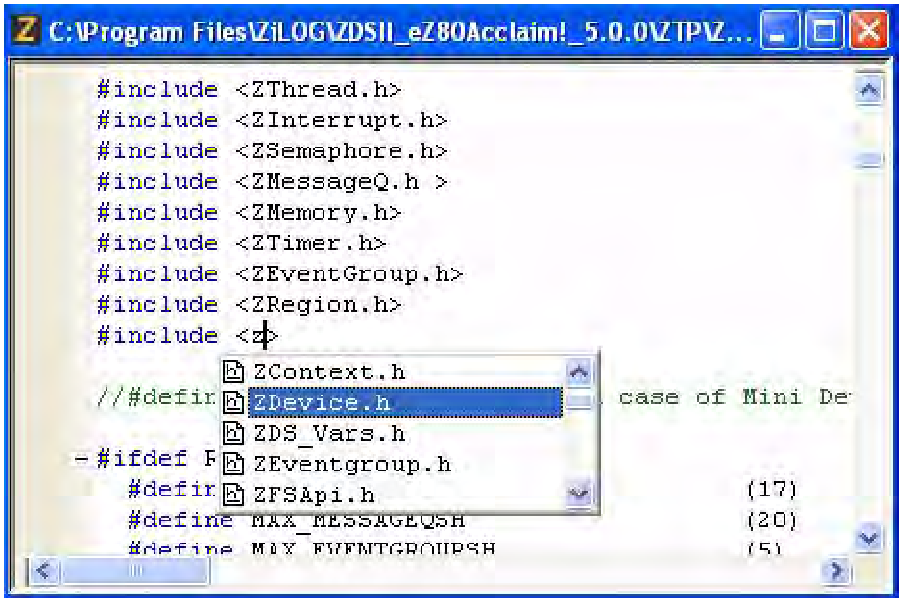


```
fwhdr.signature = (*(u16 *)f_array);
fwhdr.version= (*(u16 *)(f_array+2));
fwhdr.fw_priv_sz = (*(u32 *)(f_array+4));
fwhdr.fwpriv=     *(struct fw_priv *)(f_array+8);
fwhdr.imgsz = (*(u32 *)(f_array+8+fwhdr.fw_priv_s
fwhdr.
    fw_priv_sz
    imgsz
    rsvd
    signature
    version
```

Figure 85. Autocompletion of Members

Include file list box

The editor opens a list box of all possible header file entries after you type `#include` and a double quote or a angle bracket, as shown in Figure 86. Enter more characters to refine the choices and include the appropriate header file.



```
#include <ZThread.h>
#include <ZInterrupt.h>
#include <ZSemaphore.h>
#include <ZMessageQ.h >
#include <ZMemory.h>
#include <ZTimer.h>
#include <ZEventGroup.h>
#include <ZRegion.h>
#include <z>
            ZContext.h
//#defir    ZDevice.h          case of Mini De
            ZDS_Vars.h
#ifdef F    ZEventgroup.h
  #defir    ZFSApi.h                   (17)
  #define MAX_MESSAGEQSH              (20)
  #define MAX_EVENTGROUPSH            (5)
```

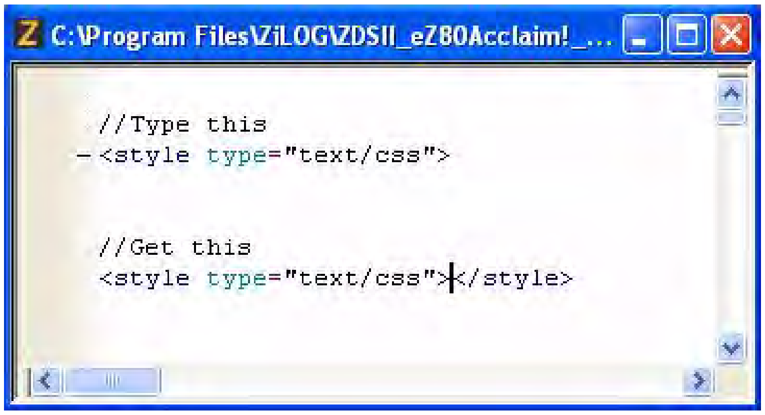
Figure 86. Autocompletion of Header File Inclusion

When you type an angle bracket following an `#include` directive, the editor shows a list of all *system include* header files.

When you type a double quote following an `#include` directive, the editor shows a list of all system and *user include* header files.

Autocompletion of Tags in an HTML file

When you enter a starting tag in an HTML file, the editor automatically adds its end tag and places the text caret in between them to allow you to enter the content. See the example in Figure 87.

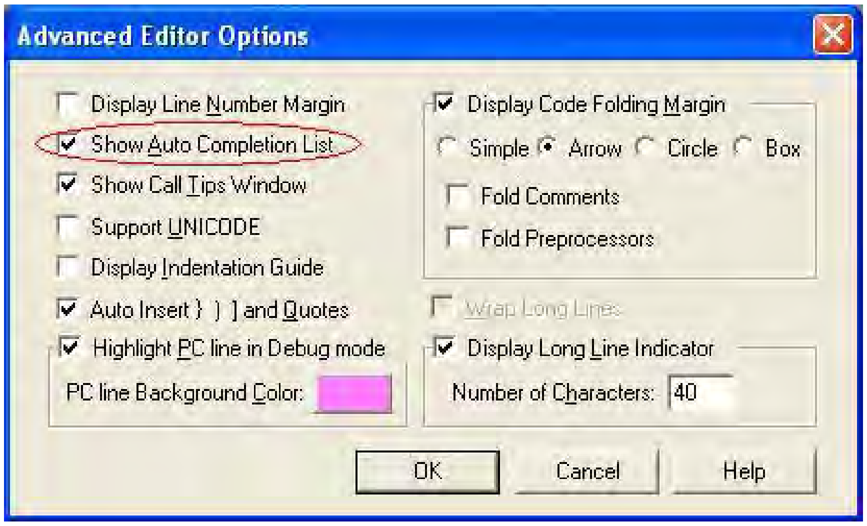


```
//Type this
<style type="text/css">


//Get this
<style type="text/css"></style>
```

Figure 87. Autocompletion of HTML Tags

To enable or disable autocompletion, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.
3. Click the **Advanced Editor Options** button. The **Advanced Editor Options** dialog box is displayed, as shown in Figure 88.
4. Select the **Show Autocompletion List** checkbox to enable the autocompletion. This option is selected by default.



Advanced Editor Options

- [ ] Display Line Number Margin
- [✓] Show Auto Completion List
- [✓] Show Call Tips Window
- [ ] Support UNICODE
- [ ] Display Indentation Guide
- [✓] Auto Insert })] and Quotes
- [✓] Highlight PC line in Debug mode
  - PC line Background Color: 🟪
- [✓] Display Code Folding Margin
  - Simple [✓ Arrow] Circle Box
  - [ ] Fold Comments
  - [ ] Fold Preprocessors
- [ ] Wrap Long Lines
- [✓] Display Long Line Indicator
  - Number of Characters: 40

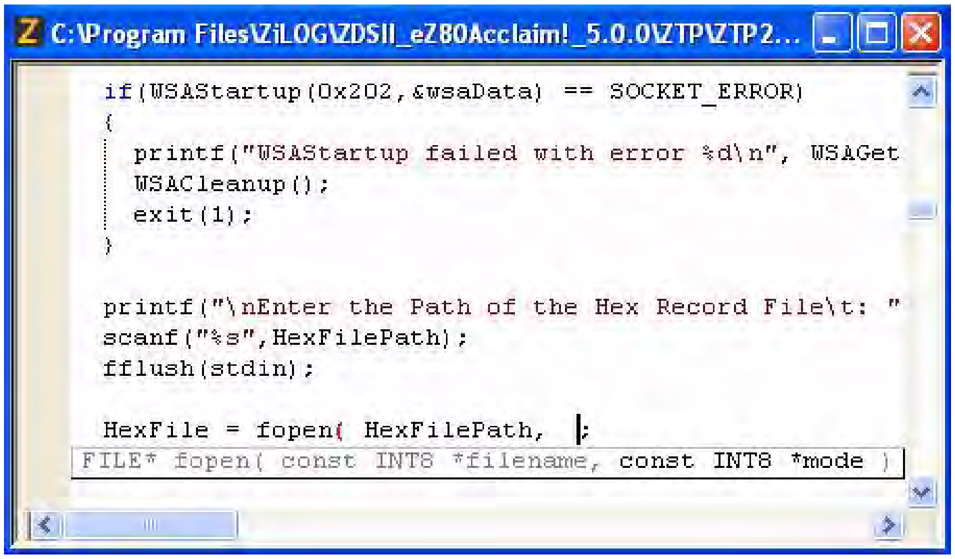
Figure 88. Advance Editor Options—Show Autocompletion List

Note: When autocompletion is disabled, you can still bring up the autocompletion list box by pressing Ctrl+Enter. Autocompletion is not supported for assembly files.

Call Tips

Call Tips is a hovering and short-lived small window that displays the prototype of a function whenever you type a function followed by a left parenthesis. As each parameter is entered via the keyboard, the Call Tips function guides you by highlighting the corresponding argument of the function prototype within the hovering window.

An example of the Call Tips window is shown in Figure 89.



C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_5.0.0\ZTP\ZTP2...

```
if(WSAStartup(0x202,&wsaData) == SOCKET_ERROR)
{
  printf("WSAStartup failed with error %d\n", WSAGet
  WSACleanup();
  exit(1);
}

printf("\nEnter the Path of the Hex Record File\t: "
scanf("%s",HexFilePath);
fflush(stdin);

HexFile = fopen( HexFilePath, ;
FILE* fopen( const INT8 *filename, const INT8 *mode )
```

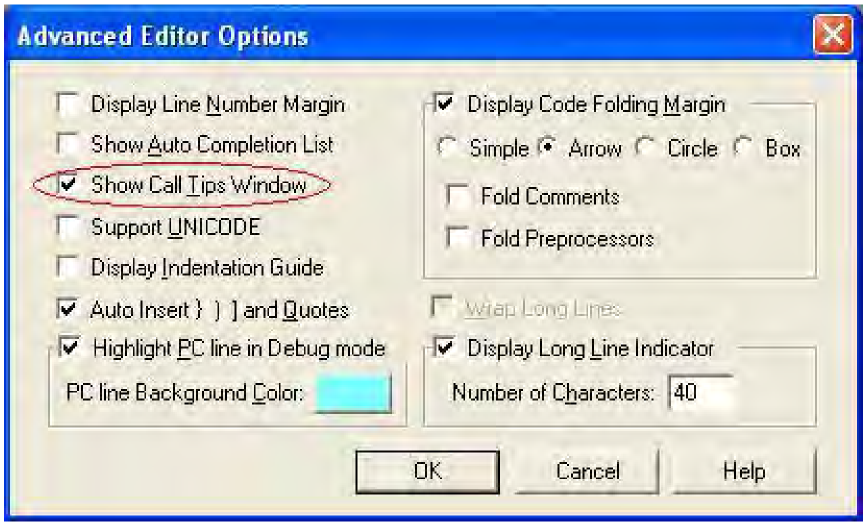
Figure 89. Call Tips Window

Call Tips becomes available for virtually all of the functions declared or defined in your project code and all standard include files. You are not required to build the project for the call tips to become available. Typing within commented lines or in a string does not bring up the Call Tips window.

If you return to the middle of a parameter list in a function call, press Ctrl+Shift+Enter to cause the call tips to reappear. To hide the call tips window, press ESC.

To enable or disable the call tips, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.
3. Click the **Advanced Editor Options** button. The **Advanced Editor Options** dialog box is displayed., as shown in Figure 90.
4. Select the **Show Call Tips** checkbox to enable the call tips. This option is selected by default.


Advanced Editor Options

- ☐ Display Line Number Margin
- ☐ Show Auto Completion List
- ☑ Show Call Tips Window
- ☐ Support UNICODE
- ☐ Display Indentation Guide
- ☑ Auto Insert })] and Quotes
- ☑ Highlight PC line in Debug mode
  - PC line Background Color: 🟦
- ☑ Display Code Folding Margin
  - Simple [✓ Arrow] Circle Box
  - ☐ Fold Comments
  - ☐ Fold Preprocessors
- ☐ Wrap Long Lines
- ☑ Display Long Line Indicator
  - Number of Characters: 40

Figure 90. Advance Editor Options—Show Call Tips Window

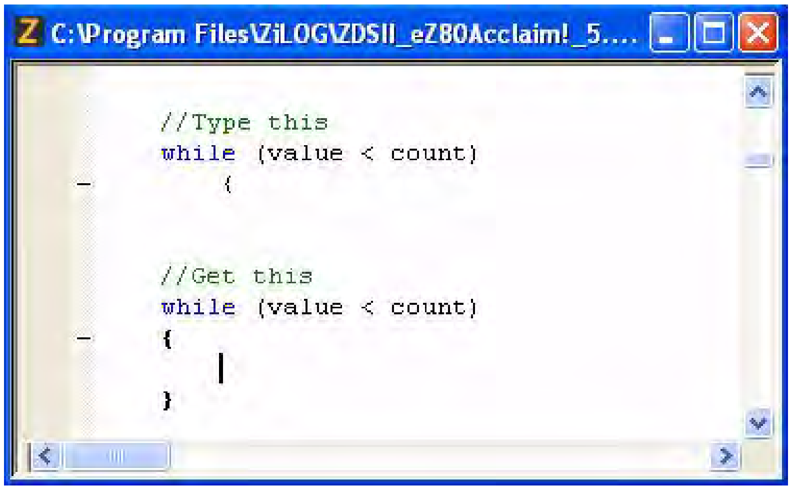
Note: When call tips are disabled, you can still bring up the **Call Tips** window by pressing Ctrl+Shift+Enter. Call tips are not supported for assembly files.

Auto Indentation

Indentation of statements is often used to clarify the program structure both in C and in assembly code; it is one of the indispensable coding standards. While the Tab key is often used to indent the statements belonging to a particular code block, manual indentation is cumbersome and time-consuming.

The ZDS II editor provides automatic indentation that indent lines in a smart way based on the syntax and formats while you are typing.

Figure 91 shows an example of autoindentation in a C file; note that the closing brace is added automatically upon entering the opening brace because **Auto Insertion of Braces and Quotes** is enabled.



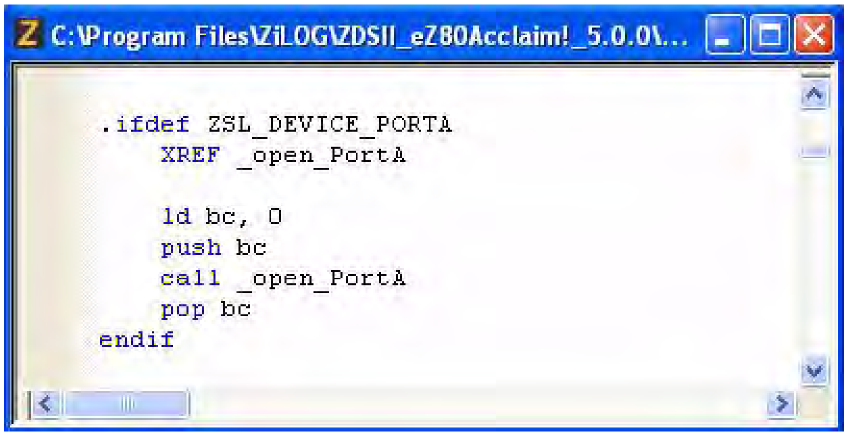
```
//Type this
while (value < count)
    {


//Get this
while (value < count)
{
    |
}
```

Figure 91. Auto Indentation in C Program

In C program code, auto indentation is supported with brace characters { } and keywords if, else, while, for, do, case, default.

In assembly program code, the auto indentation is supported with macros and conditional directives such as `ifdef`, `ifndef`, `if`, `else`, `elif`, `elseif`, `ifsame`, `ifma`, `macro`, `$while`, `$repeat`, `with`, `endif`, `endmac`, `endm`, `endmacro`, `macend`, `$wend`, `$until` and `endwith`.

Figure 92 shows an example of autoindentation with an assembly program. Note that all of the lines between `ifdef` and `endif` are automatically indented.

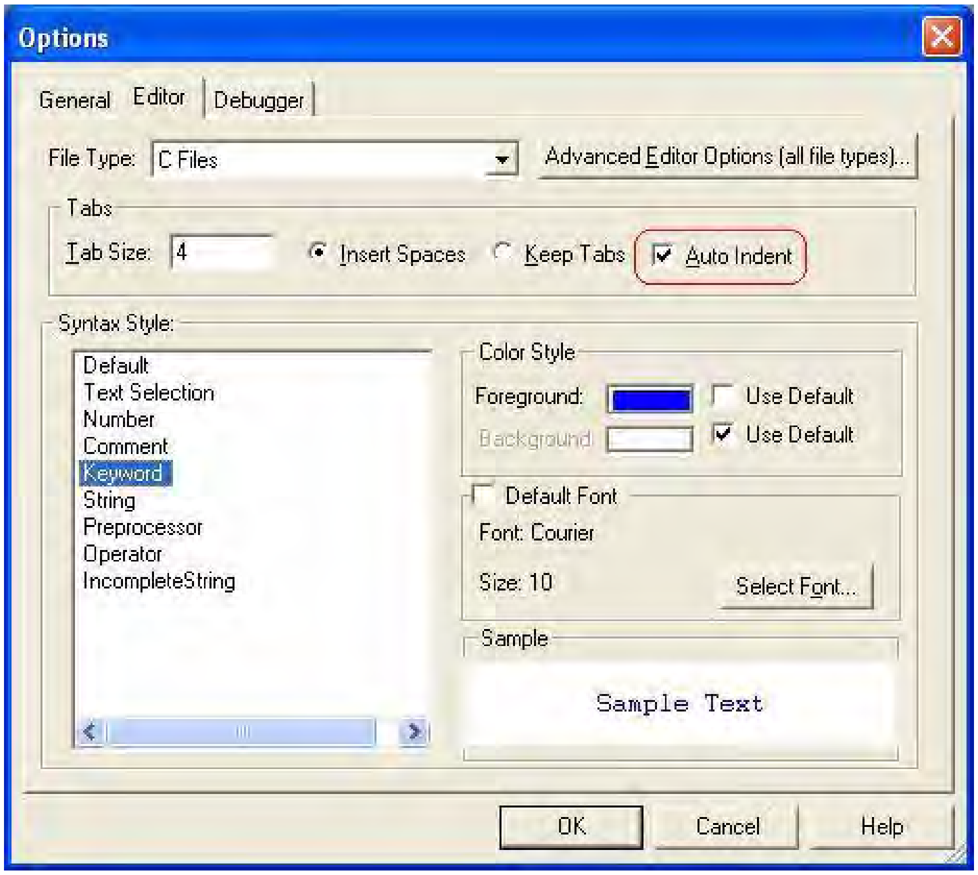


```
.ifdef ZSL_DEVICE_PORTA
    XREF _open_PortA

    ld bc, 0
    push bc
    call _open_PortA
    pop bc
endif
```

Figure 92. Auto Indentation With Assembly Program

To enable or disable auto indentation, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.
3. Select **Auto Indent** checkbox to enable the automatic indentation of code. This option is selected by default.



File Type: C Files

Tabs

Tab Size: 4 [✓ Insert Spaces] Keep Tabs [✓ Auto Indent]

Syntax Style:

Default
Text Selection
Number
Comment
Keyword
String
Preprocessor
Operator
IncompleteString

Color Style

Foreground: Use Default

Background [✓ Use Default]

Default Font

Font: Courier

Size: 10

Sample

Sample Text

Figure 93. Options Dialog Box—Auto Indent

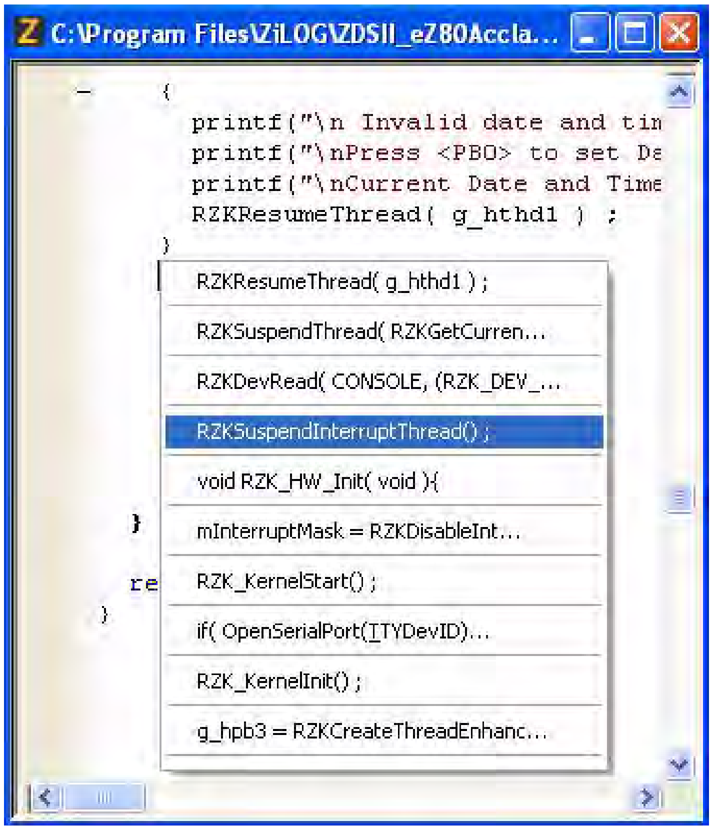
Multiple Clipboards

The limitation of built-in clipboard in windows is that at any time you can hold only one copied item, and as soon as you cut or copy something else, the previous item is removed, in fact overwritten, by the newer item.

The ZDS II editor provides multiple clipboards that can improve your productivity by allowing you to keep a history of up to 10 previous cuts and copies you have added to the system clipboard. It works alongside the regular Windows Clipboard and records every piece of data that you cut or copy.

Simply use the keyboard hotkey Ctrl+Shift+V to retrieve earlier copies. You can scan through the list of clipboarded items and select any item you prefer.

Press the Up or Down arrow in the keyboard to select your appropriate entry from the list. Press Enter to paste the selected entry at the text caret position.



```
    {
      printf("\n Invalid date and tin
      printf("\nPress <PB0> to set Da
      printf("\nCurrent Date and Time
      RZKResumeThread( g_hthd1 ) ;
    }
```

```
RZKResumeThread( g_hthd1 ) ;
RZKSuspendThread( RZKGetCurren...
RZKDevRead( CONSOLE, (RZK_DEV_...
RZKSuspendInterruptThread() ;
void RZK_HW_Init( void ){
mInterruptMask = RZKDisableInt...
RZK_KernelStart() ;
if( OpenSerialPort(TTYDevID)...
RZK_KernelInit() ;
g_hpb3 = RZKCreateThreadEnhanc...
```

Figure 94. Multiple Clipboards

Clipboards are listed in the order in which they are copied. Pasting from the pop-up list moves a clipboard to the top of the list.

Use the regular paste command, Ctrl+V, to efficiently paste the most recent clipboard content.

Clipboards are saved between instances of the IDE sessions and will not become lost, even if you restart Windows.

Line and Block Comments

In a C file, line comments are framed using two diagonal characters (//), a format which is derived from C++. Block comments are framed by the character sets "/*" and "*/" , which are inherent in C. In an assembly file, line comments are framed using semicolons (;) and there are no character sets for block comments.

The ZDS II editor provides two hot keys to comment or un-comment a line or a block of code; each is described below.

Line Comment in C file

To comment or un-comment a single line of text, place the text caret anywhere on the appropriate line, then press Ctrl+Q.

To comment or un-comment multiple lines of text, select all of the appropriate lines and press Ctrl+Q.

Note: Ctrl+Q does not un-comment lines that don't start with line comment characters at the first column.

Block Comments in a C File

To comment a block of code, select the block and press Ctrl+M.

Note: You cannot un-comment a commented block of text using this hotkey. Instead, undo the change by pressing Ctrl+Z.

Line Comments in an Assembly File

To comment or un-comment a single line of text, place the text caret anywhere on the appropriate line, then press either Ctrl+Q or Ctrl+M.

To comment or un-comment multiple lines of text, select all of the appropriate lines and then press either Ctrl+Q or Ctrl+M.

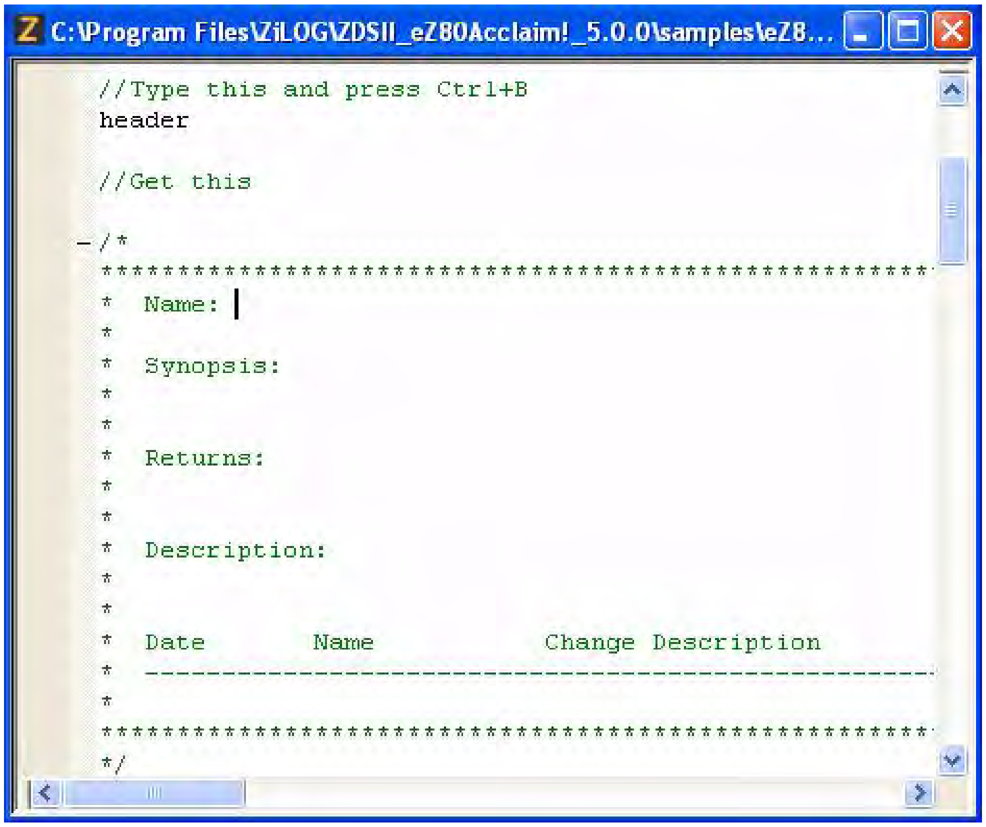
Abbreviations and Expansions

An abbreviation is a shortened word assigned to an expansion. An expansion is a text string that can be a single line of code, a code block or a comment section such as a function/file header; an expansion can even be a combination of code and comments.

ZDS II allows you to define an Abbreviation and Expansion pair (See the Adding an Abbreviation section on page 124) and to expand an abbreviation to its expansion by simply pressing a hotkey. This feature improves your productivity by saving the time involved in typing repeating code blocks and comment sections.

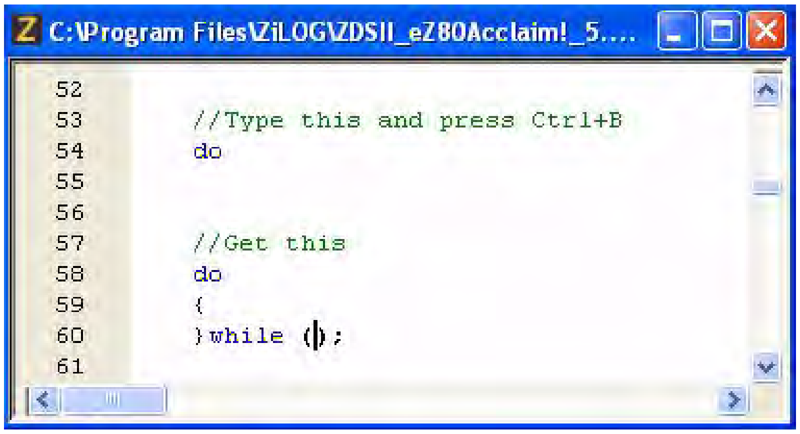
To expand an abbreviation, type an abbreviation at the appropriate location of your code in the editor, then press Ctrl+B. The abbreviation is not case-sensitive.

Figures 95 and 96 show two examples of abbreviation and expansion.


C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_5.0.0\samples\eZ8...

```
//Type this and press Ctrl+B
header

//Get this

/*
*****************************************************************
*   Name:
*
*   Synopsis:
*
*
*   Returns:
*
*
*   Description:
*
*
*   Date         Name              Change Description
*   -------------------------------------------------------------
*
*****************************************************************
*/
```

Figure 95. Abbreviation Example 1



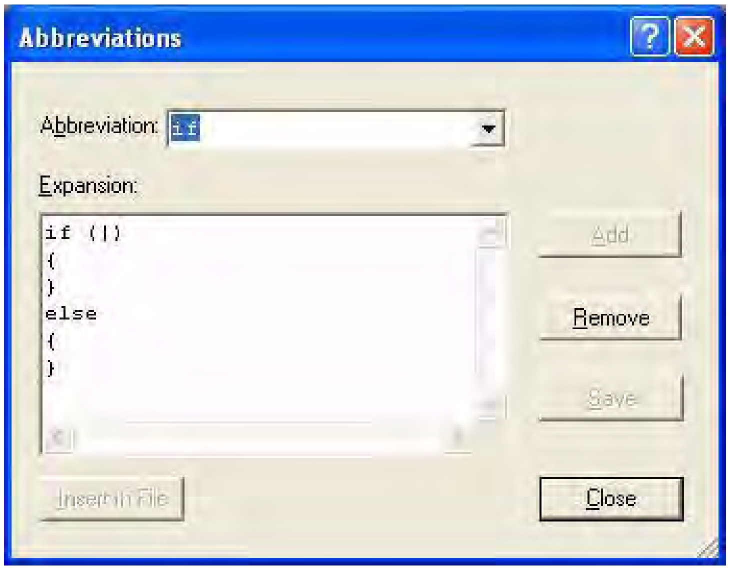
```
//Type this and press Ctrl+B
do


//Get this
do
{
}while ();
```

Figure 96. Abbreviation Example 2

Abbreviation and Expansion pairs are saved between instances of the IDE sessions and will not become lost, even if you restart Windows.

Note: ZDSII provides some common Abbreviation and Expansion pairs by default that help you to learn various generic syntactical notations applicable to Expansion text. You are free to modify or remove them.

To manage abbreviation and expansion pairs, select **Manage Abbreviations** from the **Edit** menu. The **Abbreviations** dialog box is displayed as shown in Figure 97.



```
if
if (|)
{
}
else
{
}
```

Figure 97. Abbreviations Dialog Box

In the **Abbreviations** dialog box, you can perform the following operations; each is linked to below and further described in this section.

- Adding an Abbreviation
- Modifying an Abbreviation
- Removing an Abbreviation
- Expanding an Abbreviation

Adding an Abbreviation

To add a new abbreviation and expansion pair, perform the following steps:

1. Select **Manage Abbreviations** from the **Edit** menu to display the **Abbreviations** dialog box.
2. Enter an appropriate abbreviation in the text entry combo box labeled **Abbreviation** (see Figure 97). You cannot enter a space nor special symbols except for an underscore (_) as part of the name.
3. Press the Tab key. The multi-line text edit box, labeled **Expansion**, displays your keyboard entries.
4. Type the expansion, or use another editor (such as Notepad) from which to copy and paste into the Expansion edit box.

- To indent a block of code, precede the code block with the two-character string `\t`.
- To add a blank line, enter the two-character string `\n`.
- To place the text caret, use the pipe (|) character, which helps you to continue typing within the expanded text. To include a literal pipe character, enter two pipe characters (||). If no pipe character is added, the text caret is moved to the end of the expanded text.

5. Click the **Add** button to add the new abbreviation and expansion pair.
6. Click the **Close** button to close the dialog.

Modifying an Abbreviation

To modify the expansion of an abbreviation, perform the following steps:

1. Select **Manage Abbreviations** from the **Edit** menu to display the **Abbreviations** dialog box.
2. Perform either of the following two actions:
   - Enter the abbreviation name in the **Abbreviation** combo box
   - Click the **Abbreviation** combo box down arrow and select the appropriate abbreviation from the pop-up list. The **Expansion** box displays the expansion of the selected abbreviation.
3. Modify the expansion by performing either of the following actions:
   - To indent a block of code, precede the code block with the two-character string `\t`.
   - To add a blank line, enter the two-character string `\n`.

   To place the text caret, use the pipe (|) character, which helps you to continue typing within the expanded text string. To include a literal pipe character enter two pipe characters. If no pipe character is added, the text caret is moved to the end of the expanded text.
4. Click the **Save** button.
5. Click the **Close** button to close the dialog.

Note: The abbreviation name cannot be modified.

Removing an Abbreviation

To remove an abbreviation and expansion pair, perform the following steps:

1. Select **Manage Abbreviations** from the **Edit** menu to display the **Abbreviations** dialog box.
2. Perform either of the following two actions:
   – Enter the abbreviation in the **Abbreviation** combo box.
   – Click the Abbreviation combo box down arrow and select the appropriate abbreviation name from the pop-up list. The **Expansion** box displays the expansion of the selected abbreviation.
3. Click the **Remove** button to delete the abbreviation.
4. Click the **Close** button to close the dialog.

Note: You cannot restore an abbreviation after it is deleted.

Expanding an Abbreviation

There are two ways to expand an abbreviation; one way is with a hotkey, the other is via the **Manage Abbreviations** dialog box, as this section describes.

Using the hotkey

1. Move the text caret to the appropriate location in your code.
2. Using your keyboard, enter the appropriate abbreviation.
3. Press Ctrl+B to expand the abbreviation to its expansion at the caret position.

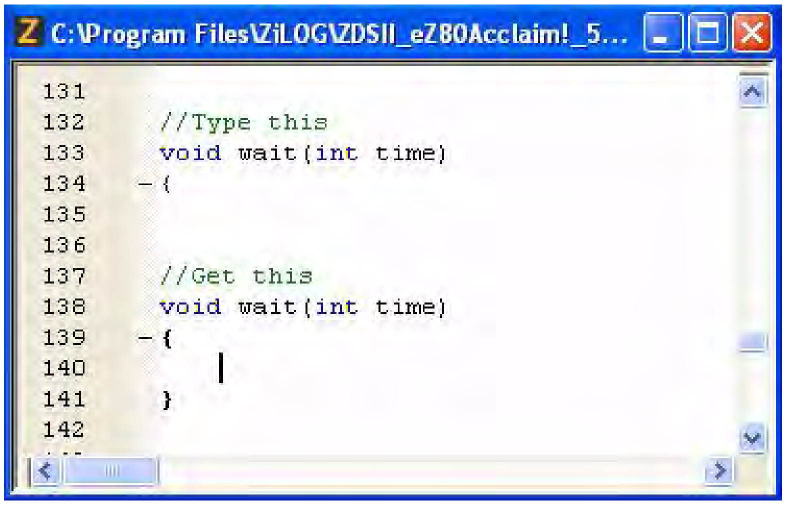
Using the Manage Abbreviations Dialog Box

1. Select **Manage Abbreviations** from the **Edit** menu to display the **Abbreviations** dialog box.
2. Perform either of the following two actions:
   – Enter the abbreviation in the **Abbreviation** combo box.
   – Click the Abbreviation combo box down arrow and select the appropriate abbreviation name from the pop-up list. The **Expansion** box displays the expansion of the selected abbreviation.
3. Click the **Insert in File** button. The expansion of the selected abbreviation is inserted at the text caret position.

Note: Because the **Abbreviations** dialog is *non-modal*, you can use **Insert in File** to insert the expansions of abbreviations across many files that belong to a project. Essentially, while the **Abbreviations** dialog remains open, you are allowed to open any file, move the text caret anywhere in that file, and insert the selected abbreviation.

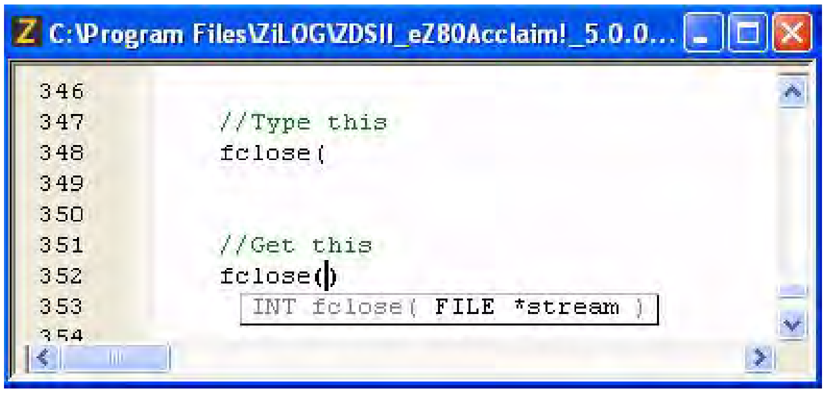
Auto Insertion of Braces and Quotes

When typing an opening symbol such as a left parenthesis “(”, left brace “{”, left bracket “[”, single left quote “'” or double left quote “"”, its matching closing symbol is automatically inserted and the text caret remains between the characters, as shown in Figures 98 and 99.



```
//Type this
void wait(int time)
{


//Get this
void wait(int time)
{

}
```

Figure 98. Auto Insertion of Closing Brace



```
//Type this
fclose(


//Get this
fclose()
 INT fclose( FILE *stream )
```

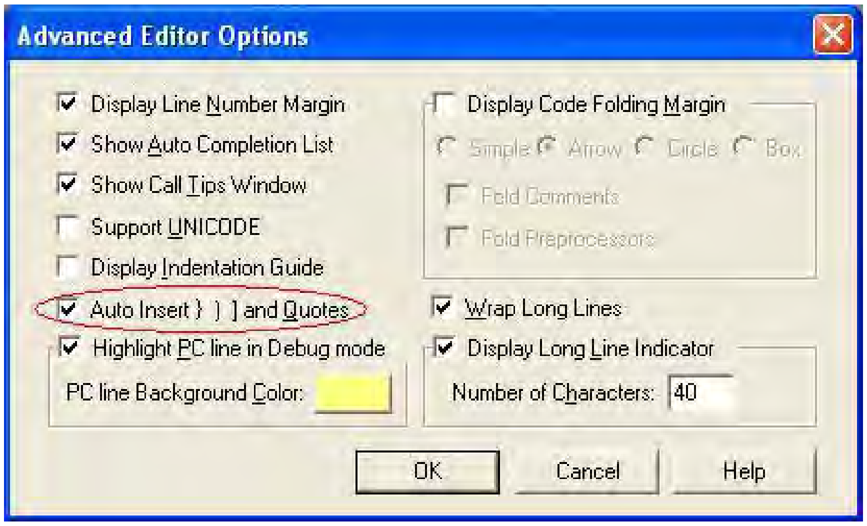
Figure 99. Auto Insertion of Closing Parenthesis

Press Delete or Backspace to delete an autoinserted character.

Note: Closing characters are not inserted inside comments and strings.

To enable or disable the auto insertion of the closing symbols })] ' and ", perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.
3. Click the **Advanced Editor Options** button. The **Advanced Editor Options** dialog box is displayed.
4. Select the **Auto Insert })] and Quotes** checkbox to enable autoinsertion of closing braces and quotes (see Figure 100). Disable this option if you prefer to manually enter all of the closing characters. This option is deselected by default.

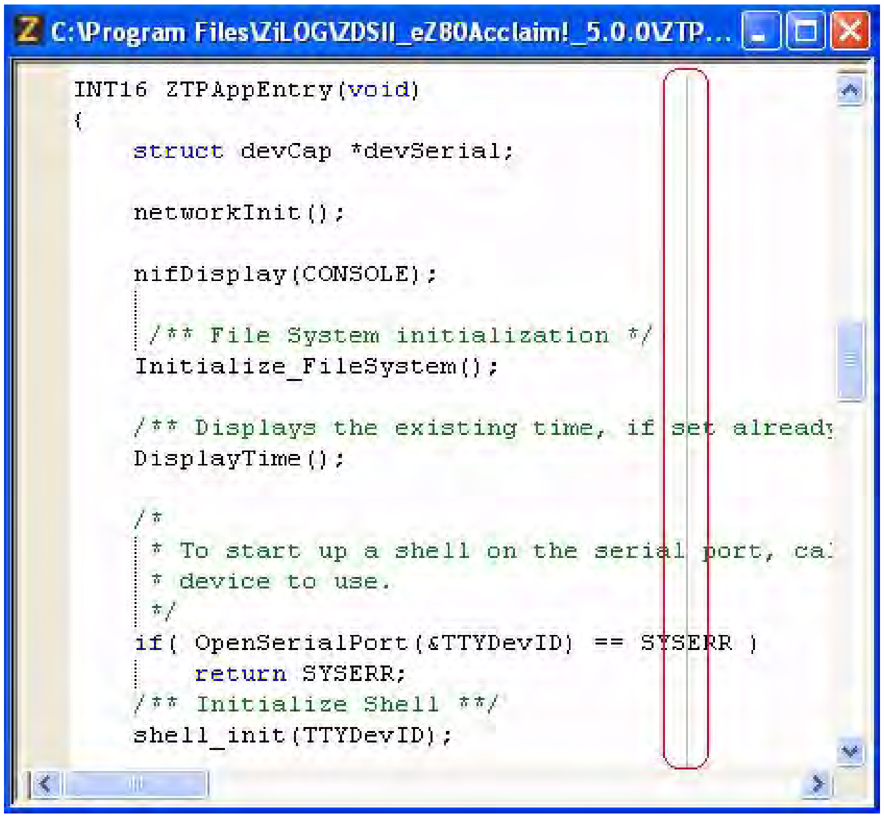

Advanced Editor Options

- ✓ Display Line Number Margin
- ✓ Show Auto Completion List
- ✓ Show Call Tips Window
- ☐ Support UNICODE
- ☐ Display Indentation Guide
- ✓ Auto Insert })] and Quotes
- ✓ Highlight PC line in Debug mode
  - PC line Background Color: 🟨
- ☐ Display Code Folding Margin
  - Simple [● Arrow] Circle Box
  - ☐ Fold Comments
  - ☐ Fold Preprocessors
- ✓ Wrap Long Lines
- ✓ Display Long Line Indicator
  - Number of Characters: 40

Figure 100. Advance Editor Options—Auto Insertion of Brace and Quotes

Long Line Indicator

A Long Line indicator is a vertical line that appears in the editor to mark a fixed number of character columns. Use this indicator to wrap all of your long lines manually for better readability. See Figure 101.



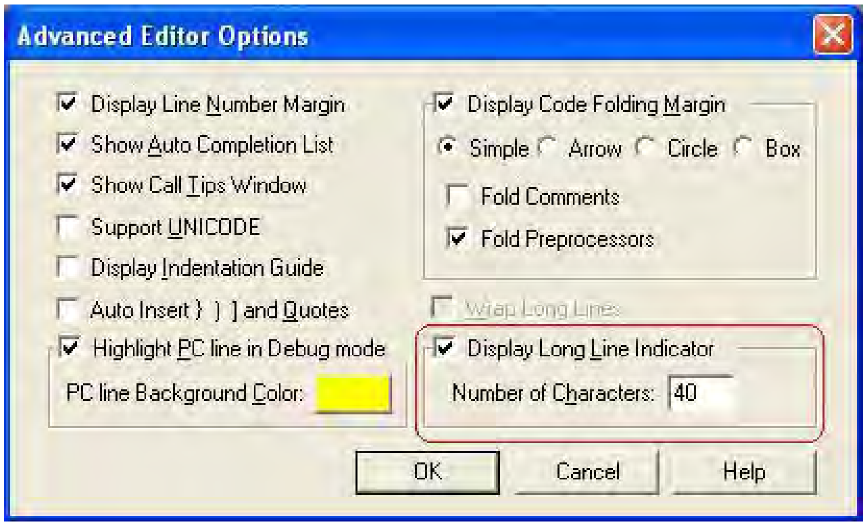
```
INT16 ZTPAppEntry(void)
{
    struct devCap *devSerial;

    networkInit();

    nifDisplay(CONSOLE);

     /** File System initialization */
    Initialize_FileSystem();

    /** Displays the existing time, if set already
    DisplayTime();

    /*
     * To start up a shell on the serial port, cal
     * device to use.
     */
    if( OpenSerialPort(&TTYDevID) == SYSERR )
        return SYSERR;
    /** Initialize Shell **/
    shell_init(TTYDevID);
```

Figure 101. Long Line Indicator

Note: The Long Line indicator works well only for monospaced fonts (for example: the Courier New font).

To show or hide the Long Line Indicator, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.
3. Click the **Advanced Editor Options** button. The **Advanced Editor Options** dialog box is displayed.
4. Select the **Display Long Line Indicator** checkbox to display the long line indicator. This option is deselected by default; see Figure 102.



☑ Display Line Number Margin
☑ Show Auto Completion List
☑ Show Call Tips Window
☐ Support UNICODE
☐ Display Indentation Guide
☐ Auto Insert })] and Quotes
☑ Highlight PC line in Debug mode
PC line Background Color: 🟨
☑ Display Code Folding Margin
● Simple ○ Arrow ○ Circle ○ Box
☐ Fold Comments
☑ Fold Preprocessors
☐ Wrap Long Lines
☑ Display Long Line Indicator
Number of Characters: 40

Figure 102. Advance Editor Options—Long Line Indicator Settings

5. To move the indicator to a particular column, enter the number of character columns in the **Number of Characters** text entry box. The allowed range of values is between 1 and 999. It is set to 80 by default.

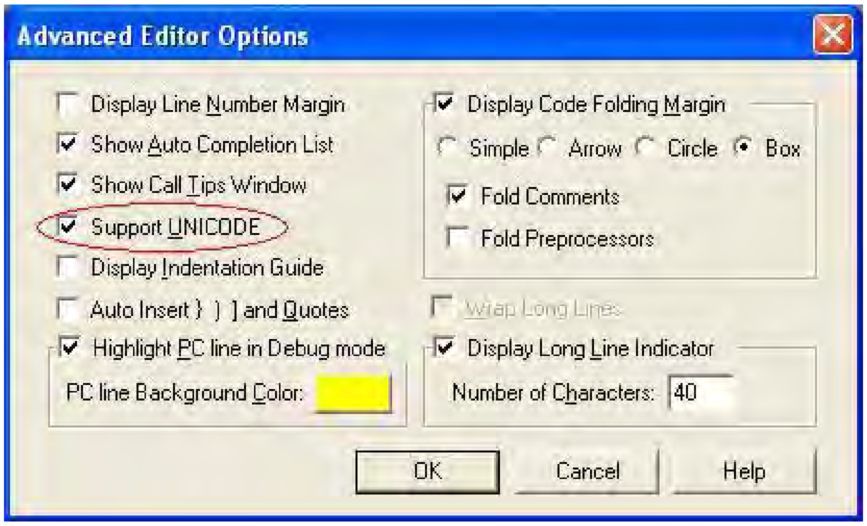
UNICODE Support

You can use a non-English language that is supported by UNICODE in the comments and strings to better document your code in your native language. You can type the sentences in your native language script in a UNICODE-supported editor, such as Microsoft Word, and copy/paste them into the ZDS II editor.

Note: You cannot use a bilingual keyboard to enter your native language scripts directly into the ZDS II editor.

To enable or disable the UNICODE support, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.
3. Click the **Advanced Editor Options** button. The **Advanced Editor Options** dialog box is displayed.
4. Select **Support UNICODE** checkbox to enable UNICODE support. This option is deselected by default.



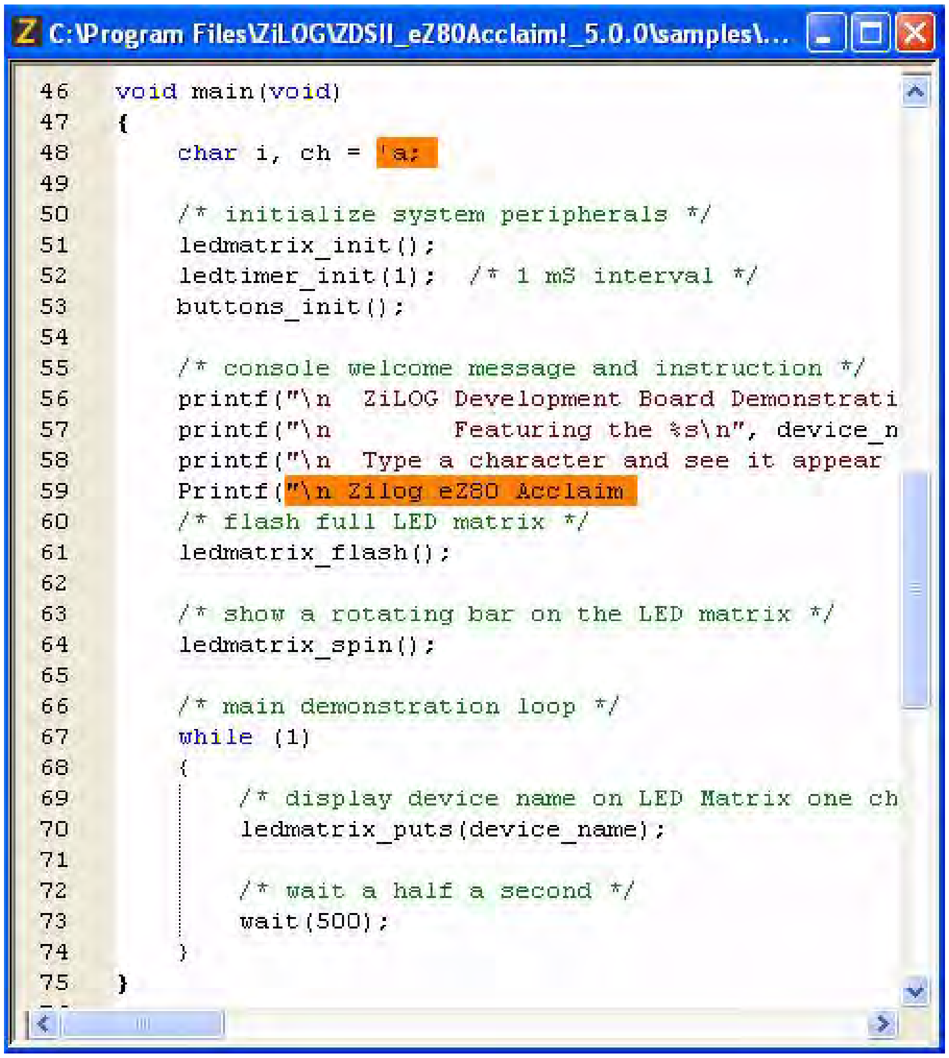
[ ] Display Line Number Margin
[✓] Show Auto Completion List
[✓] Show Call Tips Window
[✓] Support UNICODE
[ ] Display Indentation Guide
[ ] Auto Insert })] and Quotes
[✓] Highlight PC line in Debug mode
PC line Background Color: 🟨
[✓] Display Code Folding Margin
Simple Arrow Circle [✓ Box]
[✓] Fold Comments
[ ] Fold Preprocessors
[ ] Wrap Long Lines
[✓] Display Long Line Indicator
Number of Characters: 40

Figure 103. Advance Editor Options—Support UNICODE

Auto Syntax Styler

The Auto Syntax Styler displays the language constructs of your code in different colors. It enables you to read your code more easily by providing visual color cues as to the structure and purpose of the code. It also helps you to avoid any typing mistakes by employing the basic building blocks of the language constructs such as keywords, preprocessor reserved words, comments, etc.

Figure 104 shows an incomplete string and char highlighted by the Auto Syntax Styler, which allows you to easily correct mistakes in the code.

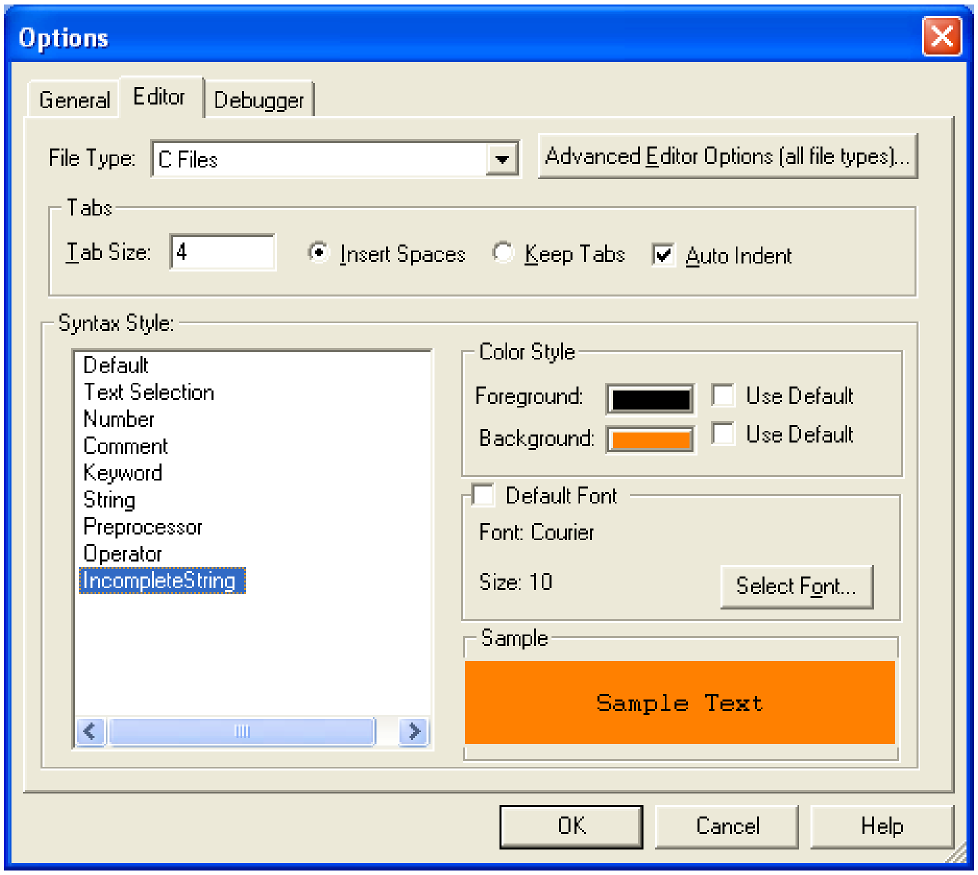

C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_5.0.0\samples\...

```
void main(void)
{
    char i, ch = 'a;

    /* initialize system peripherals */
    ledmatrix_init();
    ledtimer_init(1);  /* 1 mS interval */
    buttons_init();

    /* console welcome message and instruction */
    printf("\n  ZiLOG Development Board Demonstrati
    printf("\n        Featuring the %s\n", device_n
    printf("\n  Type a character and see it appear
    Printf("\n Zilog eZ80 Acclaim
    /* flash full LED matrix */
    ledmatrix_flash();

    /* show a rotating bar on the LED matrix */
    ledmatrix_spin();

    /* main demonstration loop */
    while (1)
    {
        /* display device name on LED Matrix one ch
        ledmatrix_puts(device_name);

        /* wait a half a second */
        wait(500);
    }
}
```

Figure 104. Auto Syntax Styler

The colors used by the editor’s Auto Syntax Styler are completely configurable. To change the color of a language construct, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.



File Type: C Files

Tabs

Tab Size: 4 [✓ Insert Spaces] Keep Tabs [✓ Auto Indent]

Syntax Style:

Default
Text Selection
Number
Comment
Keyword
String
Preprocessor
Operator
[✓ IncompleteString]

Color Style

Foreground: ■ Use Default

Background: 🟧 Use Default

Default Font

Font: Courier

Size: 10

Sample

Sample Text

Figure 105. Options Dialog Box—Editor Tab

3. Select an appropriate color from the **Color Style** list box, and make sure that the **Use Default** checkboxes are deselected.
4. Click the **Foreground** or **Background** color to display the **Color** dialog box (see Figure 106). In the **Color** dialog box, select the appropriate color.

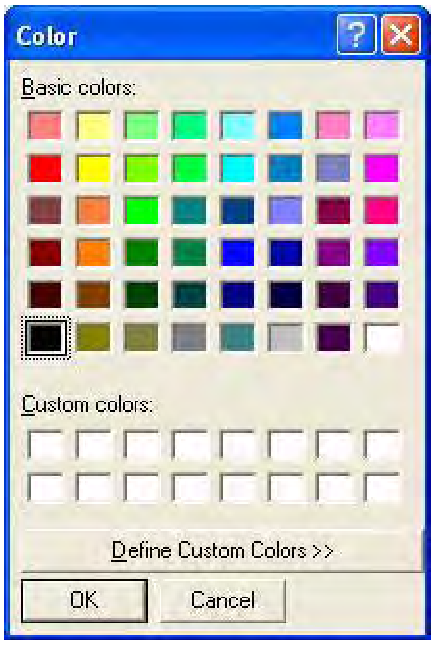


Figure 106. Color Dialog Box

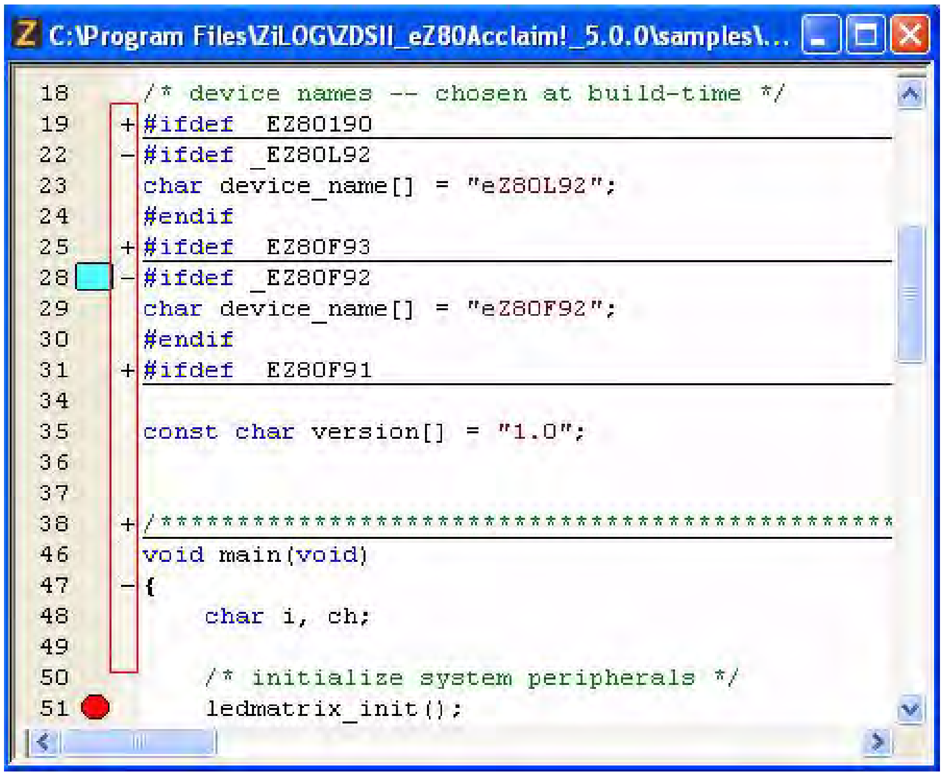
Note: If you want to use the default foreground or background color for the selected item, select the **Use Default** checkbox.

5. Click **OK** to close the **Color** dialog box.
6. Click **OK** to close the **Options** dialog box.

Code Folding Margin

Code folding allows you to selectively hide and display various sections of the code as a part of your routine editing operations. It also helps you to understand and analyze the code faster by letting you concentrate on a particular section of complex or problematic code and ignore all other sections.

The code folding margin displays the fold and unfold symbols, as shown in Figure 107.



```
/* device names -- chosen at build-time */
#ifdef  EZ80190
#ifdef _EZ80L92
char device_name[] = "eZ80L92";
#endif
#ifdef  EZ80F93
#ifdef _EZ80F92
char device_name[] = "eZ80F92";
#endif
#ifdef  EZ80F91

const char version[] = "1.0";


/**************************************************
void main(void)
{
    char i, ch;

    /* initialize system peripherals */
    ledmatrix_init();
```

Figure 107. Code Folding Margin

The folding points of a document are based on the hierarchical structure of the document contents. In C programming code, the document hierarchy is determined by the brace characters, conditional preprocessor macros, commented code block and file/function header.

Code folding is available only with C and HTML files; it is not available with assembly files.

Folding and unfolding the code does not change the content or structure of the code.

To contract or expand single foldable block of code, click the fold point, or press `Ctrl` + = while the text caret is positioned on the fold pointing line of code or within the block of code.

To contract or expand all foldable blocks of code, click anywhere on the fold margin while pressing the Ctrl key.

To show or hide the folding margin of a codeset, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.

2. Click the **Editor** tab.
3. Click the **Advanced Editor Options** button. The **Advanced Editor Options** dialog box is displayed.

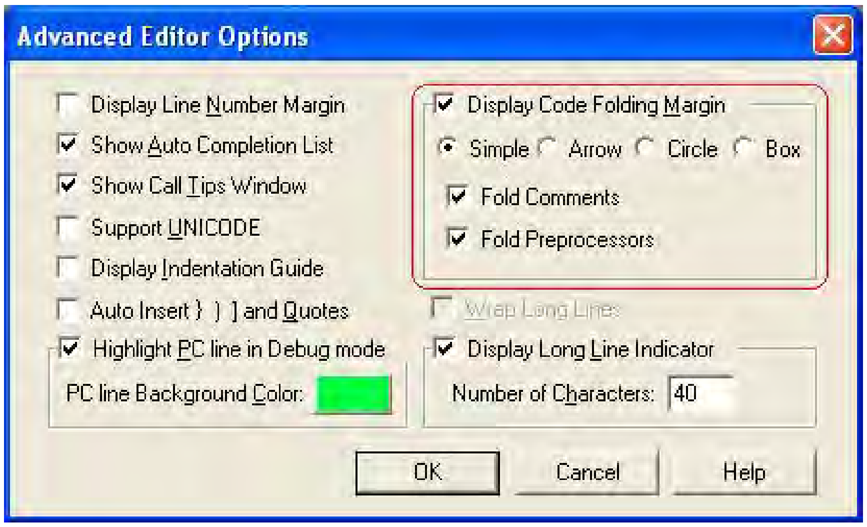

Advanced Editor Options
Display Line Number Margin
Show Auto Completion List
Show Call Tips Window
Support UNICODE
Display Indentation Guide
Auto Insert })] and Quotes
Highlight PC line in Debug mode
PC line Background Color:
Display Code Folding Margin
Simple
Arrow
Circle
Box
Fold Comments
Fold Preprocessors
Wrap Long Lines
Display Long Line Indicator
Number of Characters: 40
OK
Cancel
Help

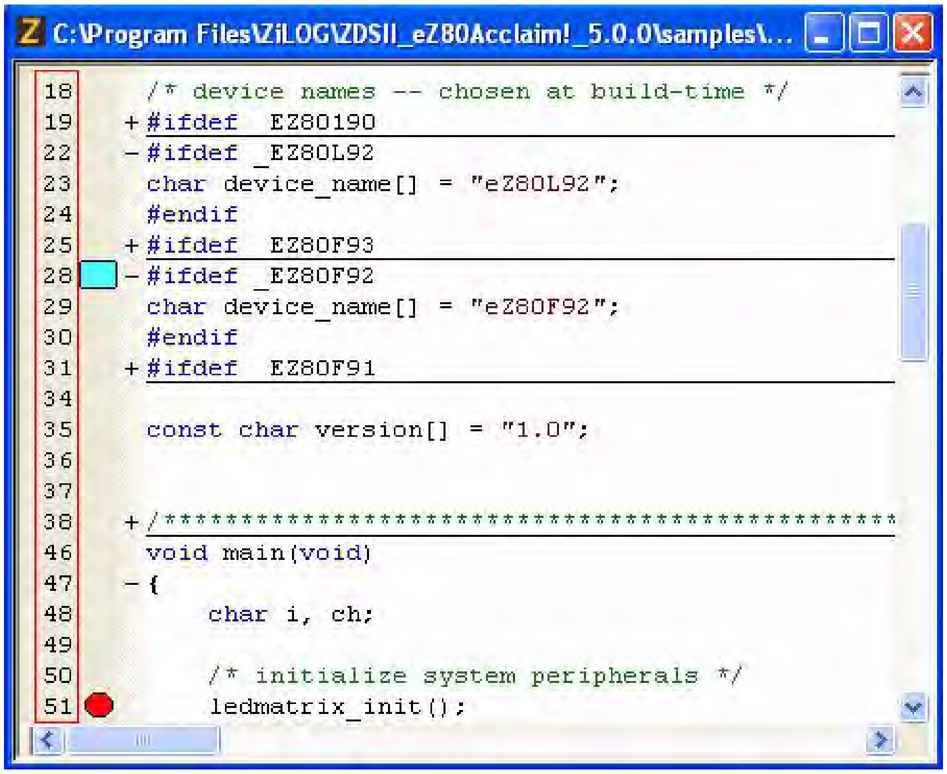
Figure 108. Advance Editor Options—Display Code Folding Margin

4. Select the **Display Code Folding Margin** checkbox to display the code folding margin. This option is selected by default.
5. Select any one of the fold symbol option buttons **Simple**, **Arrow**, **Circle** or **Box** to change the look and feel of the fold points. By default, **Simple** is selected.
6. Select the **Fold Comments** checkbox to display fold points for all of the commented lines of code and text.
7. Select the **Fold Preprocessors** checkbox to display fold points for all of the preprocessor conditional macro statements.

Note: If you enable the Code Folding Margin function, the Wrap long Lines function will be disabled automatically, and vice-versa.

Line Number Margin

Line numbers can orient you when working in a long file. It allows you to quickly navigate to a specific line of code or to identify easily a given line of code. It would also be helpful to have the line numbers appear in the margin to aid debugging.



```
18      /* device names -- chosen at build-time */
19    + #ifdef  EZ80190
22    - #ifdef  _EZ80L92
23      char device_name[] = "eZ80L92";
24      #endif
25    + #ifdef  EZ80F93
28    - #ifdef  _EZ80F92
29      char device_name[] = "eZ80F92";
30      #endif
31    + #ifdef  EZ80F91
34
35      const char version[] = "1.0";
36
37
38    + /*********************************************
46      void main(void)
47    - {
48          char i, ch;
49
50          /* initialize system peripherals */
51          ledmatrix_init();
```

Figure 109. Line Number Margin

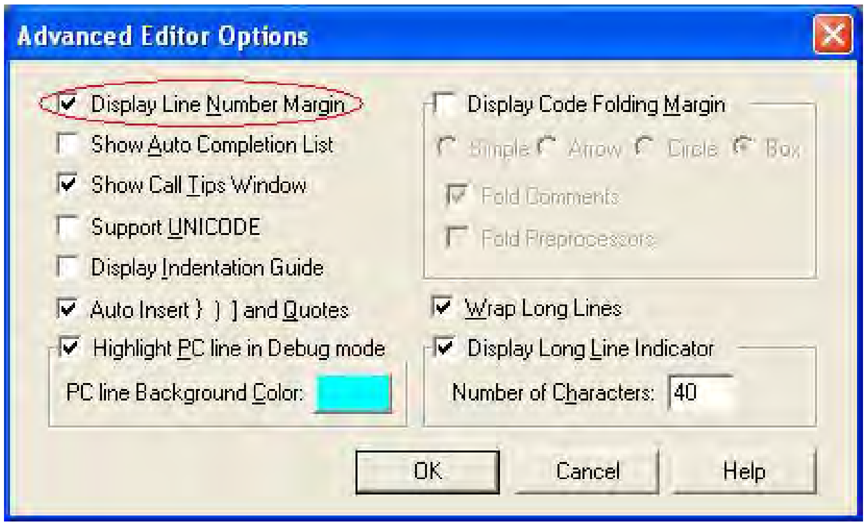
You can select a line of text by clicking the associated line number on the line number margin.

You can select multiple lines either by clicking and dragging the mouse on the line number margin or by clicking the appropriate start line number and with the Shift key pressed, clicking the appropriate end line number.

You can select all of the text in the document by clicking on the line number margin with Ctrl key pressed.

To show or hide the line number margin, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.
3. Click the **Advanced Editor Options** button. The **Advanced Editor Options** dialog box is displayed.
4. Select the **Display Line Number Margin** checkbox to display the line number margin. This option is deselected by default.



Advanced Editor Options

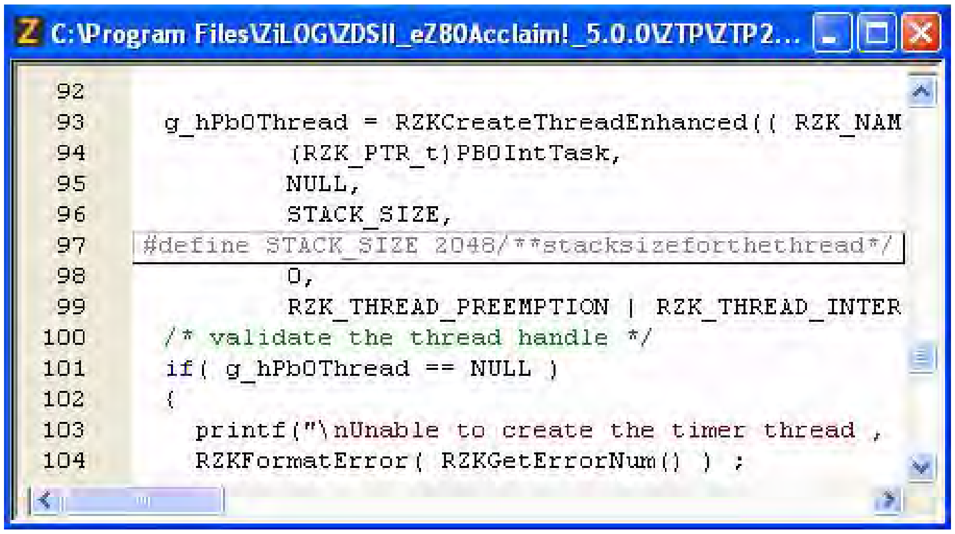
- [✓] Display Line Number Margin
- [ ] Show Auto Completion List
- [✓] Show Call Tips Window
- [ ] Support UNICODE
- [ ] Display Indentation Guide
- [✓] Auto Insert })] and Quotes
- [✓] Highlight PC line in Debug mode
  - PC line Background Color: 🟦
- [ ] Display Code Folding Margin
  - Simple / Arrow / Circle / [● Box]
  - [✓] Fold Comments
  - [ ] Fold Preprocessors
- [✓] Wrap Long Lines
- [✓] Display Long Line Indicator
  - Number of Characters: 40

Figure 110. Advance Editor Options—Display Line Number Margin

Type Info Tips

The Type Info Tips window is a hovering and short-lived small window that pops up to display the type of an identifier in your code whenever you move the mouse pointer over the identifier and let it remain there for more than a fraction of a second. The tips that display in these small windows can help you to read and write code faster as well as to locate hard-to-find errors. See the example in Figure 111.

Type Info Tips becomes available for virtually all of the variables and functions declared or defined in your project code and in all standard include files. You are not required to build a project for Type Info Tips to become available.

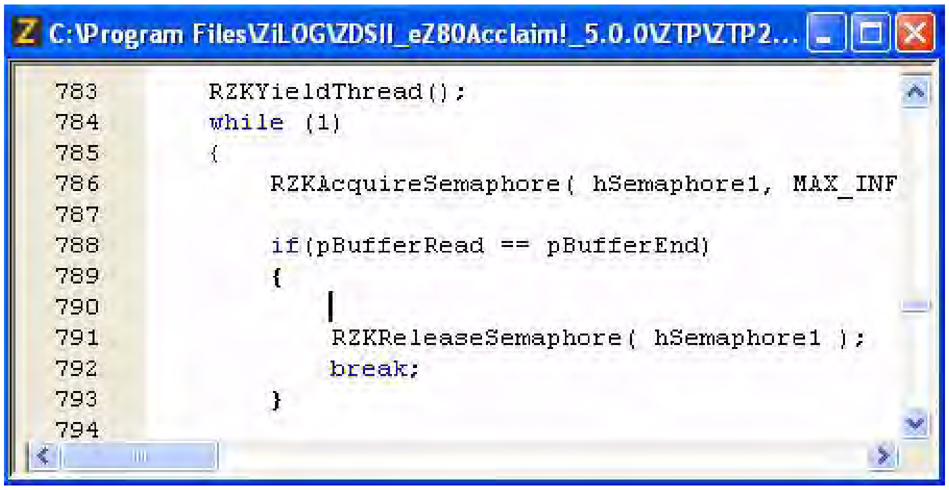


```
    g_hPbOThread = RZKCreateThreadEnhanced( ( RZK_NAM
            (RZK_PTR_t)PBOIntTask,
            NULL,
            STACK_SIZE,
#define STACK_SIZE 2048/**stacksizeforthethread*/
            0,
            RZK_THREAD_PREEMPTION | RZK_THREAD_INTER
    /* validate the thread handle */
    if( g_hPbOThread == NULL )
    {
      printf("\nUnable to create the timer thread ,
      RZKFormatError( RZKGetErrorNum() ) ;
```

Figure 111. Type Info Tips

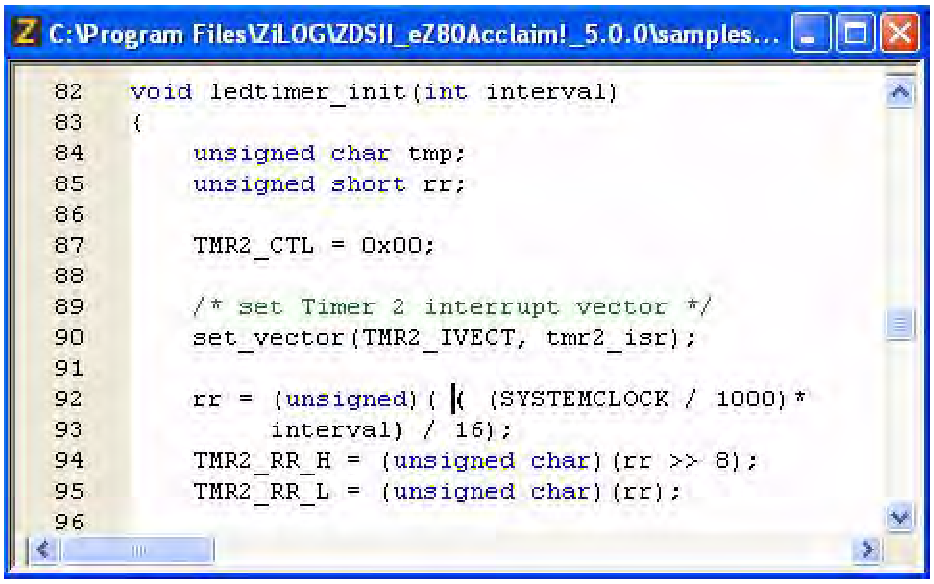
Highlighting and Finding Matched Braces

Move your text caret just inside a pair of braces { } or parentheses ( ) or square brackets [ ] and observe the matching pair as it becomes highlighted. This highlighting feature helps you to locate a block, function or expression scope easily. See the examples in Figures 112 and 113.



```
    RZKYieldThread();
    while (1)
    {
        RZKAcquireSemaphore( hSemaphore1, MAX_INF

        if(pBufferRead == pBufferEnd)
        {
            |
            RZKReleaseSemaphore( hSemaphore1 );
            break;
        }

```

Figure 112. Highlighting Matching Braces



```
void ledtimer_init(int interval)
{
    unsigned char tmp;
    unsigned short rr;

    TMR2_CTL = 0x00;

    /* set Timer 2 interrupt vector */
    set_vector(TMR2_IVECT, tmr2_isr);

    rr = (unsigned) ( (  (SYSTEMCLOCK / 1000) *
          interval) / 16);
    TMR2_RR_H = (unsigned char)(rr >> 8);
    TMR2_RR_L = (unsigned char)(rr);

```

Figure 113. Highlighting Matching Parentheses

When braces or parentheses are nested, the innermost pair containing the text caret is highlighted.

You can also locate the matching brace by pressing Ctrl+E; this keyboard shortcut is especially useful when the braces are extended over multiple pages. Place your text caret anywhere in between the braces or parentheses and then press Ctrl+E to move to the closing or opening brace or parenthesis, respectively. Use this hotkey to quickly jump between opening and closing braces or parentheses.

To select content within matching braces or parentheses, place the text caret anywhere in between the braces or parentheses, respectively, then press Ctrl+Shift+E.

Matching Preprocessor Conditional Macros

Source code is often grouped between compiler preprocessor statements. The ZDS II editor will allow you to move from inside a conditional statement to the enclosing preprocessor statements.

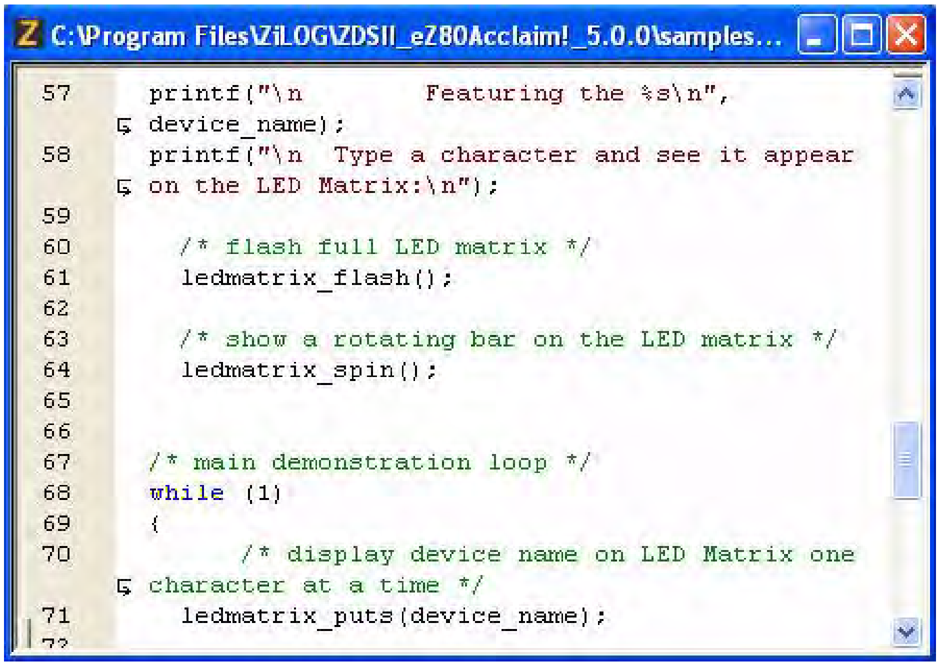
Move the text cursor to the line of a preprocessor conditional statement or to a line that is enclosed by preprocessor conditional statements, then perform either of the following actions:

- Press Ctrl+K to find the matching preprocessor conditional statements that exist forward or backward

- Press Ctrl+Shift+K to select the entire text within the matching preprocessor conditional statements and the conditional statements

Wrap Long Lines

When working on text strings that extend beyond 80 characters, the Wrap Long Lines function can become extremely useful. With this feature turned on, it will not be necessary to continually scroll horizontally, because all long lines will be wrapped to fit the size of the editing area. The editor displays a wrapping symbol at the beginning of all wrapped lines, as shown in Figure 114.

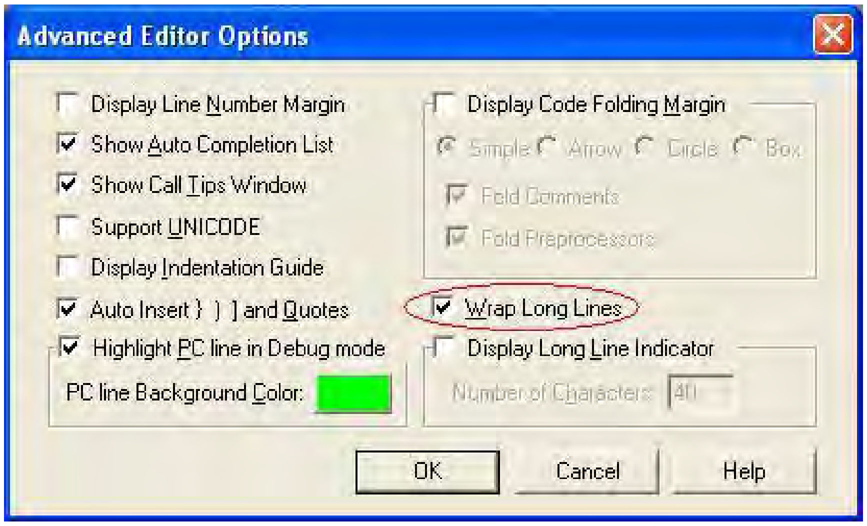


```
   printf("\n        Featuring the %s\n",
device_name);
   printf("\n  Type a character and see it appear
on the LED Matrix:\n");

     /* flash full LED matrix */
     ledmatrix_flash();

     /* show a rotating bar on the LED matrix */
     ledmatrix_spin();


   /* main demonstration loop */
   while (1)
   {
         /* display device name on LED Matrix one
character at a time */
     ledmatrix_puts(device_name);
```

Figure 114. Wrapping Long Lines

To enable or disable the Wrap Long Lines function, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.
3. Click the **Advanced Editor Options** button. The **Advanced Editor Options** dialog box is displayed, as shown in Figure 115.
4. Select the **Wrap Long Lines** checkbox to wrap all long lines. This option is deselected by default; see Figure 115.



- [ ] Display Line Number Margin
- [✓] Show Auto Completion List
- [✓] Show Call Tips Window
- [ ] Support UNICODE
- [ ] Display Indentation Guide
- [✓] Auto Insert })] and Quotes
- [✓] Highlight PC line in Debug mode
  - PC line Background Color: 🟩
- [ ] Display Code Folding Margin
  - [✓ Simple] Arrow Circle Box
  - [✓] Fold Comments
  - [✓] Fold Preprocessors
- [✓] Wrap Long Lines
- [ ] Display Long Line Indicator
  - Number of Characters: 40

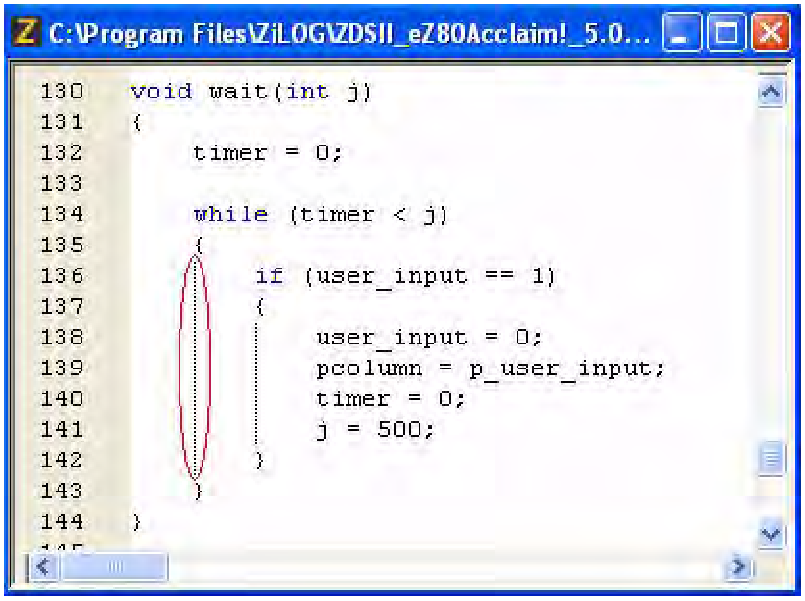
Figure 115. Advance Editor Options—Wrap Long Lines

Note: If you enable the **Wrap Long Lines** feature, the Code Folding Margin will be disabled automatically, and vice versa.

Indentation Guides

Indentation guides are finely-dotted vertical lines that can assist in defining the indentations of code blocks. These indentation guides make it easy to see which constructs line up, especially when they extend over multiple pages.

When you move the text caret in between a matching pair of braces { }, the indentation guide will be highlighted, as shown in Figure 116.

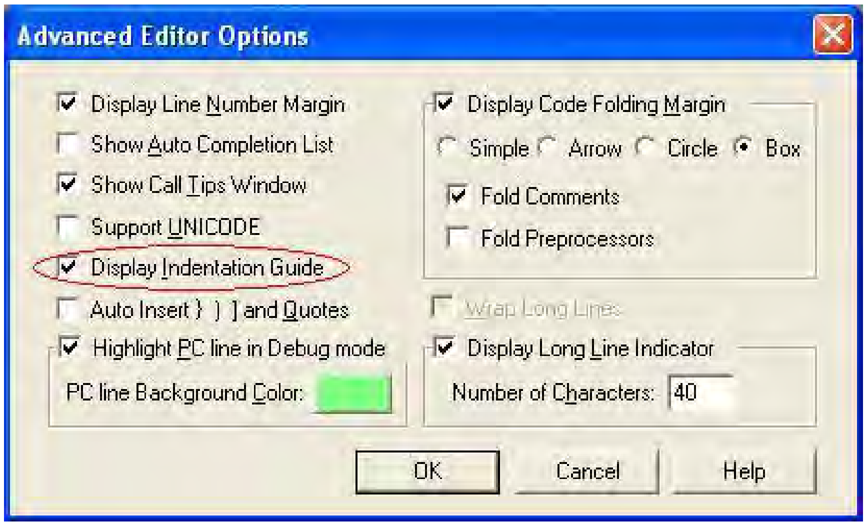


```
void wait(int j)
{
    timer = 0;

    while (timer < j)
    {
        if (user_input == 1)
        {
            user_input = 0;
            pcolumn = p_user_input;
            timer = 0;
            j = 500;
        }
    }
}
```

Figure 116. Indentation Guides

To enable or disable indentation guides, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.
3. Click the **Advanced Editor Options** button. The **Advanced Editor Options** dialog box is displayed.
4. Select the **Display Indentation Guide** checkbox to enable indentation guides. This option is deselected by default. See Figure 117.



[✓] Display Line Number Margin
[ ] Show Auto Completion List
[✓] Show Call Tips Window
[ ] Support UNICODE
[✓] Display Indentation Guide
[ ] Auto Insert })] and Quotes
[✓] Highlight PC line in Debug mode
PC line Background Color: 🟩

[✓] Display Code Folding Margin
Simple Arrow Circle [✓ Box]
[✓] Fold Comments
[ ] Fold Preprocessors
[ ] Wrap Long Lines
[✓] Display Long Line Indicator
Number of Characters: 40

Figure 117. Advance Editor Options—Display Indentation Guide

Zoom In/Out

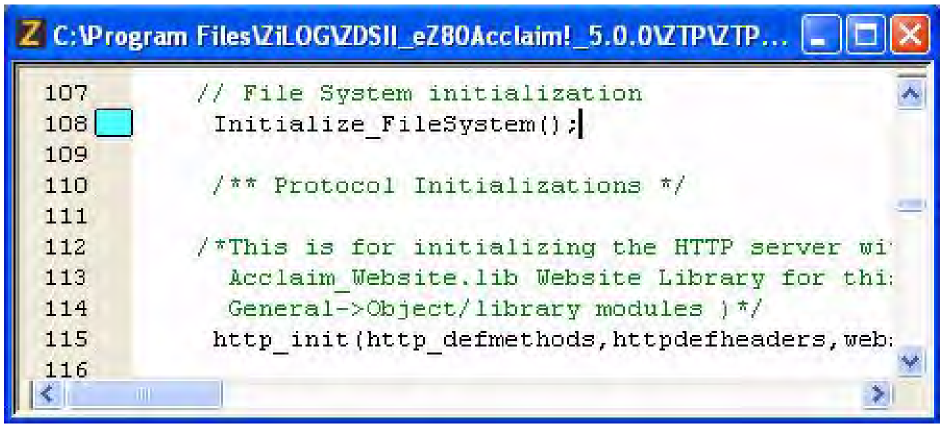
The ZDS II Editor allows you to increase or decrease the magnification of all text without changing font settings. This function can sometimes help to locate hard-to-find bugs in the syntactical notations of your program.

- To magnify text, roll the mouse wheel up while pressing the `Ctrl` key, or press `Ctrl+Num Keyboard +`.
- To shrink text, roll the mouse wheel down while pressing the `Ctrl` key, or press `Ctrl+Num Keyboard -`.
- To reset the text to the original font size, double-click the left mouse button within the editor area while pressing the `Ctrl` key, or press `Ctrl+Num Keyboard /`.

Bookmarks

You can set bookmarks to mark frequently accessed lines in your source file. After a bookmark is set, you can use menus or keyboard commands to move to it. You can remove a bookmark when you no longer need it.

Bookmarks are saved in the project workspace and therefore will be restored in between instances of the IDE sessions.



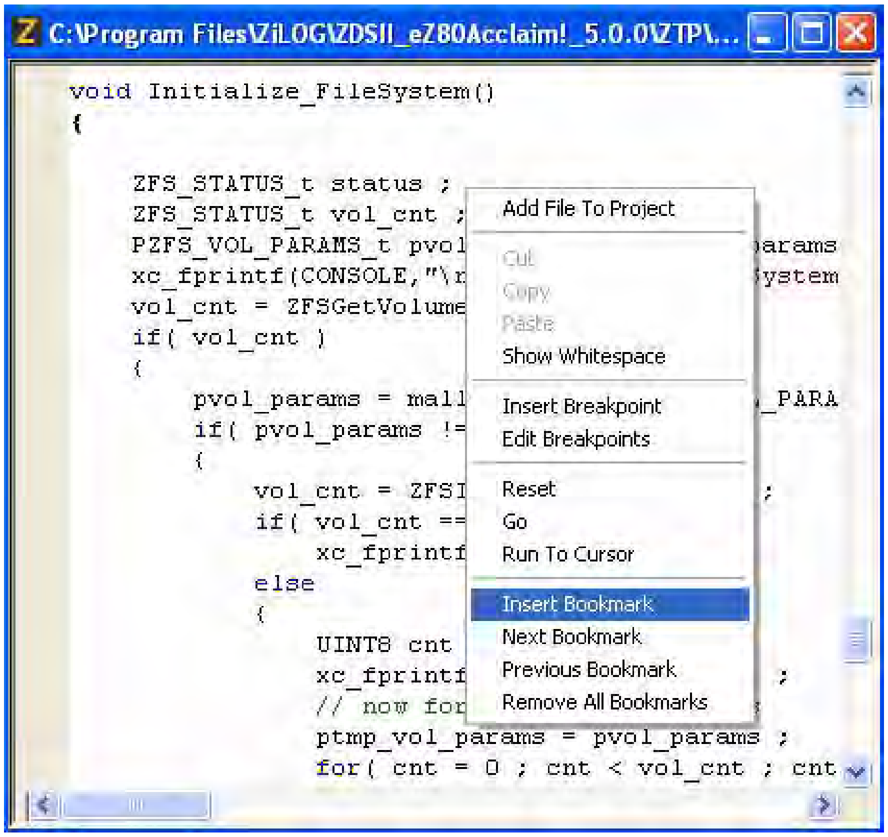
```
// File System initialization
 Initialize_FileSystem();

 /** Protocol Initializations */

/*This is for initializing the HTTP server wi
  Acclaim_Website.lib Website Library for thi
  General->Object/library modules )*/
 http_init(http_defmethods,httpdefheaders,web
```

Figure 118. Bookmark Example

To insert a bookmark, position the cursor on the appropriate line of the active file and perform either of the following actions:

- Right-click in the **Edit** window and select **Insert Bookmark** from the resulting context menu.
- Select **Toggle Bookmark** from the **Edit** menu.

Next, press Ctrl+F2.



```
void Initialize_FileSystem()
{

    ZFS_STATUS_t status ;
    ZFS_STATUS_t vol_cnt ;
    PZFS_VOL_PARAMS_t pvol                    params
    xc_fprintf(CONSOLE,"\r                   System
    vol_cnt = ZFSGetVolume
    if( vol_cnt )
    {
        pvol_params = mall                   _PARA
        if( pvol_params !=
        {
            vol_cnt = ZFSI                    ;
            if( vol_cnt ==
                xc_fprintf
            else
            {
                UINT8 cnt
                xc_fprintf                    ;
                // now for
                ptmp_vol_params = pvol_params ;
                for( cnt = 0 ; cnt < vol_cnt ; cnt
```

Figure 119. Inserting a Bookmark

To remove a bookmark, position the cursor on the line of the active file containing the bookmark to be removed and perform either of the following actions:

- Right-click in the **Edit** window and select **Remove Bookmark** from the resulting context menu.
- Select **Toggle Bookmark** from the **Edit** menu.

Next, press Ctrl+F2.

To remove all bookmarks in the active file, perform either of the following actions:

- Right-click in the **Edit** window and select **Remove All Bookmarks** from the resulting context menu.
- Select **Remove All Bookmarks** from the **Edit** menu.

Next, press Ctrl+Shift+F2.

To jump to the next bookmark in the active file, perform either of the following actions:

- Right-click in the **Edit** window and select **Next Bookmark** from the resulting context menu.
- Select **Next Bookmark** from the **Edit** menu.

Next, press F2.

The search for the bookmark is started from the current cursor position and when a bookmark is not found forward until the end of the file, the search is started from the beginning of the file and will go on until a bookmark is reached. If no bookmark is found, this command has no effect.

To jump to the previous bookmark in the active file, perform either of the following actions:

- Right-click in the **Edit** window and select **Previous Bookmark** from the resulting context menu.
- Select **Previous Bookmark** from the **Edit** menu.

Next, press Shift+F2.

The search for the bookmark is started from the current cursor position and when a bookmark is not found backward until the beginning of the file, the search is started from the end of the file and will go on until a bookmark is reached. If no bookmark is found, this command has no effect.

To select the text up to the next bookmark in the active file, press Alt+F2.

To select the text up to the previous bookmark in the active file, press Alt+Shift+F2.

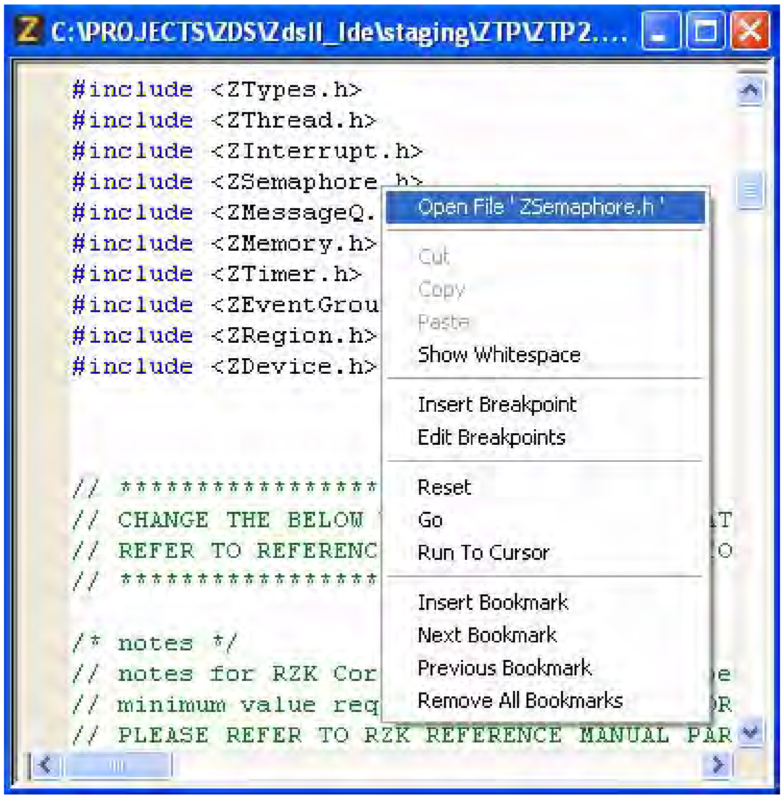
Opening an Include File

Source files more often include header files that contain a *preprocessor include* statement. The ZDS II editor allows you to jump to the include file instantaneously.

To open the include file, right-click the preprocessor include statement and perform one of the following actions:

- Click **Open File '<file_name.h>'** from the resulting context menu.
- Move the text cursor to the line containing the preprocessor include statement, and press Alt+G.

You can jump to any header file that is part of the standard include path and your project directory path. See Figure 120.



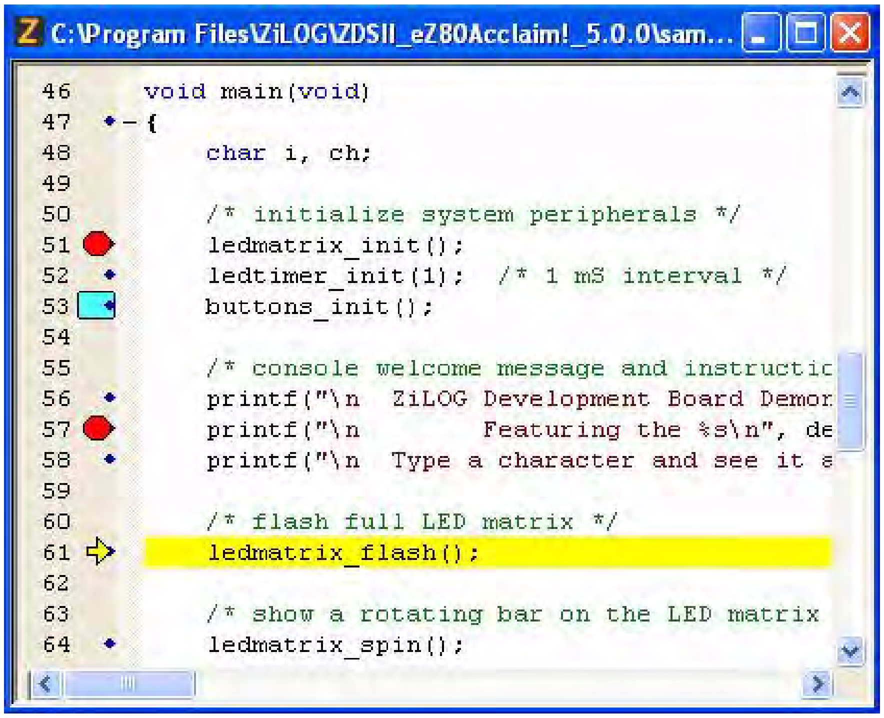
```
#include <ZTypes.h>
#include <ZThread.h>
#include <ZInterrupt.h>
#include <ZSemaphore.h>
#include <ZMessageQ.
#include <ZMemory.h>
#include <ZTimer.h>
#include <ZEventGrou
#include <ZRegion.h>
#include <ZDevice.h>


// *****************
// CHANGE THE BELOW                                   T
// REFER TO REFERENC                                  O
// *****************

/* notes */
// notes for RZK Cor                                  e
// minimum value req                                  R
// PLEASE REFER TO RZK REFERENCE MANUAL PAR
```

Figure 120. Opening an Include File

Note: The opening include file work well only with a project opened in the ZDSII. And with just a file open in the ZDSII, the search is performed only within the file's directory.

Highlighting a Program Counter Line

In debug mode, highlighting the Program Counter line helps you to locate the current PC line easily. Figure 121 shows the PC line highlighted in yellow.

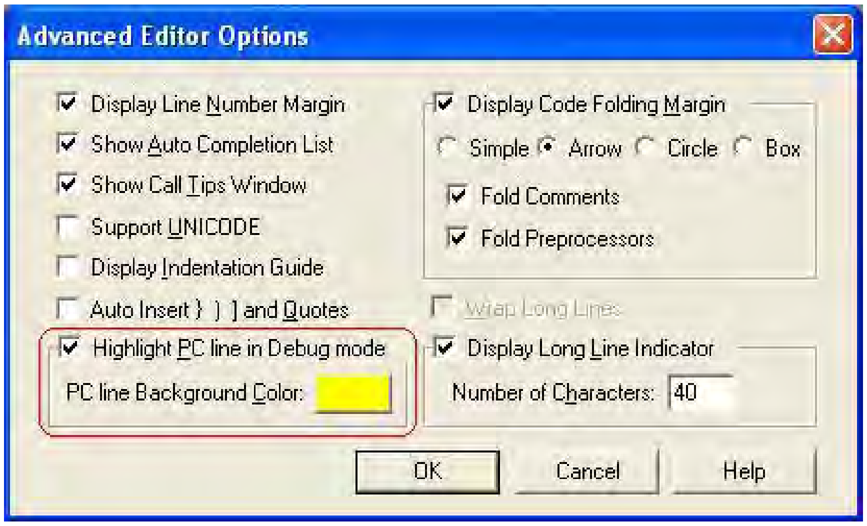


```
void main(void)
{
    char i, ch;

    /* initialize system peripherals */
    ledmatrix_init();
    ledtimer_init(1);  /* 1 mS interval */
    buttons_init();

    /* console welcome message and instructic
    printf("\n  ZiLOG Development Board Demor
    printf("\n         Featuring the %s\n", de
    printf("\n  Type a character and see it a

    /* flash full LED matrix */
    ledmatrix_flash();

    /* show a rotating bar on the LED matrix
    ledmatrix_spin();
```

Figure 121. Highlighting PC Line in Debug mode

To highlight the PC line, perform the following steps:

1. From the **Tools** menu, select **Options**. The **Options** dialog box is displayed.
2. Click the **Editor** tab.
3. Click the **Advanced Editor Options** button. The **Advanced Editor Options** dialog box is displayed.
4. Select the **Highlight PC line** checkbox in Debug mode to highlight the current PC line. This option is deselected by default. Click the **PC line Background Color** button to change the color of the highlighting.

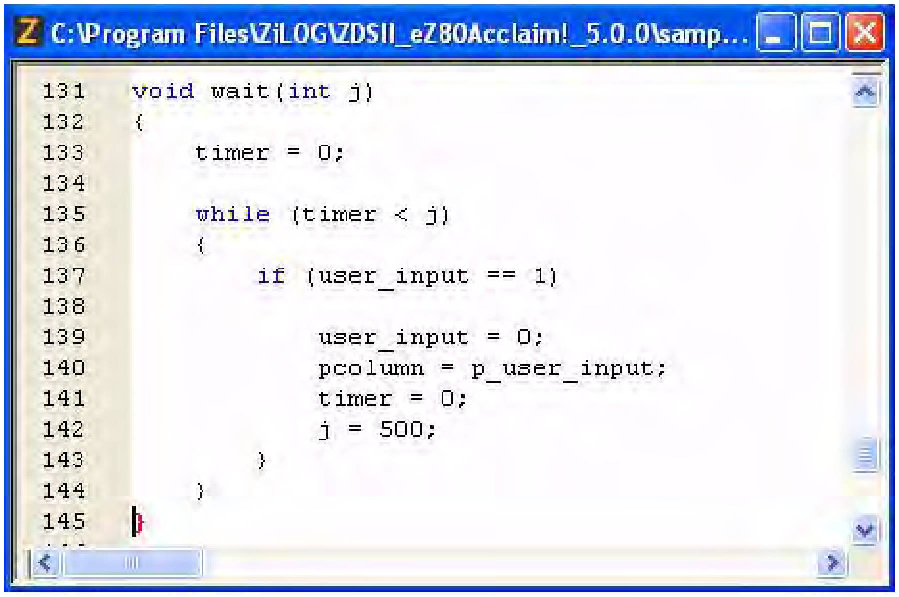

Advanced Editor Options

- [✓] Display Line Number Margin
- [✓] Show Auto Completion List
- [✓] Show Call Tips Window
- [ ] Support UNICODE
- [ ] Display Indentation Guide
- [ ] Auto Insert })] and Quotes
- [✓] Highlight PC line in Debug mode
- PC line Background Color: 🟨
- [✓] Display Code Folding Margin
- Simple [✓ Arrow] Circle Box
- [✓] Fold Comments
- [✓] Fold Preprocessors
- [ ] Wrap Long Lines
- [✓] Display Long Line Indicator
- Number of Characters: 40

Figure 122. Advance Editor Options—Highlight PC Line in Debug mode

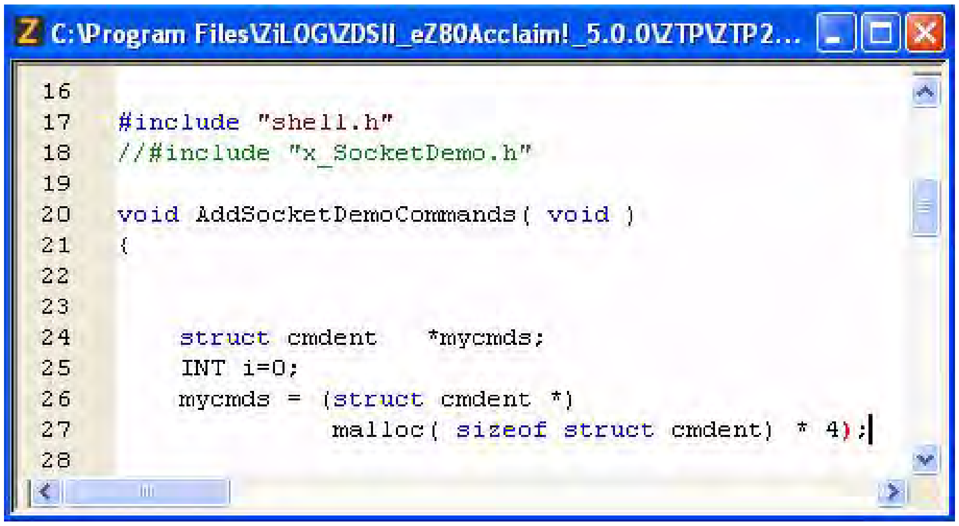
Mismatched Brace Highlighting

Move your text caret just behind the braces, parentheses or square brackets that are not properly balanced and observe those that are mismatched become highlighted in red, as shown in Figures 123 and 124.


C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_5.0.0\samp...

```
void wait(int j)
{
    timer = 0;

    while (timer < j)
    {
        if (user_input == 1)

            user_input = 0;
            pcolumn = p_user_input;
            timer = 0;
            j = 500;
        }
    }
}
```

Figure 123. Mismatched Brace Highlighting

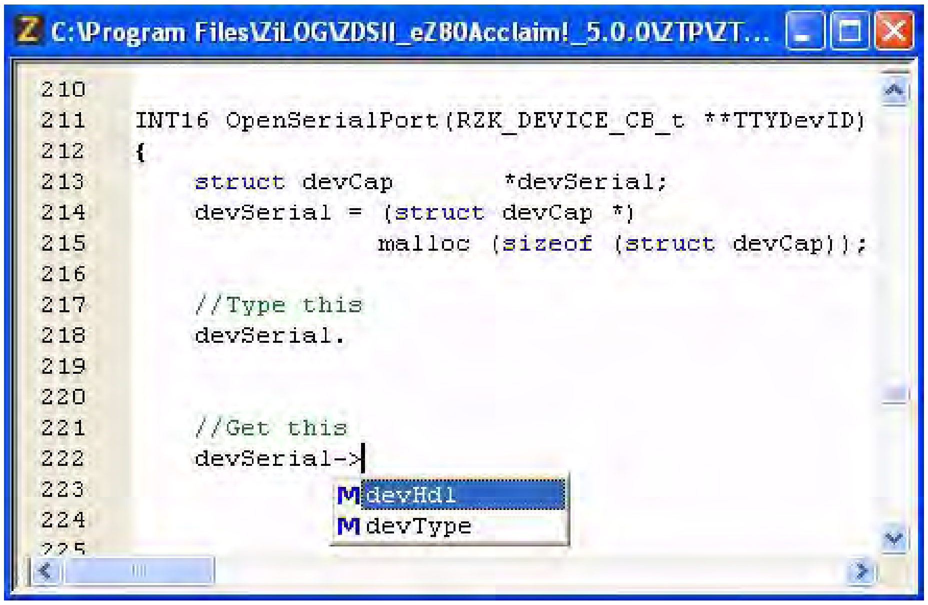

C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_5.0.0\ZTP\ZTP2...

```

#include "shell.h"
//#include "x_SocketDemo.h"

void AddSocketDemoCommands( void )
{


    struct cmdent    *mycmds;
    INT i=0;
    mycmds = (struct cmdent *)
              malloc( sizeof struct cmdent) * 4);

```

Figure 124. Mismatched Parenthesis Highlighting

Auto Conversion of "." to "→"

The ZDS II editor, upon discovering that you typed a period (.) instead of an arrow (→) after a pointer to a type of `struct` or `union` will automatically correct it and cause the **Member** list box to pop up. Automatic conversion of a period (.) to an arrow (→) avoids wasted builds and thereby allows you to be more productive with your code writing. See Figure 125.



C:\Program Files\ZiLOG\ZDSII_eZ80Acclaim!_5.0.0\ZTP\ZT...

```
INT16 OpenSerialPort(RZK_DEVICE_CB_t **TTYDevID)
{
    struct devCap       *devSerial;
    devSerial = (struct devCap *)
                malloc (sizeof (struct devCap));

    //Type this
    devSerial.


    //Get this
    devSerial->
```

Figure 125. Convert . to → Automatically

Chapter 4. Using the ANSI C-Compiler

The ZNEO C-Compiler is a conforming freestanding 1989 ANSI C implementation with some exceptions. These exceptions are described in the ANSI Standard Compliance section. In accordance with the definition of a freestanding implementation, the compiler accepts programs that confine the use of the features of the ANSI standard library to the contents of the standard headers <`float.h`>, <`limits.h`>, <`stdarg.h`> and <`stddef.h`>. The ZNEO compiler release supports more of the standard library than is required of a freestanding implementation as listed in the Run-Time Library section on page 177.

The ZNEO C-Compiler supports language extensions for easy programming of the ZNEO processor architecture, which include support for different address spaces and interrupt function designation. The language extensions are described in the Language Extensions section on page 156.

This chapter describes the various features of the ZNEO C-Compiler. It consists of the following sections:

The ZNEO C-Compiler is optimized for embedded applications in which execution speed and code size are crucial.

Language Extensions

To give you additional control over the way the ZNEO C-Compiler allocates storage and to enhance its ability to handle common real-time constructs, the compiler implements the following extensions to the ANSI C standard:

- Additional Keywords for Storage Specification

  The compiler divides the ZNEO CPU memory into four memory spaces: Near ROM, Extended ROM, Near RAM, and Extended (Far) RAM. It provides the following keywords with which you can control the storage location of data in these memory spaces:

  – `_Near` (near)
  – `_Far` (far)
  – `_Rom` (rom)
  – `_Erom` (erom)

  These keywords can also be used to specify the memory space to which a pointer is pointing to.

- Memory Models – see page 161

  The compiler supports two memory models: small and large. These models allow you to control where data are stored by default. Each application can only use one model. The model can affect the efficiency of your application. Some of the memory allocation defaults associated with a memory model can be overridden using the keywords for storage specification.

- Interrupt Support – see page 162

  The ZNEO CPU supports various interrupts. The C-Compiler provides language extensions to designate a function as interrupt service routine and provides features to set each interrupt vector.

- Placement Directives – see page 164

  The placement directives allow users to place objects at specific hardware addresses and align objects at a given alignment.

- String Placement – see page 165

  Because the ZNEO CPU has multiple address spaces, the C-Compiler provides language extensions to specify the storage for string constants.

- Inline Assembly – see page 166

  The C-Compiler provides directives for embedding assembly instructions and directives into the C program.

- Char and Short Enumerations – see page 167

The enumeration data type is defined as `int` as per ANSI C. The C-Compiler provides language extension to specify the enumeration data type to be other than `int`.

- Setting Flash Option Bytes in C – see page 168

  The ZNEO CPU has four Flash option bytes. The C-Compiler provides language extension to define these Flash option bytes.

- Supported New Features from the 1999 Standard – see page 169

  The ZNEO C-Compiler is based on the 1989 ANSI C standard. Some new features from the 1999 standard are supported in this compiler for ease of use.

Additional Keywords for Storage Specification

The `_Near`, `_Far`, `_Rom`, and `_Erom` keywords are storage class specifiers and are used to control the allocation of data objects by the compiler. They can be used on individual data objects similar to `const` and `volatile` keywords in the ANSI C standard. The storage specifiers can only be used to control the allocation of global and static data. The allocation of local data (nonstatic local) and function parameters is decided by the compiler and is described in later sections. Any storage specifier used on local and parameter data is ignored by the compiler.

The Zilog header file <`zneo.h`> defines macros to permit the use of the more familiar keywords: `near`, `far`, `rom`, and `erom`. The reason for not using these keywords directly is to avoid conflict with C identifiers in a preexisting C program. (To avoid conflicts, current ANSI recommendations are that keywords for vendor extensions to the C language begin with an underscore and capital letter.)

The data allocation for various storage class specifiers is shown in Figure 126 and described in the following sections:

- _Near – see page 158
- _Rom – see page 158
- _Erom – see page 159
- _Far – see page 159

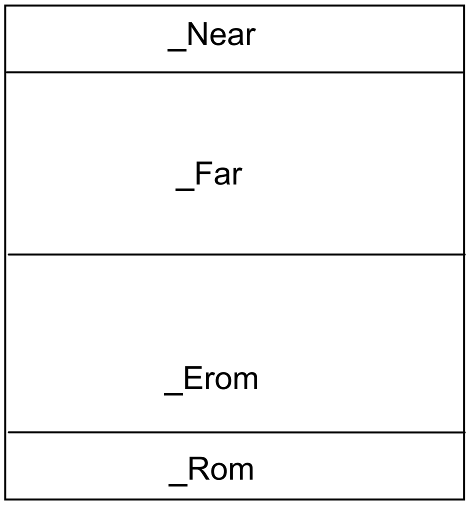


Figure 126. ZNEO C-Compiler Memory Layout

_Near

The variable with the `_Near` storage specifier is allocated in the 16-bit addressable near RAM address space. This space corresponds to the RAM assembler address space defined in the linker address space project settings. These variables lie within the 16-bit address range `8000-BFFF`, which is the 32-bit range `FF_8000-FF_BFFF`.

For example:

```
_Near int ni;  /* ni is placed in RAM address space */
```

In the ZNEO compiler, the peripheral registers (16-bit address: `C000-FFFF` and 32-bit address: `FF_C000-FF_FFFF`) are also mapped to the `_Near` storage specifier, and no separate keyword is provided.

For example:

```
#define T0CTL0    (*(unsigned char volatile _Near*)0xE306)
T0CTL0 = 0x12;
```

_Rom

The variable with the `_Rom` storage specifier is allocated in the space corresponding to the ROM assembler address space, which is defined in the linker address space project settings. These variables lie within the 16- or 32-bit addressable range `0000-7FFF`. The lower portion of this address space is used for Flash option bytes and interrupt vector table.

For example:

```
_Rom int ri;  /* ri is placed in ROM address space */
```

_Erom

The variable with the `_Erom` storage specifier is allocated in 32-bit addressed internal or external nonvolatile memory. This space corresponds to the EROM assembler address space defined in the linker address space project settings. These variables lie within the range extending from `00_8000` to the highest nonvolatile memory address.

For example:

```
_Erom int eri;  /* eri is placed in EROM address space */
```

_Far

The variable with the `_Far` storage specifier is allocated in 32-bit addressed external volatile (random access) memory. This space corresponds to the ERAM assembler address space defined in the linker address space project settings. These variables lie within the 32-bit addressed range above the highest EROM address and below `FF_8000`.

For example:

```
_Far int fi;  /* fi is placed in ERAM address space */
```

Storage Specification for Pointers

To properly access `_Near`, `_Far`, `_Rom`, and `_Erom` objects using a pointer, the compiler provides the ability to associate the storage specifier with the pointer type.

- _Near pointer

  The _Near pointer points to `_Near` data.
- _Far pointer

  The _Far pointer points to `_Far` data.
- _Rom pointer

  The _Rom pointer points to `_Rom` data.
- _Erom pointer

  The _Erom pointer points to `_Erom` data.

For example:

```
char _Near * _Far npf;
// npf is a pointer to a _Near char, npf itself is stored in _Far
memory.
```

Default Storage Specifiers

Default storage specifiers are applied if none is specified. The default storage specifiers depend on the memory model chosen. Table 11 lists the default storage specifiers for each model type.

Table 11. Default Storage Specifiers

	Function	Globals	Locals	String	Const	Parameters	Pointer
Small (S)	_Erom	_Near	_Near	_Near	_Near	_Near	_Near
Large (L)	_Erom	_Far	_Far	_Far	_Far	_Far	_Far

Space Specifier on Structure and Union Members

The space specifier for a structure or union takes precedence over the space specifier of an individual member. When the space specifier of a member does not match the space specifier of its structure or union, the space specifier of the member is ignored.

For example:

```
struct{
_Near char num;      /* Warning: _Near space specifier is ignored.
*/
_Near char * ptr;     /* Correct: ptr points to a char in _Near
memory. */
             /* ptr itself is stored in the memory space of
structure (_Far). */
} _Far mystruct;     /* All of mystruct is allocated in _Far
memory.*/
```

Pointer Conversions

A pointer to a qualified space type can be converted to a different qualified space type as shown in Table 12.

Table 12. Pointer Conversion*

	Source						
Destination	**unqualified**	**const**	**volatile**	**_Near**	**_Far**	**_Rom**	**_Erom**
unqualified	v	W	W	v	L	v	L
const	v	v	W	v	L	v	L
volatile	v	W	v	v	L	v	L
_Near	S	WS	WS	v	X	v	X
_Far	v	W	W	v	v	v	L
_Rom	S	WS	WS	v	X	v	X
_Erom	v	W	W	v	v	v	v

Notes:
1. v represents Valid.
2. W represents Warning.
3. S represents Valid in Small Model (Error in Large Model).
4. L represents Valid in Large Model (Error in Small Model).
5. WS represents Warning in Small Model (Error in Large Model).
6. WL represents Warning in Large Model (Error in Small Model).
7. X represents Error.

Memory Models

The ZNEO C-Compiler provides two memory models:

- Small memory model
- Large memory model

Each of these two models is described in this section.

Small Memory Model

In the small memory model, global variables are allocated in RAM address space. The address of these variables is 16 bits. The locals and parameters are allocated on the stack which is located in RAM address space. The address of a local or parameter is a 16-bit address. The global variables can be manually placed into the ERAM, ROM, or EROM address space by using the `_Far`, `_Rom`, and `_Erom` address specifiers, respectively. The local variables (nonstatic) and parameters are always allocated in RAM address space, and any address specifiers, if used on them, are ignored.

Use of the small memory model does not impose any restriction on your code size. The limitations of the small model are due to the somewhat limited amount of 16-bit addressable RAM. Current ZNEO CPU parts offer up to 4KB of internal RAM, and the ZDS II GUI restricts the total RAM linker address space (internal and external) to 16KB. If the local data and parameters exceed the available RAM size, then the small memory model cannot be used. If the local data and parameters are within the RAM size, but along with global data they exceed the RAM size, then the small model can still be used but only by selectively placing the global data in the extended RAM (ERAM) address space using the `_Far` keyword. Because ERAM is always located in external memory, this solution requires adding external memory to your system.

Large Memory Model

In the large memory model, global variables are allocated in the ERAM address space. The address of these variables is 32 bits. The locals and parameters are allocated on stack, which is located in ERAM address space. The address of a local or parameter is a 32-bit address. The global variables can be manually placed into the RAM, ROM, or EROM address space by using the `_Near`, `_Rom`, and `_Erom` address specifiers, respectively. The local variables (nonstatic) and parameters are always allocated in the ERAM address space, and any address specifiers, if used on them, are ignored.

In the large memory model, the local and global data and parameters can span the entire ERAM space, which can be configured at the user’s discretion to be much larger than the space available in the RAM address space. Besides the requirement to implement an external memory interface, the costs of using the large model are that 32-bit addressing is required to access the variables in ERAM, causing an increase in code size. Also, pointers to these data are 32 bits, which might increase the data space requirements if the application uses lots of pointers. It is possible that the application might run more slowly if accesses to external memory require wait states. To reduce the impact of some of these issues, you can selectively place your more frequently accessed global and static data in RAM using the `_Near` keyword.

Interrupt Support

To support interrupts, the ZNEO C-Compiler provides the Interrupt Keyword and Interrupt Vector Setup functions, as described below.

interrupt Keyword

Functions that are preceded by `#pragma interrupt` or are associated with the interrupt storage class are designated as interrupt handlers. These functions should neither take parameters nor return a value. The compiler stores the machine state at the beginning of these functions and restores the machine state at the end of these functions. Also, the compiler uses the `iret` instruction to return from these functions.

For example:

```
void interrupt isr_timer0(void)
{
}
```

or

```
#pragma interrupt
void isr_timer0(void)
{
}
```

Interrupt Vector Setup

The compiler provides an intrinsic function `SET_VECTOR` for interrupt vector setup. `SET_VECTOR` can be used to specify the address of an interrupt handler for an interrupt vector. Because the interrupt vectors of the ZNEO microcontroller are usually in ROM, they cannot be modified at run time. The `SET_VECTOR` function works by switching to a special segment and placing the address of the interrupt handler in the vector table. No executable code is generated for this statement.

The following example represents the `SET_VECTOR` intrinsic function prototype:

```
intrinsic void SET_VECTOR(int vectnum,void (*hndlr)(void));
```

An example of the use of `SET_VECTOR` is:

```
#include <zneo.h>
extern void interrupt isr_timer0(void);
void main(void)
{
      SET_VECTOR(TIMER0, isr_timer0);
}
```

The following values for `vectnum` are supported:

ADC	P7AD
C0	PWM_FAULT
C1	PWM_TIMER
C2	RESET
C3	SPI
I2C	SYSEXC
P0AD	TIMER0
P1AD	TIMER1
P2AD	TIMER2
P3AD	UART0_RX
P4AD	UART0_TX
P5AD	UART1_RX
P6AD	UART1_TX

Placement Directives

The ZNEO C-Compiler provides language extensions to declare a variable at an address and to align a variable at a specified alignment.

Placement of a Variable

The following syntax can be used to declare a global or static variable at an address:

```
char placed_char _At 0xb9ff; // placed_char is assigned an address
0xb9ff.
far struct {
 char ch;
 int ii;
} ss _At 0x080eff;                  // ss is assigned an address
0x080eff

rom char init_char _At 0x2fff = 33;
                    // init_char is in rom and initialized to 33
```

Note: Only placed variables with the `rom` or `erom` storage class specifiers can be initialized. The placed variables with `near` and `far` storage class specifier cannot be initialized. The uninitialized placed variables are not initialized to zero by the compiler startup routine.

Placement of Consecutive Variables

The compiler also provides syntax to place several variables at consecutive addresses.

For example:

```
char ch1 _At 0xbef0;
char ch2 _At …;
char ch3 _At …;
```

This example places `ch1` at address `0xbef0`, `ch2` at the next address (`0xbef1`) after `ch1`, and `ch3` at the next address (`0xbef2`) after `ch2`. The `_At …` directive can only be used after a previous `_At` or `_Align` directive.

Alignment of a Variable

The following syntax can be used to declare a global or static variable aligned at a specified alignment:

```
char ch2 _Align 2;   // ch2 is aligned at even boundary
char ch4 _Align 4;   // ch4 is aligned at a four byte boundary
```

Note: Only aligned variables with the `rom` or `erom` storage class specifiers can be initialized. The aligned variables with the `near` and `far` storage class specifiers cannot be initialized. The uninitialized aligned variables are not initialized to zero by the compiler startup routine.

String Placement

When string constants (literals) such as `"mystring"` are used in a C program, they are stored by the C-Compiler in RAM address space for the small memory model and in ERAM address space for the large memory model. However, sometimes this default placement of string constants does not allow you adequate control over your memory usage. Therefore, language extensions are provided to give you more control over string placement:

`N"mystring"`**.** This constant defines a near string. The string is stored in RAM. The address of the string is a _Near pointer.

`F"mystring"`**.** This constant defines a far string. The string is stored in ERAM. The address of the string is a _Far pointer.

`R"mystring"`**.** This constant defines a ROM string. The string is stored in ROM. The address of the string is a _Rom pointer.

`E"mystring"`. This constant defines an EROM string. The string is stored in EROM. The address of the string is a _Erom pointer.

The following example presents string placement:

```
#include <sio.h>
void funcn (_Near char *str)
{
     while (*str)
         putch (*str++);
     putch ('\n');
}

void funcf (_Far char *str)
{
     while (*str)
        putch (*str++);
     putch ('\n');
}

void funcr (_Rom char *str)
{
    while (*str)
        putch (*str++);
    putch ('\n');
}

void funcer (_Erom char *str)
{
    while (*str)
        putch (*str++);
    putch ('\n');
}

void main (void)
{
    funcn (N"nstr");
    funcf (F"fstr");
    funcr (R"rstr");
    funcer (E"erstr");
}
```

Inline Assembly

There are two methods of inserting assembly language within C code, as described below.

Inline Assembly Using the Pragma Directive

The first method uses the `#pragma` feature of ANSI C with the following syntax:

```
#pragma asm “<assembly line>”
```

`#pragma` can be inserted anywhere within the C source file. The contents of <*assembly line*> must be legal assembly language syntax. The usual C escape sequences (such as `\n`, `\t`, and `\r`) are properly translated. Currently, the compiler does not process the <*assembly line*>. Except for escape sequences, it is passed through the compiler verbatim.

Inline Assembly Using the asm Statement

The second method of inserting assembly language uses the `asm` statement:

```
asm(“<assembly line>”);
```

The `asm` statement cannot be within an expression and can be used only within the body of a function.

The <*assembly line*> can be any string.The compiler does *not* check the legality of the string.

As with the `#pragma asm` form, the compiler does not process the <*assembly line*> except for translating the standard C escape sequences.

The compiler prefixes the name of every global variable name with "_". Global variables can therefore be accessed in inline assembly by prefixing their name with "_ ". The local variables and parameters cannot be accessed in inline assembly.

Char and Short Enumerations

The enumeration data type is defined as `int` as per ANSI C. The C-Compiler provides language extensions to specify the enumeration data type to be other than `int` to save space. The following syntax is provided by the C-Compiler to declare them as `char` or `short`:

```
enum
{
    RED = 0,
    YELLOW,
    BLUE,
    INVALID
} char color;

enum
{
    NEW= 0,
    OPEN,
    FIXED,
    VERIFIED,
    CLOSED
} short status;

void main(void)
{
    if (color == RED)
        status = FIXED;
    else
        status = OPEN;
}
```

Setting Flash Option Bytes in C

The ZNEO CPU provides up to four Flash option bytes to configure the device. These Flash option bytes can be set in C, using the following syntax:

```
#include <zneo.h>
FLASH_OPTION0 = val;
FLASH_OPTION1 = val;
FLASH_OPTION2 = val;
FLASH_OPTION3 = val;
```

where:

- FLASH_OPTION0 is the Flash option byte at address 0
- FLASH_OPTION1 is the Flash option byte at address 1
- FLASH_OPTION2 is the Flash option byte at address 2
- FLASH_OPTION3 is the Flash option byte at address 3

For example:

```
#include <zneo.h>
FLASH_OPTION0 = 0xFF;
FLASH_OPTION1 = 0xFF;
FLASH_OPTION2 = 0xFF;
FLASH_OPTION3 = 0xFF;
void main (void)
{
}
```

This example sets the Flash option bytes at address 0, 1, 2, and 3 as `0xFF`. The Flash option bytes can be written only once in a program. They are set at load time. When you set these bytes, you must make sure that the settings match the actual hardware. For more information, see the product specification specific to your device.

Supported New Features from the 1999 Standard

The ZNEO compiler implements four new features introduced in the ANSI 1999 standard, also known as ISO/IEC 9899:1999; each is described below.

C++ Style Comments

Comments preceded by `//` and terminated by the end of a line, as in C++, are supported.

Trailing Comma in Enum

A trailing comma in `enum` declarations is allowed. This essentially allows a common syntactic error that does no harm. Thus, a declaration such as:

```
enum color {red, green, blue,} col;
```

is allowed (note the extra comma after `blue`).

Empty Macro Arguments

Preprocessor macros that take arguments are allowed to be invoked with one or more arguments empty, as in this example:

```
#define cat3(a,b,c) a b c
printf("%s\n", cat3("Hello ", ,"World"));
                                      // ^ Empty arg
```

Long Long Int Type

The `long long int` type is allowed. (In the ZNEO C-Compiler, this type is treated as the same as `long`, which is allowed by the standard.)

Type Sizes

The type sizes for basic data types on the ZNEO C-Compiler are:

`int`	32 bits
`short int`	16 bits
`char`	8 bits
`long`	32 bits
`float`	32 bits
`double`	32 bits

The type sizes for the pointer data types on the ZNEO C-Compiler are:

_Near pointer	16 bits
_Far pointer	32 bits
_Rom pointer	16 bits
_Erom pointer	32 bits

All data are aligned on a byte boundary.

Caution: Alignment of 16- or 32-bit objects on even boundaries is a possible future enhancement. Avoid writing code that depends on how data are aligned.

Predefined Macros

The ZNEO C-Compiler comes with the following standard predefined macro names:

__DATE__	This macro expands to the current date in the format “Mmm dd yyyy” (a character string literal), where the names of the months are the same as those generated by the `asctime` function and the first character of dd is a space character if the value is less than 10.
__FILE__	This macro expands to the current source file name (a string literal).
__LINE__	This macro expands to the current line number (a decimal constant).
__STDC__	This macro is defined as the decimal constant 1 and indicates conformance with ANSI C.
__TIME__	This macro expands to the compilation time in the format “hh:mm:ss” (a string literal).

None of these macro names can be the subject of a `#define` or a `#undef` preprocessing directive. The values of these predefined macros (except for `__LINE__` and `__FILE__`) remain constant throughout the translation unit.

The following additional macros are predefined by the ZNEO C-Compiler:

`__CONST_IN_ROM__`	This macro indicates that the const variables are placed in ROM. This macro, which is optional in some other Zilog processor architectures, must always be defined for the ZNEO.
`__MODEL__`	This macro indicates the memory model used by the compiler as follows: 0Small Model 3Large Model
`__UNSIGNED_CHARS__`	This macro is defined if the plain `char` type is implemented as `unsigned char`.
`__ZDATE__`	This macro expands to the build date of the compiler in the format YYYYMMDD. For example, if the compiler were built on May 31, 2006, then __ZDATE__ expands to 20060531. This macro gives a means to test for a particular Zilog release or to test that the compiler is released after a new feature has been added.
`__ZILOG__`	This macro is defined and set to 1 on all Zilog compilers to indicate that the compiler is provided by Zilog.
`__ZNEO__`	This macro is defined and set to 1 for the ZNEO compiler and is otherwise undefined.

All predefined macro names begin with two underscores and end with two underscores.

Examples

The following program illustrates the use of some of these predefined macros:

```
#include <stdio.h>
void main()
{
#ifdef __ZILOG__
    printf("Zilog Compiler ");
#endif
#ifdef __ZNEO__
    printf("For ZNEO ");
#endif
#ifdef __ZDATE__
    printf("Built on %d.\n", __ZDATE__);
#endif
}
```

Calling Conventions

The C-Compiler imposes a strict set of rules regarding function calls. Except for special run-time support functions, any function that calls or is called by a C function must follow these rules. Failure to adhere to these rules can disrupt the C environment and cause a C program to fail.

Function Call Mechanism

A function (caller function) performs the following sequence of tasks when it calls another function (called function):

1. Save any of the registers R0–R7 that are in use and may be required after the call; these registers may be overwritten in the called function.
2. Place the first seven scalar parameters (not structures or unions) of the called function in registers R1–R7. Push parameters beyond the seventh parameter and nonscalar parameters on the stack in reverse order (the rightmost declared argument is pushed first, and the leftmost is pushed last). This places the leftmost argument on top of the stack when the function is called. For a `varargs` function, all parameters are pushed on the stack in reverse order.
3. Call the function. The call instruction pushes the return address on the top of the stack.
4. On return from the called function, caller pops the arguments off the stack or increment the stack pointer.
5. Restore any of the registers R0–R7 that were saved in step 1.

When a byte or structure of an odd size is pushed on the stack, only the byte or structure is pushed. Future enhancements might introduce padding so that 16- or 32-bit objects are located at an even offset from the frame pointer so avoid writing code that depends on the alignment of data. If you are writing an assembly routine called out of C, it is recommended that you declare parameters as `short` rather than `char` so that offsets to parameters are not changed by such an enhancement.

The called function performs the following sequence of tasks:

1. Push the frame pointer onto the stack and allocate the local frame:
   a. Set the frame pointer to the current value of the stack pointer.
   b. Decrement the stack pointer by the size of locals and temporaries, if required.

2. Save the contents of any of the registers R8–R13 (and possibly R14; see comment below) that are going to be used inside this function.
3. Execute the code for the function.

4. Restore any of the registers R8–R14 that were saved in step 2.
5. If the function returns a scalar value, place it in the r0 register. For functions returning an aggregate, see the Special Cases section, which follows.
6. Deallocate the local frame (set the stack pointer to the current value of frame pointer) and restore the frame pointer from stack.
7. Return.

Registers R8–R13 are considered as *callee* saves; that is, they are saved and restored (if necessary) by the called function. If the called function does not set up a frame pointer, it can also use R14 as a general-purpose register but must still save it on entry and restore it on exit. The flag register is not saved and restored by the called function.

The function call mechanism described in this section is a dynamic call mechanism. In a dynamic call mechanism, each function allocates memory on stack for its locals and temporaries during the run time of the program. When the function has returned, the memory that it was using is freed from the stack. Figure 127 shows a diagram of the ZNEO C-Compiler dynamic call frame layout.

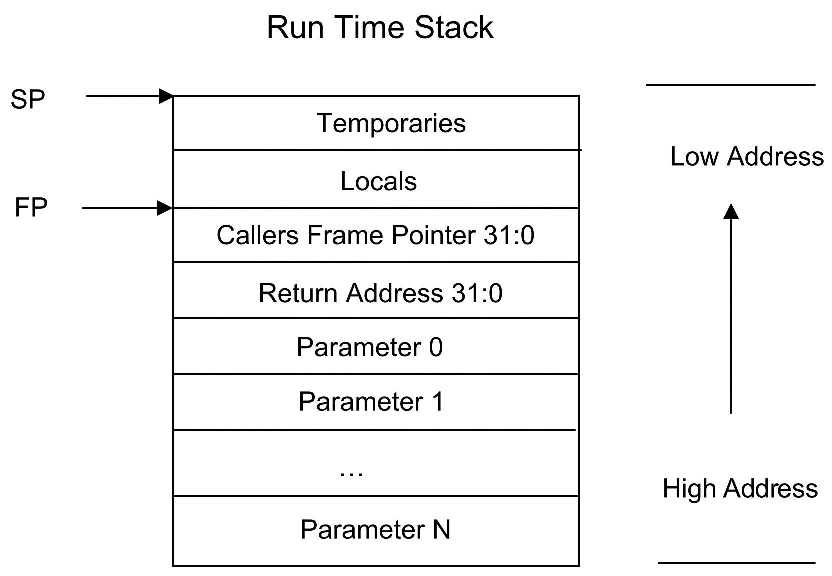


Figure 127. Call Frame Layout

Special Cases

Some function calls do not follow the mechanism described in the Function Call Mechanism section on page 172. Such cases are described in this section.

Returning Structure

If the function returns a structure, the caller allocates the space for the structure and then passes the address of the return space to the called function as an additional and first argument. To return a structure, the called function then copies the structure to the memory block pointed to by this argument.

Not Allocating a Local Frame

The compiler does not allocate a local stack frame for a function in the following case:

- The function does not have any local stack variables, stack arguments, or compiler-generated temporaries on the stack

and:

- The function does not return a structure

and:

- The function is compiled without the debug option.

Calling Assembly Functions from C

The ZNEO C-Compiler allows mixed C and assembly programming. A function written in assembly can be called from C if the assembly function follows the C calling conventions as described in the Calling Conventions section on page 172.

This section covers the following topics:

- Function Naming Convention
- Argument Locations
- Return Values
- Preserving Registers

Function Naming Convention

Assembly function names must be preceded by an underscore (_). The compiler prefixes the function names with an underscore in the generated assembly. For example, a call to `myfunc()` in C is translated to a call to `_myfunc` in assembly generated by the compiler.

Argument Locations

The assembly function assigns the location of the arguments following the C calling conventions as described in the Calling Conventions section on page 172. For example, if you are using the following C prototype:

```
void myfunc(short arga, long argb, short *argc, char argd, int
arge, int argf, char argg, long *argh, int argi)
```

then the location of the arguments are:

`arga`: R1

`argb`: R2

`argc`: R3

`argd`: R4

`arge`: R5

`argf`: R6

`argg`: R7

The remaining arguments are on stack, and their offsets from Stack Pointer (SP, R15) at the entry point of assembly function are:

`argh`: `-`4(SP)

`argi`: `-`8(SP)

The corresponding offsets from Frame Pointer (FP, R14) after a `Link #0` instruction are:

`argh`: `-`8(FP)

`argi`: `-`12(FP)

Return Values

The assembly function returns the value in the location as specified by the C calling convention as described in Calling Conventions on page 172.

For example, if you are using the following C prototype:

```
long myfunc(short arga, long argb, short *argc)
```

then the assembly function returns the long value in register R0.

Preserving Registers

The ZNEO C-Compiler implements a scheme in which the registers R8–R13 are treated as *callee save*. The assembly function must preserve any of these registers that it uses. The assembly function is not expected to save and restore the flag register.

Calling C Functions from Assembly

The C functions that are provided with the compiler library can also be used to add functionality to an assembly program. You can also create your own C functions and call them from an assembly program.

Because the compiler makes the caller function responsible for saving registers R0–R7 (see the Calling Conventions section on page 172), if the assembly code is using any of these functions and needs their contents to be preserved across the C function call, it must save them before the call and restore them afterwards.

Note: The C-Compiler precedes the function names with an underscore in the generated assembly. See the Function Naming Convention section on page 174.

The following example shows an assembly source file referencing the function `sinf`. The `sinf` function is defined in the C library.

Assembly File

```
    globals on

    xref _sinf

    segment near_data
  val:dl %3F060A96; 0.523599
  res:dl 0

    segment code
  _main:
    pushm <R1>
    ; save the registers, other than return register, if any in
use

    ld R1,val; load the argument
    call _sinf; call the c functions
    ld res,r0 ; the result is in r0
```

```
popm <R1>; restore the registers, if any were saved

ret
```

Referenced C Function Prototype

```
float sinf(float arg);
```

Command Line Options

The compiler, like the other tools in ZDS II, can be run from the command line for processing inside a script, and so on. Please see Compiler Command Line Options on page 354 for the list of compiler commands that are available from the command line.

Run-Time Library

The C-Compiler provides a collection of run-time libraries. The largest section of these libraries consists of an implementation of much of the C Standard Library. A small library of functions specific to Zilog or to the ZNEO is also provided.

The ZNEO C-Compiler is a conforming freestanding 1989 ANSI C implementation with some exceptions. In accordance with the definition of a freestanding implementation, the compiler supports the required standard header files <`float.h`>, <`limits.h`>, <`stdarg.h`>, and <`stddef.h`>. It also supports additional standard header files and Zilog-specific nonstandard header files.

The standard header files and functions are, with minor exceptions, fully compliant with the ANSI C Standard. They are described in detail in the appendix titled C Standard Library, on page 391. The deviations from the ANSI Standard in these files are summarized in the Library Files Not Required for Freestanding Implementation section on page 196. The following sections describe the use and format of the nonstandard, Zilog-specific run-time libraries:

- Zilog Header Files – see page 178
- Zilog Functions – see page 180

The Zilog-specific header files provided with the compiler are listed in Table 13 and described in the Zilog Header Files section on page 178.

Table 13. Nonstandard Headers

Header	Description	Page
`<zneo.h>`	ZNEO defines and functions	page 178
`<sio.h>`	Serial input/output functions	page 179

Note: The Zilog-specific header files are located in the following directory:

<*ZDS Installation Directory*>`\include\zilog`

where <*ZDS Installation Directory*> is the directory in which Zilog Developer Studio was installed. By default, this would be `C:\Program Files\Zilog\ZDSII_ZNEO_<version>`, where <*version*> might be `4.11.0` or `5.0.0`.

Note: All external identifiers declared in any of the headers are reserved, whether or not the associated header is included. All external identifiers and macro names that begin with an underscore are also reserved. If the program redefines a reserved external identifier, even with a semantically equivalent form, the behavior is indeterminate.

Zilog Header Files

Architecture-Specific Functions <zneo.h>

A ZNEO-specific header file `<zneo.h>` is provided that has prototypes for Zilog-specific C library functions and macro definitions.

Macros

`<zneo.h>` contains macro definitions giving the more conventional names for storage specifiers:

erom	Expands to space specifier _Erom.
near	Expands to space specifier _Near.
far	Expands to space specifier _Far.
rom	Expands to space specifier _Rom.

`<zneo.h>` has the macro definitions for all ZNEO microcontroller peripheral registers. For example:

T0H	Expands to (*(unsigned char volatile near*)0xE300)

Refer to the ZNEO product specifications for the list of peripheral registers supported.

`<zneo.h>` also has the macro definition for the ZNEO Flash option bytes:

FLASH_OPTION0	Expands to a _Rom char at address 0x0.
FLASH_OPTION1	Expands to a _Rom char at address 0x1.
FLASH_OPTION2	Expands to a _Rom char at address 0x2.
FLASH_OPTION3	Expands to a _Rom char at address 0x3.

`<zneo.h>` also has the macro definition for interrupt vector addresses:

RESET	Expands to address of Reset vector.

Refer to the ZNEO product specifications for the list of interrupt vectors supported.

Functions

intrinsic void EI(void);	Enable interrupts.
intrinsic void DI(void);	Disable interrupts.
intrinsic void RI(unsigned short istat);	Restores interrupts.
intrinsic void SET_VECTOR(int vectnum,void (*hndlr)(void));	Specifies the address of an interrupt handler for an interrupt vector.
intrinsic unsigned short TDI(void);	Tests and disables interrupts.

Nonstandard I/O Functions <sio.h>

This header contains nonstandard ZNEO-specific input/output functions.

_DEFFREQ	Expands to unsigned long default frequency.
_DEFBAUD	Expands to unsigned long default baud rate.
_UART0	Expands to an integer indicating UART0.
_UART1	Expands to an integer indicating UART1.

Functions

char getch(void) ;	Returns the data byte available in the selected UART.
int init_uart(int port,unsigned long freq, unsigned long baud);	Initializes the selected UART for specified settings and returns the error status.
int kbhit(void);	Checks for receive data available on selected UART.
int putch(char) ;	Sends a character to the selected UART and returns the error status.
int select_port(int port);	Selects the UART. Default is _UART0.

Zilog Functions

The following functions are Zilog-specific; each is described on their respective pages, as referenced below.

- DI – see page 180
- EI – see page 181
- getch – see page 181
- init_uart – see page 182
- kbhit – see page 182
- putch – see page 183
- RI – see page 183
- select_port – see page 184
- SET_VECTOR – see page 184
- TDI – see page 185

DI

`DI` is an intrinsic function that disables all interrupts and is inline-expanded by default. If the `–reduceopt` compiler option is selected, then this function is not inline-expanded and is instead implemented as a regular function.

Synopsis

```
#include <zneo.h>
intrinsic void DI(void);
```

```
Example
  #include <zneo.h>

  void main(void)
  {
    DI();/* Disable interrupts */
  }
```

EI

`EI` is an intrinsic function that enables all interrupts and is inline-expanded by default. If the `–reduceopt` compiler option is selected, then this function is not inline-expanded and is instead implemented as a regular function.

Synopsis

```
#include <zneo.h>
intrinsic void EI(void);
```

Example

```
#include <zneo.h>

void main(void)
{
  EI(); /* Enable interrupts */
}
```

getch

`getch` is a ZILOG function that waits for the next character to appear at the serial port and returns its value. This function does not wait for end-of-line to return as `getchar` does. `getch` does not echo the character received.

Synopsis

```
#include <sio.h>
char getch(void) ;
```

Returns

The next character that is received at the selected UART.

Example

```
char ch;
ch=getch();
```

Note: Before using the `getch` function, the `init_uart()` function must be called to initialize and select the UART. The default UART is _UART0.

init_uart

The `init_uart` function is a Zilog function that selects the specified UART and initializes it for specified settings and returns the error status.

Synopsis

```
    #include <sio.h>
    int init_uart(int port, unsigned long freq, unsigned long
baud);
```

Returns

Returns 0 if initialization is successful and 1 otherwise.

Example

```
#include <stdio.h>
#include <sio.h>
void main()
{
  init_uart(_UART0,_DEFFREQ,_DEFBAUD);
  printf("Hello UART0\n"); // Write to _UART0
}
```

`_DEFFREQ` is automatically set from the IDE using the setting in the **Configure Target** dialog box. See Setup on page 76.

kbhit

`kbhit` is a Zilog function that determines whether there is receive data available on the selected UART.

Synopsis

```
#include <sio.h>
int kbhit(void);
```

Returns

Returns 1 if there is receive data available on the selected UART; otherwise, it returns 0.

Example

```
int i;
i=kbhit();
```

Note: Before using the `kbhit` function, the `init_uart()` function must be called to initialize and select the UART. The default UART is _UART0.

putch

`putch` is a Zilog function that sends a character to the selected UART and returns an error status.

Synopsis

```
#include <sio.h>
int putch( char ch ) ;
```

Returns

A zero is returned on success; a nonzero is returned on failure.

Example

```
char ch = 'c' ;
int err;
err = putch( ch ) ;
```

Note: Before using the `putch` function, the `init_uart()` function must be called to initialize and select the UART. The default UART is _UART0.

RI

`RI` (restore interrupt) is an intrinsic function that restores interrupt status. It is intended to be paired with an earlier call to `TDI()`, which has previously saved the existing interrupt status. See the TDI section on page 185 for a discussion of that function. The interrupt status, which is passed as a parameter to `RI()`, consists of the Flags register extended to 16 bits for efficient stack storage. This function inline-expanded by default. If the `–reduceopt` compiler option is selected, then this function is not inline-expanded and is instead implemented as a regular function.

Synopsis

```
#include <zneo.h>
intrinsic void RI(unsigned short istat);
```

Example

```
#include <zneo.h>
```

```
    void main(void)
    {
              unsigned short istat;
              istat = TDI();       /* Test and Disable Interrupts
  */
              /* Do Something */
              RI(istat);            /* Restore Interrupts */
    }
```

select_port

`select_port` is a Zilog function that selects the UART. The default is _UART0. The `init_uart` function can be used to configure either _UART0 or _UART1 and select the UART passed as the current one for use. All calls to `putch`, `getch`, and `kbhit` use the selected UART. You can also change the selected UART using the `select_port` function without having to reinitialize the UART.

Synopsis

```
#include <sio.h>
int select_port( int port ) ;
```

Returns

A zero is returned on success; a nonzero is returned on failure.

Example

```
#include <stdio.h>
#include <sio.h>
void main(void)
{
  init_uart(_UART0,_DEFFREQ,_DEFBAUD);
  init_uart(_UART1,_DEFFREQ,_DEFBAUD);
  select_port(_UART0);
  printf("Hello UART0\n"); // Write to uart0
  select_port(_UART1);
  printf("Hello UART1\n"); // Write to uart1
}
```

SET_VECTOR

`SET_VECTOR` is an intrinsic function provided by the compiler to specify the address of an interrupt handler for an interrupt vector. Because the interrupt vectors of the ZNEO micro-controller are usually in ROM, they cannot be modified at run time. The `SET_VECTOR` function works by switching to a special segment and placing the address of the interrupt handler in the vector table. No executable code is generated for this statement. Calls to the `SET_VECTOR` intrinsic function must be placed within a function body. The `–reduceopt` compiler option does not affect the `SET_VECTOR` function handling.

Synopsis

```
#include <zneo.h>
intrinsic void SET_VECTOR(int vectnum,void (*hndlr)(void));
```

where:

- `vectnum` is the interrupt vector number for which the interrupt handler `hndlr` is to be set
- `hndlr` is the interrupt handler function pointer. The `hndlr` function must be declared to be of the interrupt type with parameters and return as `void` (no parameters and no return)

The compiler supports the following values for `vectnum` for the ZNEO CPU:

ADC	P7AD
C0	PWM_FAULT
C1	PWM_TIMER
C2	RESET
C3	SPI
I2C	SYSEXC
P0AD	TIMER0
P1AD	TIMER1
P2AD	TIMER2
P3AD	UART0_RX
P4AD	UART0_TX
P5AD	UART1_RX
P6AD	UART1_TX

Returns

None

Example

```
#include <zneo.h>
extern void interrupt isr_timer0(void);
void main(void)
{
  SET_VECTOR(TIMER0, isr_timer0); /* setup TIMER0 vector */
}
```

TDI

`TDI` (test and disable interrupts) is an intrinsic function that supports users creating their own critical sections of code. It returns the previous interrupt status and disables interrupts. It is intended to be paired with a later call to `RI()`, which will restore the previously existing interrupt status. See the RI section on page 183 for a discussion of that function.

The interrupt status, which is returned from `TDI()`, consists of the flags register extended to 16 bits for efficient stack storage.

This function is an intrinsic function and is inline-expanded by default. If the `-reduceopt` compiler option is selected, then this function is not inline-expanded and is implemented as a regular function instead.

Synopsis

```
#include <zneo.h>
intrinsic unsigned short TDI(void);
```

Example

```
#include <zneo.h>

void main(void)
{

  unsigned short istat;
  istat = TDI();        /* Test and Disable Interrupts */
  /* Do Something */
  RI(istat);            /* Restore Interrupts */
}
```

Stack Pointer Overflow

The run-time library for the ZNEO C-Compiler manipulates the SPOV register to protect program and allocated data. The default action is:

- The SPOV register is initialized to the end of the initialized data segment. Therefore, if the Stack Pointer is decremented below SPOV, it results in the Stack overflow exception.
- When `malloc()` is called, the SPOV register is increased to the highest address of allocated data; `malloc()` returns NULL if changing the SPOV results in an immediate stack overflow.
- Calling `free()` returns memory for possible use by `malloc()` but does not return memory for possible use as stack; that is, the SPOV register is not updated.
- However, if you modify the SPOV register directly, `malloc()` leaves SPOV where you have put it and does not allocate data on the stack side of SPOV.

The run-time library does not supply a handler for the stack overflow exception (or any other exception). If there is any possibility of your released application overflowing its stack, you must decide how to recover from the situation and write your own handler.

Startup Files

The start-up or C run-time initialization file is an assembly program that performs required start-up functions and then calls `main`, which is the C entry point. The start-up program performs the following C run-time initializations:

1. Initializes the stack pointer and stack overflow register.
2. Initializes the external interface if enabled (see the description of the Configure Target Dialog Box on page 76).
3. Clears the `_Near` and `_Far` uninitialized variables to zero.
4. Sets the initialized `_Near` and `_Far` variables to their initial value from `_Erom`.
5. Allocate space for interrupt vectors.
6. Allocate space for the `errno` variable used by the C run-time libraries.

Table 14 lists the start-up files provided with the ZNEO C-Compiler.

Table 14. ZNEO Startup Files

Name	Description
lib\zilog\startups.obj	C start-up object file for small model.
src\boot\common\startups.asm	C start-up source file for small model.
lib\zilog\startupl.obj	C start-up object file for large model.
src\boot\common\startupl.asm	C start-up source file for large model.
lib\zilog\startupexkl.obj	C start-up object file (large model) with external interface set up for devices with Port K.
lib\zilog\startupexl.obj	C start-up object file (large model) with external interface set up for devices without Port K.
src\boot\common\startupexl.asm	C start-up source file (large model) with external interface set up.
lib\zilog\startupexks.obj	C start-up object file (small model) with external interface set up for devices with Port K.
lib\zilog\startupexs.obj	C start-up object file (small model) with external interface set up for devices without Port K.
src\boot\common\startupexs.asm	C start-up source file (small model) with external interface set up.

Segment Naming

The compiler places code and data into separate segments in the object file. The different segments used by the compiler are listed in Table 15.

Table 15. Segments

Segment	Description
NEAR_DATA	_Near initialized global and static data
NEAR_BSS	_Near un-initialized global and static data
NEAR_TEXT	_Near constant strings
FAR_DATA	_Far initialized global and static data
FAR_BSS	_Far un-initialized global and static data
FAR_TEXT	_Far constant strings
ROM_DATA	_Rom global and static data
ROM_TEXT	_Rom constant strings
EROM_DATA	_Erom global and static data
EROM_TEXT	_Erom constant strings
CODE	_Erom code
__VECTORS	_Rom interrupt vectors
STARTUP	_Rom C startup

Linker Command Files for C Programs

This section describes how the ZNEO linker is used to link a C program. A C program consists of compiled and assembled object module files, compiler libraries, user-created libraries, and special object module files used for C run-time initializations. These files are linked based on the commands given in the linker command file.

The default linker command file is automatically generated by the ZDS II IDE whenever a `build` command is issued. It has information about the ranges of various address spaces for the selected device, the assignment of segments to spaces, order of linking, and so on. The default linker command file can be overridden by the user.

The linker processes the object modules (in the order in which they are specified in the linker command file), resolves the external references between the modules, and then locates the segments into the appropriate address spaces as per the linker command file.

The linker depicts the memory of the ZNEO CPU using a hierarchical memory model containing spaces and segments. Each memory region of the CPU is associated with a space. Multiple segments can belong to a given space. Each space has a range associated with it that identifies valid addresses for that space. The hierarchical memory model for the ZNEO CPU is shown in Figure 128. Figure 129 depicts how the linker links and locates segments in different object modules.

For a more detailed description of the linker and the various commands it supports, see the Using the Linker/Locator chapter on page 259.

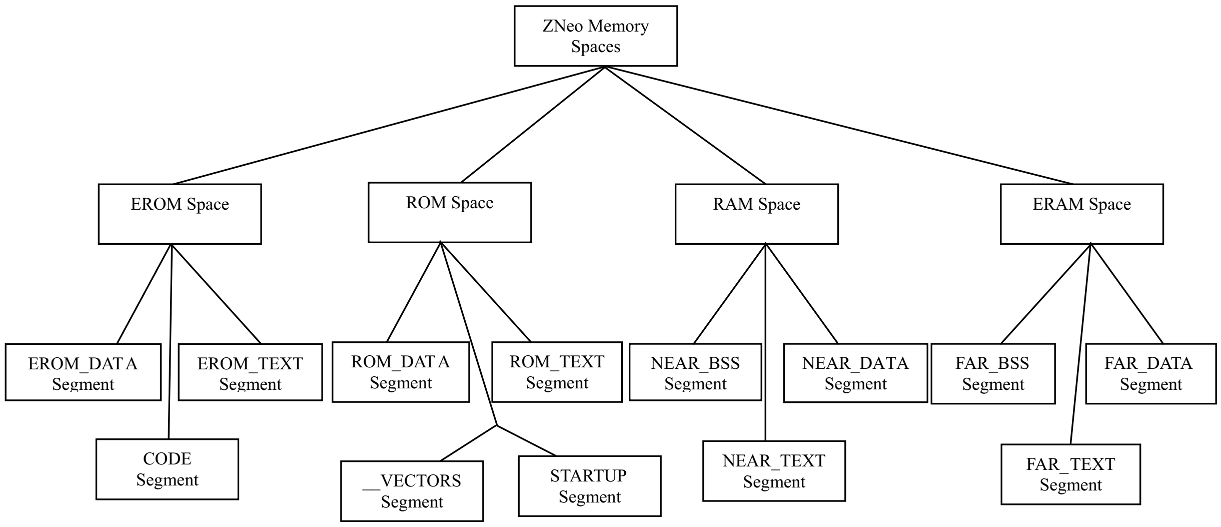


Figure 128. ZNEO Hierarchical Memory Model

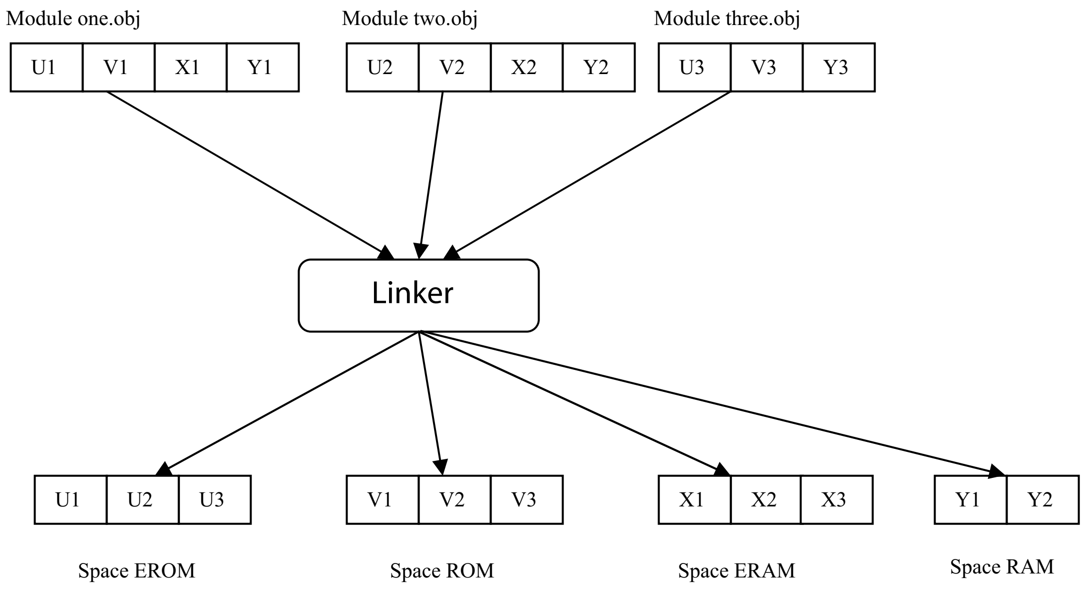


Figure 129. Multiple File Linking

Linker Referenced Files

The default linker command file generated by the ZDS II IDE references system object files and libraries based on the compilation memory model selected by the user. A list of the system object files and libraries is provided in Table 16. The linker command file automatically selects and links to the appropriate version of the C run-time and floating-point libraries (if necessary) from the list in Table 16, based on the your project settings.

Table 16. Linker Referenced Files

File	Description
`Startups.obj`	C startup for small model
`Startupl.obj`	C startup for large model
Startupexks.obj	C startup (small model) with external interface setup for devices with Port K.
Startupexs.obj	C startup (small model) with external interface setup for devices without Port K.
Startupexkl.obj	C startup (large model) with external interface setup for devices with Port K.
Startupexl.obj	C startup (large model) with external interface setup for devices without Port K.
`Fpdumy.obj`	Floating-point do-nothing stubs
`Crtl.lib`	C run-time library for large model, no debug information

Table 16. Linker Referenced Files (Continued)

File	Description
`Crtld.lib`	C run-time library for large model, with debug information
`Fpl.lib`	Floating-point library for large model, no debug information
`Fpld.lib`	Floating-point library for large model, with debug information
`Crts.lib`	C run-time library for small model, no debug information
`Crtsd.lib`	C run-time library for small model, with debug information
`Fps.lib`	Floating-point library for small model, no debug information
`Fpsd.lib`	Floating -point library for small model, with debug information
Chelps.lib	C helper routines for small model, no debug information
Chelpsd.lib	C helper routines for small model, with debug information
Chelpl.lib	C helper routines for large model, no debug information
Chelpld.lib	C helper routines for large model, with debug information

Linker Symbols

The default linker command file defines some system symbols, which are used by the C startup file to initialize the stack pointer, clear the uninitialized variables to zero, set the initialized variables to their initial value, set the heap base, and so on. Table 17 shows the list of symbols that might be defined in the linker command file, depending on the compilation memory model selected by the user.

Table 17. Linker Symbols

Symbol	Description
`_low_neardata`	Base of near_data segment after linking.
`_len_neardata`	Length of near_data segment after linking.
`_low_near_romdata`	Base of the rom copy of near_data segment after linking.
`_low_fardata`	Base of far_data segment after linking.
`_len_fardata`	Length of far_data segment after linking.
`_low_far_romdata`	Base of the rom copy of far_data segment after linking.
`_low_nearbss`	Base of near_bss segment after linking.
`_len_nearbss`	Length of near_bss segment after linking.
`_low_farbss`	Base of far_bss segment after linking.
`_len_farbss`	Length of far_bss segment after linking.

Table 17. Linker Symbols

Symbol	Description
`_far_stack`	Top of stack for large model is set as highest address of ERAM.
`_near_stack`	Top of stack for small model is set as highest address of RAM.
`_far_heapbot`	Base of heap for large model is set as highest allocated ERAM address.
`_near_heapbot`	Base of heap for small model is set as highest allocated RAM address.
`_far_heaptop`	Top of heap for large model is set as highest address of ERAM.
`_near_heaptop`	Top of heap for small model is set as highest address of RAM.
_SYS_CLK_FREQ	System clock frequency as selected in the Configure Target dialog box.
_SYS_CLK_SRC	System clock source as selected in the Configure Target dialog box.

Sample Linker Command File

The sample default linker command file for large compilation model is discussed here as a good example of the contents of a linker command file in practice and how the linker commands it contains work to configure your load file. The default linker command file is automatically generated by the ZDS II IDE. If the project name is `test.zdspro`, for example, the default linker command file name is `test_debug.linkcmd`. You can add additional directives to the linking process by specifying them in the **Additional Linker Directives** dialog box (see Additional Directives on page 62). Alternatively, you can define your own linker command file by selecting the **Use Existing** button (see Use Existing on page 63).

The most important of the linker commands and options in the default linker command file are hereby described individually, in the order in which they are typically found in the linker command file:

```
    -FORMAT=OMF695, INTEL32
    -map -maxhexlen=64 -quiet -warnoverlap -NOxref -
  unresolved=fatal
    -sort NAME=ascending -warn –debug -NOigcase
```

In this command, the linker output file format is selected to be OMF695, which is based on the IEEE 695 object file format, and INTEL32, which is the Intel Hex 32 format. This setting is generated from options selected in the **Output** page (see page 72). The `–quiet`, `-debug`, and `–noigcase` options are generated from the settings on the **General** page (see page 47). The other options shown here are all generated from the settings selected in the **Warnings** and **Output** pages (see pages 70– 72).

```
    RANGE ROM $0 : $7fff
    RANGE RAM $ffb000 : $ffbfff
    RANGE EROM $8000 : $1ffff
```

```
RANGE ERAM $800000 : $81ffff
```

The ranges for the four address spaces are defined here. These ranges are taken from the settings in Address Spaces page (see page 68).

```
CHANGE NEAR_TEXT=NEAR_DATA
CHANGE FAR_TEXT=FAR_DATA
```

The NEAR_TEXT and FAR_TEXT segments are renamed to NEAR_DATA and FAR_DATA segments, respectively, by the above command. This allows the linker to merge them with the material that has already been placed into the NEAR_DATA and FAR_DATA segments. The NEAR_TEXT and FAR_TEXT segments contain constant strings in RAM and ERAM, respectively. This reduces the number of initialized segments from four to two, and the C startup then only must initialize two segments.

```
ORDER FAR_BSS, FAR_DATA
ORDER NEAR_BSS,NEAR_DATA
```

These ORDER commands specify the link order of these segments. The FAR_BSS segment is placed at lower addresses with the FAR_DATA segment immediately following it in the ERAM space. Similarly, NEAR_DATA follows after NEAR_BSS in RAM space.

```
COPY NEAR_DATA EROM
COPY FAR_DATA EROM
```

This COPY command is a linker directive to make the linker place a copy of the initialized data segments NEAR_DATA and FAR_DATA into the EROM address space. At run time, the C start-up module then copies the initialized data back from the EROM address space to the RAM (NEAR_DATA segment) and ERAM (FAR_DATA segment) address spaces. This is the standard method to ensure that variables get their required initialization from a nonvolatile stored copy in a typical embedded application where there is no offline memory such as disk storage from which initialized variables can be loaded.

```
define _low_near_romdata = copy base of NEAR_DATA
define _low_neardata = base of NEAR_DATA
define _len_neardata = length of NEAR_DATA
define _low_far_romdata = copy base of FAR_DATA
define _low_fardata = base of FAR_DATA
define _len_fardata = length of FAR_DATA
define _low_nearbss = base of NEAR_BSS
define _len_nearbss = length of NEAR_BSS
define _low_farbss = base of FAR_BSS
define _len_farbss = length of FAR_BSS
define _far_heapbot = top of ERAM
define _far_heaptop = highaddr of ERAM
define _far_stack = highaddr of ERAM
define _near_heapbot = top of RAM
define _near_heaptop = highaddr of RAM
define _near_stack = highaddr of RAM
```

The list above comprises the linker symbol definitions described in Table 17. They allow the compiler to know the bounds of the different memory areas that must be initialized in different ways by the C start-up module.

```
"c:\sample\test"= \
C:\PROGRA~1\Zilog\ZD3E4C~1.0\lib\startupL.obj, \
.\foo.obj, \
C:\PROGRA~1\Zilog\ZD3E4C~1.0\lib\chelpLD.lib, \
C:\PROGRA~1\Zilog\ZD3E4C~1.0\lib\crtLD.lib, \
C:\PROGRA~1\Zilog\ZD3E4C~1.0\lib\fpLD.lib
```

This final command shows that, in this example, the linker output file is named `test.lod`. The source object file (`foo.obj`) is to be linked with the other modules that are required to make a complete executable load file. In this case, those other modules are the C start-up modules for the large model (`startupl.obj`), the C helper library for the large model with debug (`chelpld.lib`), the C run-time library for the large model with debug (`crtld.lib`), and the floating-point library for that same configuration (`fpld.lib`).

An important point to understand in using the linker is that if you use the Zilog default version of the C run-time library, the linker will link in only those functions that are actually called in your program. This is because the Zilog default library is organized with only one function (or in a few cases, a few closely related functions) in each module. Although the C run-time library contains a very large number of functions from the C standard library, if your application only calls two of those functions, then only those two are linked into your application (plus any functions that are called by those two functions in turn). This means it is safe for you to simply link in a large library such as `chelpLD.lib`, `crtLD.lib`, and `fpLD.lib` as in this example. You do not have to worry about any unnecessary code being linked in and do not have to do the extra work of painstakingly finding the unresolved symbols for yourself and linking only to those specific functions. See Use Default Libraries on page 67 for a further discussion of this area.

ANSI Standard Compliance

The Zilog ZNEO C-Compiler is a freestanding ANSI C compiler, complying with the 1989 ISO standard, which is also known as ANSI Standard X3.159-1989 with some deviations, which are described in the Deviations from ANSI C section on page 195.

Freestanding Implementation

A *freestanding* implementation of the C language is a concept defined in the ANSI standard itself, to accommodate the needs of embedded applications that cannot be expected to provide all of the services of the typical desktop execution environment (which is called a hosted environment in the terms of the standard). In particular, it is presumed that there are no file system and no operating system. The use of the standard term *freestanding imple-*

mentation means that the compiler must contain, at least, a specific subset of the full ANSI C features. This subset consists of those basic language features appropriate to embedded applications. Specifically the list of required header files and associated library functions is minimal, namely <`float.h`>, <`limits.h`>, <`stdarg.h`>, and <`stddef.h`>. A freestanding implementation is allowed to additionally support all or parts of other standard header files but is not required to. The ZNEO C-Compiler, for example, supports a number of additional headers from the standard library, as specified in the Library Files Not Required for Freestanding Implementation section on page 196.

A *conforming implementation* (that is, compiler) is allowed to provide extensions, as long as they do not alter the behavior of any program that uses only the standard features of the language. The Zilog ZNEO C-Compiler uses this concept to provide language extensions that are useful for developing embedded applications and for making efficient use of the resources of the ZNEO CPU. These extensions are described in the Language Extensions section on page 156.

Deviations from ANSI C

The differences between the Zilog ZNEO C-Compiler and the freestanding implementation of the ANSI C Standard consist of both extensions to the ANSI standard and deviations from the behavior described by the standard. The extensions to the ANSI standard are explained in Language Extensions on page 156.

There are a small number of areas in which the ZNEO C-Compiler does not behave as specified by the Standard. These areas are described in the following sections.

Prototype of Main

As per ANSI C, in a freestanding environment, the name and type of the function called at program startup are implementation defined. Also, the effect of program termination is implementation defined.

For compatibility with hosted applications, the ZNEO C-Compiler uses `main()` as the function called at program startup. Because the ZNEO compiler provides a freestanding execution environment, there are a few differences in the syntax for `main()`. The most important of these is that, in a typical small embedded application, `main()` never executes a return as there is no operating system for a value to be returned to and is also not intended to terminate. If `main()` does terminate, and the standard Zilog ZNEO C start-up module is in use, control simply goes to the statement:

```
_exit:
  JP _exit
```

For this reason, in the ZNEO C-Compiler, `main()` must be of type `void`; any returned value is ignored. Also, `main()` is not passed any arguments. The following example presents the prototype for `main()`:

```
void main (void);
```

By contrast, in the hosted environment, the closest allowed form for `main` is:

```
int main (void);
```

Double Treated as Float

The ZNEO C-Compiler does not support a double-precision floating-point type. The type `double` is accepted, but is treated as if it were `float`.

Library Files Not Required for Freestanding Implementation

As noted in Freestanding Implementation on page 194, only four of the standard library header files are required by the standard to be supported in a freestanding compiler such as the ZNEO C-Compiler. However, the compiler does support many of the other standard library headers as well. The supported headers are listed here. The support offered in the Zilog libraries is fully compliant with the Standard except as noted here:

- `<assert.h>`
- `<ctype.h>`
- `<errno.h>`
- `<math.h>`

  The Zilog implementation of the `math.h` library is not fully ANSI-compliant in the general limitations of the handling of floating-point numbers: namely, Zilog does not fully support floating-point NANs, INFINITYs, and related special values. These special values are part of the full ANSI/IEEE 754-1985 floating-point standard that is referenced in the ANSI C Standard.

- `<stddef.h>`
- `<stdio.h>`

  Zilog supports only the portions of `stdio.h` that make sense in the embedded environment. Specifically, Zilog defines the ANSI required functions that do not depend on a file system. For example, `printf` and `sprintf` are supplied but not `fprintf`.

- `<stdlib.h>`

  The `stdlib.h` header is ANSI-compliant in the Zilog library except that the following functions – which are of limited or no use in an embedded environment – are not supplied:

  ```
  strtoul()
  _Exit()
  atexit()
  ```

Warning and Error Messages

This section covers the following topics:

Note: If you see an internal error message, please report it to Technical Support at http://support.zilog.com. Zilog staff will use the information to diagnose or log the problem.

Preprocessor Warning and Error Messages

000 Illegal constant expression in directive. A constant expression made up of constants and macros that evaluate to constants can be the only operands of an expression used in a preprocessor directive.

001 Concatenation at end-of-file. Ignored. An attempt was made to concatenate lines with a backslash when the line is the last line of the file.

002 Illegal token. An unrecognizable token or non-ASCII character was encountered.

003 Illegal redefinition of macro <*name*>. An attempt was made to redefine a macro, and the tokens in the macro definition do not match those of the previous definition.

004 Incorrect number of arguments for macro <*name*>. An attempt was made to call a macro, but too few or too many arguments were given.

005 Unbalanced parentheses in macro call. An attempt was made to call a macro with a parenthesis embedded in the argument list that did not match up.

006 Cannot redefine <*name*> keyword. An attempt was made to redefine a keyword as a macro.

007 Illegal directive. The syntax of a preprocessor directive is incorrect.

008 Illegal "#if" directive syntax. The syntax of a `#if` preprocessor directive is incorrect.

009 Bad preprocessor file. Aborted. An unrecognizable source file was given to the compiler.

010 Illegal macro call syntax. An attempt was made to call a macro that does not conform to the syntax rules of the language.

011 Integer constant too large. An integer constant that has a binary value too large to be stored in 32 bits was encountered.

012 Identifier <*name*> is undefined. The syntax of the identifier is incorrect.

013 Illegal #include argument. The argument to a #include directive must be of the form "*pathname*" or <*filename*>.

014 Macro "<*name*>" requires arguments. An attempt was made to call a macro defined to have arguments and was given none.

015 Illegal "#define" directive syntax. The syntax of the `#define` directive is incorrect.

016 Unterminated comment in preprocessor directive. Within a comment, an end of line was encountered.

017 Unterminated quoted string. Within a quoted string, an end of line was encountered.

018 Escape sequence ASCII code too large to fit in char. The binary value of an escape sequence requires more than 8 bits of storage.

019 Character not within radix. An integer constant was encountered with a character greater than the radix of the constant.

020 More than four characters in string constant. A string constant was encountered having more than four ASCII characters.

021 End of file encountered before end of macro call. The end of file is reached before right parenthesis of macro call.

022 Macro expansion caused line to be too long. The line must be shortened.

023 "##" cannot be the first or last token in a replacement string. The macro definition cannot have the "##" operator in the beginning or end.

024 "#" must be followed by an argument name. In a macro definition, "#" operator must be followed by an argument.

025 Illegal "#line" directive syntax. In `#line` <*linenum*> directive, <*linenum*> must be an integer after macro expansion.

026 Cannot undefine macro "*name*". The syntax of the macro is incorrect.

027 End-of-file found before "#endif" directive. `#if` directive was not terminated with a corresponding `#endif` directive.

028 "#else" not within #if and #endif directives. `#else` directive was encountered before a corresponding `#if` directive.

029 Illegal constant expression. The constant expression in preprocessing directive has invalid type or syntax.

030 Illegal macro name <*name*>. The macro name does not have a valid identifier syntax.

031 Extra "#endif" found. `#endif` directive without a corresponding `#if` directive was found.

032 Division by zero encountered. Divide by zero in constant expression found.

033 Floating point constant over/underflow. In the process of evaluating a floating-point expression, the value became too large to be stored.

034 Concatenated string too long. Shorten the concatenated string.

035 Identifier longer than 32 characters. Identifiers must be 32 characters or shorter.

036 Unsupported CPU "*name*" in pragma. An unknown CPU encountered.

037 Unsupported or poorly formed pragma. An unknown `#pragma` directive encountered.

038 (User-supplied text). A user-created `#error` directive has been encountered. The user-supplied text from the directive is printed with the error message.

Front-End Warning and Error Messages

100 Syntax error. A syntactically incorrect statement, declaration, or expression was encountered.

101 Function "<*name*>" already declared. An attempt was made to define two functions with the same name.

102 Constant integer expression expected. A non-integral expression was encountered where only an integral expression can be.

103 Constant expression overflow. In the process of evaluating a constant expression, value became too large to be stored in 32 bits.

104 Function return type mismatch for "<*name*>". A function prototype or function declaration was encountered that has a different result from a previous declaration.

105 Argument type mismatch for argument <*name*>. The type of an actual parameter does not match the type of the formal parameter of the function called.

106 Cannot take address of unsubscripted array. An attempt was made to take the address of an array with no index. The address of the array is already implicitly calculated.

107 Function call argument cannot be void type. An attempt was made to pass an argument to a function that has type void.

108 Identifier "<*name*>" is not a variable or enumeration constant name. In a declaration, a reference to an identifier was made that was not a variable name or an enumeration constant name.

109 Cannot return a value from a function returning "void". An attempt was made to use a function defined as returning void in an expression.

110 Expression must be arithmetic, structure, union or pointer type. The type of an operand to a conditional expression was not arithmetic, structure, union or pointer type.

111 Integer constant too large. Reduce the size of the integer constant.

112 Expression not compatible with function return type. An attempt was made to return a value from function that cannot be promoted to the type defined by the function declaration.

113 Function cannot return value of type array or function. An attempt was made to return a value of type array or function.

114 Structure or union member may not be of function type. An attempt was made to define a member of structure or union that has type function.

115 Cannot declare a typedef within a structure or union. An attempt was made to declare a typedef within a structure or union.

116 Illegal bit field declaration. An attempt was made to declare a structure or union member that is a bit field and is syntactically incorrect.

117 Unterminated quoted string. Within a quoted string, an end of line was encountered.

118 Escape sequence ASCII code too large to fit in char. The binary value of an escape sequence requires more than 8 bits of storage.

119 Character not within radix. An integer constant was encountered with a character greater than the radix of the constant.

120 More than one character in string constant. A string constant was encountered having more than one ASCII character.

121 Illegal declaration specifier. An attempt was made to declare an object with an illegal declaration specifier.

122 Only "const" and "volatile" may be specified with a struct, union, enum, or typedef. An attempt was made to declare a struct, union, enum, or typedef with a declaration specifier other than const and volatile.

123 Cannot specify both long and short in declaration specifier. An attempt was made to specify both long and short in the declaration of an object.

124 Only type qualifiers may be specified within pointer declarations. An attempt was made to declare a pointer with a declaration specifier other than const and volatile.

125 Identifier "<*name*>" already declared within current scope. An attempt was made to declare two objects of the same name in the same scope.

126 Identifier "<*name*>" not in function argument list, ignored. An attempt was made to declare an argument that is not in the list of arguments when using the old style argument declaration syntax.

127 Name of formal parameter not given. The type of a formal parameter was given in the new style of argument declarations without giving an identifier name.

128 Identifier "<*name*>" not defined within current scope. An identifier was encountered that is not defined within the current scope.

129 Cannot have more than one default per switch statement. More than one default statements were found in a single switch statement.

130 Label "<*name*>" is already declared. An attempt was made to define two labels of the same name in the same scope.

131 Label "<*name*> not declared. A `goto` statement was encountered with an undefined label.

132 "continue" statement not within loop body. A `continue` statement was found outside a body of any loop.

133 "break" statement not within switch body or loop body. A `break` statement was found outside the body of any loop.

134 "case" statement must be within switch body. A `case` statement was found outside the body of any `switch` statement.

135 "default" statement must be within switch body. A `default` statement was found outside the body of any `switch` statement.

136 Case value <*name*> already declared. An attempt was made to declare two cases with the same value.

137 Expression is not a pointer. An attempt was made to dereference a value of an expression whose type is not a pointer.

138 Expression is not a function locator. An attempt was made to use an expression as the address of a function call that does not have a type pointer to function.

139 Expression to left of "." or "->" is not a structure or union. An attempt was made to use an expression as a structure or union, or a pointer to a structure or union, whose type was neither a structure or union, or a pointer to a structure or union.

140 Identifier "<*name*>" is not a member of <*name*> structure. An attempt was made to reference a member of a structure that does not belong to the structure.

141 Object cannot be subscripted. An attempt was made to use an expression as the address of an array or a pointer that was not an array or pointer.

142 Array subscript must be of integral type. An attempt was made to subscript an array with a non integral expression.

143 Cannot dereference a pointer to "void". An attempt was made to dereference a pointer to void.

144 Cannot compare a pointer to a non-pointer. An attempt was made to compare a pointer to a non-pointer.

145 Pointers to different types may not be compared. An attempt was made to compare pointers to different types.

146 Pointers may not be added. It is not legal to add two pointers.

147 A pointer and a non-integral may not be subtracted. It is not legal to subtract a non-integral expression from a pointer.

148 Pointers to different types may not be subtracted. It is not legal to subtract two pointers of different types.

149 Unexpected end of file encountered. In the process of parsing the input file, end of file was reached during the evaluation of an expression, statement, or declaration.

150 Unrecoverable parse error detected. The compiler became confused beyond the point of recovery.

151 Operand must be a modifiable lvalue. An attempt was made to assign a value to an expression that was not modifiable.

152 Operands are not assignment compatible. An attempt was made to assign a value whose type cannot be promoted to the type of the destination.

153 "<*name*>" must be arithmetic type. An expression was encountered whose type was not arithmetic where only arithmetic types are allowed.

154 "<*name*>" must be integral type. An expression was encountered whose type was not integral where only integral types are allowed.

155 "<*name*>" must be arithmetic or pointer type. An expression was encountered whose type was not pointer or arithmetic where only pointer and arithmetic types are allowed.

156 Expression must be an lvalue. An expression was encountered that is not an lvalue where only an lvalue is allowed.

157 Cannot assign to an object of constant type. An attempt was made to assign a value to an object defined as having constant type.

158 Cannot subtract a pointer from an arithmetic expression. An attempt was made to subtract a pointer from an arithmetic expression.

159 An array is not a legal lvalue. Cannot assign an array to an array.

160 Cannot take address of a bit field. An attempt was made to take the address of a bit field.

161 Cannot take address of variable with "register" class. An attempt was made to take the address of a variable with "register" class.

162 Conditional expression operands are not compatible. One operand of a conditional expression cannot be promoted to the type of the other operand.

163 Casting a non-pointer to a pointer. An attempt was made to promote a non-pointer to a pointer.

164 Type name of cast must be scalar type. An attempt was made to cast an expression to a non-scalar type.

165 Operand to cast must be scalar type. An attempt was made to cast an expression whose type was not scalar.

166 Expression is not a structure or union. An expression was encountered whose type was not structure or union where only a structure or union is allowed.

167 Expression is not a pointer to a structure or union. An attempt was made to dereference a pointer with the arrow operator, and the expression's type was not pointer to a structure or union.

168 Cannot take size of void, function, or bit field types. An attempt was made to take the size of an expression whose type is void, function, or bit field.

169 Actual parameter has no corresponding formal parameter. An attempt was made to call a function whose formal parameter list has fewer elements than the number of arguments in the call.

170 Formal parameter has no corresponding actual parameter. An attempt was made to call a function whose formal parameter list has more elements than the number of arguments in the call.

171 Argument type is not compatible with formal parameter. An attempt was made to call a function with an argument whose type is not compatible with the type of the corresponding formal parameter.

172 Identifier "<*name*>" is not a structure or union tag. An attempt was made to use the dot operator on an expression whose type was not structure or union.

173 Identifier "<*name*>" is not a structure tag. The tag of a declaration of a structure object does not have type structure.

174 Identifier "<*name*>" is not a union tag. The tag of a declaration of a union object does not have type union.

175 Structure or union tag "<*name*>" is not defined. The tag of a declaration of a structure or union object is not defined.

176 Only one storage class may be given in a declaration. An attempt was made to give more than one storage class in a declaration.

177 Type specifier cannot have both "unsigned" and "signed". An attempt was made to give both `unsigned` and `signed` type specifiers in a declaration.

178 "unsigned" and "signed" may be used in conjunction only with "int", "long" or "char". An attempt was made to use signed or unsigned in conjunction with a type specifier other than `int`, `long`, or `char`.

179 "long" may be used in conjunction only with "int" or "double". An attempt was made to use long in conjunction with a type specifier other than int or double.

180 Illegal bit field length. The length of a bit field was outside of the range 0-32.

181 Too many initializers for object. An attempt was made to initialize an object with more elements than the object contains.

182 Static objects can be initialized with constant expressions only. An attempt was made to initialize a static object with a non-constant expression.

183 Array "<*name*>" has too many initializers. An attempt was made to initialize an array with more elements than the array contains.

184 Structure "<*name*>" has too many initializers. An attempt was made to initialize a structure with more elements than the structure has members.

185 Dimension size may not be omitted. An attempt was made to omit the dimension of an array which is not the rightmost dimension.

186 First dimension of "<*name*>" may not be omitted. An attempt was made to omit the first dimension of an array which is not external and is not initialized.

187 Dimension size must be greater than zero. An attempt was made to declare an array with a dimension size of zero.

188 Only "register" storage class is allowed for formal parameter. An attempt was made to declare a formal parameter with storage class other than register.

189 Cannot take size of array with missing dimension size. An attempt was made to take the size of an array with an omitted dimension.

190 Identifier "<*name*>" already declared with different type or linkage. An attempt was made to declare a tentative declaration with a different type than a declaration of the same name; or, an attempt was made to declare an object with a different type from a previous tentative declaration.

191 Cannot perform pointer arithmetic on pointer to void. An attempt was made to perform pointer arithmetic on pointer to void.

192 Cannot initialize object with "extern" storage class. An attempt was made to initialize variable with `extern` storage class.

193 Missing "<*name*>" detected. An attempt was made to use a variable without any previous definition or declaration.

194 Recursive structure declaration. A structure member can not be of same type as the structure itself.

195 Initializer is not assignment compatible. The initializer type does not match with the variable being initialized.

196 Empty parameter list is an obsolescent feature. Empty parameter lists are not allowed.

197 No function prototype "<*name*>" in scope. The function <*name*> is called without any previous definition or declaration.

198 "old style" formal parameter declarations are obsolescent. Change the parameter declarations.

201 Only one memory space can be specified. An attempt was made to declare a variable with multiple memory space specifier.

202 Unrecognized/invalid type specifier. An attempt was made to declare a variable with unknown type specifier.

204 Ignoring space specifiers (e.g. near, far, rom) on local, parameter or struct member. An attempt was made to declare a local, parameter, or struct member with a memory space specifier. The space specifier for a local or parameter is decided based on the memory model chosen. The space specifier for a struct member is decided based on the space specifier of the entire struct. Any space specifier on local, parameter, or struct member is ignored.

205 Ignoring const or volatile qualifier. An attempt was made to assign a pointer to a type with const qualifier to a pointer to a type with no const qualifier, or an attempt was made to assign a pointer to a type with volatile qualifier to a pointer to a type with no volatile qualifier.

206 Cannot initialize typedef. An attempt was made to initialize a typedef.

207 Aggregate or union objects may be initialized with constant expressions only.

An attempt was made to initialize an array or struct with non constant expression.

208 Operands are not cast compatible. An attempt was made to cast a variable to an incompatible type, for example, casting a _Far pointer to a _Near pointer.

209 Ignoring space specifier (e.g. near, far) on function. An attempt was made to designate a function as a _Near or a _Far function.

210 Invalid use of placement or alignment option. An attempt was made to use a placement or alignment option on a local or parameter.

212 No previous use of placement or alignment options. An attempt was made to use the _At … directive without any previous use of the _At address directive.

213 Function "<*name*>" must return a value. An attempt was made to return from a non void function without providing a return value.

214 Function return type defaults to int. The return type of the function was not specified so the default return type was assumed. A function that does not return anything should be declared as void.

215 Signed/unsigned mismatch. An attempt was made to assign a pointer to a signed type with a pointer to an unsigned type and vice versa.

Optimizer Warning and Error Messages

250 Missing format parameter to (s)printf. This message is generated when a call to printf or sprintf is missing the format parameter and the inline generation of printf calls is requested. For example, a call of the form

```
printf();
```

251 Cannot preprocess format to (s)printf. This message is generated when the format parameter to printf or sprintf is not a string literal and the inline generation of printf calls is requested. For example, the following code causes this warning:

```
static char msg1 = "x = %4d";
char buff[sizeof(msg1)+4];
sprintf(buff,msg1,x); // WARNING HERE
```

This warning is generated because the line of code is processed by the real printf or sprintf function, so that the primary goal of the inline processing, reducing the code size by removing these functions, is not met.

When this message is displayed, you have three options:

- Deselect the **Generate Printfs Inline** checkbox (see Generate Printfs Inline on page 58) so that all calls to `printf` and `sprintf` are handled by the real `printf` or `sprintf` functions.

- Recode to pass a string literal. For example, the code in the example can be revised as follows:

  ```
  define MSG1 "x = %4d"
  char buff[sizeof(MSG1)+4];
  sprintf(buff,MSG1,x);      // OK
  ```

- Keep the **Generate Printfs Inline** checkbox selected and ignore the warning. This loses the primary goal of the option but results in the faster execution of the calls to printf or sprintf that can be processed at compile time, a secondary goal of the option.

252 Bad format string passed to (s)printf. This warning occurs when the compiler is unable to parse the string literal format and the inline generation of printf calls is requested. A normal call to printf or sprintf is generated (which might also be unable to parse the format).

253 Too few parameters for (s)printf format. This error is generated when there are fewer parameters to a call to printf or sprintf than the format string calls for and the inline generation of printf calls is requested. For example:

```
printf("x = %4d\n");
```

254 Too many parameters for (s)printf format. This warning is generated when there are more parameters to a call to printf or sprintf than the format string calls for and the inline generation of printf calls is requested. For example:

```
printf("x = %4d\n", x, y);
```

The format string is parsed, and the extra arguments are ignored.

255 Missing declaration of (s)printf helper function, variable, or field. This warning is generated when the compiler has not seen the prototypes for the printf or sprintf helper functions it generates calls to. This occurs if the standard include file stdio.h has not been included or if stdio.h from a different release of ZDS II has been included.

256 Cannot preprocess calls to vprintf or vsprintf. This message is generated when the code contains calls to vprintf or vsprintf and the inline generation of printf calls is requested. The reason for this warning and the solutions are similar to the ones for message 201: Can’t preprocess format to (s)printf.

257 Not all paths through “<*name*>” return a value. The function declared with a return type is not returning any value at least on one path in the function.

Code Generator Warning and Error Messages

303 Case value <*number*> already defined. If a case value consists of an expression containing a `sizeof`, its value is not known until code generation time. Thus, it is possible to have two cases with the same value not caught by the front end. Review the `switch` statement closely.

309 Interrupt function <*name*> cannot have arguments. A function declared as an interrupt function cannot have function arguments.

313 Bitfield Length exceeds *x* bits. The compiler only accepts bit-field lengths of 8 bits or less for char bit-fields, 16 bit or less for short bit-fields and 32 bit or less for int and long bit-fields.

Chapter 5. Using the Macro Assembler

You use the Macro Assembler to translate ZNEO assembly language files with the `.asm` extension into relocatable object modules with the `.obj` extension. After your relocatable object modules are complete, you convert them into an executable program using the linker/locator. The Macro Assembler can be configured using the **Assembler** page of the **Project Settings** dialog box (see page 49).

Note: The Command Processor allows you to use commands or script files to automate the execution of a significant portion of the IDE's functionality. For more information about the Command Processor, see Using the Command Processor – see page 359.

The following topics are covered in this section:

- Address Spaces and Segments – see page 212
- Output Files – see page 214
- Source Language Structure – see page 216
- Expressions – see page 220
- Directives – see page 226
- Conditional Assembly – see page 244
- Macros – see page 247
- Labels – see page 250
- Source Language Syntax – see page 252
- Warning and Error Messages – see page 255

Note: For more information about ZNEO CPU instructions, see the Instruction Set Description section in the ZNEO CPU User Manual (UM0188).

Address Spaces and Segments

The ZNEO architecture divides all memory into multiple memory regions which are depicted by address spaces in the assembler. Each address space can have various segments associated with it. A segment is a contiguous set of memory locations within an address space. The segments can be predefined by the assembler or user-defined.

Allocating Processor Memory

All memory locations, whether data or code, must be defined within a segment. There are two types of segments:

Absolute segments. An absolute segment is any segment with a fixed origin. The origin of a segment can be defined with the ORG directive. All data and code in an absolute segment is located at the specified physical memory address.

Relocatable segments. A relocatable segment is a segment without a specified origin. At link time, linker commands are used to specify where relocatable segments are to be located within their space. Relocatable segments can be assigned to different physical memory locations without re-assembling.

Address Spaces

The assembler provides the address spaces listed in Table 18, which represent the memory regions of the ZNEO microcontroller.

Table 18. ZNEO Address Spaces

Space ID	Description
ROM	16-bit addressable read-only memory.
RAM	16-bit addressable read/write memory.
EROM	32-bit addressable code memory.
ERAM	32-bit addressable extended memory.
IODATA	16-bit addressable IO data memory.

Code and data are allocated to these spaces by using segments attached to the space.

Segments

Segments are used to represent regions of memory. Only one segment is considered active at any time during the assembly process. A segment must be defined before setting it as the current segment. Every segment is associated with one and only one address space.

Predefined Segments

For convenience, the segments listed in Table 19 are predefined by the assembler.

Table 19. Predefined Segments

Segment ID	Space	Alignment	Type	Contents
CODE	EROM	2 bytes	Relocatable	Code
EROM_DATA	EROM	1 byte	Relocatable	Constant data, tables, and strings
EROM_TEXT	EROM	1 byte	Relocatable	Constant strings
ROM_DATA	ROM	1 byte	Relocatable	Constant data, tables, and strings
ROM_TEXT	ROM	1 byte	Relocatable	Constant strings
__VECTORS	ROM	1 byte	Absolute	Interrupt vector table
NEAR_DATA	RAM	1 byte	Relocatable	Initialized near data
NEAR_BSS	RAM	1 byte	Relocatable	Uninitialized near data
NEAR_TEXT	RAM	1 byte	Relocatable	Near strings
FAR_DATA	RAM	1 byte	Relocatable	Initialized far data
FAR_BSS	RAM	1 byte	Relocatable	Uninitialized far data
FAR_TEXT	RAM	1 byte	Relocatable	Far strings
IOSEG	IODATA	1 byte	Relocatable	I/O data

User-Defined Segments

You can define a new segment using the following directives:

```
DEFINE MYSEG,SPACE=ROM
SEGMENT MYSEG
```

MYSEG becomes the current segment when the assembler processes the `SEGMENT` directive, and *MYSEG* remains the current segment until a new `SEGMENT` directive appears. *MYSEG* can be used as a segment name in the linker command file.

You can define a new segment in ERAM space using the following directives:

```
DEFINE MYDATA,SPACE=ERAM
SEGMENT MYDATA
```

The `DEFINE` directive creates a new segment and attaches it to a space. For more information about using the DEFINE directive, see the DEFINE section on page 231. The `SEG-`

MENT directive attaches code and data to a segment. The SEGMENT directive makes that segment the current segment. Any code or data following the directive resides in the segment until another SEGMENT directive is encountered. For more information about the SEGMENT directive, see the SEGMENT section on page 235.

A segment can also be defined with a boundary alignment and/or origin.

Alignment. Aligning a segment tells the linker to place all instances of the segment in your program on the specified boundary.

Note: Although a module can enter and leave a segment many times, each module still has only one instance of a segment.

Origin. When a segment is defined with an origin, the segment becomes an absolute segment, and the linker places it at the specified physical address in memory.

Assigning Memory at Link Time

At link time, the linker groups those segments of code and data that have the same name and places the resulting segment in the address space to which it is attached. However, the linker handles relocatable segments and absolute segments differently:

Relocatable segments. If a segment is relocatable, the linker decides where in the address space to place the segment.

Absolute segments. If a segment is absolute, the linker places the segment at the absolute address specified as its origin.

Note: At link time, you can redefine segments with the appropriate linker commands.

Output Files

The assembler creates the following files and names them the name of the source file but with a different extension:

- <*source*>`.lst` contains a readable version of the source and object code generated by the assembler. The assembler creates <*source*>`.lst` unless you deselect the **Generate Listing File (.lst)** checkbox in the **Assembler** page of the **Project Settings** dialog box. See Generate Assembly Listing Files (.lst) – see page 51.

- <*source*>`.obj` is an object file in relocatable OMF695 format. The assembler creates <*source*>`.obj`.

Caution: Do not use source input files with `.lst` or `.obj` extensions. The assembler does not assemble files with these extensions, and therefore the data contained in the files is lost.

Source Listing (.lst) Format

The listing file name is the same as the source file name with a `.lst` file extension. Assembly directives allow you to tailor the content and amount of output from the assembler.

Each page of the listing file (`.lst`) contains the following elements:

- Heading with the assembler version number
- Source input file name
- Date and time of assembly

Source lines in the listing file are preceded by the following elements:

- Include level
- Plus sign (+) if the source line contains a macro
- Line number
- Location of the object code created
- Object code

The include level starts at level A and works its way down the alphabet to indicate nested includes. The format and content of the listing file can be controlled with directives included in the source file:

- TITLE
- NOLIST
- LIST
- MACLIST ON/OFF
- CONDLIST ON/OFF

Note: Error and warning messages follow the source line containing the error(s). A count of the errors and warnings detected is included at the end of the listing output file.

The addresses in the assembly listing are relative. To convert the relative addresses into absolute addresses, select the **Show Absolute Addresses in Assembly Listings** checkbox on the **Output** page of the **Project Settings** dialog box. This option uses the information in the `.src` file (generated by the compiler when the Generate Assembly Source Code checkbox is selected [see Project Settings—Listing Files Page – see page 53]) and the `.map` file to change all of the relative addresses in the assembly listing into absolute addresses.

Object Code (.obj) File

The object code output file name is the same as the source file name with an `.obj` extension. This file contains the relocatable object code in OMF695 format and is ready to be processed by the linker and librarian.

Source Language Structure

This section describes the form of an assembly source file.

General Structure

A line in an assembly source file is either a source line or a comment line. The assembler ignores blank lines. Each line of input consists of ASCII characters terminated by a carriage return. An input line cannot exceed 512 characters.

A backslash (\) at the end of a line is a line continuation. The following line is concatenated onto the end of the line with the backslash, as exemplified in the C programming language. If you place a space or any other character after the backslash, the following line is not treated as a continuation.

Source Line

A source line is composed of an optional label followed by an instruction or a directive. It is possible for a source line to contain only a label field.

Comment Line

A semicolon (;) terminates the scanning action of the assembler. Any text following the semicolon is treated as a comment. A semicolon that appears as the first character causes the entire line to be treated as comment.

Label Field

A label must meet at least one of the following conditions:

- It must be followed by a colon.
- It must start at the beginning of the line, with no preceding white space (start in column 1). When an instruction is in the first column, it is treated as an instruction and not a label.

The first character of a label can be a letter, an underscore (_) , a dollar sign ($), a question mark (?), a period (.) or a pound sign (#). Following characters can include letters, digits, underscores, dollar signs ($), question marks (?), periods (.), or pound signs (#). The label can be followed by a colon (:) that completes the label definition. A label can only be defined once. The maximum label length is 129 characters. See the Labels section on page 250 for more information.

Labels that can be interpreted as hexadecimal numbers are not allowed. For example,

```
ADH:
ABEFH:
```

cannot be used as labels.

For more information, see the Labels section on page 250 and the Hexadecimal Numbers section on page 222.

Instruction

An instruction contains one valid assembler instruction that consists of a mnemonic and its arguments. When an instruction is in the first column, it is treated as an instruction and not a label. Use commas to separate the operands. Use a semicolon or carriage return to terminate the instruction. For more information about ZNEO CPU instructions, see the Instruction Set Description section of the ZNEO CPU User Manual (UM0188).

Directive

A directive tells the assembler to perform a specified task. Use a semicolon or carriage return to terminate the directive. Use spaces or tabs to separate the directive from its operands. See the Directives section on page 226 for more information.

Case Sensitivity

In the default mode, the assembler treats all symbols as case-sensitive. Select the **Ignore Case of Symbols** checkbox of the **General** page in the **Project Settings** dialog box to have the assembler ignore the case of user-defined identifiers (see Ignore Case of Symbols – see page 49). Assembler reserved words are not case-sensitive.

Assembler Rules

Reserved Words

The following list contains reserved words that the assembler uses. You cannot use these words as symbol names or variable names. Also, reserved words are not case-sensitive.

.ALIGN	.ASCII	.ASCIZ	.ASECT
.ASG	.BES	.BLOCK	.BSS
.BYTE	.chip	.COPY	.cpu
.DATA	.DEF	.define	.double
.DW24	.ELIF	.ELSE	.ELSEIF
.EMSG	.ENDIF	.ENDM	.ENDMAC
.ENDMACRO	.ENDSTRUCT	.EQU	.EVAL
.EVEN	.EXTERN	.FCALL	.file
.float	.FRAME	.GLOBAL	.IF
.IFNTRUE	.INCLUDE	.INT	.LIST
.LONG	.MACEND	.MACRO	.MAXBRANCH
.MLIST	.MMSG	.MNOLIST	.NEWBLOCK
.NOLIST	.ORG	.PAGE	.PUBLIC
.REF	.SBLOCK	.SECT	.SET
.SHORT_STACK_FRAME	.SPACE	.STRING	.STRUCT
.TAG	.TEXT	.trio	.USECT
.VAR	.VECTOR	.wmsg	.WORD
.word24	ALIGN	ASCII	ASCIZ
ASECT	B	BFRACT	BLKB
BLKL	BLKP	BLKW	BYTE
C	CHIP	COMMENT	COND
CONDLIST	CPU	DB	DBYTE
DD	DEFB	DEFINE	DF
DL	DS	DW	DW24

ELIF	ELSE	ELSEIF	END
ENDC	ENDIF	ENDM	ENDMAC
ENDMODULE	ENDS	EQ	EQU
EQUAL	ERROR	EXIT	EXTERN
EXTERNAL	FCB	FILE	FP
FRACT	GE	GLOBAL	GLOBALS
GREGISTER	GT	IF	IFDEF
IFDIFF	IFE	IFEQ	IFFALSE
IFMA	IFN	IFNDEF	IFNDIFF
IFNFALSE	IFNMA	IFNSAME	IFNTRUE
IFNZ	IFSAME	IFTRUE	IFZ
INCLUDE	LE	LEADZERO	LFRACT
LIST	LONG	LOW	LOW16
LT	MACCNTR	MACDELIM	MACEND
MACEXIT	MACFIRST	MACLIST	MACNOTE
MAXBRANCH	MESSAGE	MI	NB
NC	NE	NEWPAGE	NOCONDLIST
NOLIST	NOMACLIST	NOSPAN	NOV
NZ	OFF	ON	ORG
ORIGIN	OV	PAGE	PL
POPSEG	PRINT	PT	public
PUSHSEG	PW	R0	R10
R11	R12	R13	R14
R15	R2	R3	R4
R5	R6	R7	R8
R9	RR0	RR1	RR10
RR11	RR12	RR13	RR14
RR15	RR2	RR3	RR4
RR5	RR6	RR7	RR8
RR9	SCOPE	SECTION	SEGMENT
SET	SP	STRING	SUBTITLE
TITLE	UBFRACT	UFRACT	UGE
UGT	ULE	ULFRACT	ULT
VAR	VECTOR	WARNING	word
XDEF	XREF	Z	

Note: Do *not* use the instruction mnemonics or assembler directives as symbol or variable names.

Assembler Numeric Representation

Numbers are represented internally as signed 32-bit integers. The assembler detects an expression operand that is out of range for the intended field and generates appropriate error messages.

Character Strings

Character strings consist of printable ASCII characters enclosed by double (") or single (') quotes. A double quote used within a string delimited by double quotes and a single quote used within a string delimited by single quotes must be preceded by a back slash (\). A single quoted string consisting of a single character is treated as a character constant. The assembler does not insert null character (0's) at the end of a text string automatically unless a 0 is inserted, and a character string cannot be used as an operand. For example:

```
DB "STRING" ; a string
DB 'STRING',0 ; C printable string
DB "STRING\"S" ; embedded quote
DB 'a','b','c' ; character constants
```

Expressions

In most cases, where a single integer value can be used as an operand, an expression can also be used. The assembler evaluates expressions in 32-bit signed arithmetic. Logical expressions are bit-wise operators.

The assembler detects division-by-zero errors and reports an error message. The following sections describe the syntax of writing an expression.

Arithmetic Operators

`<<`	Left Shift
`>>`	Arithmetic Right Shift
`**`	Exponentiation
`*`	Multiplication
`/`	Division
`%`	Modulus

+	Addition
-	Subtraction

Note: You must put spaces before and after the modulus operator to separate it from the rest of the expression.

Relational Operators

For use only in conditional assembly expressions.

`==`	Equal	Synonyms: `.eq.`, `.EQ.`
`!=`	Not Equal	Synonyms: `.ne.`, `.NE.`
`>`	Greater Than	Synonyms: `.gt.`, `.GT.`
`<`	Less Than	Synonyms: `.lt.`, `.LT.`
`>=`	Greater Than or Equal	Synonyms: `.ge.`, `.GE.`
`<=`	Less Than or Equal	Synonyms: `.le.`, `.LE.`

Boolean Operators

&	Bit-wise AND	Synonyms: `.and.`, `.AND.`
|	Bit-wise inclusive OR	Synonyms: `.or.`, `.OR.`
^	Bit-wise exclusive XOR	Synonyms: `.xor.`, `.XOR.`
~	Complement	
!	Boolean NOT	Synonyms: `.not.`, `.NOT.`

LOW and LOW16 Operators

The LOW and LOW16 operators can be used to extract the least significant byte or 16-bit word from an integer expression. The LOW operator extracts the byte starting at bit 0 of the expression; the LOW16 operator extracts the 16-bit word starting at bit 0 of the expression.

For example:

```
x equ %123456
```

```
# LOW (X) ; 8 bits of X starting at bit 0 = 56H
# LOW16 (X) ; 16 bits of X starting at bit 0 = 3456H
```

Decimal Numbers

Decimal numbers are signed 32-bit integers consisting of the characters `0`–`9` inclusive between `-2147483648` and `2147483647`. Positive numbers are indicated by the absence of a sign. Negative numbers are indicated by a minus sign (`-`) preceding the number. Underscores (_) can be inserted between digits to improve readability. For example:

```
1234 ; decimal
-1234 ; negative decimal
1_000_000; decimal number with underscores
_123_; NOT an integer but a name. Underscore can be neither
first nor last character.
```

Hexadecimal Numbers

Hexadecimal numbers are signed 32-bit integers ending with the `h` or `H` suffix or starting with a `%` character and consisting of the characters `0`–`9` and `A`–`F`. A hexadecimal number can have 1 to 8 characters. Positive numbers are indicated by the absence of a sign. Negative numbers are indicated by a minus sign (`-`) preceding the number. Underscores (_) can be inserted between hexadecimal digits to improve readability. For example:

```
ABCDEFFFFH ; hexadecimal
%ABCDEFFF ; hexadecimal
-0FFFFh ; negative hexadecimal
ABCD_EFFFH; hexadecimal number with underscore
ADC0D_H; NOT a hexadecimal number but a name; hexadecimal digit
must follow underscore
```

Binary Numbers

Binary numbers are signed 32-bit integers ending with the character `b` or `B` and consisting of the characters `0` and `1`. A binary number can have 32 characters. Positive numbers are indicated by the absence of a sign. Negative numbers are indicated by a minus sign (`-`) preceding the number. Underscores (_) can be inserted between binary digits to improve readability. For example:

```
-0101b ; negative binary number
0010_1100_1010_1111B; binary number with underscores
```

Octal Numbers

Octal numbers are signed 32-bit integers ending with the character `o` or `O`, and consisting of the characters `0`–`7`. An octal number can have 1 to 11 characters. Positive numbers are indicated by the absence of a sign. Negative numbers are indicated by a minus sign (-) preceding the number. Underscores (_) can be inserted between octal digits to improve readability. For example:

```
12340 ; octal number
-12340 ; negative octal number
1_2340; octal number with underscore
```

Character Constants

A single printable ASCII character enclosed by single quotes (') can be used to represent an ASCII value. This value can be used as an operand value. For example:

```
'A' ; ASCII code for "A"
'3' ; ASCII code for "3"
```

Operator Precedence

Table 20 shows the operator precedence in descending order, with operators of equal precedence on the same line. Operators of equal precedence are evaluated left to right. Parentheses can be used to alter the order of evaluation.

Table 20. Operator Precedence

Level 1	`()`						
Level 2	`~`	`unary-`	`!`	`low`			
Level 3	`**`	`*`	`/`	`%`			
Level 4	`+`	`-`	`&`	`|`	`^`	`>>`	`<<`
Level 5	`<`	`>`	`<=`	`>=`	`==`	`!=`	

Shift Left (<<) and OR (|) have the same operator precedence and are evaluated from left to right. If you must alter the order of evaluation, add parentheses to ensure the appropriate operator precedence. For example:

```
ld r0, # 1<<2 | 1<<2 | 1<<1
```

The constant expression in the preceding instruction evaluates to 2A H.

If you want to perform the Shift Left operations before the OR operation, use parentheses as follows:

```
ld r0, #(1<<2)|(1<<2)|(1<<1)
```

The modified constant expression evaluates to 6 H.

Address Spaces and Instruction Encoding

The ZNEO instruction set provides different encodings for many instructions depending on whether an address or immediate data can be represented as an 8-, 16-, or 32-bit value. In most cases, the ZNEO assembler selects the encoding that results in the smallest representation of the instruction.

In doing so, the assembler makes use of the address space information for labels occurring in instructions. Labels in a 32-bit address space (EROM or ERAM) are encoded as 32-bit values, while labels in 16-bit address spaces (ROM, RAM, or IODATA) are encoded as 16-bit values. An otherwise undeclared label is assumed to be in a 16-bit address space.

If you want to override the assembler's encoding, you can indicate the address space for a label or for an absolute address with a colon (:) followed by the name of an address space. For example:

```
  LD R0,#myLabel    ; if myLabel undeclared, gets 16-bit encoding
  LD R0,#myLabel:EROM ; forces 32-bit encoding
```

In particular, absolute addresses in the range `8000H-FFFFH` are considered as 32-bit unsigned values. If an address in internal RAM or the IO space is intended, you can obtain the desired result in either of the following ways:

```
  OR C001H:IODATA, R0 ; Specify the required space
  OR FFFF_C0001,R0     ; or sign extend the address
```

The following example illustrates these features:

```
  SEGMENT NEAR_DATA
nw1: DL %1          ; An address in near data, space is RAM
  SEGMENT FAR_DATA
fw1: DL %2          ; An address in far data, space is ERAM

SEGMENT CODE
  ADD nw1,R0      ; nw1 is in RAM, uses 16-bit encoding
  ADD fw1,R1      ; fw1 is in ERAM, uses 32-bit encoding
  LD R0,nw2      ; nw2 will be in RAM, uses 16-bit encoding
  LD R1,fw2      ; fw2 will be in ERAM, uses 32-bit encoding
  SUB R0,rw1      ; rw1 declared to be in ROM, uses 16-bit encoding
  SUB R1,erw1     ; erw1 declared to be in EROM, uses 32-bit
encoding
  LD xxx,R0      ; xxx undeclared, 16-bit encoding assumed
  LD yyy:EROM,R1   ; yyy undeclared, 32-bit encoding forced.

  SEGMENT NEAR_BSS
nw2: DS 4            ; nw2 is in near bss, space is RAM
  SEGMENT FAR_BSS
fw2: DS 4           ; fw2 is far bss, space is ERAM

  XREF rw1:ROM   ; rw1 declared to be in ROM
  XREF erw1:EROM ; erw1 declared to be in EROM
```

Register Lists for PUSHM and POPM Instructions

The ZNEO processor provides the `PUSHM` and `POPM` instructions to push or pop multiple registers. (Actually, the ZNEO processor provides the `PUSHMHI`, `PUSHMLO`, `POPMHI`, and `POPMLO` instructions, which each push or pop half the register set.The ZNEO assembler accepts `PUSHM` and `POPM` and generates the lower level instructions.) The list of registers to be pushed or popped can be expressed either as a register list or as an immediate value. If an immediate value is used, the least significant bit represents R0, and the most significant bit represents R15. The following examples illustrate the syntax for register lists:

```
myFunction:
  PUSHM <R0-R2,R8,R12>  ; Save registers R0,R1,R2,R8, and R12
                        ; Code for myFunction
  POPM <R0-R2,R8,R12>  ; Restore Registers
  RET                  ; and return

; The same example, using immediate values:
hisFunction:
  PUSHM #1107H          ; Save registers R0,R1,R2,R8, and R12
                        ; Code for hisFunction
  POPM #1107H          ; Restore Registers
  RET                  ; and return

; The same example, using equates:
RegList    EQU "<R0-R2,R8,R12>"
herFunction:
  PUSHM RegList         ; Save registers
                        ; Code for herFunction
  POPM RegList         ; Restore Registers
  RET                  ; and return
```

Instruction Alignment

Because all ZNEO instructions must be aligned to an even address, the ZNEO assembler implicitly inserts an ALIGN directive in front of each instruction. Thus, the following example assembles correctly:

```
  ; Some code
  RET
_L1: DW %1234 ; Some data, perhaps used by previous routine
  DB %3      ; Warning, next address is odd

myFunction:     ; OK, implicitly aligned to next even address
  LINK #%20
```

Of course, it does no harm to insert an ALIGN 2 or a .EVEN directive ahead of a function, just to be safe.

Directives

Directives control the assembly process by providing the assembler with commands and information. These directives are instructions to the assembler itself and are not part of the microprocessor instruction set. Each of the supported assembler directives is described on the following pages:

- ALIGN – see page 227
- .COMMENT – see page 227
- CPU – see page 227
- Data Directives – see page 228
- DEFINE – see page 231
- DS – see page 232
- END – see page 232
- EQU – see page 233
- INCLUDE – see page 233
- LIST – see page 234
- NOLIST – see page 234
- ORG – see page 235
- SEGMENT – see page 235
- .SHORT_STACK_FRAME – see page 236
- TITLE – see page 236
- VAR – see page 237
- VECTOR – see page 237
- XDEF – see page 238
- XREF – see page 238
- Structures and Unions in Assembly Code – see page 239

ALIGN

Forces the object following to be aligned on a byte boundary that is a multiple of <*value*>.

Synonym

```
.align
```

Syntax

<*align_directive*> = > `ALIGN` <*value*>

Example

```
ALIGN 2
DW EVEN_LABEL
```

.COMMENT

The `.COMMENT` assembler directive classifies a stream of characters as a comment.

The `.COMMENT` assembler directive causes the assembler to treat an arbitrary stream of characters as a comment. The delimiter can be any printable ASCII character. The assembler treats as comments all text between the initial and final delimiter, as well as all text on the same line as the final delimiter.

You must not use a label on this directive.

Synonym

```
COMMENT
```

Syntax

`.COMMENT` *delimiter* [*text*] *delimiter*

Example

```
.COMMENT $ An insightful comment of great import $
```

This text is a comment, delimited by a dollar sign, and spanning multiple source lines. The dollar sign (`$`) is a delimiter that marks the line as the end of the comment block.

CPU

Defines to the assembler which member of the ZNEO family is targeted. From this directive, the assembler can determine which instructions are legal as well as the locations of the interrupt vectors within the ROM space, __VECTOR segment.

Note: The `CPU` directive is used to determine the physical location of the interrupt vectors.

Syntax

<cpu_definition> = > `CPU` = *<cpu_name>*

Example

```
CPU = Z16F2811
```

Data Directives

Data directives allow you to reserve space for specified types of data.

Syntax

<data directive> = > *<type>* *<value_list>*
<type> → `DB`
=> `DL`
=> `DW`
=> `DW24`
=> `BLKB`
=> `BLKL`
=> `BLKW`
<value_list> => *<value>*
=> *<value_list>*,*<value>*
<value> => *<expression>*|*<string_const>*

BLKB Declaration Type

Syntax

`BLKB`*count* [, <*init_value*>]

Examples

```
BLKB 16 ; Allocate 16 uninitialized bytes.
BLKB 16, -1 ; Allocate 16 bytes and initialize them to -1.
```

BLKL Declaration Type

Syntax

`BLKL`*count* [, <*init_value*>]

Examples

```
BLKL 16 ; Allocate 16 uninitialized longs.
BLKL 16, -1 ; Allocate 16 longs and initialize them to -1.
```

BLKW Declaration Type

Syntax

`BLKW`*count* [, <*init_value*>]

Examples

```
BLKW 16 ; Allocate 16 uninitialized words.
BLKW 16, -1 ; Allocate 16 words and initialize them to -1.
```

DB Declaration Type

Synonyms

`.byte`, `.ascii`, `DEFB`, `FCB`, `STRING`, `.STRING`, `byte`, `.asciz`

Syntax

`DB` byte data (8 bits)

Examples

```
DB "Hello World" ; Reserve and initialize 11 bytes.
DB 1,2 ; Reserve 2 bytes. Initialize the
       ; first word with a 1 and the second with a 2.
DB %12 ; Reserve 1 byte. Initialize it with %12.
```

Note: There is no trailing null for the DB declaration type. A trailing null is added for `.asciz` declaration types.

DL Declaration Type

Synonyms

`.long, long`

Syntax

`DL long (32 bits)`

Examples

```
  DL 1,2 ; Reserve 2 long words. Initialize the
         ; first with a 1 and last with a 2.
  DL %12345678 ; Reserve space for 1 long word and
               ; initialize it to %12345678.
```

DW Declaration Type

Synonyms

`.word`, `word`, `.int`

Syntax

`DW`word data (16 bits)

Examples

```
  DW "Hello World" ; Reserve and initialize 11 words.
  DW "Hello" ; Reserve and initialize 5 words.
  DW 1,2 ; Reserve 2 words. Initialize the
         ; first word with a 1 and the second with a 2.
  DW %1234 ; Reserve 1 word and initialize it with %1234.
```

Note: There is no trailing null for the DW declaration type. Each letter gets 16 bits with the upper 8 bits zero.

DW24 Declaration Type

Synonyms

.word24, .trio, .DW24

Syntax

`DW24`word data (24 bits)

Examples

```
dw24 %123456    ; Reserve one 24-bit entity and initialize it with
%123456
.trio %789abc       ; Reserve one 24-bit entity and initialize it
with %798abc
```

DEFINE

Defines a segment with its associated address space, alignment, and origin. You must define a segment before you can use it, unless it is a predefined segment. If a clause is not given, use the default for that definition. For more information about the `SEGMENT` directive, see SEGMENT – see page 235. For a list of predefined segments, see Predefined Segments – see page 213.

Synonym

```
.define
```

Syntax

<*segment_definition*> =>
`DEFINE`<*ident*>[<*space_clause*>][*align_clause*>][<*org_clause*>]

Examples

```
DEFINE near_code ; Uses the defaults of the current
                 ; space, byte alignment and relocatable.
DEFINE irq_table,ORG=%FFF8 ; Uses current space, byte alignment,
                           ; and absolute starting address at
                           ; memory location %FFF8.
```

ALIGN Clause

Allows you to select the alignment boundary for a segment. The linker places modules in this segment on the defined boundary. The multiple, given in bytes, must be a power of two (1, 2, 4, 8, and so on).

Syntax

<*align_clause*> => ,`ALIGN` = <*int_const*>

Example

```
DEFINE fdata,SPACE = ERAM,ALIGN = 2
; Aligns on 2-byte boundary, relocatable.
```

ORG Clause

Allows you to specify where the segment is to be located, making the segment an absolute segment. The linker places the segment at the memory location specified by the `ORG.` The default is no `ORG,` and thus the segment is relocatable.

Syntax

<*org_clause*> => ,`ORG` = <*int_const*>

Synonym

`ORIGIN`

Example

```
DEFINE near_code,ORG = %FFF8
; Uses current space, byte alignment, and absolute starting
; address at memory location %FFF8.
```

SPACE Clause

A `SPACE` clause defines the address space in which the segment resides. The linker groups together segments with the same space identification. See Table 18, “ZNEO Address Spaces,” on page 212 for available spaces.

Syntax

<*space_clause*> => ,`SPACE` = <*indent*>

Example

```
DEFINE fdata,SPACE = ERAM,ALIGN = 2
; Aligns on a 2-byte boundary, relocatable.
```

DS

Defines storage locations that do not need to be initialized.

Synonym

```
.block
```

Syntax

<*define_storage*> => `DS` <*value*>

Example

```
NAME DS 10 ; Reserve 10 bytes of storage.
```

END

Informs the assembler of the end of the source input file. If the operand field is present, it defines the start address of the program. During the linking process, only one module can define the start address; otherwise, an error results. The END directive is optional for those modules that do not define the start address.

Synonym

`.end`

Syntax

<end_directive> => `END`[*<expression>*]

Example

```
  END _start ; Use the value of _start as the program start
address.
```

EQU

Assigns symbolic names to numeric or string values. Any name used to define an equate must not have been previously defined. Other equates and label symbols are allowed in the expression, provided they are previously defined.

Synonyms

`.equ`, `.EQU`, `EQUAL`, `.equal`

Syntax

<label> `EQU` *<expression>*

Examples

```
  length EQU 6 ; 1st dimension of rectangle
  width EQU 11 ; 2nd dimension of rectangle
  area EQU length * width ; area of the rectangle

  reg EQU r7 ; symbolic name of a register
```

INCLUDE

Allows the insertion of source code from another file into the current source file during assembly. The included file is assembled into the current source file immediately after the directive. When the EOF (End of File) of the included file is reached, the assembly resumes on the line after the `INCLUDE` directive.

The file to include is named in the string constant after the `INCLUDE` directive. The file name can contain a path. If the file does not exist, an error results, and the assembly is aborted. A recursive `INCLUDE` also results in an error.

`INCLUDE` files are contained in the listing (`.lst`) file unless a `NOLIST` directive is active.

Synonyms

`.include`, `.copy, COPY`

Syntax

<*include_directive*> => `INCLUDE`[<*string_const*>]

Examples

```
INCLUDE "calc.h" ; include calc header file
INCLUDE "\test\calc.h" ; contains a path name
INCLUDE calc.h ; ERROR, use string constant
```

LIST

Instructs the assembler to send output to the listing file. This mode stays in effect until a `NOLIST` directive is encountered. No operand field is allowed. This mode is the default mode.

Synonyms

`.list`, `.LIST`

Syntax

<*list_directive*> => `LIST`

Example

```
LIST
NOLIST
```

NOLIST

Turns off the generation of the listing file. This mode remains in effect until a `LIST` directive is encountered. No operand is allowed.

Synonym

`.NOLIST`

Syntax

<*nolist_directive*> => `NOLIST`

Example

```
LIST
NOLIST
```

ORG

The `ORG` assembler directive sets the assembler location counter to a specified value in the address space of the current segment.

The `ORG` directive must be followed by an integer constant, which is the value of the new origin.

Synonyms

`ORIGIN`, `.ORG`

Syntax

<*org_directive*> => `ORG` <*int_const*>

Examples

```
ORG %1000 ; Sets the location counter at %1000 in the address space
of current segment
ORG LOOP    ; ERROR, use an absolute constant
```

On encountering the `ORG` assembler directive, the assembler creates a new absolute segment with a name starting with `$$$org`. This new segment is placed in the address space of the current segment, with origin at the specified value and alignment as 1.

Note: Zilog recommends that segments requiring the use of `ORG` be declared as absolute segments from the outset by including an `ORG` clause in the `DEFINE` directive for the segment.

SEGMENT

Specifies entry into a previously defined segment.

The `SEGMENT` directive must be followed by the segment identifier. The default segment is used until the assembler encounters a `SEGMENT` directive. The internal assembler program counter is reset to the previous program counter of the segment when a `SEGMENT` directive is encountered. See Table 19, “Predefined Segments,” on page 213 for the names of predefined segments.

Synonyms

`SECTION`

Syntax

<*segment_directive*> => `SEGMENT` <*ident*>

Example

```
SEGMENT code ; predefined segment
DEFINE data ; user-defined
```

.SHORT_STACK_FRAME

The ZNEO LD and LEA instructions permit a special encoding when an argument is an offset from the frame pointer and the offset can be expressed as a signed 6-bit value (-32 to +31). Normally, the ZNEO assembler chooses the smaller encoding whenever possible; otherwise, the assembler chooses the more general 14-bit encoding.

Also, when the ZNEO assembler encounters a LINK instruction allocating more than 256 bytes of stack frame, it silently substitutes a sequence of LINK and LEA or SUB instructions to obtain the desired result.

For especially tight code, you might prefer to be alerted with an error message when an offset from the frame pointer on an LD or LEA instruction cannot be encoded in a 6-bit field or when the stack size requested in a LINK instruction cannot be encoded in the 8-bit field allowed for a LINK instruction. The .SHORT_STACK_FRAME directive supports such a preference.

Syntax

<*shortfp_directive*> => `.SHORT_STACK_FRAME ON|OFF`

Example

```
.SHORT_STACK_FRAME ON ; Turn short stack frames on
; Section of code that needs to be very tight
.SHORT_STACK_FRAME OFF ; Turn short stack frames off
```

TITLE

Causes a user-defined `TITLE` to be displayed in the listing file. The new title remains in effect until the next `TITLE` directive. The operand must be a string constant.

Synonym

`.title`

Syntax

<*title_directive*> => `TITLE` <*string_const*>

Example

```
TITLE "My Title"
```

VAR

The VAR directive works similarly to an EQU directive except you are allowed to change the value of the label. In the following example, STRVAR is assigned three different values. This would cause an error if EQU was used instead of VAR.

Synonym

.VAR, SET, .SET

Syntax

<label> VAR *<expression>*

Example

```
                              A     6    SEGMENT NEAR_DATA
                              A     7    ALIGN 2
       000000FF               A     8    STRVARVAR FFH
000000 FF                     A     9    DBSTRVAR
                              A    10    SEGMENT ROM_TEXT
000000                        A    11    L__0:
000000 4641494C 4544          A    12    DB"FAILED"
000006 00                     A    13    DB0
                              A    14    SEGMENT NEAR_DATA
                              A    15    ALIGN 2
       00000000               A    16    STRVAR VAR L__0
                              A    17
000002                        A    18    _fail_str:
000002 00                     A    19    DBSTRVAR
                              A    20    SEGMENT ROM_TEXT
000007                        A    21    L__1:
000007 50415353 4544          A    22    DB"PASSED"
00000D 00                     A    23    DB0
       00000007               A    24    STRVAR VAR L__1
                              A    25    SEGMENT NEAR_DATA
                              A    26    ALIGN 2
000004                        A    27    _pass_str:
000004 07                     A    28    DBSTRVAR
```

VECTOR

Initializes an interrupt or reset vector to a program address.

The CPU directive is used to determine the physical location of the interrupt vectors.

Syntax

<vector_directive> => VECTOR *<vector name>* = *<expression>*

<*vector name*> specifies which vector is being selected. For ZNEO, <*vector name*> must be one of the following elements:

ADC	P7AD
C0	PWM_FAULT
C1	PWM_TIMER
C2	RESET
C3	SPI
I2C	SYSEXC
P0AD	TIMER0
P1AD	TIMER1
P2AD	TIMER2
P3AD	UART0_RX
P4AD	UART0_TX
P5AD	UART1_RX
P6AD	UART1_TX

Examples

```
VECTOR SPI = spi_handler
VECTOR P2AD = p2ad_handler
```

XDEF

Defines a list of labels in the current module as an external symbol that are to be made publicly visible to other modules at link time. The operands must be labels that are defined somewhere in the assembly file.

Synonyms

`.global`, `GLOBAL`, `.GLOBAL`, `.public`, `.def`, `public`

Syntax

<*xdef_directive*> => `XDEF` <*ident list*>

Examples

```
XDEF label
XDEF label1,label2,label3
```

XREF

Specifies that a list of labels in the operand field are defined in another module. The reference is resolved by the linker. The labels must not be defined in the current module. This directive optionally specifies the address space in which the label resides.

Synonyms

`.extern`, `EXTERN`, `EXTERNAL`, `.ref`

Syntax

<xref_directive> => `XREF` *<ident_space_list>*
<ident_space_list> => *<ident_space>*
=> *<ident_space_list>*, *<ident_space>*
<ident_space> => *<ident>* [:*<space>*]

Examples

```
XREF label
XREF label1,label2,label3
XREF label:ROM
```

Structures and Unions in Assembly Code

The assembler provides a set of directives to group data elements together, similar to high-level programming language constructs like a C structure or a Pascal record. These directives allow you to declare a structure or union type consisting of various elements, assign labels to be of previously declared structure or union type, and provide multiple ways to access elements at an offset from such labels.

The assembler directives associated with structure and union support are listed in Table 21:

Table 21. Structure and Union Assembler Directives

Assembler Directive	Description
`.STRUCT`	Group data elements in a structure type.
`.ENDSTRUCT`	Denotes end of structure or union type.
`.UNION`	Group data elements in a union type.
`.TAG`	Associate label with a structure or union type.
`.WITH`	A section in which the specified label or structure tag is implicit.
`.ENDWITH`	Denotes end of with section.

These structure and union directives are described in the following sections:

- .STRUCT and .ENDSTRUCT Directives – see page 240
- .TAG Directive – see page 241
- .UNION Directive – see page 242
- .WITH and .ENDWITH Directives – see page 243

.STRUCT and .ENDSTRUCT Directives

A structure is a collection of various elements grouped together under a single name for convenient handling. The `.STRUCT` and `.ENDSTRUCT` directives can be used to define the layout for a structure in assembly by identifying the various elements and their sizes. The `.STRUCT` directive assigns symbolic offsets to the elements of a structure. It does not allocate memory. It merely creates a symbolic template that can be used repeatedly.

The `.STRUCT` and `.ENDSTRUCT` directives have the following form:

[*stag*] `.STRUCT` [*offset* | : *parent*]

[*name_1*] `DS` *count1*

[*name_2*] `DS` *count2*

[*tname*] `.TAG` *stagx* [*count*]

...

[*name_n*] `DS` *count3*

[*ssize*] `.ENDSTRUCT` [*stag*]

The label *stag* defines a symbol to use to reference the structure; the expression *offset*, if used, indicates a starting offset value to use for the first element encountered; otherwise, the starting offset defaults to zero.

If *parent* is specified rather than *offset*, the *parent* must be the name of a previously defined structure, and the *offset* is the size of the parent structure. In addition, each name in the *parent* structure is inserted in the new structure.

Each element can have an optional label, such as *name_1*, which is assigned the value of the element's offset into the structure and which can be used as the symbolic offset. If *stag* is missing, these element names become global symbols; otherwise, they are referenced using the syntax `stag.name`. The directives following the optional label can be any space reserving directive such as `DS`, or the `.TAG` directive (defined below), and the structure offset is adjusted accordingly.

The label *ssize*, if provided, is a label in the global name space and is assigned the size of the structure.

If a label *stag* is specified with the `.ENDSTRUCT` directive, it must match the label that is used for the `.STRUCT` directive. The intent is to allow for code readability with some checking by the assembler.

An example structure definition is:

```
DATE     .STRUCT
MONTH    DS 1
DAY      DS 1
```

YEAR `DS` *2*

DSIZE `.ENDSTRUCT` *DATE*

Note: Directives allowed between `.STRUCT` and `.ENDSTRUCT` are directives that specify size, principally `DS`, `ALIGN`, `ORG`, and `.TAG` and their aliases. Also, `BLKB`, `BLKW`, and `BLKL` directives with one parameter are allowed because they indicate only size.

The following directives are not allowed within `.STRUCT` and `.ENDSTRUCT`:

- Initialization directives (`DB`, `DW`, `DL`, `DF`, and `DD`) and their aliases
- `BLKB`, `BLKW`, and `BLKL` with two parameters because they perform initialization
- Equates (`EQU` and `SET`)
- Macro definitions (`MACRO`)
- Segment directives (`SEGMENT` and `FRAME`)
- Nested `.STRUCT` and `.UNION` directives
- CPU instructions (for example, `LD` and `NOP`)

.TAG Directive

The `.TAG` directive declares or assigns a label to have a structure type. This directive can also be used to define a structure/union element within a structure. The `.TAG` directive does not allocate memory; however, the `.TAG` directive inside a structure reserves space in the structure.

The `.TAG` directive to define a structure/union element has the following form:

[*stag*] `.STRUCT` [*offset* | : *parent*]

[*name_1*] `DS` *count1*

[*name_2*] `DS` *count2*

...

[*tname*] `.TAG` *stagx* [*count*]

...

[*ssize*] `.ENDSTRUCT` [*stag*]

The `.TAG` directive to assign a label to have a structure type has the following form:

[*tname*] `.TAG` *stag* ; Apply *stag* to *tname*

[*tname*] `DS` *ssize* ; Allocate space for *tname*

Once applied to label *tname*, the individual structure elements are applied to *tname* to produce the desired offsets using *tname* as the structure base. For example, the label `tname.name_2` is created and assigned the value `tname + stag.name_2`. If there are any alignment requirements with the structure, the `.TAG` directive attaches the required alignment to the label. The optional *count* on the `.TAG` directive is meaningful only inside a structure definition and implies an array of the `.TAG` structure.

Note: Keeping the space allocation separate allows you to place the `.TAG` declarations that assign structure to a label in the header file in a similar fashion to the `.STRUCT` and `XREF` directives. You can then include the header file in multiple source files wherever the label is used. Make sure to perform the space allocation for the label in only one source file.

Examples of the `.TAG` directive are as follows:

```
DATE   .STRUCT
MONTH   DS 1
DAY     DS 1
YEAR    DS 2
DSIZE   .ENDSTRUCT DATE


NAMELEN EQU 30


EMPLOYEE .STRUCT
NAME      DS NAMELEN
SOCIAL    DS  10
START    .TAG DATE
SALARY    DS 1
ESIZE     .ENDSTRUCT EMPLOYEE


NEWYEARS .TAG DATE
NEWYEARS DS DSIZE
```

The `.TAG` directive in the last example above creates the symbols `NEWYEARS.MONTH`, `NEWYEARS.DAY`, and `NEWYEARS.YEAR`. The space for `NEWYEARS` is allocated by the `DS` directive.

.UNION Directive

The `.UNION` directive is similar to the `.STRUCT` directive, except that the offset is reset to zero on each label. A `.UNION` directive cannot have an offset or parent union. The keyword to terminate a `.UNION` definition is `.ENDSTRUCT`.

The .UNION directive has the following form:

[*stag*] `.UNION`

[*name_1*] `DS` *count1*

[*name_2*] `DS` *count2*

[*tname*] `.TAG` *stagx* [*count*]

...

[*name_n*] `DS` *count3*

[*ssize*] `.ENDSTRUCT` [*stag*]

An example of the `.UNION` directive usage is:

```
BYTES  .STRUCT
B0 DS 1
B1 DS 1
B2 DS 1
B3 DS 1
BSIZE  .ENDSTRUCT BYTES


LONGBYTES .UNION
LDATA BLKL 1
BDATA .TAG BYTES
LSIZE .ENDSTRUCT LONGBYTES
```

.WITH and .ENDWITH Directives

Using the fully qualified names for fields within a structure can result in very long names. The `.WITH` directive allows the initial part of the name to be dropped.

The `.WITH` and `.ENDWITH` directives have the following form:

  `.WITH` *name*

`;` *directives*

  `.ENDWITH` [*name*]

The identifier name may be the name of a previously defined `.STRUCT` or `.UNION`, or an ordinary label to which a structure has been attached using a `.TAG` directive. It can also be the name of an equate or label with no structure attached. Within the `.WITH` section, the assembler attempts to prepend "*name*." to each identifier encountered, and selects the modified name if the result matches a name created by the `.STRUCT`, `.UNION`, or `.TAG` directives.

The `.WITH` directives can be nested, in which case the search is from the deepest level of nesting outward. In the event that multiple names are found, a warning is generated and the first such name is used.

If name is specified with the `.ENDWITH` directive, the name must match that used for the `.WITH` directive. The intent is to allow for code readability with some checking by the assembler.

Examine the following `COMPUTE_PAY` routine.

```
COMPUTE_PAY:
; Enter with pointer to an EMPLOYEE in R2, days in R1
; Return with pay in R0
LD.SB R0,EMPLOYEE.SALARY(R2)
MUL R0,R1
RET
```

The preceding routine could be written using the `.WITH` directive as follows:

```
COMPUTE_PAY:
; Enter with pointer to an EMPLOYEE in R2, days in R1
; Return with pay in R0
.WITH EMPLOYEE
LD.SB R0, SALARY(R2)
 MUL R0,R1
 RET
.ENDWITH EMPLOYEE
```

Conditional Assembly

Conditional assembly is used to control the assembly of blocks of code. Entire blocks of code can be enabled or disabled using conditional assembly directives.

Conditional Assembly Directives

The following conditional assembly directives are allowed:

- IF
- IFDEF
- IFSAME
- IFMA

Any symbol used in a conditional directive must be previously defined by an `EQU` or `VAR` directive. Relational operators can be used in the expression. Relational expressions evaluate to 1 if true, and 0 if false.

If a condition is true, the code body is processed. Otherwise, the code body after an `ELSE` is processed, if included.

The `ELIF` directive allows a case-like structure to be implemented.

Note: Conditional assembly can be nested.

IF

Evaluates a Boolean expression. If the expression evaluates to 0, the result is false; otherwise, the result is true.

Synonyms

`.if`, `.IF`, `IFN`, `IFNZ`, `COND`, `IFTRUE`, `IFNFALSE`, `.IFTRUE`

Syntax

`IF` [<*cond_expression*> <*code_body*>]
[`ELIF` <*cond_expression*> <*code_body*>]
[`ELSE` <*code_body*>]
`ENDIF`

Example

```
IF XYZ ; process code body if XYZ is not 0
 .
 .
 .
<Code Body>
 .
 .
ENDIF
IF XYZ !=3 ; code body 1 if XYZ is not 3
 .
 .
 .
<Code Body 1>
 .
 .
 .
ELIF ABC ; XYZ=3 and ABC is not 0,
 .
 .
 .
<Code Body 2>
 .
```

```
  .
  .
ELSE ; otherwise code body 3
  .
  .
  .
<Code Body 3>
  .
  .
  .
ENDIF
```

IFDEF

Checks for label definition. Only a single label can be used with this conditional. If the label is defined, the result is true; otherwise, the result if false.

Syntax

`IFDEF` <*label*>
 <*code_body*>
`[ELSE`
 <*code_body*>]
`ENDIF`

Example

```
IFDEF XYZ ; process code body if XYZ is defined
  .
  .
  .
<Code Body>
  .
  .
  .
ENDIF
```

IFSAME

Checks to see if two string constants are the same. If the strings are the same, the result is true; otherwise, the result is false. If the strings are not enclosed by quotes, the comma is used as the separator.

Syntax

`IFSAME` <*string_const*> , <*string_const*>
 <*code_body*>
`[ELSE`

```
  <code_body>]
ENDIF
```

IFMA

Used only within a macro, this directive checks to determine if a macro argument has been defined. If the argument is defined, the result is true. Otherwise, the result is false. If <*arg_number*> is 0, the result is TRUE if no arguments were provided; otherwise, the result is FALSE.

Note: `IFMA` refers to argument numbers that are one based (that is, the first argument is numbered one).

Syntax

```
IFMA <arg_number>
  <code_body>
[ELSE
  <code_body>]
ENDIF
```

Macros

Macros allow a sequence of one or more assembly source lines to be represented by a single assembler symbol. In addition, arguments can be supplied to the macro to specify or alter the assembler source lines generated once the macro is expanded. The following sections describe how to define and invoke macros.

Macro Definition

A macro definition must precede the use of the macro. The macro name must be the same for both the definition and the `ENDMACRO` line. The argument list contains the formal arguments that are substituted with actual arguments when the macro is expanded. The arguments can be optionally prefixed with the substitution character (\) in the macro body.

During the invocation of the macro, a token substitution is performed, replacing the formal arguments (including the substitution character, if present) with the actual arguments.

Syntax

<*macroname*>[:]`MACRO`[<*arg*>(,<*arg*>)...]
  <*macro_body*>
`ENDMAC`[RO]<*macroname*>

Example

```
  store: MACRO reg1,reg2,reg3
         ADD reg1,reg2
         LD reg3,reg1
         ENDMAC store
```

Note: The following example contains a subtle error:

```
  BadMac MACRO a,b,c
  DL a,b,c
  MACEND BadMac
```

Recall that b and c are reserved words on ZNEO, used for condition codes on the JP instruction. Thus, they cannot be used as macro arguments. To avoid this and similar errors, it is recommended that you avoid single character names.

Concatenation

To facilitate unambiguous symbol substitution during macro expansion, the concatenation character (&) can be suffixed to symbol names. The concatenation character is a syntactic device for delimiting symbol names that are points of substitution and is devoid of semantic content. The concatenation character, therefore, is discarded by the assembler, when the character has delimited a symbol name. For example:

```
  val_part1 equ 55h
  val_part2 equ 33h
The assembly is:
  value macro par1, par2
    DB par1&_&par2
    macend

    value val,part1
    value val,part2
The generated list file is:
                           A     9         value val,part1
  000000 55                A+    9         DB val_part1
                           A+    9         macend
                           A    10         value val,part2
  000001 33                A+   10         DB val_part2
```

```
                         A+    10         macend
```

Macro Invocation

A macro is invoked by specifying the macro name and following the name with the desired arguments. Use commas to separate the arguments.

Syntax

<*macroname*>[<*arg*>[(,<*arg*>)]...]

Example

```
store R1,R2,R3
```

This macro invocation causes registers R1 and R2 to be added and the result stored in register R3.

Local Macro Labels

Local macro labels allow labels to be used within multiple macro expansions without duplication. When used within the body of a macro, symbols preceded by two dollar signs ($$) are considered local to the scope of the macro and therefore are guaranteed to be unique. The two dollars signs are replaced by an underscore followed by a macro invocation number.

Syntax

$$ <*label*>

Example

```
LJMP: MACRO cc,label
      JP cc,$$lab
      JP label
$$lab:
      ENDMAC
```

Optional Macro Arguments

A macro can be defined to handle omitted arguments using the `IFMA` (if macro argument) conditional directive within the macro. The conditional directive can be used to detect if an argument was supplied with the invocation.

Example

```
MISSING_ARG: MACRO ARG0,ARG1,ARG2
             IFMA 2
```

```
                LD ARG0,ARG1
                ELSE
                LD ARG0,ARG2
                ENDIF
                ENDMACRO MISSING_ARG
```

Invocation

```
MISSING_ARG R1, ,R2 ; missing second arg
```

Result

```
LD R1,R2
```

Note: `IFMA` refers to argument numbers that are one based (that is, the first argument is numbered one).

Exiting a Macro

The `MACEXIT` directive is used to immediately exit a macro. No further processing is performed. However, the assembler checks for proper `if-then` conditional directives. A `MACEXIT` directive is normally used to terminate a recursive macro.

The following example is a recursive macro that demonstrates using `MACEXIT` to terminate the macro.

Example

```
RECURS_MAC: MACRO ARG1,ARG2
            IF ARG1==0
                MACEXIT
            ELSE
              RECURS_MAC ARG1-1, ARG2
              DB ARG2
            ENDIF
            ENDMACRO RECURS_MAC
RECURS_MAC 1, ’a’
```

Labels

Labels are considered symbolic representations of memory locations and can be used to reference such memory locations within an expression. Labels can be anonymous, local, imported or exported by directive, and be contained within a defined space, as described below.

Anonymous Labels

The ZDS II assembler supports anonymous labels. Table 22 lists the reserved symbols provided for this purpose.

Table 22. Anonymous Labels

Symbol	Description
$$	Anonymous label. This symbol can be used as a label an arbitrary number of times.
$B	Anonymous label backward reference. This symbol references the most recent anonymous label defined before the reference.
$F	Anonymous label forward reference. This symbol references the most recent anonymous label defined after the reference.

Local Labels

Any label beginning with a dollar sign (`$`) or ending with a question mark (`?`) is considered to be a local label. The scope of a local label ends when a SCOPE directive is encountered, thus allowing the label name to be reused. A local label cannot be imported or exported.

Example

```
$LOOP:    JP $LOOP ; Infinite branch to $LOOP
LAB?:     JP LAB?     ; Infinite branch to LAB?
          SCOPE       ; New local label scope
$LOOP:    JP $LOOP ; Reuse $LOOP
LAB?:     JP LAB?    ; Reuse LAB?
```

Importing and Exporting Labels

Labels can be imported from other modules using the `EXTERN` or `XREF` directive. A space can be provided in the directive to indicate the label's location. Otherwise, the space of the current segment is used as the location of the label.

Labels can be exported to other modules by use of the `PUBLIC` or `XDEF` directive.

Label Spaces

The assembler makes use of a label's space when checking the validity of instruction operands. Certain instruction operands require that a label be located in a specific space because that instruction can only operate on data located in that space. A label is assigned to a space by one of the following methods:

- The space of the segment in which the label is defined.
- The space provided in the `EXTERN` or `XREF` directive.
- If no space is provided with the `EXTERN` or `XREF` directive, the space check is not performed on the label.

Source Language Syntax

The syntax description that follows is offered here to outline the general assembler syntax. It does not define assembly language instructions.

<source_line>	→	<if_statement>
	→	[<Label_field>]<instruction_field><EOL>
	→	[<Label_field>]<directive_field><EOL>
	→	<Label_field><EOL>
	→	<EOL>
<if_statement>	→	<if_section>
	→	[<else_statement>]
	→	`ENDIF`
<if_section>	→	<if_conditional>
		<code-body>
<if_conditional>	→	`IF`<cond_expression>|
		`IFDEF`<ident>|
		`IFSAME`<string_const>,<string_const>|
		`IFMA`<int_const>
<else_statement>	→	`ELSE` <code_body>|
		`ELIF`<cond_expression>
		<code_body>
		[<else_statement>]
<cond_expression>	→	<expression>|
		<expression><relop><expression>
<relop>	→	`==` | `<` | `>` | `<=` | `=>` | `!=`
<code_body>	→	<source_line>`@`
<label_field>	→	<ident>`:`
<instruction_field>	→	<mnemonic>[<operand>]`@`
<directive_field>	→	<directive>
<mnemonic>	→	valid instruction mnemonic
<operand>	→	<addressing_mode>
	→	<expression>
<addressing_mode>	→	valid instruction addressing mode

<directive>	→	ALIGN<int_const>
	→	<array_definition>
	→	CONDLIST(ON|OFF)
	→	END[<expression>]
	→	<ident>EQU<expression>
	→	ERROR<string_const>
	→	EXIT<string_const>
	→	.FCALL<ident>
	→	FILE<string_const>
	→	.FRAME<ident>,<ident>,<space>
	→	GLOBALS (ON|OFF)
	→	INCLUDE<string_const>
	→	LIST (ON|OFF)
	→	<macro_def>
	→	<macro_invoc>
	→	MACDELIM<char_const>
	→	MACLIST (ON|OFF)
	→	NOLIST
	→	ORG<int_const>
	→	<public_definition>
	→	<scalar_definition>
	→	SCOPE
	→	<segment_definition>
	→	SEGMENT<ident>
	→	SUBTITLE<string_const>
	→	SYNTAX=<target_microprocessor>
	→	TITLE<string_const>
	→	<ident>VAR<expression>
	→	WARNING<string_const>
<array_definition>	→	<type>'['<elements>']'
	→	[<initvalue>(,<initvalue>)@]
<type>	→	DB
	→	DL
	→	DW
	→	DW24
<elements>	→	[<int_const>]
<initvalue>	→	['['<instances>']']<value>
<instances>	→	<int_const>
<value>	→	<expression>|<string_const>

<expression>	→	'('<expression>')'
	→	<expression><binary_op><expression>
	→	<unary_op><expression>
	→	<int_const>
	→	<label>
	→	HIGH<expression>
	→	LOW<expression>
	→	OFFSET<expression>
<binary_op>	→	+
	→	-
	→	*
	→	/
	→	>>
	→	<<
	→	&
	→	|
	→	^
<i>	→	-
	→	~
	→	!
<int_const>	→	digit(digit|'_')@
	→	hexdigit(hexdigit|'_')@H
	→	bindigit(bindigit|'_')@B
	→	<char_const>
<char_const>	→	'any'
<label>	→	<ident>
<string_const>	→	"('\"'|any)@"
<ident>	→	(letter|'_')(letter|'_'|digit|'.')@
<ident_list>	→	<ident>(,<ident>)@
<macro_def>	→	<ident>MACRO[<arg>(<arg>)] <code_body> ENDMAC[RO]<macname>
<macro_invoc>	→	<macname>[<arg>](,<arg>)]
<arg>	→	macro argument
<public_definition>	→	PUBLIC<ident list> EXTERN<ident list>
<scalar_definition>	→	<type>[<value>]
<segment_definition >	→	DEFINE<ident>[<space_clause>] [<align_clause>][<org_clause>]
<space_clause>	→	,SPACE=<space>
<align_clause>	→	,ALIGN=<int_const>

<org_clause>	→	`,ORG=`*<int_const>*
<space>	→	(RAM|ROM|ERAM|EROM|IODATA)

Warning and Error Messages

This section covers warning and error messages for the assembler.

400 Symbol already defined. The symbol has been previously defined.

401 Syntax error. General-purpose error when the assembler recognizes only part of a source line. The assembler might generate multiple syntax errors per source line.

402 Symbol XREF'd and XDEF'd. Label previously marked as externally defined or referenced. This error occurs when an attempt is made to both XREF and XDEF a label.

403 Symbol not a segment. The segment has not been previously defined or is defined as some other symbol type.

404 Illegal EQU. The name used to define an equate has been previously defined or equates and label symbols in an equate expression have not been previously defined.

405 Label not defined. The label has not been defined, either by an XREF or a label definition.

406 Illegal use of XREF's symbol. XDEF defines a list of labels in the current module as an external symbol that are to be made publicly visible to other modules at link time; XREF specifies that a list of labels in the operand field are defined in another module.

407 Illegal constant expression. The constant expression is not valid in this particular context. This error normally occurs when an expression requires a constant value that does not contain labels.

408 Memory allocation error. Not enough memory is available in the specified memory range.

409 Illegal `.elif` directive. There is no matching `.if` for the `.elif` directive.

410 Illegal `.else` directive. There is no matching `.if` for the `.else` directive.

411 Illegal `.endif` directive. There is no matching `.if` for the `.endif` directive.

412 EOF encountered within an `.if`. End-of-file encountered within a conditional directive.

416 Unsupported/illegal directives. General-purpose error when the assembler recognizes only part of a source line. The assembler might generate multiple errors for the directive.

417 Unterminated quoted string. You must terminate a string with a double quote.

418 Illegal symbol name. There are illegal characters in a symbol name.

419 Unrecognized token. The assembler has encountered illegal/unknown character(s).

420 Constant expression overflow. A constant expression exceeded the range of –2147483648 to 2147483648.

421 Division by zero. The divisor equals zero in an expression.

422 Address space not defined. The address space is not one of the defined spaces.

423 File not found. The file cannot be found in the specified path, or, if no path is specified, the file cannot be located in the current directory.

424 XREF or XDEF label in const exp. You cannot use an XREF or XDEF label in an EQU directive.

425 EOF found in macro definition. End of file encountered before ENDMAC(RO) reached.

426 MACRO/ENDMACRO name mismatch. The declared MACRO name does not match the ENDMAC(RO) name.

427 Invalid MACRO arguments. The argument is not valid in this particular instance.

428 Nesting same segment. You cannot nest a segment within a segment of the same name.

429 Macro call depth too deep. You cannot exceed a macro call depth of 25.

430 Illegal ENDMACRO found. No macro definition for the ENDMAC(RO) encountered.

431 Recursive macro call. Macro calls cannot be recursive.

432 Recursive include file. Include directives cannot be recursive.

433 ORG to bad address. The ORG clause specifies an invalid address for the segment.

434 Symbol name too long. The maximum symbol length (33 characters) has been exceeded.

435 Operand out-of-range error. The assembler detects an expression operand that is out of range for the intended field and generates appropriate error messages.

436 Relative branch to XREF label. Do not use the JP instruction with XREF.

437 Invalid array index. A negative number or zero has been used for an array instance index. You must use positive numbers.

438 Label in improper space. Instruction requires label argument to be located in certain address space. The most common error is to have a code label when a data label is required or vice versa.

439 Vector not recognized. The vector name is illegal.

444 Too many initializers. Initializers for array data allocation exceeds array element size.

445 Missing `.$endif` at EOF. There is no matching `.$endif` for the `.$if` directive.

448 Segment stack overflow. Do not allocate returned structures on the stack.

456 Unsupported initialization of RAM. A data directive has been used to try to initialize data in a segment which is located in the RAM, ERAM or IODATA address spaces. Because data in these spaces is volatile (not preserved through system reset or power cycles), the data initialization will not occur unless a copy of the segment is made in the nonvolatile ROM or EROM space then copied into the RAM, ERAM or IODATA space by your system start-up code.

461 Unexpected end-of-file in comment. End-of-file encountered in a multi-line comment

462 Macro redefinition. The macro has been redefined.

464 Obsolete feature encountered. An obsolete feature was encountered.

470 Missing token error. A token must be added.

475 User error. General-purpose error.

476 User warning. General-purpose warning.

480 Relist map file error. A map file will not be generated.

481 Relist file not found error. The map file cannot be found in the specified path, or, if no path is specified, the map file cannot be located in the current directory.

482 Relist symbol not found. Any symbol used in a conditional directive must be previously defined by an `EQU` or `VAR` directive.

483 Relist aborted. A map file will not be generated.

490 Stall or hazard conflict found. A stall or hazard conflict was encountered.

499 General purpose switch error. There was an illegal or improperly formed command line option.

Chapter 6. Using the Linker/Locator

The ZNEO developer's environment linker/locator creates a single executable file from a set of object modules and object libraries. It acts as a linker by linking together object modules and resolving external references to public symbols. It also acts as a locator because it allows you to specify where code and data is stored in the target processor at run time. The executable file generated by the linker can be loaded onto the target system and debugged using the Zilog Developer Studio II.

This section describes the following topics:

- Linker Functions – see page 259
- Invoking the Linker – see page 260
- Linker Commands – see page 260
- Linker Expressions – see page 273
- Sample Linker Map File – see page 280
- Troubleshooting the Linker – see page 295
- Warning and Error Messages – see page 297

Linker Functions

The following five major types of objects are manipulated during the linking process.

Libraries. Object libraries are collections of object modules created by the Librarian.

Modules. Modules are created by assembling a file with the assembler or compiling a file with the compiler and then assembling it.

Address spaces. Each module consists of various address spaces. Address spaces correspond to either a physical or logical block of memory on the target processor. For example, a Harvard architecture that physically divides memory into program and data stores has two physical blocks—each with its own set of addresses. Logical address spaces are often used to divide a large contiguous block of memory to separate data and code. In this case, the address spaces partition the physical memory into two logical address spaces. The memory range for each address space depends on the particular ZNEO family member. For more information about address spaces on ZNEO, see Address Spaces – see page 212.

Groups. A group is a collection of logical address spaces. They are typically used for convenience in locating a set of address spaces together.

Segments. Each address space consists of various segments. Segments are named logical partitions of data or code that form a continuous block of memory. Segments with the same name residing in different modules are concatenated together at link time. Segments are assigned to an address space and can be relocatable or absolute. Relocatable segments can be randomly allocated by the linker; absolute segments are assigned a physical address within its address space. See Segments – see page 213 for more information about using predefined segments, defining new segments, and attaching code and data to segments.

The linker performs the following functions:

- Reads in relocatable object modules and library files and linker commands.
- Resolves external references.
- Assigns absolute addresses to relocatable segments of each address space and group.
- Generates a single executable module to download into the target system.
- Generates a map file.

Invoking the Linker

The linker is automatically invoked when your project is open and you click the Build button or **Rebuild All** button on the Build toolbar (see the Build Toolbar section on page 18). The linker then links the corresponding object modules of the various source files in your project and any additional object/library modules specified in the **Objects and Libraries** page in the **Project Settings** dialog box (discussed on page 64).The linker uses the linker command file to control how these object modules and libraries are linked. The linker command file is automatically generated by ZDS II if the Always Generate from Settings button is selected (see the Always Generate from Settings section on page 61). You can add additional linker commands with the **Additional Linker Directives** dialog box (discussed on page 62). If you want to override the automatically generated linker command file, select the **Use Existing** button (see the Use Existing section on page 63).

The linker can also be invoked from the DOS command line or through the ZDS II Command Processor. For more information about invoking the linker from the DOS command line, see Running ZDS II from the Command Line – see page 349. To invoke the linker through the ZDS II Command Processor, see Using the Command Processor – see page 359.

Linker Commands

This section describes the commands of a linker command file:

- <outputfile>=<module list> – see page 262

- CHANGE – see page 262
- COPY – see page 263
- DEBUG – see page 265
- DEFINE – see page 265
- FORMAT – see page 265
- GROUP – see page 266
- HEADING – see page 266
- LOCATE – see page 266
- MAP – see page 267
- MAXHEXLEN – see page 267
- MAXLENGTH – see page 268
- NODEBUG – see page 268
- NOMAP – see page 268
- NOWARN – see page 268
- ORDER – see page 269
- RANGE – see page 269
- SEARCHPATH – see page 270
- SEQUENCE – see page 270
- SORT – see page 271
- SPLITTABLE – see page 271
- UNRESOLVED IS FATAL – see page 272
- WARN – see page 272
- WARNING IS FATAL – see page 272
- WARNOVERLAP – see page 273

Note: Only the <*outputfile*>=<*module list*> and the `FORMAT` commands are required. All commands and operators are *not* case-sensitive.

<outputfile>=<module list>

This command defines the executable file, object modules, and libraries involved in the linking process. <*module list*> is a list of object module or library path names to be linked together to create the output file. <*output file*> is the base name of the output file generated. The extension of the output file name is determined by the FORMAT command.

Example

```
sample=c:\ZDSII_ZNEO_4.11.0\lib\zilog\startups.obj, \
    test.obj, \
    c:\ZDSII_ZNEO_4.11.0\lib\standard\chelpsd.lib, \
    c:\ZDSII_ZNEO_4.11.0\lib\standard\crtsd.lib, \
    c:\ZDSII_ZNEO_4.11.0\lib\standard\fpsd.lib
```

This command links the two object modules and three library modules to generate the linked output file `sample.lod` in IEEE 695 format when the `format=OMF695` command is present.

Notes: In the preceding example, the \ (backslash) at the end of the first line allows the <*module list*> to extend over several lines in a linker command file.

The backslash to continue the <*module list*> over multiple lines is not supported when this command is entered on the DOS command line.

CHANGE

The `CHANGE` command is used to rename a group, address space, or segment. The `CHANGE` command can also be used to move an address space to another group or to move a segment to another address space.

Syntax

`CHANGE` <*name*> `=` <*newname*>

<*name*> can be a group, address space, or segment.

<*newname*> is the new name to be used in renaming a group, address space, or segment; the name of the group where an address space is to be moved; or the name of the address space where a segment is to be moved.

Note: The linker recognizes the special space `NULL`. `NULL` is not one of the spaces that an object file or library contains in it. If a segment name is changed to `NULL` using the CHANGE command to the linker, the segment is deleted from the linking process. This can be useful if you must link only part of an executable or not write out a particular part of the executable. The predefined space `NULL` can also be used to prevent initialization of data while reserving the segment in the original space using the COPY command. See also the examples for the `COPY` command.

Examples

To change the name of a segment (for example, `strseg`) to another segment name (for example, `codeseg`), use the following command:

```
CHANGE strseg=codeseg
```

To move a segment (for example, `codeseg`) to a different address space (for example, RAM), use the following command:

```
CHANGE codeseg=RAM
```

To not allocate a segment (for example, `dataseg`), use the following command:

```
CHANGE dataseg=NULL
```

COPY

The `COPY` command is used to make a copy of a segment into a specified address space. This is most often used to make a copy of initialized RAM in ROM so that it can be initialized at run time.

Syntax

`COPY` <*segment*> <*name*>*[at<expression>]*

<*segment*> can only be a segment.
<*name*> can only be an address space.

Examples

To make a copy of a code segment in ROM, observe the following procedure:

1. In the assembly code, define a code segment (for example, `codeseg`) to be a segment located in RAM. This is the run-time location of `codeseg`.
2. Use the following linker command:

```
COPY codeseg ROM
```

   The linker places the actual contents associated with `codeseg` in ROM (the load time location) and associates RAM (the run-time location) addresses with labels in `codeseg`.

Note: You must copy the `codeseg` contents from ROM to RAM at run time as part of the start-up routine. You can use the `COPY BASE OF` and `COPY TOP OF` linker expressions to get the base address and top address of the contents in ROM. You can use the BASE OF and TOP OF linker expressions to get the base address and top address of `codeseg`.

To copy multiple segments to ROM, observe the following procedure:

1. In the assembly code, define the segments (for example, `strseg`, `text`, and `codeseg`) to be segments located in RAM. This is the run-time location of the segments.
2. Use the following linker commands:

```
CHANGE strseg=codeseg
CHANGE text=codeseg
COPY codeseg ROM
```

   The linker renames `strseg` and `text` as `codeseg` and performs linking as described in the previous example. You must write only one loop to perform the copy from ROM to RAM at run time, instead of three (one loop each for `strseg`, `text`, and `codeseg`).

To allocate a string segment in ROM but not generate the initialization:

1. In the assembly code, define the string segment (for example, `strsect`) to be a segment located in ROM.
2. Use the following linker command:

```
COPY strsect NULL
```

   The linker associates all of the labels in `strsect` with an address in ROM and does not generate any loadable data for `strsect`. This is useful when ROM is already programmed separately, and the address information is required for linking and debugging.

Notes: The linker recognizes the special space `NULL`. `NULL` is not one of the spaces that an object file or library contains in it. If a segment name is changed to `NULL` using the CHANGE command to the linker, the segment is deleted from the linking process. This can be useful if you must link only part of an executable or not write out a particular part of the executable. The predefined space `NULL` can also be used to prevent initialization of data while reserving the segment in the original space using the COPY command.

Refer to the Linker Expressions section on page 273 for the format to write an expression.

DEBUG

The `DEBUG` command causes the linker to generate debug information for the debugger. This option is applicable only if the executable file is IEEE 695.

Syntax

[-]`DEBUG`

DEFINE

The `DEFINE` command creates a user-defined public symbol at link time. This command is used to resolve external references (`XREF`) used in assemble time.

Syntax

`DEFINE` <*symbol name*> `=` <*expression*>

<*symbol name*> is the name assigned to the public symbol.

<*expression*> is the value assigned to the public symbol.

Example

```
DEFINE copy_size = copy top of data_seg - copy base of data_seg
```

Note: Refer to the Linker Expressions section on page 273 for the format to write an expression.

FORMAT

The `FORMAT` command sets the executable file of the linker according to the following table.

File Type	Option	File Extension
IEEE 695 format	`OMF695`	`.lod`
Intel 32-bit	`INTEL32`	`.hex`

The default setting is `IEEE 695`.

Syntax

[-]`FORMAT=`<*type*>

Example

```
FORMAT = OMF695, INTEL32
```

GROUP

The `GROUP` command provides a method of collecting multiple address spaces into a single manageable entity.

Syntax

`GROUP` <*groupname*> = <*name*>*[*,<*name*>*]*

<*groupname*> can only be a group.

<*name*> can only be an address space.

HEADING

The `HEADING` command enables or disables the form feed (`\f`) characters in the map file output.

Syntax

```
-[NO]heading
```

LOCATE

The `LOCATE` command specifies the address where a group, address space, or segment is to be located. If multiple locates are specified for the same space, the last specification takes precedence. A warning is flagged on a `LOCATE` of an absolute segment.

Note: The `LOCATE` of a segment overrides the `LOCATE` of an address space. A `LOCATE` does not override an absolute segment.

Syntax

`LOCATE` <*name*> *AT* <*expression*>

<*name*> can be a group, address space, or segment.

<*expression*> is the address to begin loading.

Example

```
LOCATE ROM AT $1000
```

Note: Refer to the Linker Expressions section on page 273 for the format to write an expression.

MAP

The `MAP` command causes the linker to create a link map file. The link map file contains the location of address spaces, segments, and symbols. The default is to create a link map file. `NOMAP` suppresses the generation of a link map file.

For the ZNEO link map file, the C prefix indicates EROM, the T prefix indicates ROM, the E prefix indicates ERAM, and the R prefix indicates RAM.

Syntax

`-MAP` = *[<mapfile>]*

mapfile has the same name as the executable file with the `.map` extension unless an optional <*mapfile*> is specified.

Example

```
MAP = myfile.map
```

Link Map File

A sample map file is shown in the Sample Linker Map File – see page 280.

MAXHEXLEN

The `MAXHEXLEN` command causes the linker to fix the maximum data record size for the Intel hex output. The default is 64 bytes.

Syntax

```
[-]MAXHEXLEN < IS | = >  < 16 | 32 | 64 | 128 | 255 >
```

Examples

```
-maxhexlen=16
```

or

```
MAXHEXLEN IS 16
```

MAXLENGTH

The `MAXLENGTH` command causes a warning message to be issued if a group, address space, or segment is longer than the specified size. The `RANGE` command sets address boundaries. The `MAXLENGTH` command allows further control of these boundaries.

Syntax

`MAXLENGTH` <*name*> <*expression*>

<*name*> can be a group, address space, or segment.

<*expression*> is the maximum size.

Example

```
MAXLENGTH CODE $FF
```

Note: Refer to the Linker Expressions section on page 273 for the format to write an expression.

NODEBUG

The `NODEBUG` command suppresses the linker from generating debug information. This option is applicable only if the executable file is IEEE 695.

Syntax

```
[-]NODEBUG
```

NOMAP

The `NOMAP` command suppresses generation of a link map file. The default is to generate a link map file.

Syntax

```
[-]NOMAP
```

NOWARN

The `NOWARN` command suppresses warning messages. The default is to generate warning messages.

Syntax

`[-]NOWARN`

ORDER

The `ORDER` command establishes a linking sequence and sets up a dynamic `RANGE` for contiguously mapped address spaces. The base of the `RANGE` of each consecutive address space is set to the top of its predecessor.

Syntax

`ORDER` *<name>[,<name-list>]*

<name> can be a group, address space, or segment. *<name-list>* is a comma-separated list of other groups, address spaces, or segments. However, a `RANGE` is established only for an address space.

Example

```
ORDER GDATA,GTEXT
```

where `GDATA` and `GTEXT` are groups.

In this example, all address spaces associated with `GDATA` are located before (that is, at lower addresses than) address spaces associated with `GTEXT`.

RANGE

The `RANGE` command sets the lower and upper bounds of a group, address space, or segment. If an address falls outside of the specified `RANGE`, the system displays a message.

Note: You must use white space to separate the colon from the `RANGE` command operands.

Syntax

RANGE *<name><expression> : <expression>[,<expression> : <expression>...]*

<name> can be a group, address space, or segment. The first *<expression>* marks the lower boundary for a specified address `RANGE`. The second *<expression>* marks the upper boundary for a specified address `RANGE`.

Example

```
RANGE EROM $008000 : $01FFFF,$040000 : $04FFFF
```

If a `RANGE` is specified for a segment, this range must be within any `RANGE` specified by that segment's address space.

Note: Refer to the Linker Expressions section on page 273 for the format to write an expression.

SEARCHPATH

The `SEARCHPATH` command establishes an additional search path to be specified in locating files. The search order is.

1. Current directory
2. Environment path
3. Search path

Syntax

`SEARCHPATH ="`<*path*>`"`

Example

```
SEARCHPATH="C:\ZDSII_ZNEO_4.11.0\lib\standard"
```

SEQUENCE

The `SEQUENCE` command forces the linker to allocate a group, address space, or segment in the order specified.

Syntax

`SEQUENCE` <*name*>[,<*name_list*>]

<*name*> is either a group, address space, or segment.

<*name_list*> is a comma-separated list of group, address space, or segment names.

Example

```
SEQUENCE NEAR_DATA,NEAR_TEXT,NEAR_BSS
```

Note: If the sequenced segments do *not* all receive space allocation in the first pass through the available address ranges, then the sequence of segments is *not* maintained.

SORT

The SORT command sorts the external symbol listing in the map file by name or address order. The default is to sort in ascending order by name.

Syntax

```
[-]SORT  <ADDRESS | NAME> [IS | =] <ASCENDING | UP | DESCENDING |
DOWN>
```

NAME indicates sorting by symbol name.

ADDRESS indicates sorting by symbol address.

Examples

The following examples show how to sort the symbol listing by the address in ascending order:

```
SORT ADDRESS ASCENDING
```

or

```
-SORT ADDRESS = UP
```

SPLITTABLE

The SPLITTABLE command allows (but does not force) the linker to split a segment into noncontiguous pieces to fit into available memory slots. Splitting segments can be helpful in reducing the overall memory requirements of the project. However, problems can arise if a noncontiguous segment is copied from one space to another using the COPY command. The linker issues a warning if it is asked to COPY any noncontiguous segment.

If SPLITTABLE is not specified for a given segment, the linker allocates the entire segment contiguously.

The SPLITTABLE command takes precedence over the ORDER and SEQUENCE commands.

By default, ZDS II segments are nonsplittable. When multiple segments are made splittable, the linker might re-order segments regardless of what is specified in the ORDER (or SEQUENCE) command. In this case, you must perform one of following actions:

- Modify the memory map of the system so there is only one discontinuity and only one splittable segment in which case the ORDER command is followed
- Modify the project so a specific ordering of segments is not required, in which case multiple segments can be marked splittable

Syntax

`SPLITTABLE` *segment_list*

Example

```
SPLITTABLE CODE, ROM_TEXT
```

UNRESOLVED IS FATAL

The `UNRESOLVED IS FATAL` command causes the linker to treat *undefined external symbol* warnings as fatal errors. The linker quits generating output files immediately if the linker cannot resolve any undefined symbol. By default, the linker proceeds with generating output files if there are any undefined symbols.

Syntax

```
[-] < UNRESOLVED > < IS | = > <FATAL>
```

Examples

```
-unresolved=fatal
```

or

```
UNRESOLVED IS FATAL
```

WARN

The `WARN` command specifies that warning messages are to be generated. An optional warning file can be specified to redirect messages. The default setting is to generate warning messages on the screen and in the map file.

Syntax

[-]`WARN` = *[<warn filename>]*

Example

```
-WARN=warnfile.txt
```

WARNING IS FATAL

The `WARNING IS FATAL` command causes the linker to treat all warning messages as fatal errors. The linker does not generate output file(s) if there are any warnings while linking. By default, the linker proceeds with generating output files even if there are warnings.

Syntax

[-]< WARNING | WARN> < IS | = > <FATAL>

Examples

```
-warn=fatal
```

or

```
WARNING IS FATAL
```

WARNOVERLAP

The `WARNOVERLAP` command enables or disables the warnings when overlap occurs while binding segments. The default is to display the warnings whenever a segment gets overlapped.

Syntax

```
-[NO]warnoverlap
```

Linker Expressions

This section describes the operators and their operands that form legal linker expressions:

- + (Add) – see page 274
- & (And) – see page 274
- BASE OF – see page 275
- COPY BASE – see page 276
- COPY TOP – see page 276
- / (Divide) – see page 276
- FREEMEM – see page 276
- HIGHADDR – see page 277
- LENGTH – see page 277
- LOWADDR – see page 277
- * (Multiply) – see page 278
- Decimal Numeric Values – see page 278
- Hexadecimal Numeric Values – see page 278

The following note applies to many of the <*expression*> commands discussed in this section.

Note: To use BASE, TOP, COPY BASE, COPY TOP, LOWADDR, HIGHADDR, LENGTH, and FREEMEM <*expression*> commands, you must have completed the calculations on the expression. This is done using the SEQUENCE and ORDER commands. Do not use an expression of the segment or space itself to locate the object in question.

Examples

```
  /* Correct example using segments */
  SEQUENCE seg2, seg1 /* Calculate seg2 before seg1 */
  LOCATE seg1 AT TOP OF seg2 + 1

  /* Do not do this: cannot use expression of seg1 to locate seg1
*/
  LOCATE seg1 AT (TOP OF seg2 - LENGTH OF seg1)
```

+ (Add)

The + (Add) operator is used to perform the addition of two expressions.

Syntax

<*expression*> + <*expression*>

& (And)

The & (And) operator is used to perform a bitwise & of two expressions.

Syntax

<expression> & <expression>

BASE OF

The BASE OF operator provides the lowest used address of a group, address space, or segment, excluding any segment copies when <*name*> is a segment. The value of BASE OF is treated as an expression value.

Syntax

BASE OF <*name*>

<*name*> can be a group, address space, or segment.

BASE OF Versus LOWADDR OF

By default, allocation for a given memory group, address space, or segment starts at the lowest defined address for that memory group, address space, or segment. If you explicitly define an assignment within that memory space, allocation for that space begins at that defined point and then occupies subsequent memory locations; the explicit allocation becomes the default BASE OF value. BASE OF <*name*> gives the lowest *allocated* address in the space; LOWADDR OF <*name*> gives the lowest *physical* address in the space.

For example:

```
RANGE ROM $0 : $7FFF
RANGE RAM $8000 : $BFFF

/* RAM allocation */
LOCATE s_uninit_data at $8000
LOCATE s_nvrblock at $9000
DEFINE __low_data = BASE OF s_uninit_data
```

Using

```
LOCATE s_uninit_data at $8000
```

or

```
LOCATE s_uninit_data at LOWADDR OF RAM
```

gives the same address (the lowest possible address) when `RANGE RAM $8000:$BFFF`.

If

```
LOCATE s_uninit_data at $8000
```

is changed to

```
LOCATE s_uninit_data at BASE OF RAM
```

the lowest used address is `$9000` (because `LOCATE s_nvrblock at $9000` and `s_nvrblock` is in RAM).

COPY BASE

The COPY BASE operator provides the lowest used address of a copy segment, group, or address space. The value of COPY BASE is treated as an expression value.

Syntax

`COPY BASE OF` <*name*>

<*name*> can be either a group, address space, or segment.

COPY TOP

The `COPY TOP` operator provides the highest used address of a copy segment, group, or address space. The value of COPY TOP is treated as an expression value.

Syntax

`COPY TOP OF` <*name*>

<*name*> can be a group, address space, or segment.

/ (Divide)

The / (Divide) operator is used to perform division.

Syntax

<expression> / *<expression>*

FREEMEM

The FREEMEM operator provides the lowest address of unallocated memory of a group, address space, or segment. The value of FREEMEM is treated as an expression value.

Syntax

FREEMEM OF <*name*>

<*name*> can be a group, address space, or segment.

HIGHADDR

The HIGHADDR operator provides the highest possible address of a group, address space, or segment. The value of HIGHADDR is treated as an expression value.

Syntax

`HIGHADDR OF` <*name*>

<*name*> can be a group, address space, or segment.

LENGTH

The LENGTH operator provides the length of a group, address space, or segment. The value of LENGTH is treated as an expression value.

Syntax

`LENGTH OF` <*name*>

<*name*> can be a group, address space, or segment.

LOWADDR

The LOWADDR operator provides the lowest possible address of a group, address space, or segment. The value of LOWADDR is treated as an expression value.

Syntax

`LOWADDR OF` <*name*>

<*name*> can be a group, address space, or segment.

BASE OF Versus LOWADDR OF

By default, allocation for a given memory group, address space, or segment starts at the lowest defined address for that memory group, address space, or segment. If you explicitly define an assignment within that memory space, allocation for that space begins at that defined point and then occupies subsequent memory locations; the explicit allocation becomes the default BASE OF value. BASE OF <*name*> gives the lowest *allocated* address in the space; LOWADDR OF <*name*> gives the lowest *physical* address in the space.

For example:

```
  RANGE ROM $0 : $7FFF
  RANGE RAM $8000 : $BFFF
```

```
/* RAM allocation */
LOCATE s_uninit_data at $8000
LOCATE s_nvrblock at $9000
DEFINE __low_data = BASE OF s_uninit_data
```

Using

```
LOCATE s_uninit_data at $8000
```

or

```
LOCATE s_uninit_data at LOWADDR OF RAM
```

gives the same address (the lowest possible address) when `RANGE RAM $8000:$BFFF`.

If

```
LOCATE s_uninit_data at $8000
```

is changed to

```
LOCATE s_uninit_data at BASE OF RAM
```

the lowest used address is `$9000` (because `LOCATE s_nvrblock at $9000` and `s_nvrblock` is in RAM).

* (Multiply)

The * (Multiply) operator is used to multiply two expressions.

Syntax

<expression> * *<expression>*

Decimal Numeric Values

Decimal numeric constant values can be used as an expression or part of an expression. Digits are collections of numeric digits from 0 to 9.

Syntax

<digits>

Hexadecimal Numeric Values

Hexadecimal numeric constant values can be used as an expression or part of an expression. Hex digits are collections of numeric digits from 0 to 9 or A to F.

Syntax

$<hexdigits>

| (Or)

The | (Or) operator is used to perform a bitwise inclusive | (Or) of two expressions.

Syntax

<expression> | *<expression>*

<< (Shift Left)

The << (Shift Left) operator is used to perform a left shift. The first expression to the left of << is the value to be shifted. The second expression is the number of bits to the left the value is to be shifted.

Syntax

<expression> << *<expression>*

>> (Shift Right)

The >> (Shift Right) operator is used to perform a right shift. The first expression to the left of >> is the value to be shifted. The second expression is the number of bits to the right the value is to be shifted.

Syntax

<expression> >> *<expression>*

- (Subtract)

The - (Subtract) operator is used to subtract one expression from another.

Syntax

<expression> - *<expression>*

TOP OF

The TOP OF operator provides the highest allocated address of a group, address space, or segment, excluding any segment copies when <*name*> is a segment. The value of TOP OF is treated as an expression value.

Syntax

`TOP OF` <*name*>

<*name*> can be a group, address space, or segment.

If you declare a segment to begin at `TOP OF` another segment, the two segments share one memory location. `TOP OF` give the address of the last used memory location in a segment, not the address of the next available memory location. For example,

```
LOCATE segment2 at TOP OF segment1
```

starts `segment2` at the address of the last used location of `segment1`. To avoid both segments sharing one memory location, use the following syntax:

```
LOCATE segment2 at (TOP OF segment1) + 1
```

^ (Bitwise Exclusive Or)

The ^ operator is used to perform a bitwise exclusive OR on two expressions.

Syntax

<*expression*> ^ <*expression*>

~ (Not)

The ~ (Not) operator is used to perform a one’s complement of an expression.

Syntax

~ <*expression*>

Sample Linker Map File

```
IEEE 695 OMF Linker Version 6.20 (05120604)
Copyright (C) 1999-2004 Zilog, Inc. All Rights Reserved

LINK MAP:

DATE:      Wed Dec 07 13:51:43 2005
```

```
PROCESSOR: assembler
FILES:     C:\PROGRA~1\Zilog\ZDSII_~1.0\lib\zilog\startupexkS.obj
           .\main.obj
           .\Z16F2800100ZCOG.obj
           C:\PROGRA~1\Zilog\ZDSII_~1.0\lib\std\chelpSD.lib
           C:\PROGRA~1\Zilog\ZDSII_~1.0\lib\std\crtSD.lib
           C:\PROGRA~1\Zilog\ZDSII_~1.0\lib\std\fpSD.lib



COMMAND LIST:
=============
/* Linker Command File - Z16F2800100ZCOG Debug */

/* Generated by: */
/* ZDS II - ZNEO 4.11.0 (Build 05120701) */
/* IDE component: b:4.10:05120701 */

/* compiler options */
/* -chartype:U -define:_Z16F2811AL -define:_Z16K_SERIES */
/* -define:_Z16F_SERIES -genprintf -keepasm -NOkeeplst -NOlist */
/* -NOlistinc -model:S */
/* -
stdinc:"C:\PROGRA~1\Zilog\ZDSII_~1.0\include\std;C:\PROGRA~1\Zilog\ZDSII_~1.0\
include\zilog" */
/* -NOregvar -reduceopt -debug -cpu:Z16F2811AL */
/* -asmsw:" -cpu:Z16F2811AL -define:_Z16F2811AL=1 -define:_Z16K_SERIES=1 -
define:_Z16F_SERIES=1 -
include:C:\PROGRA~1\Zilog\ZDSII_~1.0\include\std;C:\PROGRA~1\Zilog\ZDSII_~1.0\
include\zilog" */

/* assembler options */
/* -define:_Z16F2811AL=1 -define:_Z16K_SERIES=1 */
/* -define:_Z16F_SERIES=1 */
/* -
include:"C:\PROGRA~1\Zilog\ZDSII_~1.0\include\std;C:\PROGRA~1\Zilog\ZDSII_~1.0
\include\zilog" */
/* -list -NOlistmac -name -pagelen:56 -pagewidth:80 -quiet -warn */
/* -debug -NOigcase -cpu:Z16F2811AL */

-FORMAT=OMF695,INTEL32
-map -maxhexlen=64 -quiet -warnoverlap -NOxref -unresolved=fatal
-sort NAME=ascending -warn -debug -NOigcase

RANGE ROM $000000 : $007FFF
RANGE RAM $FFB000 : $FFBFFF
RANGE IODATA $FFC000 : $FFFFFF
RANGE EROM $008000 : $01FFFF
RANGE ERAM $800000 : $81FFFF

CHANGE NEAR_TEXT=NEAR_DATA
CHANGE FAR_TEXT=FAR_DATA

ORDER FAR_BSS, FAR_DATA
ORDER NEAR_BSS,NEAR_DATA
COPY NEAR_DATA EROM
COPY FAR_DATA EROM

define _0_exit = 0
define _low_near_romdata = copy base of NEAR_DATA
define _low_neardata = base of NEAR_DATA
define _len_neardata = length of NEAR_DATA
define _low_far_romdata = copy base of FAR_DATA
define _low_fardata = base of FAR_DATA
define _len_fardata = length of FAR_DATA
define _low_nearbss = base of NEAR_BSS
define _len_nearbss = length of NEAR_BSS
define _low_farbss = base of FAR_BSS
define _len_farbss = length of FAR_BSS
define _near_heaptop = highaddr of RAM
define _far_heaptop = highaddr of ERAM
define _far_stack = highaddr of ERAM
define _near_stack = highaddr of RAM
define _near_heapbot = top of RAM
define _far_heapbot = top of ERAM
DEFINE _SYS_CLK_SRC = 2
DEFINE _SYS_CLK_FREQ = 20000000

DEFINE __EXTCT_INIT_PARAM = $c0

DEFINE __EXTCS0_INIT_PARAM = $8012
DEFINE __EXTCS1_INIT_PARAM = $8001
DEFINE __EXTCS2_INIT_PARAM = $0000
DEFINE __EXTCS3_INIT_PARAM = $0000
DEFINE __EXTCS4_INIT_PARAM = $0000
DEFINE __EXTCS5_INIT_PARAM = $0000
DEFINE __PFAF_INIT_PARAM = $ff
DEFINE __PGAF_INIT_PARAM = $ff
DEFINE __PDAF_INIT_PARAM = $ff00
DEFINE __PAAF_INIT_PARAM = $0000
DEFINE __PCAF_INIT_PARAM = $0000
DEFINE __PHAF_INIT_PARAM = $0300
DEFINE __PKAF_INIT_PARAM = $0f

"C:\PROGRA~1\Zilog\ZDSII_~1.0\samples\QUICKS~1\Debug\Z16F2800100ZCOG"=
C:\PROGRA~1\Zilog\ZDSII_~1.0\lib\zilog\startupexkS.obj,  .\main.obj,
.\Z16F2800100ZCOG.obj, C:\PROGRA~1\Zilog\ZDSII_~1.0\lib\std\chelpSD.lib,
C:\PROGRA~1\Zilog\ZDSII_~1.0\lib\std\crtSD.lib,
C:\PROGRA~1\Zilog\ZDSII_~1.0\lib\std\fpSD.lib



SPACE ALLOCATION:
=================

Space                  Base         Top          Size
------------------ ----------- ----------- ---------
EROM                  C:008000    C:008502      503h
RAM                   R:FFB000    R:FFB042       43h
ROM                     T:0000      T:013F      140h


SEGMENTS WITHIN SPACE:
======================

EROM                  Type                 Base        Top          Size
------------------ ------------------- ----------- ----------- ---------
_100zcog_TEXT      normal data            C:008062    C:0082F9      298h
_sputch_TEXT       normal data            C:008494    C:0084AB       18h
_uputch_TEXT       normal data            C:008474    C:008493       20h
CODE               normal data            C:008000    C:008019       1ah
ei_TEXT            normal data            C:008460    C:008467        8h
getchar_TEXT       normal data            C:008468    C:008473        ch
main_TEXT          normal data            C:00801A    C:008061       48h
mstring_TEXT       normal data            C:0084AC    C:0084D9       2eh
NEAR_DATA          * segment copy *       C:0084F6    C:008502        dh
putchar_TEXT       normal data            C:00843E    C:00845F       22h
sio_TEXT           normal data            C:0082FA    C:00843D      144h
t_putch_TEXT       normal data            C:0084DA    C:0084F5       1ch


RAM                   Type                 Base        Top          Size
------------------ ------------------- ----------- ----------- ---------
NEAR_BSS           normal data            R:FFB000    R:FFB035       36h
NEAR_DATA          normal data            R:FFB036    R:FFB042        dh


ROM                   Type                 Base        Top          Size
------------------ ------------------- ----------- ----------- ---------
____flash_option0_ absolute data            T:0000      T:0000        1h
____flash_option1_ absolute data            T:0001      T:0001        1h
____flash_option2_ absolute data            T:0002      T:0002        1h
____flash_option3_ absolute data            T:0003      T:0003        1h
__VECTORS_04       absolute data            T:0004      T:0007        4h
```

```
__VECTORS_08         absolute data         T:0008      T:000B        4h
__VECTORS_0C         absolute data         T:000C      T:000F        4h
__VECTORS_10         absolute data         T:0010      T:0013        4h
__VECTORS_14         absolute data         T:0014      T:0017        4h
__VECTORS_18         absolute data         T:0018      T:001B        4h
__VECTORS_1C         absolute data         T:001C      T:001F        4h
__VECTORS_20         absolute data         T:0020      T:0023        4h
__VECTORS_24         absolute data         T:0024      T:0027        4h
__VECTORS_28         absolute data         T:0028      T:002B        4h
__VECTORS_2C         absolute data         T:002C      T:002F        4h
__VECTORS_30         absolute data         T:0030      T:0033        4h
__VECTORS_34         absolute data         T:0034      T:0037        4h
__VECTORS_38         absolute data         T:0038      T:003B        4h
__VECTORS_3C         absolute data         T:003C      T:003F        4h
__VECTORS_40         absolute data         T:0040      T:0043        4h
__VECTORS_44         absolute data         T:0044      T:0047        4h
__VECTORS_48         absolute data         T:0048      T:004B        4h
__VECTORS_4C         absolute data         T:004C      T:004F        4h
__VECTORS_50         absolute data         T:0050      T:0053        4h
__VECTORS_54         absolute data         T:0054      T:0057        4h
__VECTORS_58         absolute data         T:0058      T:005B        4h
__VECTORS_5C         absolute data         T:005C      T:005F        4h
__VECTORS_60         absolute data         T:0060      T:0063        4h
__VECTORS_64         absolute data         T:0064      T:0067        4h
__VECTORS_68         absolute data         T:0068      T:006B        4h
__VECTORS_6C         absolute data         T:006C      T:006F        4h
ROM_TEXT             normal data           T:0070      T:007B        ch
startup              normal data           T:007C      T:013F       c4h


SEGMENTS WITHIN MODULES:
========================

Module: ..\..\src\boot\common\startupexs.asm (File: startupexkS.obj) Version:
1:0 12/06/2005 16:57:27

    Name                                         Base         Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_04                          T:0004      T:0007        4h
    Segment: NEAR_BSS                            R:FFB000    R:FFB003        4h
    Segment: startup                               T:007C      T:013F       c4h


Module: ..\SOURCE\MAIN.C (File: main.obj) Version: 1:0 12/07/2005 13:51:42

    Name                                         Base         Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: main_TEXT                           C:00801A    C:008061       48h
    Segment: NEAR_BSS                          R:FFB004    R:FFB007        4h
    Segment: ROM_TEXT                            T:0070      T:007B        ch


Module: ..\SOURCE\Z16F2800100ZCOG.C (File: Z16F2800100ZCOG.obj) Version: 1:0
12/07/2005 13:51:43

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: _100zcog_TEXT                     C:008062    C:0082F9      298h
    Segment: __VECTORS_18                        T:0018      T:001B        4h
    Segment: NEAR_BSS                          R:FFB008    R:FFB014        dh
    Segment: NEAR_DATA                         R:FFB036    R:FFB03D        8h


Module: COMMON\EI.C (Library: crtSD.lib) Version: 1:0 12/06/2005 16:58:06

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: ei_TEXT                           C:008460    C:008467        8h


Module: COMMON\FLASH0.C (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: ____flash_option0_segment           T:0000      T:0000        1h


Module: COMMON\FLASH1.C (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: ____flash_option1_segment           T:0001      T:0001        1h


Module: COMMON\FLASH2.C (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: ____flash_option2_segment           T:0002      T:0002        1h


Module: COMMON\FLASH3.C (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45
```

```
    Name                                     Base        Top         Size
    --------------------------------------- ----------- ----------- ---------
    Segment: ____flash_option3_segment          T:0003      T:0003         1h


Module: COMMON\GETCHAR.C (Library: crtSD.lib) Version: 1:0 12/06/2005 16:57:59

    Name                                     Base        Top         Size
    --------------------------------------- ----------- ----------- ---------
    Segment: getchar_TEXT                      C:008468    C:008473        ch


Module: COMMON\PRINT_GLOBALS.C (Library: crtSD.lib) Version: 1:0 12/06/2005
16:58:01

    Name                                     Base        Top         Size
    --------------------------------------- ----------- ----------- ---------
    Segment: NEAR_BSS                          R:FFB015    R:FFB035       21h
    Segment: NEAR_DATA                         R:FFB03F    R:FFB042        4h


Module: COMMON\PRINT_PUTCH.C (Library: crtSD.lib) Version: 1:0 12/06/2005
16:58:01

    Name                                     Base        Top         Size
    --------------------------------------- ----------- ----------- ---------
    Segment: t_putch_TEXT                      C:0084DA    C:0084F5      1ch


Module: COMMON\PRINT_PUTROMSTRING.C (Library: crtSD.lib) Version: 1:0 12/06/
2005 16:58:02

    Name                                     Base        Top         Size
    --------------------------------------- ----------- ----------- ---------
    Segment: mstring_TEXT                      C:0084AC    C:0084D9      2eh


Module: COMMON\PRINT_SPUTCH.C (Library: crtSD.lib) Version: 1:0 12/06/2005
16:58:01

    Name                                     Base        Top         Size
    --------------------------------------- ----------- ----------- ---------
    Segment: _sputch_TEXT                      C:008494    C:0084AB      18h


Module: COMMON\PRINT_UPUTCH.C (Library: crtSD.lib) Version: 1:0 12/06/2005
16:58:01
```

```
    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: _uputch_TEXT                      C:008474    C:008493       20h


Module: COMMON\PUTCHAR.C (Library: crtSD.lib) Version: 1:0 12/06/2005 16:58:02

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: putchar_TEXT                      C:00843E    C:00845F       22h


Module: COMMON\SIO.C (Library: crtSD.lib) Version: 1:0 12/06/2005 16:58:03

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: NEAR_DATA                         R:FFB03E    R:FFB03E        1h
    Segment: sio_TEXT                          C:0082FA    C:00843D      144h


Module: common\ucase.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: CODE                              C:008000    C:008019       1ah


Module: common\vect08.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_08                        T:0008      T:000B        4h


Module: common\vect0c.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_0C                        T:000C      T:000F        4h


Module: common\vect10.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45
```

```
    Name                                         Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_10                         T:0010      T:0013        4h


Module: common\vect14.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                         Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_14                         T:0014      T:0017        4h


Module: common\vect1c.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                         Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_1C                         T:001C      T:001F        4h


Module: common\vect20.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                         Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_20                         T:0020      T:0023        4h


Module: common\vect24.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                         Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_24                         T:0024      T:0027        4h


Module: common\vect28.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                         Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_28                         T:0028      T:002B        4h


Module: common\vect2c.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45
```

```
    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_2C                        T:002C      T:002F        4h


Module: common\vect30.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005 
16:59:45

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_30                        T:0030      T:0033        4h


Module: common\vect34.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005 
16:59:45

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_34                        T:0034      T:0037        4h


Module: common\vect38.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005 
16:59:45

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_38                        T:0038      T:003B        4h


Module: common\vect3c.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005 
16:59:45

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_3C                        T:003C      T:003F        4h


Module: common\vect40.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005 
16:59:45

    Name                                        Base        Top      Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_40                        T:0040      T:0043        4h


Module: common\vect44.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005 
16:59:45
```

```
    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_44                        T:0044      T:0047        4h


Module: common\vect48.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_48                        T:0048      T:004B        4h


Module: common\vect4c.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_4C                        T:004C      T:004F        4h


Module: common\vect50.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_50                        T:0050      T:0053        4h


Module: common\vect54.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_54                        T:0054      T:0057        4h


Module: common\vect58.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_58                        T:0058      T:005B        4h


Module: common\vect5c.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45
```

```
    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_5C                        T:005C      T:005F        4h

Module: common\vect60.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_60                        T:0060      T:0063        4h

Module: common\vect64.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_64                        T:0064      T:0067        4h

Module: common\vect68.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_68                        T:0068      T:006B        4h

Module: common\vect6c.asm (Library: chelpSD.lib) Version: 1:0 12/06/2005
16:59:45

    Name                                        Base        Top       Size
    --------------------------------------- ----------- ----------- ---------
    Segment: __VECTORS_6C                        T:006C      T:006F        4h

EXTERNAL DEFINITIONS:
=====================

Symbol                             Address   Module          Segment
-------------------------------- ----------- --------------- -----------------
---------------
_0_exit                             00000000 (User Defined)
___print_buff                R:FFB015 PRINT_GLOBALS   NEAR_BSS
___print_fmt                 R:FFB022 PRINT_GLOBALS   NEAR_BSS
___print_leading_char          R:FFB033 PRINT_GLOBALS   NEAR_BSS
___print_len                 R:FFB032 PRINT_GLOBALS   NEAR_BSS
```

```
___print_out              R:FFB034 PRINT_GLOBALS  NEAR_BSS
___print_putch            C:0084DA PRINT_PUTCH   t_putch_TEXT
___print_putromstring      C:0084AC PRINT_PUTROMSTR mstring_TEXT
___print_sputch           C:008494 PRINT_SPUTCH   _sputch_TEXT
___print_uputch           C:008474 PRINT_UPUTCH   _uputch_TEXT
___print_xputch            R:FFB03F PRINT_GLOBALS  NEAR_DATA
__EXTCS0_INIT_PARAM              00008012 (User Defined)
__EXTCS1_INIT_PARAM              00008001 (User Defined)
__EXTCS2_INIT_PARAM              00000000 (User Defined)
__EXTCS3_INIT_PARAM              00000000 (User Defined)
__EXTCS4_INIT_PARAM              00000000 (User Defined)
__EXTCS5_INIT_PARAM              00000000 (User Defined)
__EXTCT_INIT_PARAM               000000C0 (User Defined)
__flash_option0                    T:0000 FLASH0
____flash_option0_segment
__flash_option1                    T:0001 FLASH1
____flash_option1_segment
__flash_option2                    T:0002 FLASH2
____flash_option2_segment
__flash_option3                    T:0003 FLASH3
____flash_option3_segment
__PAAF_INIT_PARAM                00000000 (User Defined)
__PCAF_INIT_PARAM                00000000 (User Defined)
__PDAF_INIT_PARAM                0000FF00 (User Defined)
__PFAF_INIT_PARAM                000000FF (User Defined)
__PGAF_INIT_PARAM                000000FF (User Defined)
__PHAF_INIT_PARAM                00000300 (User Defined)
__PKAF_INIT_PARAM                0000000F (User Defined)
__ucase                    C:008000 ucase         CODE
__VECTOR_04                 T:0000 startupexs     __VECTORS_04
__VECTOR_08                 T:0000 vect08        __VECTORS_08
__VECTOR_0C                 T:000C vect0c        __VECTORS_0C
__VECTOR_10                 T:0000 vect10        __VECTORS_10
__VECTOR_14                 T:0000 vect14        __VECTORS_14
__VECTOR_18                T:0000 Z16F2800100ZCOG __VECTORS_18
__VECTOR_1C                 T:0000 vect1c        __VECTORS_1C
__VECTOR_20                 T:0000 vect20        __VECTORS_20
__VECTOR_24                 T:0000 vect24        __VECTORS_24
__VECTOR_28                 T:0000 vect28        __VECTORS_28
__VECTOR_2C                 T:0000 vect2c        __VECTORS_2C
__VECTOR_30                 T:0000 vect30        __VECTORS_30
__VECTOR_34                 T:0000 vect34        __VECTORS_34
__VECTOR_38                 T:0000 vect38        __VECTORS_38
__VECTOR_3C                 T:0000 vect3c        __VECTORS_3C
__VECTOR_40                 T:0000 vect40        __VECTORS_40
__VECTOR_44                 T:0000 vect44        __VECTORS_44
__VECTOR_48                 T:0000 vect48        __VECTORS_48
__VECTOR_4C                 T:0000 vect4c        __VECTORS_4C
```

```
__VECTOR_50                         T:0000 vect50        __VECTORS_50
__VECTOR_54                         T:0000 vect54        __VECTORS_54
__VECTOR_58                         T:0000 vect58        __VECTORS_58
__VECTOR_5C                         T:0000 vect5c        __VECTORS_5C
__VECTOR_60                         T:0000 vect60        __VECTORS_60
__VECTOR_64                         T:0000 vect64        __VECTORS_64
__VECTOR_68                         T:0000 vect68        __VECTORS_68
__VECTOR_6C                         T:0000 vect6c        __VECTORS_6C
__VECTOR_adc                        T:0000 vect2c        __VECTORS_2C
__VECTOR_c0                         T:0000 vect6c        __VECTORS_6C
__VECTOR_c1                         T:0000 vect68        __VECTORS_68
__VECTOR_c2                         T:0000 vect64        __VECTORS_64
__VECTOR_c3                         T:0000 vect60        __VECTORS_60
__VECTOR_i2c                        T:0000 vect24        __VECTORS_24
__VECTOR_p0ad                       T:0000 vect4c        __VECTORS_4C
__VECTOR_p1ad                       T:0000 vect48        __VECTORS_48
__VECTOR_p2ad                       T:0000 vect44        __VECTORS_44
__VECTOR_p3ad                       T:0000 vect40        __VECTORS_40
__VECTOR_p4ad                       T:0000 vect3c        __VECTORS_3C
__VECTOR_p5ad                       T:0000 vect38        __VECTORS_38
__VECTOR_p6ad                       T:0000 vect34        __VECTORS_34
__VECTOR_p7ad                       T:0000 vect30        __VECTORS_30
__VECTOR_pwm_fault                  T:0000 vect5c        __VECTORS_5C
__VECTOR_pwm_timer                  T:0000 vect50        __VECTORS_50
__VECTOR_reset                      T:0000 startupexs    __VECTORS_04
__VECTOR_spi                        T:0000 vect28        __VECTORS_28
__VECTOR_sysexc                     T:0000 vect08        __VECTORS_08
__VECTOR_timer0                    T:0000 Z16F2800100ZCOG __VECTORS_18
__VECTOR_timer1                     T:0000 vect14        __VECTORS_14
__VECTOR_timer2                     T:0000 vect10        __VECTORS_10
__VECTOR_uart0_rx                   T:0000 vect1c        __VECTORS_1C
__VECTOR_uart0_tx                   T:0000 vect20        __VECTORS_20
__VECTOR_uart1_rx                   T:0000 vect54        __VECTORS_54
__VECTOR_uart1_tx                   T:0000 vect58        __VECTORS_58
_c_startup                          T:007C startupexs    startup
_ch                                R:FFB004 MAIN         NEAR_BSS
_EI                                C:008460 EI           ei_TEXT
_errno                             R:FFB000 startupexs   NEAR_BSS
_exit                               T:013E startupexs    startup
_far_heapbot                             007FFFFF (User Defined)
_far_heaptop                             0081FFFF (User Defined)
_far_stack                               0081FFFF (User Defined)
_getch                             C:0083F6 SIO          sio_TEXT
_getchar                           C:008468 GETCHAR      getchar_TEXT
_getState                         C:008062 Z16F2800100ZCOG _100zcog_TEXT
_gpio_init                        C:00815A Z16F2800100ZCOG _100zcog_TEXT
_init_uart                         C:00831A SIO          sio_TEXT
_kbhit                             C:0083CC SIO          sio_TEXT
```

```
_len_farbss                                00000000 (User Defined)
_len_fardata                               00000000 (User Defined)
_len_nearbss                               00000036 (User Defined)
_len_neardata                              0000000D (User Defined)
_low_far_romdata                           00000000 (User Defined)
_low_farbss                                00000000 (User Defined)
_low_fardata                               00000000 (User Defined)
_low_near_romdata                          000084F6 (User Defined)
_low_nearbss                               00FFB000 (User Defined)
_low_neardata                              00FFB036 (User Defined)
_main                               C:00801A MAIN              main_TEXT
_near_heapbot                              00FFB042 (User Defined)
_near_heaptop                              00FFBFFF (User Defined)
_near_stack                                00FFBFFF (User Defined)
_putch                              C:0083AC SIO               sio_TEXT
_putchar                            C:00843E PUTCHAR           putchar_TEXT
_select_port                        C:0082FA SIO               sio_TEXT
_setState                           C:0080A0 Z16F2800100ZCOG _100zcog_TEXT
_State                              R:FFB011 Z16F2800100ZCOG NEAR_BSS
_SYS_CLK_FREQ                              01312D00 (User Defined)
_SYS_CLK_SRC                               00000002 (User Defined)
_sysclk_init                        C:00819C Z16F2800100ZCOG _100zcog_TEXT
_SysClkFreq                         R:FFB009 Z16F2800100ZCOG NEAR_BSS
_SysClkSrc                          R:FFB008 Z16F2800100ZCOG NEAR_BSS
_system_init                        C:0082B0 Z16F2800100ZCOG _100zcog_TEXT
_Ticks                              R:FFB00D Z16F2800100ZCOG NEAR_BSS
_timer_init                         C:008272 Z16F2800100ZCOG _100zcog_TEXT
_timer_isr                          C:0081FC Z16F2800100ZCOG _100zcog_TEXT

125 external symbol(s).


START ADDRESS:
==============
(T:007C) set in module ..\..\src\boot\common\startupexs.asm.


END OF LINK MAP:
================
0 Errors
0 Warnings
```

Troubleshooting the Linker

Review the following questions to learn more about common situations you may encounter when using the linker.

How do I speed up the linker?

Use the following tips to lower linker execution times:

- If you do not need a link map file, deselect the **Generate Map File** checkbox in the **Project Settings** dialog box (**Output** page); see page 73.
- Make sure that all DOS windows are minimized.

How do I generate debug information without generating code?

Use the COPY or CHANGE command in the linker to copy or change a segment to the predefined NULL space. If you copy the segment to the NULL space, the region is still allocated but no data is written for it. If you change the segment to the NULL space, the region is not allocated at all.

The following examples are of commands in the linker command file:

```
COPYmyseg    NULL
CHANGE       myseg = NULL
```

How much memory is my program using?

Unless the **Generate Map File** checkbox is unchecked in the **Project Settings** dialog box (**Output** page), the linker creates a link map file each time it is run. The link map file name is the same as your executable file with the `.map` extension and resides in the same directory as your project file. The link map has a wealth of information about the memory requirements of your program. Views of memory usage from the space, segment, and module perspective are given as are the names and locations of all public symbols. See the Generate Map File section on page 73 and the MAP command on page 267.

How do I create a hex file?

Select **Intel Hex32 Records** from the **Executable Formats** area in the **Project Settings** dialog box (see the Project Settings—Output Page section on page 72).

How do I determine the size of my actual hex code?

Refer to the map file. Unless the **Generate Map File** checkbox is unchecked in the **Project Settings** dialog box (see the Generate Map File section on page 73), the linker creates a link map file each time it is run. The link map file name is the same as your executable file with the `.map` extension and resides in the same directory as your project file.

Warning and Error Messages

Note: If you see an internal error message, please report it to Technical Support at http://support.zilog.com. Zilog staff will use the information to diagnose or log the problem.

This section covers warning and error messages for the linker/locator.

700 Absolute segment "<*name*>" is not on a MAU boundary. The named segment is not aligned on a Minimum Addressable Unit boundary. Padding or a correctly aligned absolute location must be supplied.

701 <*address range error message*>. A group, section, or address space is larger than is specified maximum length.

704 Locate of a type is invalid. Type "<*typename*>". It is not permitted to specify an absolute location for a type.

708 "<*name*>" is not a valid group, space, or segment.

An invalid record type was encountered. Most likely, the object or library file is corrupted.

710 Merging two located spaces "<*space1*> <*space2*>" is not allowed. When merging two or more address spaces, at most one of them can be located absolutely.

711 Merging two located groups "<*group1*> <*group2*>". When merging two or more groups, at most one can be located absolutely.

712 Space "<*space*>" is not located on a segment base. The address space is not aligned with a segment boundary.

713 Space "<*space*>" is not defined. The named address space is not defined.

714 Multiple locates for "<*name*>" have been specified. Multiple absolute locations have been specified for the named group, section, or address space.

715 Module "<*name*>" contains errors or warnings. Compilation of the named module produced a nonzero exit code.

717 Invalid expression. An expression specifying a symbol value could not be parsed.

718 "<*segment*>" is not in the specified range. The named segment is not within the allowed address range.

719 "<*segment*>" is an absolute or located segment. Relocation was ignored. An attempt was made to relocate an absolutely located segment.

720 "<*name*> calls <*name*>" graph node which is not defined. This message provides detailed information about how an undefined function name is called.

721 Help file "<*name*>" not found. The named help file could not be found. You may need to reinstall the development system software.

723 "<*name*>" has not been ordered. The named group, section, or address space does not have an order assigned to it.

724 Symbol <*name*> (<*file*>) is not defined. The named symbol is referenced in the given file, but not defined. Only the name of the file containing the first reference is listed within the parentheses; it can also be referenced in other files.

726 Expression structure could not be stored. Out of memory. Memory to store an expression structure could not be allocated.

727 Group structure could not be stored. Out of memory. Memory to store a group structure could not be allocated.

730 Range structure could not be stored. Out of memory. Memory to store a range structure could not be allocated.

731 File "<*file*>" is not found. The named input file or a library file name or the structure containing a library file name was not found.

732 Error encountered opening file "<*file*>". The named file could not be opened.

736 Recursion is present in call graph. A loop has been found in the call graph, indicating recursion.

738 Segment "<*segment*>" is not defined. The referenced segment name has not been defined.

739 Invalid space "<*space*>" is defined. The named address space is not valid. It must be either a group or an address space.

740 Space "<*space*>" is not defined. The referenced space name is not defined.

742 <*error message*>. A general-purpose error message.

743 Vector "<*vector*>" not defined. The named interrupt vector could not be found in the symbol table.

745 Configuration bits mismatch in file <*file*>. The mode bit in the current input file differs from previous input files.

746 Symbol <*name*> not attached to a valid segment. The named symbol is not assigned to a valid segment.

747 <*message*>. General-purpose error message for reporting out-of-range errors. An address does not fit within the valid range.

748 <*message*>. General-purpose error message for OMF695 to OMF251 conversion. The requested translation could not proceed.

749 Could not allocate global register. A global register was requested, but no register of the desired size remains available.

751 Error opening output file "<*outfile*>". The named load module file could not be opened.

753 Segment '<*segment*>' being copied is splittable. A segment, which is to be copied, is being marked as splittable, but start-up code might assume that it is contiguous.

Chapter 7. Configuring Memory for Your Program

The ZNEO CPU architecture provides a single unified address space for both internal and external memory and I/O. Several address ranges within this space can be configured for various purposes, providing a great deal of flexibility for creating a program configuration tailored to your target design and application needs. The cost of this flexibility is that you must understand how to set up the project settings and initialization to support your preferred configuration. This chapter provides the information required to configure your project to support the programming model that best fits your needs.

This chapter covers the following topics:

- ZNEO Memory Layout – see page 301
- Programmer's Model of ZNEO Memory – see page 303
- Program Configurations – see page 308

The first two sections describe the relationship between the ZNEO CPU's physical memory layout and the functional address ranges available to the programmer. Understanding this relationship is the key to correctly configuring your project. The last section presents several examples of program configuration, covering the configurations that are expected to be most commonly used.

ZNEO Memory Layout

The ZNEO CPU has a unique memory architecture with a unified 24-bit physical address space. (ZNEO CPU effective addresses are 32 bits wide, but current devices ignore bits [31:24].) The physical address space can address four types of memory and I/O, as follows:

- Internal nonvolatile memory
- Internal RAM
- Internal I/O memory and special-function registers (SFRs)
- External memory and memory mapped peripherals

The internal memory and I/O are always present in ZNEO devices, and are located at specific address ranges in the unified address space. External memory or I/O is optional, and its location in the address space is determined by the target hardware design.

To promote code efficiency, the ZNEO CPU supports shorter 16-bit data addressing for the address ranges `00_0000H-00_7FFFH` and `FF_8000H-FF_FFFFH`. 32-bit addressing can also be used in these ranges. The range `00_8000H–FF_7FFFH` requires 32-bit addresses. Figure 130 on page 302 shows the typical address layout of memory types available in the ZNEO architecture.

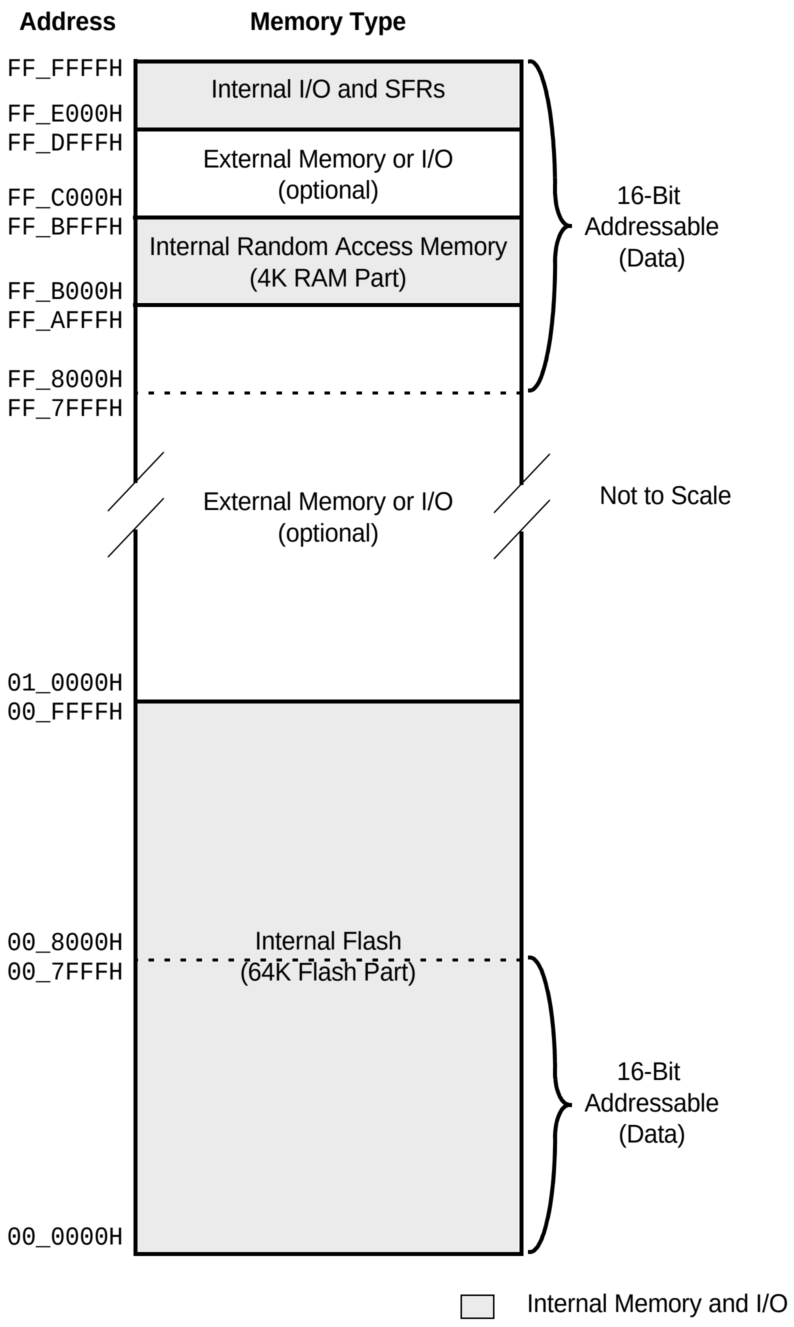

Address
Memory Type
FF_FFFFH
FF_E000H
FF_DFFFH
FF_C000H
FF_BFFFH
FF_B000H
FF_AFFFH
FF_8000H
FF_7FFFH
01_0000H
00_FFFFH
00_8000H
00_7FFFH
00_0000H
Internal I/O and SFRs
External Memory or I/O (optional)
Internal Random Access Memory (4K RAM Part)
External Memory or I/O (optional)
Internal Flash (64K Flash Part)
16-Bit Addressable (Data)
Not to Scale
16-Bit Addressable (Data)
Internal Memory and I/O

Figure 130. Typical ZNEO Physical Memory Layout

ZNEO CPU-based devices have internal nonvolatile memory starting at address `00_0000H`. For example, a device equipped with 128K of Flash has internal nonvolatile memory starting at address `00_0000H` and ending at address `01_FFFFH`.

ZNEO CPU-based devices have internal RAM ending at address `FF_BFFFH`, while the beginning address (and hence the total extent of this area) is device dependent. For example, a device equipped with 4K of RAM has internal RAM starting at address `FF_B000H` and ending at address `FF_BFFFH`.

ZNEO CPU-based devices reserve 8K of addresses for internal memory-mapped I/O, located at addresses `FF_E000H-FF_FFFFH`. This memory contains CPU control registers and other SFRs, on-chip peripherals, and memory-mapped I/O ports.

Finally, ZNEO CPU-based devices provide an external interface that allows seamless connection to external memory and/or peripherals. External memory can be nonvolatile memory such as Flash, volatile (random access) memory, or both.

The external interface supports multiple chip select signals (CSx), which the target system designer can use as required to enable different devices on the external interface. One chip select is asserted whenever an external memory or I/O address is accessed. Each chip select is available for use in a particular address range with a particular priority, but note that the actual external address ranges available on a target system depend on its design. For details about chip select priorities and address spaces, refer to the individual product specification for your ZNEO CPU-based device.

To use ZDS II, a detailed understanding of chip selects is not required. It is only necessary to enable and configure the chip selects used by the target system, and to add the actual external address ranges, as implemented on the target, to the Address Spaces page of the **Project Settings** dialog box (see page 68).

Programmer’s Model of ZNEO Memory

Different address ranges in the 24-bit ZNEO CPU memory space are suited for different functions, depending on whether the corresponding memory is volatile or nonvolatile, whether it can be addressed using 16 or 32 bits, and whether it is reserved or otherwise convenient for I/O. The following considerations affect the suitability of an address range for various memory functions:

- Volatile (random access) memory contents can be changed easily, so it is used for storing variable data, and can also contain program code downloaded temporarily or copied from nonvolatile memory.
- Nonvolatile memory is not easily changed, but is also unaffected by power loss, so it is used for storing constants, variable initializers, program code, option bits, and vectors.

- Data in 16-bit addressable memory can be addressed with short pointers and instructions, so using these spaces for data results in more compact code. However, 16-bit addresses can access only two 32KB areas – one in low memory and one in high memory – so data-intensive applications might also need to use some 32-bit addressed memory for data.
- Program code is always fetched using the 32-bit program counter; therefore there are no addressing restrictions for program code. Placing as little program code as possible into the 16-bit addressable (for data) space leaves more memory available in that space for data.
- The ZNEO microcontroller I/O and special function registers are located in the highest 8KB of its 24-bit address space, and extra chip selects are available for external I/O in the space immediately below internal I/O. Locating I/O functions in the high-memory 16-bit addressable space allows efficient access to I/O devices.

ZDS II uses five configurable memory ranges to associate a functional purpose to each part of the target system's physical memory map. You can configure the address ranges for each function in the Linker page of the **Project Settings** dialog box (see the Project Settings—Address Spaces Page section on page 68). Each address range has a corresponding mnemonic that is used with the assembly language SPACE keyword. The five address ranges and their SPACE mnemonics are:

Constant data (ROM). This range is typically `00_0000H-00_1FFFH` for devices with 32 KB of internal Flash, `00_0000H-00_3FFFH` for devices with 64KB of internal Flash, and `00_0000H-00_7FFFH` for devices with 128 KB of internal Flash. The lower boundary must be `00_0000H`. The upper boundary can be lower than `00_7FFFH`, but no higher. Data in this range is addressable using 16 or 32 bits.

The ROM address range includes the ZNEO CPU option bytes and vector table. The C-Compiler uses the ROM range for constant data, data tables, and start-up code. The assembly language programmer can place any executable code in the ROM range if desired.

The ROM range typically includes only internal Flash, but can include external nonvolatile memory if, for example, internal Flash is disabled.

Note: To use any external memory provided on the target system, you must configure the memory's chip select in the Configure Target dialog box. See the Project Settings—Debugger Page section on page 74.

Program space (EROM). Identifies any 32-bit addressed nonvolatile memory space outside the ROM range. This range is typically `00_2000H-00_7FFFH` for devices with 32 KB of internal Flash, `00_4000H-00_FFFFH` for devices with 64 KB of internal Flash, and `00_8000H-01_FFFFH` for devices with 128 KB of internal Flash. Specify a larger range only if the target system provides external nonvolatile memory.

The EROM range extends to the highest nonvolatile memory address in the target system. ZDS II requires the highest EROM address to fall below the specified ERAM (if present) and RAM ranges.

Normally, the EROM range begins immediately above the ROM range, but this is not required. The EROM range can include both internal Flash and external nonvolatile memory, if present. The EROM range is the primary location for storing executable code in most applications.

Extended RAM (ERAM). The ERAM address range identifies any 32-bit addressed random access memory on the target. In current ZNEO CPU-based devices, ERAM is always external memory. The ERAM range must begin above the highest EROM address. Also, the ZDS II GUI does not allow an ERAM starting address below `80_0000H`. An address gap is allowed between the EROM and ERAM ranges. The C-Compiler does not support gaps (holes) within the ERAM range.

The highest ERAM address must fall below the specified RAM address range. (Any external volatile memory that is present at or above FF_8000H is 16-bit addressable, so it should be assigned to the RAM range.) The ERAM address range can be used for data, stack, or executable code. For details, see the Program Configurations section on page 308.

Internal RAM (RAM). Typically `FF_B700H-FF_BFFFH` for 2KB internal RAM or `FF_B000H-FF_BFFFH` for 4KB internal RAM. Despite its name, this range can be expanded up to `FF_8000H-FF_BFFFH` if the target system provides external random access memory to fill out this address range. This GUI field does not allow a high RAM address boundary above `FF_BFFFH`.

The RAM address range is addressable using either 16 or 32 bits (the ZNEO CPU sign-extends 16-bit addresses). The C-Compiler does not support gaps (holes) within the RAM range.

ZDS II uses the RAM address range for nonpermanent storage of data during program execution. ZDS II can be configured to place code in the RAM address range, if desired. For more information, see the Program Configurations section on page 308.

Special Function Registers and IO (IODATA). Typically `FF_C000H-FF_FFFFH`. The microcontroller reserves addresses `FF_E000H` and above for its special function registers, on-chip peripherals, and I/O ports. The ZDS II GUI expects addresses `FF_C000H` to `FF_DFFFH` to be used for external I/O (if any) on the target system.

The IODATA address range is addressable using 16 or 32 bits (the ZNEO CPU sign-extends 16-bit addresses). ZDS II does not support placing executable code in the IODATA space.

Figure 131 on page 306 illustrates typical contents of the five ZDS II address ranges and an example of how they might map to a target's physical memory.

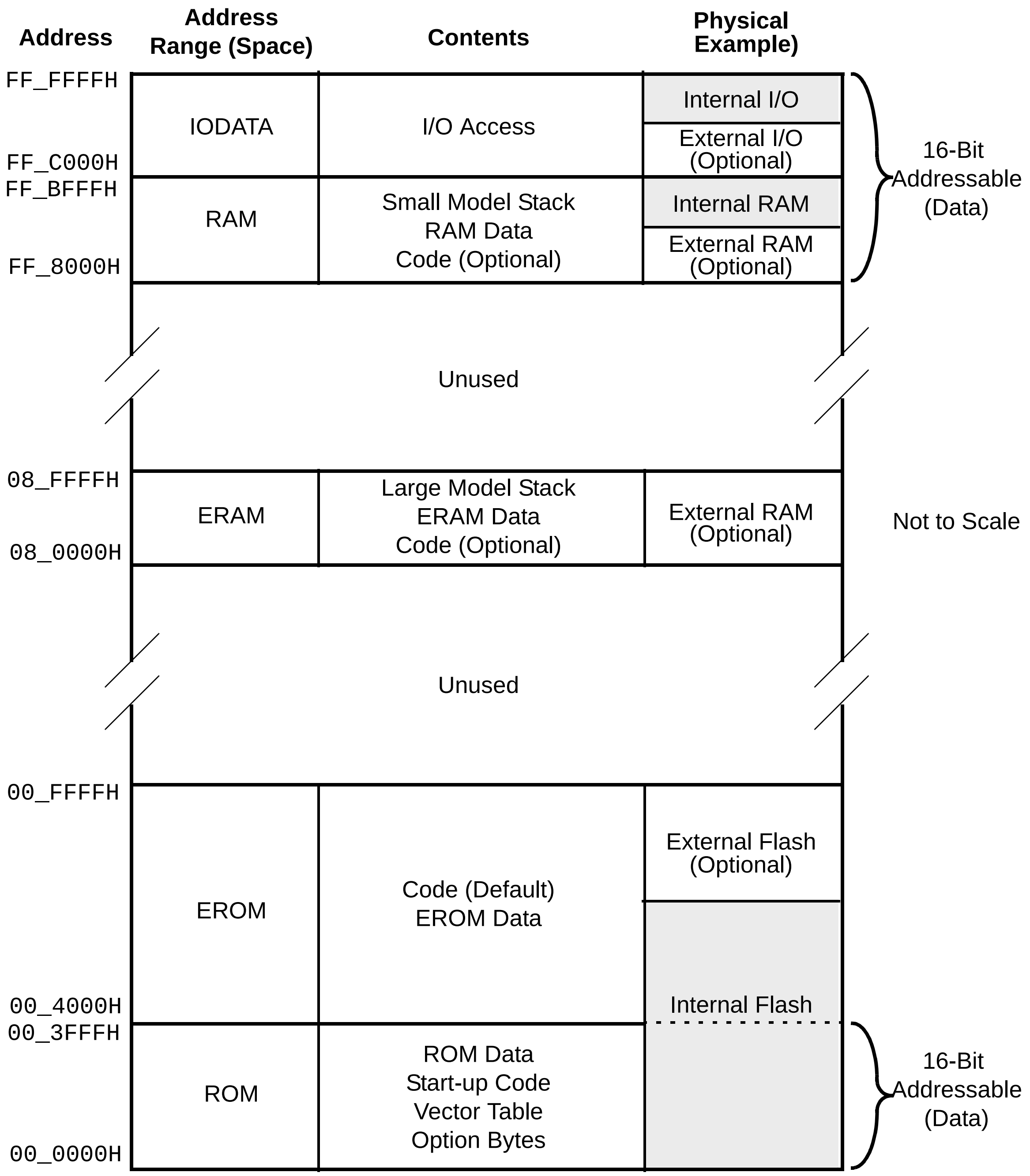


Figure 131. Typical ZNEO Programmer's Model—General

The following default segments are provided by the assembler and associated with specific address spaces as shown in the following table:

Address Range (Space)	Segment
ROM	ROM_DATA, ROM_TEXT, __VECTORS
EROM	CODE, EROM_DATA, EROM_TEXT
RAM	NEAR_DATA, NEAR_BSS, NEAR_TEXT
ERAM	FAR_DATA, FAR_BSS, FAR_TEXT
IODATA	IOSEG

For a detailed description of these segments, see the Predefined Segments section on page 213.

You can define your own segments in these address spaces using the DEFINE assembler directive. For example:

```
DEFINE romseg, SPACE=ROM  ; Defines the new segment romseg
SEGMENT romseg            ; Sets the current segment to romseg
```

See the User-Defined Segments section on page 213 for further details.

Unconventional Memory Layouts

It might be necessary for a target design to locate memory or I/O in an address range that the ZDS II GUI does not directly accommodate. Some GUI address range limitations can be circumvented, as follows:

- The GUI rejects an ERAM range lower boundary setting below `80_0000H`. If the target locates external volatile memory below this address, an unrestricted ERAM range can be configured by adding an edited linker RANGE command to the **Additional Linker Commands** field of the ZDS II project settings. For example, if the target provides 8MB of extended RAM at address `70_0000H`, you would use this command:

  ```
  RANGE ERAM $700000 : $FEFFFF
  ```

- The GUI rejects a RAM range upper boundary setting above `FF_BFFFH`. If the target uses the space above this address for external volatile memory instead of external I/O, a higher RAM upper boundary (up to `FF_DFFFH`) can be configured by adding an edited linker RANGE command to the **Additional Linker Commands** field of the ZDS II project settings. The debugger memory window always displays addresses above `FF_BFFFH` as part of the I/O Data space, however. For example, if the target provides 24KB of combined internal and external RAM beginning at `FF_8000H`, you would use this command:

  ```
  RANGE RAM $FF8000 : $FFDFFF
  ```

- The ZDS II GUI assumes external I/O is located in the range `FF_C000H` to `FF_DFFFH`. Any external I/O that is located elsewhere can be accessed using absolute addresses. The debugger memory window displays all addresses below `FF_C000H` as part of the Memory space, however.

Program Configurations

With the information given so far in this chapter as background, you are now ready to plan the memory configuration for your own application. At this point, you should have determined what external memory, if any, is present on your target system, enabled and configured its chip selects, and added its address space to the linker address ranges. For information about these settings, see the Project Settings—Debugger Page section on page 74 and the Project Settings—Address Spaces Page section on page 68.

Now it is necessary to implement your program's memory configuration using the development environment and tools. This section presents several examples of this process, beginning with the default program configuration provided by the Zilog development tools. For each configuration, there is a discussion of reasons to choose the configuration and how to implement the configuration in both C and assembly language projects.

Two distinct features of a program configuration are how its code is downloaded and how its initial values are copied. Downloading refers to what the development tools do when they copy your compiled or assembled code and data from your host computer to the ZNEO CPU's internal or external memory in preparation for a debugging or test session. The memory space (ZDS II address range) or spaces into which your program is downloaded depend on the memory model that best fits your application. You might also choose to change those spaces as your application evolves from the early stages of development closer to production.

The copying of initial values, on the other hand, is carried out by the start-up code in your application. Usually, the start-up code copies values from nonvolatile memory to a location in volatile memory where they are accessed in the main body of your program. In addition to copying data that must have a specific value on program startup, the start-up code can also set otherwise uninitialized values to zero (to conform with the C standard). Unlike the downloading step, the copying step continues to occur in your production code, typically when the end-user device is powered up or reset.

Default Program Configuration

The default program configuration provided by ZNEO development tools can be used for production code as well as development. It is designed to provide a general case that meets the needs of many users. In this configuration, your compiled or assembled program is all downloaded to the ROM or EROM functional address spaces; see Figure 132.

The start-up code (which is provided in the form of both source code and the standard, precompiled C start-up module) is downloaded to ROM, and the rest of the executable program code is downloaded to EROM. The ROM/EROM data (declared using the `_Rom` or `_Erom` keywords in a C application or appropriate SEGMENT directives in an assembly application) are downloaded into their respective address spaces. The initialized values of all initialized RAM/ERAM data are also downloaded into EROM. The Flash option bytes and the vector table are downloaded into ROM at addresses appropriate for the specified device. The start-up module code is executed from ROM, while the rest of the executable program code is executed from EROM. The start-up code sets up the external interface if required, based on the target setup parameters (see Setup – see page 76), and copies the initial values of RAM/ERAM data from EROM to their respective run-time addresses in RAM/ERAM.

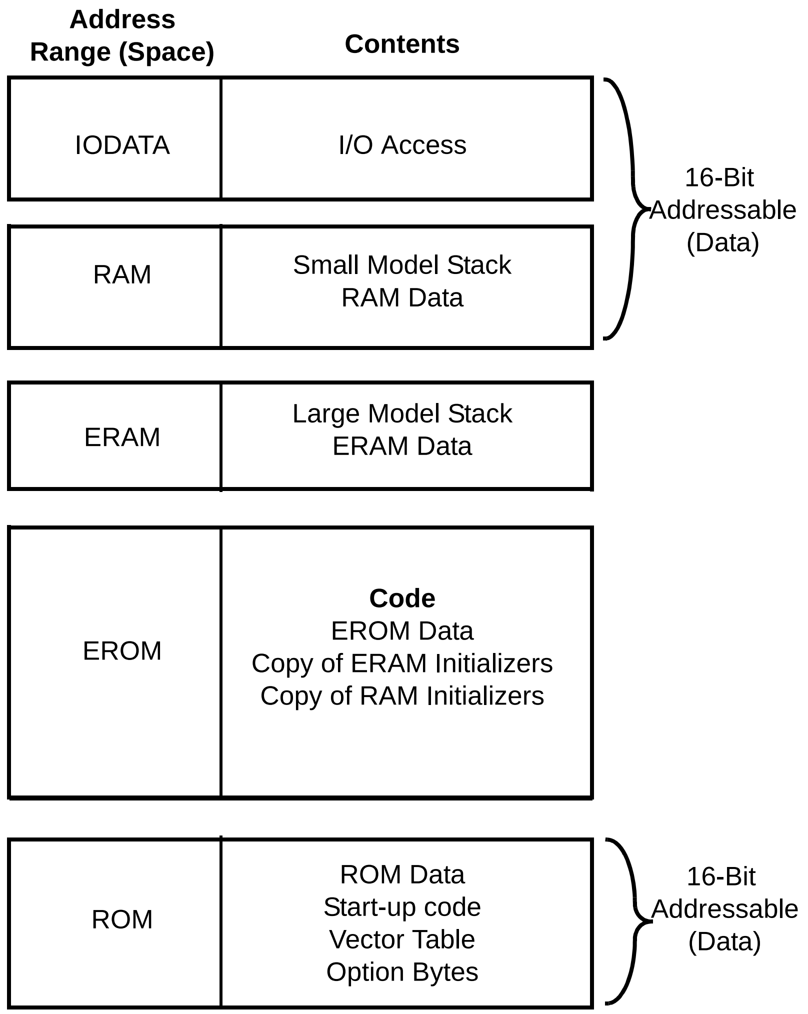


Figure 132. Programmer's Model—Default Program Configuration

Note: Figure 132 and the other program configuration figures omit physical address and interface information to emphasize that, once the address ranges have been defined in the project setup, most memory locations can be expressed symbolically in terms of the functional space (address range) in which they are located.

C Program Setup

This is the default setup provided by the development tools. You do not have to perform any additional steps to achieve this configuration. A point to note is that if the C program is a large model program, the data and stack reside in ERAM by default, so selecting the large model is not appropriate unless the target has external random access memory that you have configured in the target setup and added to the ERAM linker address range.

Assembly Program Setup

As an assembly user, you can choose either to use the default segments or to define and use your own segments in various address spaces. To avoid problems, you must observe the following guidelines to achieve the same configuration as that used in the default C program configuration:

- Locate the reset vector routine in the internal memory portion of ROM. For example:

```
vector reset = _startup

define ROMSEG, space=ROM, org=%70
           segment ROMSEG
_startup:             ; startup code here
; any external memory interface initialization should be done here
; any RAM/ERAM data copying should be done here
```

- Locate the rest of the executable program in EROM. For example:

```
segment CODE
; program code goes here
```

- Locate the initialized constant data in ROM/EROM.
- Only allocate space for data in RAM/ERAM using the DS assembler directive and make sure to not perform any initializations using the DB/DW/DL directive for such data. Any initializations of RAM/ERAM data should be done in your code. For example:

```
  segment NEAR_BSS
val:  DS 4                    ; allocate 4 uninitialized bytes: OK
  segment code
ld r0, #%12345678
ld val, r0                    ; initialize ram location val as part of code: OK
```

```
  segment NEAR_BSS
val: dl  %12345678          ; incorrect usage for production code, as RAM does
                            ; not retain value across power cycles: NOT OK
```

If your application has a lot of such initializations, this approach can be tedious. In that case, an alternative is to use the linker COPY directive and perform the physical copy for all of the RAM data as a single loop in the startup. An example can be found in the C startup provided in the installation under `src\boot\common\startups.asm`:

```
;
; Copy ROM data into internal RAM
;

  LEA     R0,_low_near_romdata
  LEA     R1,_low_neardata
  LD      R2,#_len_neardata+1
  JP      lab10
lab9:
  LD.B    R3,(R0++)
  LD.B    (R1++),R3
lab10:
  DJNZ    R2,lab9
```

The installation also includes the following linker directives:

```
COPY NEAR_DATA EROM
define _low_near_romdata = copy base of NEAR_DATA
define _low_neardata = base of NEAR_DATA
define _len_neardata = length of NEAR_DATA
```

In the above linker directives, `NEAR_DATA` is the RAM segment containing initialized data:

```
segment NEAR_DATA
val1: dl %12345678
val2: dl %23456789
val3: dl %3456789A
…
```

The linker `COPY` directive only designates the load addresses and the run-time addresses for the segments. The actual copying of the data must be performed by the user code as shown above.

- Assembly users can perform their own external interface setup in the start-up code. This can be made easier if you take advantage of some definitions provided by the Configure Target dialog box (see the Setup section on page 76). An example can be found in the C startup provided in the installation under `src\boot\common\startupexs.asm`. The ZDS II IDE generates the following linker directives based on your settings in the Target Configure dialog box:

```
DEFINE __EXTCT_INIT_PARAM = $c0
```

```
 DEFINE __EXTCS0_INIT_PARAM = $8012
 DEFINE __EXTCS1_INIT_PARAM = $8001
 DEFINE __EXTCS2_INIT_PARAM = $0000
 DEFINE __EXTCS3_INIT_PARAM = $0000
 DEFINE __EXTCS4_INIT_PARAM = $0000
 DEFINE __EXTCS5_INIT_PARAM = $0000
 DEFINE __PFAF_INIT_PARAM = $ff
 DEFINE __PGAF_INIT_PARAM = $ff
 DEFINE __PDAF_INIT_PARAM = $ff00
 DEFINE __PAAF_INIT_PARAM = $0000
 DEFINE __PCAF_INIT_PARAM = $0000
 DEFINE __PHAF_INIT_PARAM = $0300
 DEFINE __PKAF_INIT_PARAM = $0f
```

You can initialize the external interface registers based on these defines as part of your startup. For example:

```
LDR0,#__EXTCT_INIT_PARAM
LD.BEXTCT,R0; Setup EXTCT

LDR0,#EXTCS0; Setup EXTCS0-EXTCS5
LD      (R0++),#((__EXTCS0_INIT_PARAM <<16)|__EXTCS1_INIT_PARAM)
LD      (R0++),#((__EXTCS2_INIT_PARAM <<16)|__EXTCS3_INIT_PARAM)
LD      (R0++),#((__EXTCS4_INIT_PARAM <<16)|__EXTCS5_INIT_PARAM)
                          ; Setup Port Alternate functions.
LDR0,#__PAAF_INIT_PARAM
LD.WPAAF,R0
LDR0,#__PCAF_INIT_PARAM
LD.WPCAF,R0
LDR0,#__PDAF_INIT_PARAM
LD.WPDAF,R0
LDR0,#__PFAF_INIT_PARAM
LD.BPFAFL,R0
LDR0,#__PGAF_INIT_PARAM
LD.BPGAFL,R0
LDR0,#__PHAF_INIT_PARAM
LD.WPHAF,R0
LDR0,#__PKAF_INIT_PARAM
LD.BPKAFL,R0
```

Following the above guidelines, an assembly user can achieve the Default Program Configuration for production code.

Download to ERAM Program Configuration

The Download to ERAM Program Configuration, shown in Figure 133, can be used only during development and not for production code. It is similar to the Default Program Configuration; the only difference is that aside from the start-up code, the rest of the executable program is downloaded into ERAM and executed from ERAM.

Because ERAM cannot retain values across power cycles, this configuration is meant for development only. By using this configuration, you can avoid erasing and burning the executable code in Flash multiple times as part of the development cycle.

This program configuration requires that the target system contain external RAM that has been configured in the target setup and added to the ERAM linker address range.

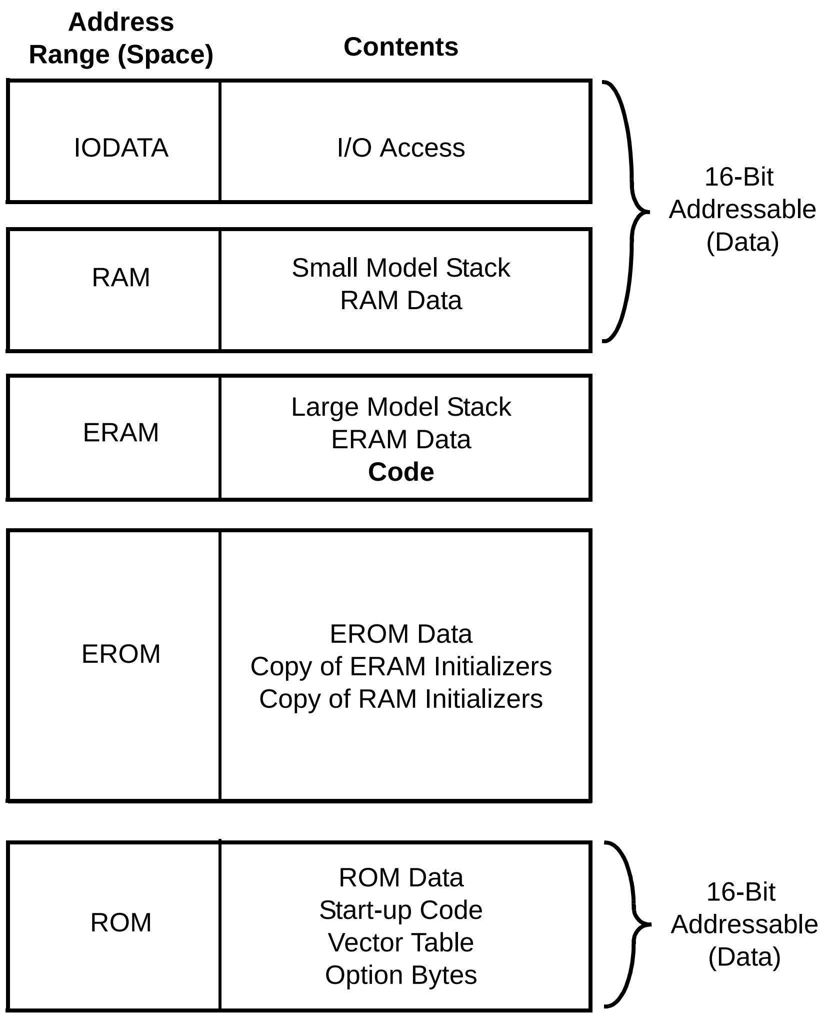


Figure 133. Programmer's Model—Download to ERAM Program Configuration

C Program Setup

The C program setup for the Download to ERAM Program Configuration is similar to the Default Program Configuration with some additional steps as described in this section.

The C-Compiler by default generates the executable program code under one unified segment named CODE. The CODE segment belongs to EROM address space. To set up this configuration, you must move this CODE segment from EROM to ERAM address space. Do this by adding the following linker command in the **Additional Linker Directives** dialog box (see the Additional Directives section on page 62):

```
change code=ERAM /* The linker will then allocate code segment in
ERAM */
```

To go back to the Default Program Configuration for production code, remove this directive from the **Additional Linker Directives** dialog box.

Special Case: Partial Download to ERAM

A special case of this configuration is to download program code from just one C source file into ERAM, while retaining the rest of the code in EROM. This example is included for completeness because similar cases of partitioning the code are discussed for the other configurations that are covered in this chapter.

Perform the following steps:

1. Select the **Distinct Code Segment for Each Module** checkbox in the **Advanced** page in the **Project Settings** dialog box (discussed on page 59). This option directs the C-Compiler to generate different code segment names for each file.
2. Use the linker CHANGE directive to move the particular segment to ERAM.

   For example, to download the code for `main.c` to ERAM, add the following linker command in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

   ```
   change main_TEXT = ERAM
   ```

   To go back to the Default Program Configuration for production code, remove this directive from the **Additional Linker Directives** dialog box.

Assembly Program Setup

The Assembly program setup for the Download to ERAM Program Configuration is similar to the Default Program Configuration with some additional guidelines, as described in this section.

Write all of the executable program (non-start-up) code under the CODE segment. This segment belongs to EROM address space. To set up this configuration, you must move the CODE segment from EROM to the ERAM address space by adding the following linker command in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

```
change code=ERAM /* The linker will then allocate code segment in
ERAM */
```

To go back to the Default Program Configuration for production code, remove this directive from the **Additional Linker Directives** dialog box.

Special Case: Partial Download to ERAM

A special case of this configuration is to download program code from just one assembly segment into ERAM, while retaining the rest of the code in EROM.

Perform the following steps:

1. Use a distinct segment name for the particular segment.

   For example:

   ```
   Define main_TEXT, space=EROM
   segment main_TEXT
   ; code goes here
   ```

2. Use the linker CHANGE directive to move the particular segment to ERAM. For example:

   To download the code for the `main_TEXT` segment to ERAM, add the following linker command in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

   ```
   change main_TEXT = ERAM
   ```

   To go back to the Default Program Configuration for production code, remove this directive from the **Additional Linker Directives** dialog box.

Download to RAM Program Configuration

The Download to RAM Program Configuration, shown in Figure 134, can be used only during development and not for production code. It is similar to the Default Program Configuration, the only difference is that the rest of the executable program code is downloaded in RAM and executed from RAM.

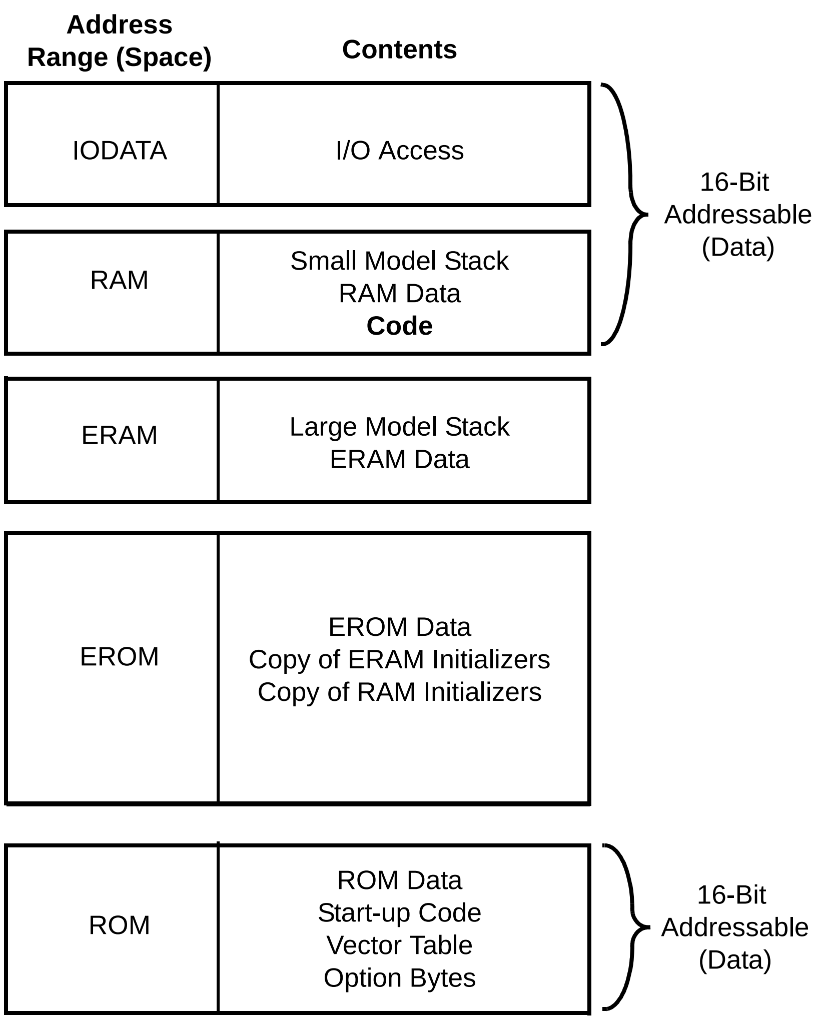


Figure 134. Programmer's Model—Download to RAM Program Configuration

Because RAM cannot retain values across power cycles, this configuration is meant for development only. By using this configuration, you can avoid erasing and burning the executable code in Flash multiple times as part of development cycle. In this respect, this configuration obviously is very similar to the Download to ERAM Program Configuration discussed in Download to ERAM Program Configuration – see page 312. The main difference is that RAM is typically internal memory, while ERAM is external. Therefore, to use the Download to ERAM Program Configuration, you must provide external memory and configure its interface. By using the Download to RAM Program Configuration, you avoid that task. On the other hand, the amount of internal memory is rather limited on many

ZNEO devices, so the Download to RAM Program Configuration could be limited to small applications or portions of applications during development.

C Program Setup

The C program setup for Download to RAM Program Configuration is similar to the Default Program Configuration with some additional steps, as described in this section.

The C-Compiler by default generates the executable program code under one unified segment named CODE. The CODE segment belongs to EROM address space. To set up this configuration, you must move the CODE segment from EROM to the RAM address space. Do this by adding the following linker command in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

```
change code=RAM /* The linker will then allocate code segment in
RAM */
```

To go back to the Default Program Configuration for production code, remove this directive from the **Additional Linker Directives** dialog box.

Special Case: Partial Download to RAM

A special case of this configuration is when you want to download program code from just one C source file into RAM, while retaining the rest of the code in EROM. This allows you to experiment with different partitions of your code, for example, if you are considering the Partial Copy to RAM Program Configuration discussed in the Special Case: Partial Copy to RAM section on page 325. Perform the following steps:

1. Select the **Distinct Code Segment for Each Module** checkbox in the **Advanced** page in the **Project Settings** dialog box (discussed on page 59).

   This option directs the C-Compiler to generate different code segment names for each file.

2. Use the linker CHANGE directive to move the particular segment to RAM. For example:

   To download the code for `main.c` to RAM, add the following linker command in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

   ```
    change main_TEXT = RAM
   ```

   To go back to the Default Program Configuration for production code, remove this directive from the **Additional Linker Directives** dialog box.

Assembly Program Setup

The Assembly program setup for the Download to RAM Program Configuration is similar to the Default Program Configuration with some additional guidelines, as described in this section.

Write all of the executable program code (non-start-up code) under the CODE segment. This segment belongs to EROM address space. To set up this configuration, you must move the CODE segment from EROM to the RAM address space. Do this by adding the following linker command in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

```
change code=RAM /* The linker will then allocate code segment in
RAM */
```

To go back to the Default Program Configuration for production code, remove this directive from the **Additional Linker Directives** dialog box.

Special Case: Partial Download to RAM

A special case of this configuration is when you want to download program code from just one assembly segment into RAM, while retaining the rest of the code in EROM. This allows you to experiment with different partitions of your code, for instance if you are considering the Partial Copy to RAM Program Configuration discussed in the Special Case: Partial Copy to RAM section on page 325. Perform the following steps:

1. Use a distinct segment name for the particular segment. For example:

   ```
   Define main_TEXT, space=EROM
   segment main_TEXT
   ; code goes here
   ```

2. Use the linker CHANGE directive to move the particular segment to RAM.

   For example, to download the code for the `main_TEXT` segment to RAM, add the following linker command in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

   ```
   change main_TEXT = RAM
   ```

   To go back to the Default Program Configuration for production code, remove this directive from the **Additional Linker Directives** dialog box.

Copy to ERAM Program Configuration

The Copy to ERAM Program Configuration, shown in Figure 135, can be used for production as well as development code. It is somewhat similar to the Default Program Configuration, the only difference being that aside from the start-up code, the rest of the executable program is downloaded into EROM, copied from EROM to ERAM by the start-up code, and then executed from ERAM. The reason for choosing this configuration is that while internal Flash on ZNEO-CPU-based devices can be accessed quickly enough to keep up with the CPU, that might not be true of external Flash. If you have both external Flash (as part of the EROM address range) and external RAM (as part of ERAM) in your application and if the external RAM is faster than the external Flash, the user program might execute faster on ERAM. Therefore, this configuration might be advantageous if you want to execute your program from external memory (for example, because your program is too large to fit into internal Flash), but you do not want your execution speed to be limited by the access speed of external Flash memory.

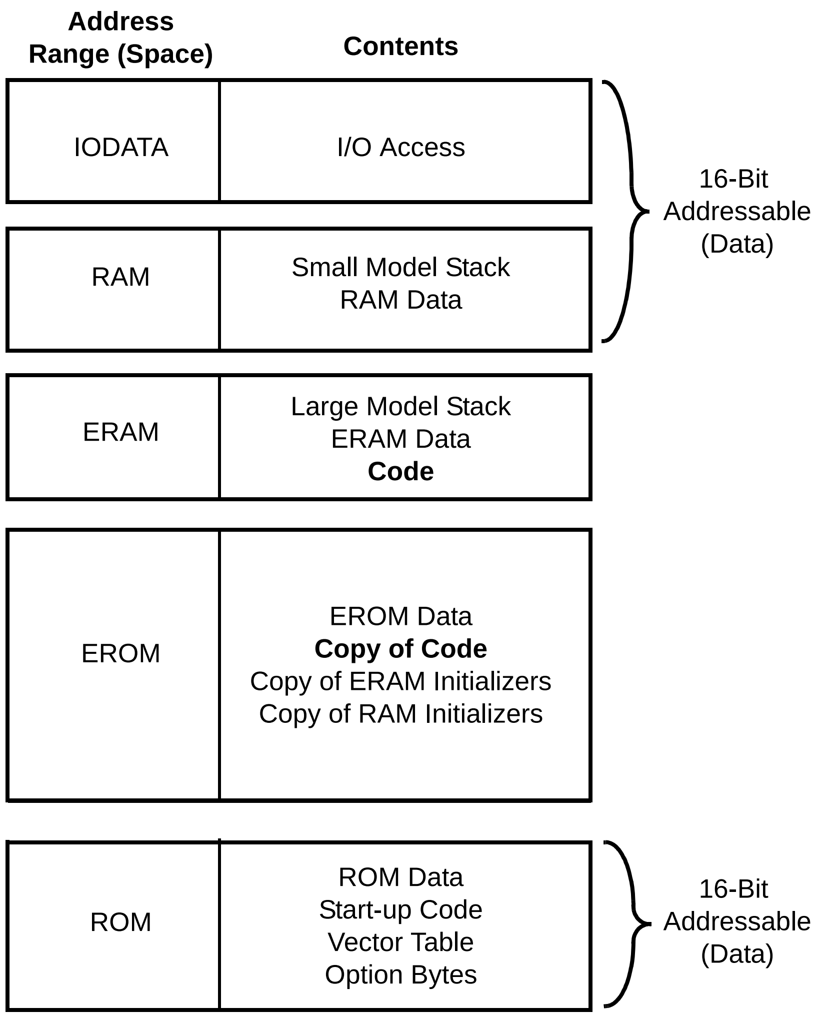


Figure 135. Programmer's Model—Copy to ERAM Program Configuration

C Program Setup

The C program setup for Copy to ERAM Program Configuration is similar to the Default Program Configuration with some additional steps as described in this section.

The C-Compiler by default generates the executable program code under one unified segment named CODE. The CODE segment belongs to EROM address space.

1. To set up this configuration, you must place the CODE segment in ERAM at run time and EROM at load time. Do this by adding the following linker commands in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

```
change code=ERAM          /* Run time CODE is in ERAM space */
copy code EROM             /* Load time CODE is in EROM space */

define _low_code_copy = copy base of CODE
define _low_code = base of CODE
define _len_code = length of CODE
```

2. The linker COPY directive only designates the load addresses and the run-time addresses for the segment. The actual copying of the segment must be performed by the start-up code. For example if you are using the small model, copy and modify the C startup provided in the installation under `src\boot\common\startupexs.asm`, rewriting the relevant section as follows:

```
;
; Copy CODE into ERAM
;

  LEA     R0,_low_code_copy
  LEA     R1,_low_code
  LD      R2,#_len_code+1
  JP      clab1
clab0:
  LD.B    R3,(R0++)
  LD.B    (R1++),R3
clab1:
  DJNZ    R2,clab0

  XREF    _low_code_copy:EROM
  XREF    _len_code:ERAM
  XREF    _low_code:ERAM
```

3. Finally, add your modified start-up module to your project. Select the **Included in Project** button in the **Objects and Libraries** page of the **Project Settings** dialog box (see C Start-up Module – see page 66) and also add the modified source code file to your project using the Add Files command from the **Project** menu.

Special Case: Partial Copy to ERAM

A special case of this configuration is when you want to copy program code from just one C source file into ERAM, while retaining the rest of the code in EROM. This allows you to gain the potential speed advantages of external RAM over external Flash for a specific module. Perform the following steps:

1. Select the **Distinct Code Segment for Each Module** checkbox in the **Advanced** page in the **Project Settings** dialog box (see page 59).

This directs the C-Compiler to generate different code segment names for each file.

2. Add the linker commands to place the particular segment in ERAM at run time and EROM at load time.

   For example, to copy the code for `main.c` to ERAM, add the following linker commands in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

```
change main_TEXT=ERAM /* Run time main_TEXT is in ERAM space */
copy main_TEXT EROM  /* Load time main_TEXT is in EROM space */

define _low_main_copy = copy base of main_TEXT
define _low_main = base of main_TEXT
define _len_main = length of main_TEXT
```

3. The linker COPY directive only designates the load addresses and the run-time addresses for the segment. The actual copying of the segment must be performed by the start-up code. For example, if you are using the small model, copy and modify the C startup provided in the installation under `src\boot\common\startupexs.asm`, rewriting the relevant section as follows:

```
;
; Copy main_TEXT into ERAM
;

  LEA     R0,_low_main_copy
  LEA     R1,_low_main
  LD      R2,#_len_main+1
  JP      mlab1
mlab0:
  LD.B    R3,(R0++)
  LD.B    (R1++),R3
mlab1:
  DJNZ    R2,mlab0

  XREF    _low_main_copy:EROM
  XREF    _len_main:ERAM
  XREF    _low_main:ERAM
```

4. Finally, add your modified start-up module to your project by selecting the **Included in Project** button in the **Objects and Libraries** page of the **Project Settings** dialog box (see C Start-up Module – see page 66) and also add the modified source code file to your project using the Add Files command from the **Project** menu.

Assembly Program Setup

The Assembly program setup for the Copy to ERAM Program Configuration is similar to the Default Program Configuration with some additional guidelines, as described in this section.

Write all of the executable program code (non-start-up code) under the CODE segment. This segment belongs to EROM address space. To set up this configuration, you must copy the CODE segment from EROM to the ERAM address space. Perform steps 2 and 3 from the C Program Setup for this configuration, as described in the C Program Setup section on page 319. In step 3, add the code to your assembly start-up code instead of the standard C startup.

Special Case: Partial Copy to ERAM

A special case of this configuration is when you want to copy program code from just one assembly segment into ERAM, while retaining the rest of the code in EROM. This allows you to gain the potential speed advantages of external RAM over external Flash for a specific module. Perform the following steps:

1. Use a distinct segment name for the particular segment. For example:

   ```
   Define main_TEXT, space=EROM
   segment main_TEXT
   ; code goes here
   ```

2. To copy the `main_TEXT` segment from EROM to the ERAM address space, perform steps 2 and 3 from the C Program Setup, Special Case, for this configuration as described in the C Program Setup section on page 319. In step 3, add the code to your assembly start-up code instead of the standard C startup.

Copy to RAM Program Configuration

The Copy to RAM Program Configuration, shown in Figure 136, can be used for production as well as development code. It is somewhat similar to the Default Program Configuration, the only difference being that the executable program code is downloaded in EROM, copied from EROM to RAM by the start-up code, and then executed from RAM. The reason for choosing this configuration is that while internal Flash on ZNEO-CPU-based devices can be accessed quickly enough to keep up with the CPU, that might not be true of external Flash. If you have external Flash (as part of the EROM address space) and if the internal RAM is faster than the external Flash, the user program might execute faster from RAM.

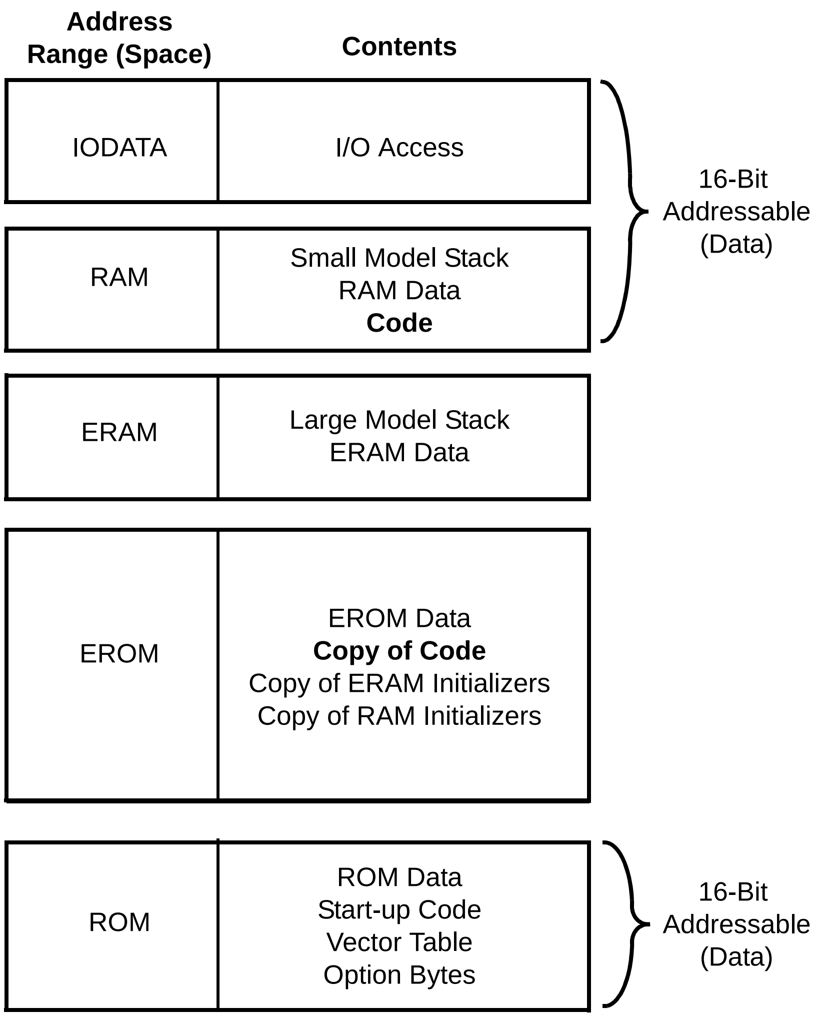


Figure 136. Programmer's Model—Copy to RAM Program Configuration

In this respect, this configuration obviously is very similar to the Copy to ERAM Program Configuration discussed in Copy to ERAM Program Configuration – see page 318. The main difference is that RAM is typically internal memory, while ERAM is external. Therefore, to use the Copy to ERAM Program Configuration, you must provide external memory and configure its interface. By using the Copy to RAM Program Configuration, you avoid that work. On the other hand, the amount of internal memory is rather limited on many ZNEO devices, so the Copy to RAM Program Configuration might tend to be limited to small applications or portions of applications during development.

C Program Setup

The C program setup for Copy to RAM Program Configuration is similar to the Default Program Configuration with some additional steps as described in this section.

The C-Compiler by default generates the executable program code under one unified segment named CODE. The CODE segment belongs to the EROM address space.

1. To set up this configuration, you must place the CODE segment in RAM at run time and EROM at load time. Do this by adding the following linker commands in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

```
change code=RAM            /* Run time CODE is in RAM space */
copy code EROM            /* Load time CODE is in EROM space */

define _low_code_copy = copy base of CODE
define _low_code = base of CODE
define _len_code = length of CODE
```

2. The linker COPY directive only designates the load addresses and the run-time addresses for the segment. The actual copying of the segment must be performed by the start-up code. For example, if you are using the small model, copy and modify the C startup provided in the installation under `src\boot\common\startupexs.asm`, rewriting the relevant section as follows:

```
;
; Copy CODE into RAM
;

  LEA     R0,_low_code_copy
  LEA     R1,_low_code
  LD      R2,#_len_code+1
  JP      clab1
clab0:
  LD.B    R3,(R0++)
  LD.B    (R1++),R3
clab1:
  DJNZ    R2,clab0

  XREF    _low_code_copy:EROM
  XREF    _len_code
  XREF    _low_code:RAM
```

3. Finally, add your modified start-up module to your project by selecting the **Included in Project** button in the **Objects and Libraries** page of the **Project Settings** dialog box (see C Start-up Module – see page 66) and also add the modified source code file to your project using the Add Files command from the **Project** menu.

Special Case: Partial Copy to RAM

A special case of this configuration is when you want to copy program code from just one C source file into RAM, while retaining the rest of the code in EROM. This allows you to investigate whether there might be power or speed savings to be realized by partitioning your application in this way. Perform the following steps:

1. Select the **Distinct Code Segment for Each Module** checkbox in the **Advanced** page in the **Project Settings** dialog box (see page 59) to direct the C-Compiler to generate different code segment names for each file.
2. Add the linker commands to place the particular segment in RAM at run time and EROM at load time.

   For example, to copy the code for `main.c` to RAM, add the following linker commands in the **Additional Linker Directives** dialog box (see Additional Directives – see page 62):

```
change main_TEXT=RAM /* Run time main_TEXT is in RAM space */
copy main_TEXT EROM  /* Load time main_TEXT is in EROM space */

define _low_main_copy = copy base of main_TEXT
define _low_main = base of main_TEXT
define _len_main = length of main_TEXT
```

3. The linker COPY directive only designates the load addresses and the run-time addresses for the segment. The actual copying of the segment must be performed by the start-up code. For example, if you are using the small model, copy and modify the C startup provided in the installation under `src\boot\common\startupexs.asm`, rewriting the relevant section as follows:

```
;
; Copy main_TEXT into ERAM
;

  LEA     R0,_low_main_copy
  LEA     R1,_low_main
  LD      R2,#_len_main+1
  JP      mlab1
mlab0:
  LD.B    R3,(R0++)
  LD.B    (R1++),R3
mlab1:
  DJNZ    R2,mlab0

  XREF    _low_main_copy:EROM
  XREF    _len_main
  XREF    _low_main:RAM
```

4. Finally, add your modified start-up module to your project by selecting the **Included in Project** button in the **Objects and Libraries** page of the **Project Settings** dialog box (see C Start-up Module – see page 66) and also add the modified source code file to your project using the Add Files command from the **Project** menu.

Assembly Program Setup

The Assembly program setup for the Copy to RAM Program Configuration is similar to the Default Program Configuration with some additional guidelines, as described in this section.

Write all of the executable program code (non-start-up code) under the CODE segment. This segment belongs to EROM address space. To set up this configuration, you must copy the CODE segment from EROM to RAM address space. Perform steps 2 and 3 from the C Program Setup for this configuration, as described in the C Program Setup section on page 324. In step 3, add the code to your assembly start-up code instead of the standard C startup.

Special Case: Partial Copy to RAM

A special case of this configuration is when you want to copy program code from just one assembly segment into RAM, while retaining the rest of the code in EROM. This allows you to investigate whether there might be power or speed savings to be realized by partitioning your application in this way. Perform the following steps:

1. Use a distinct segment name for the particular segment. For example:

```
Define main_TEXT, space=EROM
segment main_TEXT
; code goes here
```

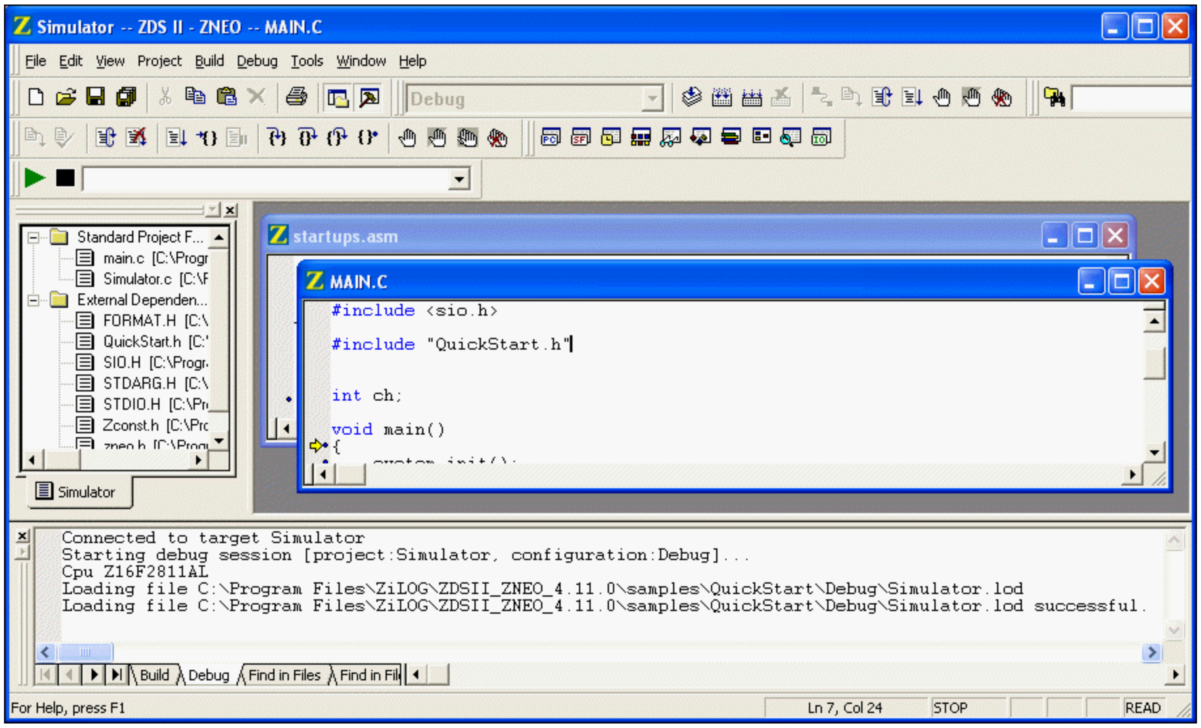
2. To copy the `main_TEXT` segment from EROM to the RAM address space, perform steps 2 and 3 from the C Program Setup, Special Case, for this configuration as described in the C Program Setup section on page 324. In step 3, add the code to your assembly start-up code instead of the standard C startup.

Chapter 8. Using the Debugger

The source-level debugger is a program that allows you to find problems in your code at the C or assembly level. You can also write batch files to automate debugger tasks (for details about the debugger, see the appendix titled Using the Command Processor, on page 359). The following topics are covered in this section:

- Status Bar – see page 328
- Code Line Indicators – see page 328
- Debug Windows – see page 329
- Using Breakpoints – see page 343

From the **Build** menu, select **Debug**, then select **Reset** to transition to Debug Mode. The Debug toolbar and Debug Windows toolbar are displayed, as shown in Figure 137. (For more information, see the Debug Output Window section on page 30.)



```
#include <sio.h>
#include "QuickStart.h"

int ch;

void main()
{
```

```
Connected to target Simulator
Starting debug session [project:Simulator, configuration:Debug]...
Cpu Z16F2811AL
Loading file C:\Program Files\ZiLOG\ZDSII_ZNEO_4.11.0\samples\QuickStart\Debug\Simulator.lod
Loading file C:\Program Files\ZiLOG\ZDSII_ZNEO_4.11.0\samples\QuickStart\Debug\Simulator.lod successful.
```

Figure 137. Debug and Debug Window Toolbars

The Debug toolbar is described in the Debug Toolbar section on page 20, and the Debug Windows toolbar is described in the Debug Windows Toolbar section on page 22.

Note: Project code cannot be rebuilt while in Debug Mode. The ZDS II will prompt you if you request a build during a debug session. If you edit code during a debug session and then attempt to execute the code, ZDS II will stop the current debug session, rebuild the project, and then attempt to start a new debug session if you elect to do so when prompted.

Status Bar

The status bar displays the current status of your program's execution. The status can be STOP, STEP or RUN. In STOP mode, your program is not executing. In STEP mode, a Step Into, Step Over, or Step Out command is in progress. In RUN mode, a Go command has been issued, your program is executing, and no breakpoint has yet been encountered.

Code Line Indicators

The Edit window displays your source code with line numbers and code line indicators. The debugger indicates the status of each line visually with the following code line indicators:

- A red octagon indicates an active breakpoint at the code line; a white octagon indicates a disabled breakpoint
- Blue dots are displayed to the left of all valid code lines; these are lines where breakpoints can be set, the program can be run to, and so on

Note: Some source lines do not display blue dots because the code has been optimized out of the executable (and the corresponding debug information).

- A program counter code line indicator (yellow arrow) indicates the code line at which the program counter is located
- A program counter code line indicator on a breakpoint (yellow arrow on a red octagon) indicates a code line indicator has stopped on a breakpoint

If the program counter steps into another file in your program, the Edit window switches to the new file automatically.

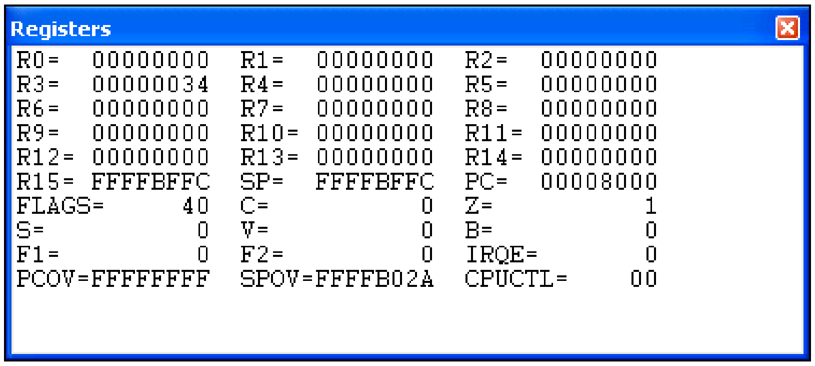
Debug Windows

The Debug Windows toolbar (described in the Debug Windows Toolbar section on page 22) allows you to display the following Debug windows:

- Registers Window – see page 329
- Special Function Registers Window – see page 330
- Clock Window – see page 331
- Memory Window – see page 331
- Watch Window – see page 337
- Locals Window – see page 339
- Call Stack Window – see page 340
- Symbols Window – see page 340
- Simulated UART Output Window – see page 342

Registers Window

Click the **Registers Window** button to show or hide the Registers window. The Registers window, shown in Figure 138, displays all of the registers in the standard ZNEO architecture.



```
R0=  00000000  R1=  00000000  R2=  00000000
R3=  00000034  R4=  00000000  R5=  00000000
R6=  00000000  R7=  00000000  R8=  00000000
R9=  00000000  R10= 00000000  R11= 00000000
R12= 00000000  R13= 00000000  R14= 00000000
R15= FFFFBFFC  SP=  FFFFBFFC  PC=  00008000
FLAGS=      40  C=          0  Z=          1
S=           0  V=          0  B=          0
F1=          0  F2=         0  IRQE=       0
PCOV=FFFFFFFF  SPOV=FFFFB02A  CPUCTL=     00
```

Figure 138. Registers Window

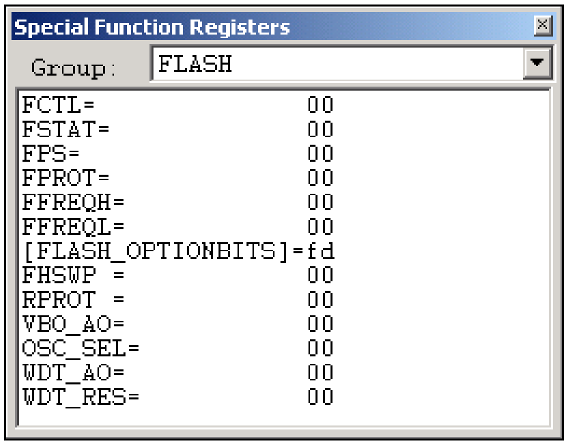
To change register values, perform the following brief procedure.

1. In the Registers window, highlight the value you want to change.
2. Enter the new value and press the **Enter** key. The changed value is displayed in red.

Special Function Registers Window

Click the Special Function Registers Window button to open up to ten Special Function Registers windows. Each Special Function Registers window, shown in Figure 139, displays the special function registers in the standard ZNEO architecture that belong to the selected group. Addresses `F00` through `FFF` are reserved for special function registers (SFRs).

Use the **Group** drop-down list to view a particular group of SFRs.



```
FCTL=                  00
FSTAT=                 00
FPS=                   00
FPROT=                 00
FFREQH=                00
FFREQL=                00
[FLASH_OPTIONBITS]=fd
FHSWP  =               00
RPROT  =               00
VBO_AO=                00
OSC_SEL=               00
WDT_AO=                00
WDT_RES=               00
```

Figure 139. Special Function Registers Window

Notes: There are several SFRs that, when read, are cleared or clear an associated register. To prevent the debugger from changing the behavior of the code, a special group of SFRs was created that groups these state changing registers. The group is called SPECIAL_CASE. If this group is selected, the behavior of the code changes, and the program must be reset.

To use the FLASH_OPTIONBITS group, you must reset the device for the changes to take effect. Use the FLASH_OPTIONBITS group to view the values of all of the Flash option bit registers, with the exception of the following registers:

- Temperature sensor trim register
- Precision oscillator trim register
- Flash capacity configuration register

To change special function register values, perform the following brief procedure.

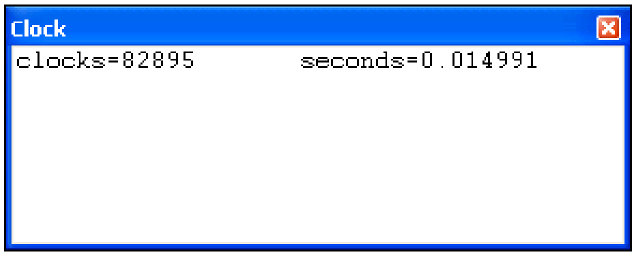
1. In the **Special Function Registers** window, highlight the value you want to change.
2. Enter the new value and press the **Enter** key. The changed value is displayed in red.

Clock Window

Click the **Clock Window** button to show or hide the Clock window.

The Clock window displays the number of states executed since the most recent reset. You can reset the contents of the Clock window at any time by selecting the number of cycles (Figure 140 displays 82895 cycles), entering a `0`, and pressing the **Enter** key. Updated values are displayed in red.

Note: The Clock window will only display clock data when the Simulator is the active debug tool.

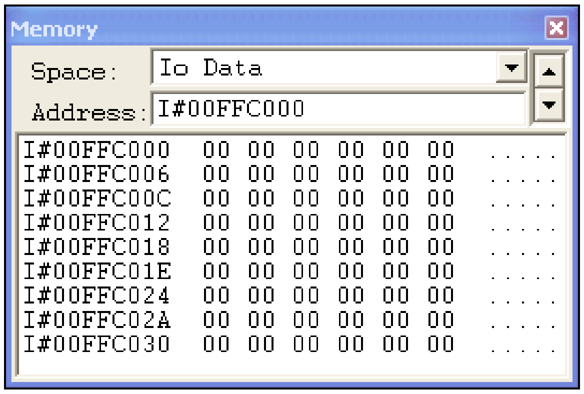


```
clocks=82895        seconds=0.014991
```

Figure 140. Clock Window

Memory Window

Click the **Memory Window** button to open up to ten Memory windows; see Figure 141.



```
I#00FFC000  00 00 00 00 00 00  .....
I#00FFC006  00 00 00 00 00 00  .....
I#00FFC00C  00 00 00 00 00 00  .....
I#00FFC012  00 00 00 00 00 00  .....
I#00FFC018  00 00 00 00 00 00  .....
I#00FFC01E  00 00 00 00 00 00  .....
I#00FFC024  00 00 00 00 00 00  .....
I#00FFC02A  00 00 00 00 00 00  .....
I#00FFC030  00 00 00 00 00 00  .....
```

Figure 141. Memory Window

Each Memory window displays data located in the target's memory. The ASCII text for memory values is shown in the last column. The address is displayed in the far left column with an I# to denote the IOData address space or with an M# to denote the Memory address space.

Note: Use the Memory window to perform the following brief procedure.

- Change Values – see page 332
- View the Values for Other Memory Spaces – see page 333
- View or Search for an Address – see page 333
- Fill Memory – see page 334
- Save Memory to a File – see page 335
- Load a File into Memory – see page 335
- Perform a Cyclic Redundancy Check – see page 336

Note: The Page Up and Page Down keys (on your keyboard) are not functional in the Memory window. Instead, use the up and down arrow buttons to the right of the Space and Address fields.

Change Values

To change the values in the Memory window, perform the following brief procedure.

1. In the window, highlight the value you want to change. The values begin in the second column after the Address column.
2. Enter the new value and press the **Enter** key. The changed value is displayed in red.

View the Values for Other Memory Spaces

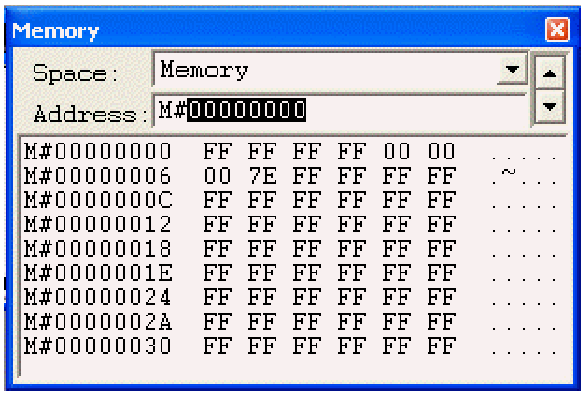
To view the values for other memory spaces, use one of the following procedures:

- Replace the initial letter with a different valid memory prefix and press the entry key. For example, type I for the IOData memory space.
- Select the space name in the Space drop-down list.

View or Search for an Address

To quickly view or search for an address in the Memory window, perform the following brief procedure.

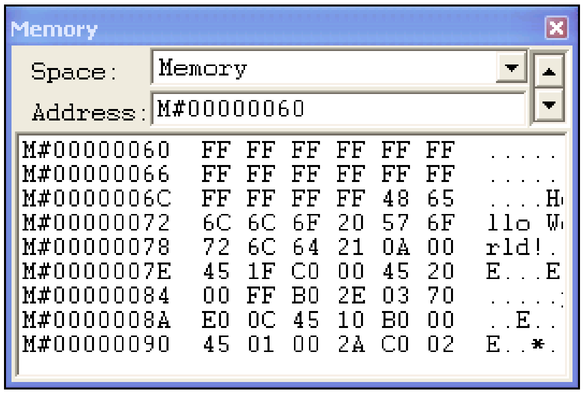
1. In the Memory window, highlight the address in the Address field, as shown in Figure 142.



```
M#00000000  FF FF FF FF 00 00  .....
M#00000006  00 7E FF FF FF FF  .~...
M#0000000C  FF FF FF FF FF FF  .....
M#00000012  FF FF FF FF FF FF  .....
M#00000018  FF FF FF FF FF FF  .....
M#0000001E  FF FF FF FF FF FF  .....
M#00000024  FF FF FF FF FF FF  .....
M#0000002A  FF FF FF FF FF FF  .....
M#00000030  FF FF FF FF FF FF  .....
```

Figure 142. Memory Window—Starting Address

2. Enter the address you want to find and press the **Enter** key; for example, find `60`. The system moves the selected address to the top of the Memory window, as shown in Figure 143.



```
M#00000060
M#00000060  FF FF FF FF FF FF  .....
M#00000066  FF FF FF FF FF FF  .....
M#0000006C  FF FF FF FF 48 65  ....H
M#00000072  6C 6C 6F 20 57 6F  llo W
M#00000078  72 6C 64 21 0A 00  rld!.
M#0000007E  45 1F C0 00 45 20  E...E
M#00000084  00 FF B0 2E 03 70  .....
M#0000008A  E0 0C 45 10 B0 00  ..E..
M#00000090  45 01 00 2A C0 02  E..*.
```

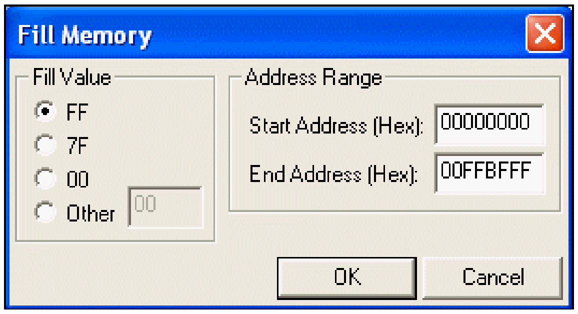
Figure 143. Memory Window—Requested Address

Fill Memory

Use this procedure to write a common value in all of the memory spaces in the specified address range, for example, to clear memory for the specified address range.

To fill a specified address range of memory, perform the following brief procedure.

1. Select the memory space in the **Space** drop-down list.
2. Right-click in the **Memory** window list box to display the context menu.
3. Select **Fill Memory**. The **Fill Memory** dialog box is displayed; see Figure 144.



Fill Value: [✓ FF] 7F 00 Other 00

Start Address (Hex): 00000000

End Address (Hex): 00FFBFFF

Figure 144. Fill Memory Dialog Box

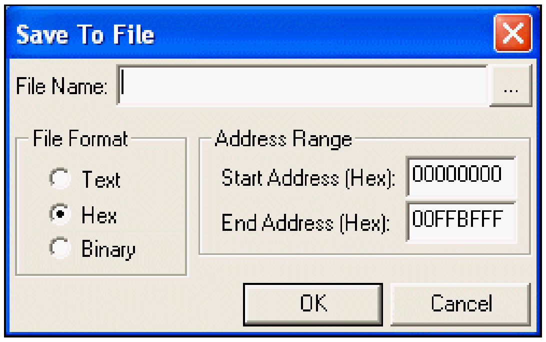
4. In the **Fill Value** area, select the characters to fill memory with or select the **Other** button. If you select the **Other** button, enter the fill characters in the **Other** field.
5. Enter the start address in hexadecimal format in the **Start Address (Hex)** field and enter the end address in hexadecimal format in the **End Address (Hex)** field. This address range is used to fill memory with the specified value.

6. Click **OK** to fill the selected memory.

Save Memory to a File

Perform the following procedure to save memory specified by an address range to a binary, hexadecimal, or text file.

1. Select the memory space in the **Space** drop-down list.
2. Right-click in the **Memory** window list box to display the context menu.
3. Select **Save to File**. The **Save to File** dialog box is displayed; see Figure 145.



File Name:

File Format: Text [✓ Hex] Binary

Start Address (Hex): 00000000

End Address (Hex): 00FFBFFF

Figure 145. Save to File Dialog Box

4. In the **File Name** field, enter the path and name of the file you want to save the memory to or click the **Browse** button (...) to search for a file or directory.
5. Enter the start address in hexadecimal format in the **Start Address (Hex)** field and enter the end address in hexadecimal format in the **End Address (Hex)** field to specify the address range of memory to save to the specified file.
6. Select whether to save the file as text, hex (hexadecimal), or binary.
7. Click **OK** to save the memory to the specified file.

Load a File into Memory

Perform the following steps to load or to initialize memory from an existing binary, hexadecimal, or text file.

1. Select the memory space in the **Space** drop-down list.
2. Right-click in the **Memory** window list box to display the context menu.
3. Select **Load from File**. The **Load from File** dialog box is displayed; see Figure 146.

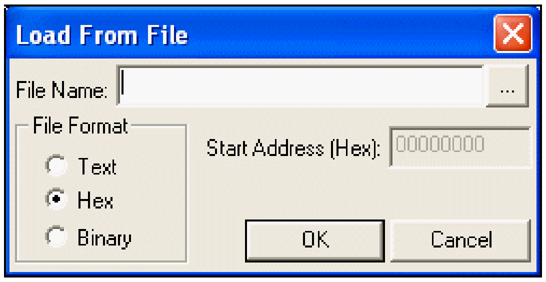


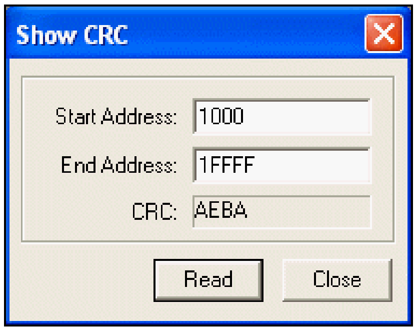
Figure 146. Load from File Dialog Box

4. In the **File Name** field, enter the path and name of the file to load or click the **Browse** button (...) to search for the file.
5. In the **Start Address (Hex)** field, enter the start address.
6. Select whether to load the file as text, hex (hexadecimal), or binary.
7. Click **OK** to load the file's contents into the selected memory.

Perform a Cyclic Redundancy Check

Observe the following procedure to perform a cyclic redundancy check (CRC).

1. Select the memory space in the **Space** drop-down list.
2. Right-click in the **Memory** window list box to display the context menu.
3. Select **Show CRC**. The **Show CRC** dialog box is displayed; see Figure 147.



Start Address: 1000
End Address: 1FFFF
CRC: AEBA

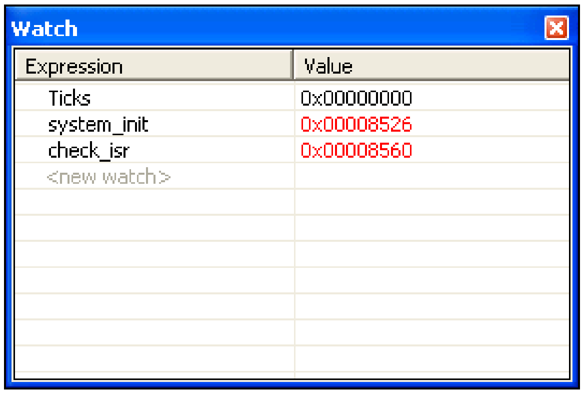
Figure 147. Show CRC Dialog Box

4. Enter the start address in the **Start Address** field. The start address must be on a 4K boundary. If the address is not on a 4K boundary, ZDS II produces an error message.

5. Enter the end address in the **End Address** field. If the end address is not a 4K increment, it is rounded up to a 4K increment.
6. Click **Read**. The checksum is displayed in the **CRC** field.

Watch Window

Click the Watch Window button to show or hide the Watch window, shown in Figure 148.



Expression	Value
Ticks	0x00000000
system_init	0x00008526
check_isr	0x00008560
<new watch>	

Figure 148. Watch Window

The Watch window displays all of the variables and their values defined using the WATCH command. If the variable is not in scope, the variable is not displayed. The values in the Watch window change as the program executes. Updated values appear in red.

The `0x` prefix indicates that the values are displayed in hexadecimal format. If you want the values to be displayed in decimal format, select **Hexadecimal Display** from the context menu.

Notes: If the Watch window displays unexpected values, deselect the **Use Register Variables** checkbox. See Use Register Variables – see page 57.

Use the Watch window to perform the following procedures.

- Add New Variables – see page 338
- Change Values – see page 338
- Remove an Expression – see page 338
- View a Hexadecimal Value – see page 338
- View a Decimal Value – see page 339

- View an ASCII Value – see page 339
- View a NULL-Terminated ASCII (ASCIZ) String – see page 339

Add New Variables

To add new variables in the Watch window, use one of the following two procedures.

- Click `<new watch>` in the **Expression** column, enter the expression, and press the **Enter** key.
- Select the variable in the source file, then drag and drop it into the **Watch** window.

Change Values

To change values in the Watch window, perform the following brief procedure.

1. In the window, highlight the value you want to change.
2. Enter the new value and press the **Enter** key. The changed value is displayed in red.

Remove an Expression

To remove an expression from the Watch window, perform the following brief procedure.

1. In the **Expression** column, click the expression you want to remove.
2. Press the **Delete** key to clear both columns.

View a Hexadecimal Value

To view the hexadecimal values of an expression, perform the following brief procedure.

1. Enter a hexadecimal expression in the format `hex` *expression* in the **Expression** column; for example, enter `hex tens`.

Note: You can also enter just the expression (for example, `tens`) to view the hexadecimal value of any expression. Hexadecimal format is the default.

2. Press the **Enter** key. The hexadecimal value displays in the **Value** column.

Note: To view the hexadecimal values for all expressions, select **Hexadecimal Display** from the context menu.

View a Decimal Value

To view the decimal values of an expression, perform the following brief procedure.

1. Enter a decimal expression in the format `dec` *expression* in the **Expression** column; for example, enter `dec huns`.
2. Press the **Enter** key. The decimal value displays in the **Value** column.

Note: To view the decimal values for all expressions, select **Hexadecimal Display** from the context menu.

View an ASCII Value

To view the ASCII values of an expression, perform the following brief procedure.

1. Enter an ASCII expression in the format `ascii` *expression* in the **Expression** column; for example, enter `ascii ones`.
2. Press the **Enter** key. The ASCII value displays in the **Value** column.

View a NULL-Terminated ASCII (ASCIZ) String

To view the NULL-terminated ASCII (ASCIZ) values of an expression, perform the following brief procedure.

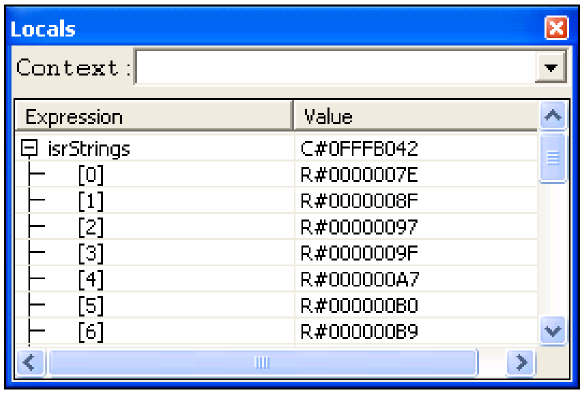
1. Enter an ASCIZ expression in the format `asciz` *expression* in the **Expression** column; for example, enter `asciz ones`.
2. Press the **Enter** key. The ASCIZ value displays in the **Value** column.

Locals Window

Click the Locals Window button to show or hide the Locals window. The Locals window, shown in Figure 149, displays all local variables that are currently in scope. Updated values appear in red.

The `0x` prefix indicates that the values are displayed in hexadecimal format. If you want the values to be displayed in decimal format, select **Hexadecimal Display** from the context menu.

Note: If the Locals window displays unexpected values, deselect the **Use Register Variables** checkbox. See Use Register Variables – see page 57.

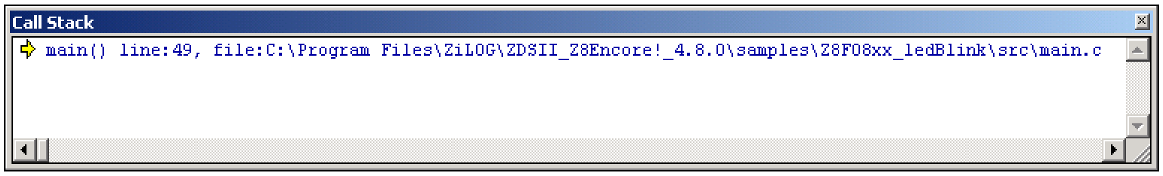


Expression	Value
isrStrings	C#0FFFB042
[0]	R#0000007E
[1]	R#0000008F
[2]	R#00000097
[3]	R#0000009F
[4]	R#000000A7
[5]	R#000000B0
[6]	R#000000B9

Figure 149. Locals Window

Call Stack Window

Click the Call Stack Window button to show or hide the Call Stack window, shown in Figure 150. If you want to copy text from the Call Stack Window, select the text and then select **Copy** from the context menu.



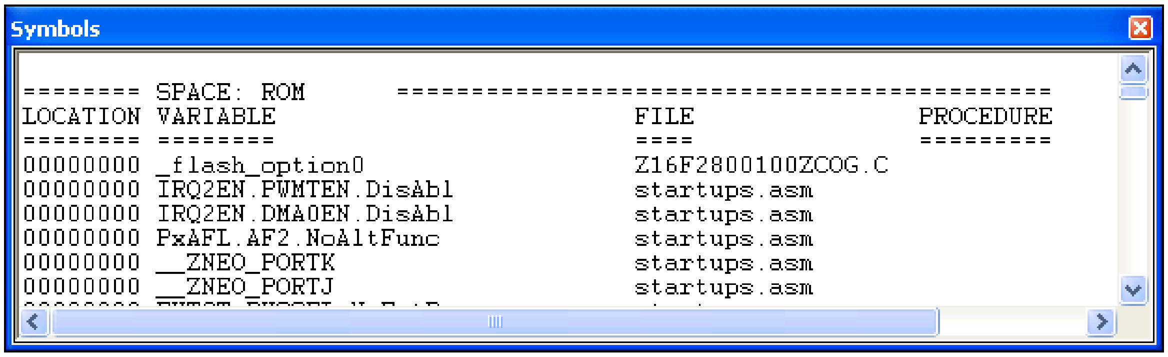
```
main() line:49, file:C:\Program Files\ZiLOG\ZDSII_Z8Encore!_4.8.0\samples\Z8F08xx_ledBlink\src\main.c
```

Figure 150. Call Stack Window

The Call Stack window allows you to view function frames that have been pushed onto the stack. Information in the Call Stack window is updated every time a debug operation is processed.

Symbols Window

Click the Symbols Window button to show or hide the Symbols window; see Figure 151.



```
======== SPACE: ROM      ============================================
LOCATION VARIABLE                       FILE                 PROCEDURE
======== ========                       ====                 =========
00000000 _flash_option0                 Z16F2800100ZCOG.C
00000000 IRQ2EN.PWMTEN.DisAbl           startups.asm
00000000 IRQ2EN.DMA0EN.DisAbl           startups.asm
00000000 PxAFL.AF2.NoAltFunc            startups.asm
00000000 __ZNEO_PORTK                   startups.asm
00000000 __ZNEO_PORTJ                   startups.asm
```

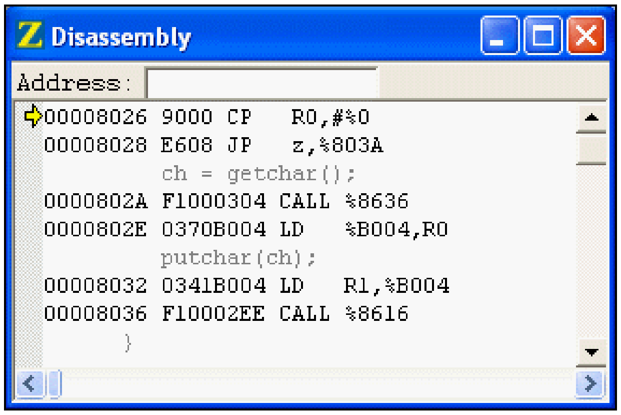
Figure 151. Symbols Window

> **Note:** Close the Symbols window before running a command script.

The Symbols window displays the address for each symbol in the program.

Disassembly Window

Click the Disassembly Window button to show or hide the Disassembly window; see Figure 152.



```
00008026 9000 CP   R0,#%0
00008028 E608 JP   z,%803A
          ch = getchar();
0000802A F1000304 CALL %8636
0000802E 0370B004 LD   %B004,R0
          putchar(ch);
00008032 0341B004 LD   R1,%B004
00008036 F10002EE CALL %8616
       }
```

Figure 152. Disassembly Window

The Disassembly window displays the assembly code associated with the code shown in the Code window. For each line in this window, the address location, the machine code, the assembly instruction, and its operands are displayed.

When you right-click in the Disassembly window, the context menu allows you to perform the following brief procedure.

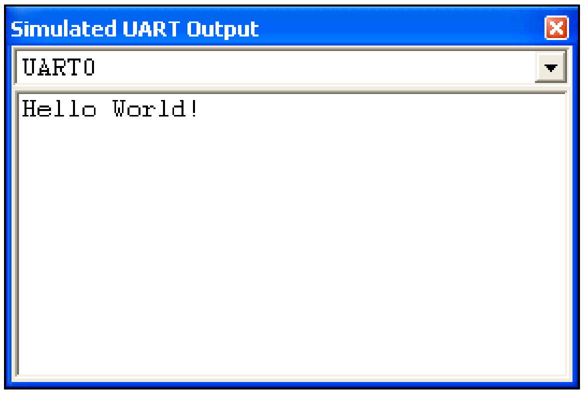
- Copy text
- Go to the source code
- Insert, edit, enable, disable, or remove breakpoints

Note: For more information about breakpoints, see the Using Breakpoints section on page 343.

- Reset the debugger
- Stop debugging
- Start or continue running the program (Go)
- Run to the cursor
- Pause the debugging (Break)
- Step into, over, or out of program instructions
- Set the next instruction at the current line
- Enable and disable source annotation and source line numbers
- Print the contents of the disassembly window
- Save the contents of the disassembly window to a file

Simulated UART Output Window

Click the Simulated UART Output Window button to show or hide the Simulated UART Output window; see Figure 153.



```
Hello World!
```

Figure 153. Simulated UART Output Window

The Simulated UART Output window displays the simulated output of the selected UART. Use the drop-down list to view the output for a particular UART.

Right-clicking in the Simulated UART Output window displays a context menu that provides access to the following features:

- Clear the buffered output for the selected UART
- Copy selected text to the Windows clipboard

Note: The Simulated UART Output window is available only when the Simulator is the active debug tool.

Using Breakpoints

This section to describes how to work with breakpoints while you are debugging. The following topics are covered:

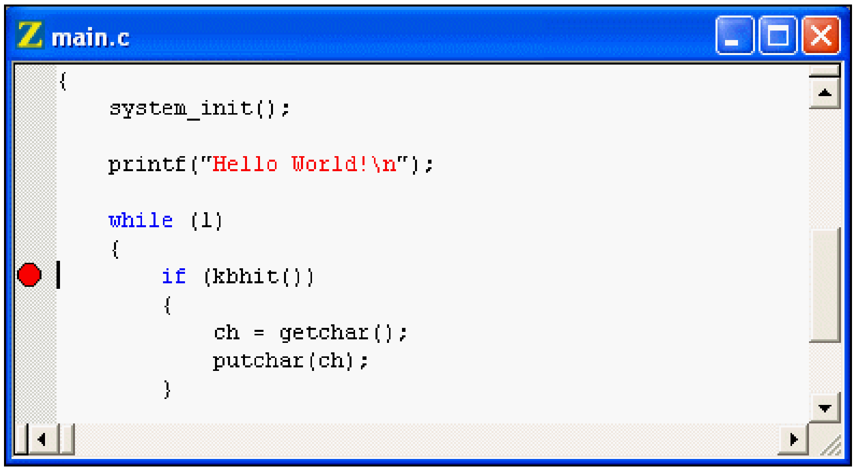
- Inserting Breakpoints – see page 344
- Viewing Breakpoints – see page 344
- Moving to a Breakpoint – see page 345
- Enabling Breakpoints – see page 345
- Disabling Breakpoints – see page 346
- Removing Breakpoints – see page 346

Inserting Breakpoints

There are three ways to place a breakpoint in your file:

- Click the line of code in which you want to insert the breakpoint. You can set a breakpoint in any line with a blue dot displayed to the left of the line (shown in Debug mode only). Next, click the **Insert/Remove Breakpoint** button ( ) on the Build or Debug toolbar.
- Click the line in which you want to add a breakpoint and select **Insert Breakpoint** from the context menu. You can set a breakpoint in any line with a blue dot displayed to the left of the line (shown in Debug mode only).
- Double-click in the gutter to the left of the line in which you want to add a breakpoint. You can set a breakpoint in any line with a blue dot displayed to the left of the line (shown in Debug mode only).

A red octagon shows that you have set a breakpoint at that location; see Figure 154.



```
{
    system_init();

    printf("Hello World!\n");

    while (1)
    {
        if (kbhit())
        {
            ch = getchar();
            putchar(ch);
        }
```

Figure 154. Setting a Breakpoint

Viewing Breakpoints

There are two ways to view breakpoints in your project:

- Select **Manage Breakpoints** from the **Edit** menu to display the **Breakpoints** dialog box; see Figure 155.
- Select **Edit Breakpoints** from the context menu to display the **Breakpoints** dialog box.

You can use the **Breakpoints** dialog box to view, go to, enable, disable, or remove breakpoints in an active project when in or out of Debug mode.

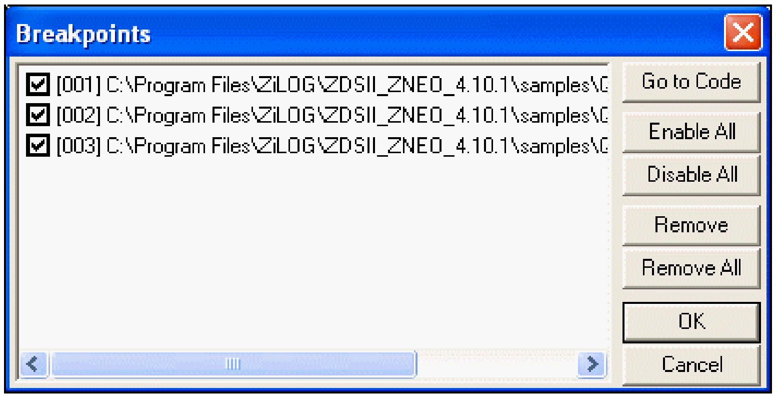

Breakpoints
[001] C:\Program Files\ZiLOG\ZDSII_ZNEO_4.10.1\samples\
[002] C:\Program Files\ZiLOG\ZDSII_ZNEO_4.10.1\samples\
[003] C:\Program Files\ZiLOG\ZDSII_ZNEO_4.10.1\samples\
Go to Code
Enable All
Disable All
Remove
Remove All
OK
Cancel

Figure 155. Viewing Breakpoints

Moving to a Breakpoint

To quickly move the cursor to a breakpoint you have previously set in your project, perform the following brief procedure.

1. Select **Manage Breakpoints** from the **Edit** menu. The **Breakpoints** dialog box is displayed.
2. Highlight the breakpoint you want.
3. Click **Go to Code**. Your cursor moves to the line in which the breakpoint is set.

Enabling Breakpoints

To make all breakpoints in a project active, perform the following brief procedure.

1. Select **Manage Breakpoints** from the **Edit** menu.The **Breakpoints** dialog box is displayed.
2. Click **Enable All**. Check marks are displayed to the left of all enabled breakpoints.
3. Click **OK**.

There are three ways to enable one breakpoint:

- Double-click the white octagon to remove the breakpoint and then double-click where the octagon was to enable the breakpoint

- Place your cursor in the line in the file where you want to activate a disabled breakpoint and click the **Enable/Disable Breakpoint** button on the Build or Debug toolbar
- Place your cursor in the line in the file where you want to activate a disabled breakpoint and select **Enable Breakpoint** from the context menu

The white octagon becomes a red octagon to indicate that the breakpoint is enabled.

Disabling Breakpoints

There are two ways to make all breakpoints in a project inactive:

- Select **Manage Breakpoints** from the **Edit** menu to display the **Breakpoints** dialog box. Click **Disable All**. Disabled breakpoints are still listed in the **Breakpoints** dialog box. Click **OK**.
- Click the **Disable All Breakpoints** button on the Debug toolbar.

There are two ways to disable one breakpoint:

- Place your cursor in the line in the file where you want to deactivate an active breakpoint and click the **Enable/Disable Breakpoint** button on the Build or Debug toolbar.
- Place your cursor in the line in the file where you want to deactivate an active breakpoint and select **Disable Breakpoint** from the context menu.

The red octagon becomes a white octagon to indicate that the breakpoint is disabled.

Removing Breakpoints

There are two ways to delete all breakpoints in a project:

- Select **Manage Breakpoints** from the **Edit** menu to display the **Breakpoints** dialog box. Click **Remove All**, then click **OK**. All breakpoints are removed from the **Breakpoints** dialog box, as well as all project files.
- Click the **Remove All Breakpoints** button on the Build or Debug toolbar.

There are four ways to delete a single breakpoint:

- Double-click the red octagon to remove the breakpoint.
- Select **Manage Breakpoints** from the **Edit** menu to display the **Breakpoints** dialog box. Click **Remove**, then click **OK**. The breakpoint is removed from the **Breakpoints** dialog box and the file.

- Place your cursor in the line in the file where there is a breakpoint and click the **Insert/ Remove Breakpoint** button on the Build or Debug toolbar.
- Place your cursor in the line in the file where there is a breakpoint and select **Remove Breakpoint** from the context menu.

Appendix A. Running ZDS II from the Command Line

You can run ZDS II from the command line. ZDS II generates a make file (*project*_`Debug.mak` or *project*_`Release.mak`, depending on the project configuration) every time you build or rebuild a project. For a project named `test.zdsproj` set up in the Debug configuration, ZDS II generates a make file named `test_Debug.mak` in the project directory. You can use this make file to run your project from the command line.

This section covers the following topics:

Building a Project from the Command Line

To build a project from the command line, observe the following procedure:

1. Add the ZDS II bin directory (for example, `C:\Program Files\Zilog\ZDSII_Z8Encore!_4.11.0\bin`) to your path by setting the PATH environment variable. The make utility is available in this directory.
2. Open the project using the IDE.
3. Export the make file for the project using the Export Makefile command in the **Project** menu.
4. Open a DOS window and change to the intermediate files directory.
5. Build the project using the make utility on the command line in a DOS window.
6. To build a project by compiling only the changed files, use the following command:
   ```
   make -f sampleproject_Debug.mak
   ```
   To rebuild the entire project, use the following command:

```
make rebuildall -f sampleproject_Debug.mak
```

Running the Compiler from the Command Line

To run the compiler from the command line:

1. Open the make file in a text editor.
2. Copy the options in the CFLAGS section.
3. In a Command Prompt window, enter the path to the compiler, the options from the CFLAGS section (on a single line and without backslashes), and your C file. For example:

```
C:\PROGRA~1\Zilog\ZDSII_ZNEO_4.11.0\bin\eZ8cc -alias -asm -
const:RAM
-debug -define:_z8f64 -NOexpmac -NOfplib -intsrc -intrinsic -
NOkeepasm
-NOkeeplst -NOlist -NOlistinc -maxerrs:50 -NOmodsect -promote -
quiet -NOstrict -NOwatch -optsize -localopt -localcse -localfold -
localcopy -peephole
-globalopt -NOglobalcse -NOglobalfold -NOglobalcopy -NOloopopt -
NOsdiopt
-NOjmpopt -
stdinc:"..\include;C:\PROGRA~1\Zilog\ZDSII_ZNEO_4.11.0\include" -
usrinc:"..\include" -cpu:z8f64 -bitfieldsize:24 -charsize:8 -
doublesize:32
-floatsize:32 -intsize:24 -longsize:32 -shortsize:16 -asmsw:"-
cpu:z8f64" test.c
```

Notes: If you use DOS, use double quotation marks for the `-stdinc` and `-usrinc` commands for the C-Compiler.

For example,

```
-stdinc:"C:\ez8\include"
```

If you use *cygwin*, use single quotation marks on both sides of a pair of braces for the `-stdinc` and `-usrinc` commands for the C-Compiler.

For example,

```
-stdinc:'{C:\ez8\include}'
```

Running the Assembler from the Command Line

To run the assembler from the command line:

1. Open the make file in a text editor.
2. Copy the options in the AFLAGS section.
3. In a Command Prompt window, enter the path to the assembler, the options from the AFLAGS section (on a single line and without backslashes), and your assembly file. For example,

```
C:\PROGRA~1\Zilog\ZDSII_ZNEO_4.11.0\bin\eZ8asm -debug -genobj -
NOigcase
-include:"..\include" -list -NOlistmac -name -pagelen:56 -
pagewidth:80 -quiet -warn -NOzmasm -cpu:z8f64 test.asm
```

Running the Linker from the Command Line

To run the linker from the command line:

1. Open the make file in a text editor.
2. In a Command Prompt window, enter the path to the linker and your linker file. For example,

```
C:\PROGRA~1\Zilog\ZDSII_ZNEO_4.11.0\bin\eZ8lnk
@e:\ez8\rtl\testfiles\test\test.linkcmd
```

Assembler Command Line Options

Table 23 describes the assembler command line options.

Notes: If you use DOS, use double quotation marks for the `-stdinc` and `-usrinc` commands for the C compiler. For example,

```
-stdinc:"C:\ez8\include"
```

If you use *cygwin*, use single quotation marks on both sides of a pair of braces for the `-stdinc` and `-usrinc` commands for the C compiler.

For example,

```
-stdinc:'{C:\ez8\include}'
```

Table 23. Assembler Command Line Options

Option Name	Description
`-cpu:`*name*	Sets the CPU.
`-debug`	Generates debug information for the symbolic debugger. The default setting is `-nodebug`.
`-define:`*name*[=*value*]	Defines a symbol and sets it to the constant value. For example: `-define:DEBUG=0` This option is equivalent to the C #define statement. The alternate syntax, `-define:`*myvar*, is the same as `-define:`*myvar*`=1`.
`-FP=Qn`	The ZNEO architecture allows you to specify which quad register is to be used as the frame pointer, which is implicitly used in the LINK and UNLNK instructions. The assembler accepts FP as a quad register, and the FP command line option and directive tell the assembler which quad register to use as the frame pointer.
`-genobj`	Generates an object file with the `.obj` extension. This is the default setting.
`-help`	Displays the assembler help screen.
`-igcase`	Suppresses case sensitivity of user-defined symbols. When this option is used, the assembler converts all symbols to uppercase. This is the default setting.
`-include:`*path*	Allows the insertion of source code from another file into the current source file during assembly.
`-list`	Generates an output listing with the `.lst` extension. This is the default setting.
`-listmac`	Expands macros in the output listing. This is the default setting.
`-listoff`	Does not generate any output in list file until a directive in assembly file sets the listing as on.
`-MAXBRANCH=`<*expression*>	The MAXBRANCH directive allows control over how large a branch the assembler generates in the jump translation phase, especially when the label branched to is in a separate assembly unit, so the assembler must assume the largest possible branch. A possible use of the command line option is in an application expected to fit in 64K of code space so that no branches of more than a 16-bit offset are required. Use the directive to override the command line option to impose either a stricter or more lenient requirement.
`-metrics`	Keeps track of how often an instruction is used. This is a static rather than a dynamic measure of instruction frequency.
`-name`	Displays the name of the source file being assembled.

Table 23. Assembler Command Line Options (Continued)

Option Name	Description
`-nodebug`	Does not create a debug information file for the symbolic debugger. This is the default setting.
`-nogenobj`	Does not generate an object file with the `.obj` extension. The default setting is `genobj`.
`-noigcase`	Enables case sensitivity of user-defined symbols. The default setting is `igcase`.
`-nolist`	Does not create a list file. The default setting is `list`.
`-nolistmac`	Does not expand macros in the output listing. The default setting is `listmac`.
`-noquiet`	Displays title and other information. This is the default.
`-nosdiopt`	Does not perform span-dependent optimizations. All size optimizable instructions use the largest instruction size. The default is `sdiopt`.
`-nowarns`	Suppresses the generation of warning messages to the screen and listing file. A warning count is still maintained. The default is to generate warning messages.
`-pagelength:`*n*	Sets the new page length for the list file. The page length must immediately follow the = (with no space between). The default is `56`. For example: `-pagelength=60`
`-pagewidth:`*n*	Sets the new page width for the list file. The page width must immediately follow the = (with no space between). The default and minimum page width is 80. The maximum page width is `132`. For example: `-pagewidth=132`
`-quiet`	Suppresses title information that is normally displayed to the screen. Errors and warnings are still displayed. The default setting is to display title information.
`-relist:`*mapfile*	Generates an absolute listing by making use of information contained in a linker map file. This results in a listing that matches linker-generated output. *mapfile* is the name of the map file created by the linker. For example: `-relist:product.map`
`-sdiopt`	Performs span-dependent optimizations. The smallest instruction size allowed is selected for all size optimizable instructions. This is the default setting.
`-trace`	Debug information for internal use.
`-version`	Prints the version number of the assembler.
`-warns`	Toggles display warnings.

Compiler Command Line Options

Table 24 describes the compiler command line options.

Notes: If you use DOS, use double quotation marks for the `-stdinc` and `-usrinc` commands for the C compiler.

For example,

```
-stdinc:"C:\ez8\include"
```

If you use *cygwin*, use single quotation marks on both sides of a pair of braces for the `-stdinc` and `-usrinc` commands for the C compiler.

For example,

```
-stdinc:'{C:\ez8\include}'
```

Table 24. Compiler Command Line Options

Option Name	Description
`-asm`	Assembles compiler-generated assembly file. This switch results in the generation of an object module. The assembly file is deleted if no assemble errors are detected and the `keepasm` switch is not given. The default is `asm`.
`-asmsw:"`*sw*`"`	Passes *sw* to the assembler when assembling the compiler-generated assembly file.
`-chartype:[s|u]`	Selects whether plain char is implemented as signed or unsigned. The default is unsigned.
`-cpu:`*cpu*	Sets the CPU.
`-debug`	Generates debug information for the symbolic debugger.
`-define:`*def*	Defines a symbol and sets it to a constant value. For example: `-define:myvar= 0` The alternate syntax, `-define:`*myvar*, is the same as `-define:`*myvar*`=1`.
`-genprintf`	The format string is parsed at compile time, and direct inline calls to the lower level helper functions are generated. The default is `genprintf`.
`-help`	Displays the compiler help screen.
`-keepasm`	Keeps the compiler-generated assembly file.
`-keeplst`	Keeps the assembly listing file (`.lst`).
`-list`	Generates a `.lis` source listing file.

Table 24. Compiler Command Line Options (Continued)

Option Name	Description
`-listinc`	Displays included files in the compiler listing file.
`-model:`*model*	Selects the memory model. Select `S` for a small memory model or `L` for a large memory model. The default is `S`.
`-modsect`	Generate distinct code segment name for each module.
`-noasm`	Does not assemble the compiler-generated assembly file.
`-nodebug`	Does not generate symbol debug information. This is the default.
-nogenprint	A call to `printf()` or `sprintf()` parses the format string at run time to generate the required output.
`-nokeepasm`	Deletes the compiler-generated assembly file. This is the default.
`-nokeeplst`	Does not keep the assembly listing file (`.lst`). This is the default.
`-nolist`	Does not produce a source listing. All errors are identified on the console. This is the default.
`-nolistinc`	Does not show include files in the compiler listing file. This is the default.
-nomodsect	Does not generate a distinct code segment name for each module. The code segment is named as “code” for every module; this is the default.
`-noquiet`	Displays the title information. This is the default.
`-noregvar`	Turns off the use of register variables.
`-quiet`	Suppresses title information that is normally displayed to the screen. Errors and warnings are still displayed.
`-regvar`	Turns on the use of register variables. This is the default.
`-stdinc:"`*path*`"`	Sets the path for the standard include files. This defines the location of include files using the `#include` *file*`.h` syntax. Multiple paths are separated by semicolons. For example: `-stdinc:"c:\rtl;c:\myinc"` In this example, the compiler looks for the include file in 1. the current directory 2. the `c:\rtl` directory 3. the `c:\myinc` directory If the file is not found after searching the entire path, the compiler flags an error. Omitting this switch tells the compiler to search only the current and default directory.

Table 24. Compiler Command Line Options (Continued)

Option Name	Description
`-usrinc:"`*path*`"`	Sets the search path for user include files. This defines the location of include files using the `#include "`*file*`.h"` syntax. Multiple paths are separated by semicolons. For example: `-usrinc:"c:\rtl;c:\myinc"` In this example, the compiler looks for the include file in 1. the current directory 2. the `c:\rtl` directory 3. the `c:\myinc` directory If the file is not found after searching the entire path, the compiler flags an error. Omitting this switch tells the compiler to search only the current directory.
`-version`	Prints the version number of the compiler.

Librarian Command Line Options

Table 25 describes the librarian command line options.

Notes: If you use DOS, use double quotation marks for the `-stdinc` and `-usrinc` commands for the C compiler.

For example,

```
-stdinc:"C:\ez8\include"
```

If you use cygwin, use single quotation marks on both sides of a pair of braces for the `-stdinc` and `-usrinc` commands for the C compiler.

For example,

```
-stdinc:'{C:\ez8\include}'
```

Table 25. Librarian Command Line Options

Option Name	Description
`-help`	Displays the librarian help screen.
`-list`	Generates an output listing with the `.lst` extension. This is the default setting.
`-noquiet`	Displays the title information.
`-nowarn`	Suppresses warning messages.

Table 25. Librarian Command Line Options (Continued)

Option Name	Description
`-quiet`	Suppresses title information that is normally displayed to the screen. Errors and warnings are still displayed. The default setting is to display title information.
`-version`	Displays the version number of the librarian.
`-warn`	Displays warnings.

Linker Command Line Options

Table 26 describes the linker command line options.

Notes: If you use DOS, use double quotation marks for the `-stdinc` and `-usrinc` commands for the C compiler.

For example,

```
-stdinc:"C:\ez8\include"
```

If you use cygwin, use single quotation marks on both sides of a pair of braces for the `-stdinc` and `-usrinc` commands for the C compiler.

For example,

```
-stdinc:'{C:\ez8\include}'
```

Table 26. Linker Command Line Options

Option Name	Description
`copy` *segment* = *space*	Makes a copy of a segment into a specified address space.
`-debug`	Turns on debug information generation.
`define` *symbol* = *expr*	Defines a symbol and sets it to the constant value. For example: `-define:DEBUG=0` This option is equivalent to the C #define statement. The alternate syntax, `-define:`*myvar*, is the same as `-define:`*myvar*`=1`.
`-format:[intel|intel32|omf695]`	Sets the format of the hex file output of the linker to `intel` or `intel32` (Intel Hex records) or `omf695` (IEEE695 format).

Table 26. Linker Command Line Options (Continued)

Option Name	Description
`-igcase`	Suppresses case sensitivity of user-defined symbols. When this option is used, the linker converts all symbols to uppercase. This is the default setting.
`locate` *segment at expr*	Specifies the address where a group, address space, or segment is to be located.
`-nodebug`	Turns off debug information generation.
`-noigcase`	Enables case sensitivity of user-defined symbols. The default setting is `igcase`.
`order` *segment_list or space_list*	Establishes a linking sequence and sets up a dynamic range for contiguously mapped address spaces.
`range` *space* = *address_range*	Sets the lower and upper bounds of a group, address space, or segment.
`sequence` *segment_list or space_list*	Allocates a group, address space, or segment in the order specified.

Appendix B. Using the Command Processor

The Command Processor allows you to use commands or script files to automate the execution of a significant portion of the functionality of the integrated development environment (IDE). This section covers the following topics:

- Sample Command Script File – see page 363
- Supported Script File Commands – see page 364
- Running the Flash Loader from the Command Processor – see page 387

You can run commands in either of the following ways:

- Using the Command Processor toolbar in the IDE to run a single command. Commands entered into the Command Processor toolbar are executed after you press the Enter (or Return) key or click the Run Command button. The toolbar is described in Command Processor Toolbar – see page 19.
- Using the `batch` command to run a command script file from the Command Processor toolbar in the IDE. For example:

  ```
  batch "c:\path\to\command\file\runall.cmd"
  batch "commands.txt"
  ```

- Passing a command script file to the IDE when it is started.

  You must precede the script file with an at symbol (`@`) when passing the path and name of the command script file to the IDE on the command line. For example:

  ```
  zds2ide @c:\path\to\command\file\runall.cmd
  zds2ide @commands.txt
  ```

Processed commands are echoed, and associated results are displayed in the Command Output window in the IDE and, if logging is enabled (see the log command on page 373), in the log file as well.

Commands are not case-sensitive.

In directory or path-based parameters, you can use `\`, `\\`, or `/` as separators as long as you use the same separator throughout a single parameter. The following examples show legal usage:

```
cd "..\path\to\change\to"
cd "..\\path\\to\\change\\to"
cd "../path/to/change/to"
```

The following examples show illegal usage:

```
cd "..\path/to\change/to"
cd "..\\path\to\change\to"
```

Table 27 lists ZDS II menu commands and dialog box options that have corresponding script file commands. Each of these commands is described on the corresponding linked page.

Table 27. Script File Commands

ZDS II Menu	ZDS II Commands	Dialog Box Options	Script File Commands	Page
File	New Project		new project	374
	Open Project		open project	375
	Exit		exit	372
Edit	Manage Breakpoints		list bp	373
		Go to Code		
		Enable All		
		Disable All		
		Remove	cancel bp	366
		Remove All	cancel all	366
Project	Add Files		add file	364
	Settings (General page)	CPU Family		378
		CPU	option general cpu	
		Show Warnings	option general warn	
		Generate Debug Information	option general debug	
		Ignore Case of Symbols	option general igcase	
		Intermediate Files Directory	option general outputdir	
	Settings (Assembler page)	Includes	option assembler include	376
		Defines	option assembler define	
		Generate Assembly Listing Files (.lst)	option assembler list	
		Expand Macros	option assembler listmac	
		Page Width	option assembler pagewidth	
		Page Length	option assembler pagelen	
	Settings (Code Generation page)	Limit Optimizations for Easier Debugging	option compiler reduceopt	377
		Memory Model	option compiler model	

Table 27. Script File Commands (Continued)

ZDSII Menu	ZDSII Commands	Dialog Box Options	Script File Commands	Page
Project (cont'd)	Settings (Listing Files page)	Generate C Listing Files (.lis) With Include Files Generate Assembly Source Code Generate Assembly Listing Files (.lst)	option compiler list option compiler listinc option compiler keepasm option compiler keeplst	377
	Settings (Preprocessor page)	Preprocessor Definitions Standard Include Path User Include Path	option compiler define option compiler stdinc option compiler usrinc	377
	Settings (Advanced page)	Use Register Variables Generate Printfs Inline Distinct Code Segment for Each Module Default Type of Char	option compiler regvar option compiler genprintf option compiler modsect option compiler chartype	377
	Settings (Librarian page)	Output File Name	option librarian outfile	378
	Settings (Commands page)	Always Generate from Settings Additional Directives Edit (Additional Linker Directives dialog box) Use Existing	option linker createnew option linker useadddirective option linker directives option linker linkctlfile	379
	Settings (Objects and Libraries page)	Additional Object/Library Modules Standard Included in Project Use Standard Startup Linker Commands C Runtime Library Floating Point Library	option linker objlibmods option linker startuptype option linker startuptype option linker startuplnkcmds option linker usecrun option linker fplib	379
	Settings (Address Spaces page)	Constant Data (ROM) Internal Ram (RAM) SFRs and IO (IOData) Program Space (EROM) Extended RAM (ERAM)	option linker rom option linker ram option linker iodata option linker erom option linker eram	379
	Settings (Warnings page)	Treat All Warnings as Fatal Treat Undefined Symbols as Fatal Warn on Segment Overlap	option linker warnisfatal option linker undefisfatal option linker warnoverlap	379

Table 27. Script File Commands (Continued)

ZDSII Menu	ZDSII Commands	Dialog Box Options	Script File Commands	Page
Project (cont'd)	Settings (Output page)	Output File Name Generate Map File Sort Symbols By Show Absolute Addresses in Assembly Listings Executable Formats Fill Unused Hex File Bytes with 0xFF Maximum Bytes per Hex File Line	option linker of option linker map option linker sort option linker relist option linker exeform option linker padhex option linker maxhexlen	379
	Settings (Debugger page)	Use Page Erase Before Flash-ing Target Setup Add Copy Delete Debug Tool Setup	 target set target options target create target copy debugtool set debugtool set	 385 384 383 383 369 369
	Export Makefile		makfile makefile	374
Build	Build		build	366
	Rebuild All		rebuild	381
	Stop Build		stop	383
	Set Active Configuration		set config	382
	Manage Configurations		set config delete config	382 370
Debug	Stop Debugging		quit	381
	Reset		reset	381
	Go		go	372
	Break		stop	383
	Step Into		stepin	382
	Step Over		step	382
	Step Out		stepout	383

Table 27. Script File Commands (Continued)

ZDSII Menu	ZDSII Commands	Dialog Box Options	Script File Commands	Page
Tools	Flash Loader			387
	Calculate File Checksum		checksum	367
	Show CRC		crc	367

Sample Command Script File

A script file is a text-based file that contains a collection of commands. The file can be created with any editor that can save or export files in a text-based format. Each command must be listed on its own line. Anything following a semicolon (`;`) is considered a comment.

The following example presents a command script file:

```
; change to correct default directory
cd "m:\ZNEO\test\focustests"
open project "focus1.zdsproj"
log "focus1.log" ; Create log file
log on ; Enable logging
rebuild
reset
bp done
go
wait 2000 ; Wait 2 seconds
print "pc = %x" reg PC
log off ; Disable logging
quit ; Exit debug mode
close project
wait 2000
open project "focus2.zdsproj"
reset
bp done
go
wait 2000 ; Wait 2 seconds
log "focus2.log" ; Open log file
log on ; Enable logging
print "pc = %x" reg PC
log off ; Disable logging
quit
exit; Exit debug mode
```

This script consecutively opens two projects, sets a breakpoint at label `done`, runs to the breakpoint, and logs the value of the PC register. After the second project is complete, the script exits the IDE. The first project is also rebuilt.

Supported Script File Commands

The Command Processor supports the following script file commands:

add file	delete config	savemem
batch	examine (?) for Expressions	set config
bp	examine (?) for Variables	step
build	exit	stepin
cancel all	fillmem	stepout
cancel bp	go	stop
cd	list bp	target copy
checksum	loadmem	target create
crc	log	target get
debugtool copy	makfile or makefile	target help
debugtool create	new project	target list
debugtool get	open project	target options
debugtool help	option	target save
debugtool list	print	target set
debugtool save	pwd	target setup
debugtool set	quit	wait
debugtool setup	rebuild	wait bp
defines	reset	

In the following syntax descriptions, items enclosed in angle brackets (< >) must be replaced with actual values, items enclosed in square brackets ([ ]) are optional, double quotes (`"`) indicate where double quotes must exist, and all other text must be included as it is presented.

add file

The `add file` command adds the given file to the currently open project. If the full path is not supplied, the current working directory is used. The following example presents the syntax of the `add file` command:

```
    add file "<[path\]<filename>"
```

For example:

```
    add file "c:\project1\main.c"
```

batch

The `batch` command runs a script file through the Command Processor. If the full path is not supplied, the current working directory is used. The following example presents the syntax of the `batch` command:

```
batch [wait] "<[path\]<filename>"
```

The `wait` parameter blocks processing of the current script until the invoked batch file completes, making this parameter useful when nesting script files.

For example:

```
BATCH "commands.txt"
batch wait "d:\batch\do_it.cmd"
```

bp

The `bp` command sets a breakpoint at a given label or line in a file. The syntax can take one of the following forms:

```
bp line <line number>
```

sets/removes a breakpoint on the given line of the active file.

```
bp <symbol>
```

sets a breakpoint at the given symbol. This version of the `bp` command can only be used during a debug session.

For example:

```
bp main
bp line 20
```

Starting in ZDS II version 4.11.0, you can also use the `bp when` command to set access breakpoints. You can have up to three access breakpoints set at one time. Access breakpoints can be set and cleared while the target is running. The following example presents the syntax of the `bp when` command:

`bp when` [`READ`|`WRITE`] <*address*> [`MASK` <*mask*>]

If not specified, `MASK` defaults to `0xFFFFFE` (1 bit masked).

The valid `MASK` range is `0xFF8000` (15 bits masked) to `0xFFFFFF` (0 bits masked).

For example:

`BP WHEN READ 0xFFBFF0 MASK 0xFFFFF0` break when read address in range `0xFFBFF0-0xFFBFFF`

`BP WHEN WRITE 0x1234` break when write to address `0x1234`

`BP WHEN 0xFFE030` break when read or write address `0xFFE030` (SFR IRQ0)

`BP WHEN READ &my_var` break when read variable `my_var`

You can use the `cancel bp` or `cancel all` commands to clear access breakpoints. See the cancel all section on page 366 and the cancel bp section on page 366.

build

The `build` command builds the currently open project. This command blocks the execution of other commands until the build process is complete. The following example presents the syntax of the `build` command:

```
build
```

cancel all

The `cancel all` command clears all breakpoints in the currently loaded project. The following example presents the syntax of the `cancel all` command:

```
cancel all
```

Starting in version 4.11.0, you can also use the `cancel all` command to clear all breakpoints, including access breakpoints. For example:

```
cancel all
```

To clear a specified breakpoint, see cancel bp. To set access breakpoints, see bp.

cancel bp

The `cancel bp` command clears the breakpoint at the bp list index. Use the `list bp` command to retrieve the index of a particular breakpoint. The following example presents the syntax of the `cancel bp` command:

```
cancel bp <index>
```

For example:

```
cancel bp 3
```

Starting in version 4.11.0, you can also use the `cancel bp when` command to clear an access breakpoint set at a specified address. The following example presents the syntax of the `cancel bp when` command:

```
cancel bp when <address>
```

To clear all access breakpoints, see cancel all. To set access breakpoints, see bp.

cd

The `cd` command changes the working directory to *dir*. The following example presents the syntax of the `cd` command:

```
cd "<dir>"
```

For example:

```
cd "c:\temp"
cd "../another_dir"
```

checksum

The `checksum` command calculates the checksum of a hex file. The following example presents the syntax of the `checksum` command:

```
checksum "<filename>"
```

For example, if you use the following command:

```
checksum "ledblink.hex"
```

The file checksum for the example is:

```
0xCEA3
```

crc

The `CRC` command performs a cyclic redundancy check (CRC). The syntax can take one of two forms:

- `crc` calculates the CRC for the whole Flash memory.
- `crc STARTADDRESS="<address>" ENDADDRESS="<endaddress>"` calculates the CRC for 4K-increment blocks. `STARTADDRESS` must be on a 4K boundary; if the address is not on a 4K boundary, ZDS II produces an error message. `ENDADDRESS` must be a 4K increment; if the end address is not a 4K increment, it is rounded up to a 4K increment.

For example:

```
crc STARTADDRESS="1000" ENDADDRESS="1FFF"
```

debugtool copy

The `debugtool copy` command creates a copy of an existing debug tool with the given new name. The syntax can take one of two forms:

- `debugtool copy NAME="<new debug tool name>"` creates a copy of the active debug tool named the value given for `NAME`.
- `debugtool copy NAME="<new debug tool name>" SOURCE="<existing debug tool name>"` creates a copy of the `SOURCE` debug tool named the value given for `NAME`.

For example:

```
debugtool copy NAME="Sim3" SOURCE="Z16F2811AL"
```

debugtool create

The `debugtool create` command creates a new debug tool with the given name and using the given communication type: `usb`, `tcpip`, `ethernet`, or `simulator`. The following example presents the syntax of the `debugtool create` command:

`debugtool create NAME="<`*`debug tool name`*`>" COMMTYPE="<`*`comm type`*`>"`

For example:

```
debugtool create NAME="emulator2" COMMTYPE="ethernet"
```

debugtool get

The `debugtool get` command displays the current value for the given data item for the active debug tool. Use the `debugtool setup` command to view available data items and current values. The following example presents the syntax of the `debugtool get` command:

`debugtool get "<`*`data item`*`>"`

For example:

```
debugtool get "ipAddress"
```

debugtool help

The `debugtool help` command displays all debugtool commands. The following example presents the syntax of the `debugtool help` command:

```
debugtool help
```

debugtool list

The `debugtool list` command lists all available debug tools. The syntax can take one of two forms:

- `debugtool list` displays the names of all available debug tools.

- `debugtool list COMMTYPE="<type>"` displays the names of all available debug tools using the given communications type: `usb`, `tcpip`, `ethernet`, or `simulator`.

For example:

```
debugtool list COMMTYPE="ethernet"
```

debugtool save

The `debugtool save` command saves a debug tool configuration to disk. The syntax can take one of two forms:

- `debugtool save` saves the active debug tool.
- `debugtool save NAME ="<Debug Tool Name>"` saves the given debug tool.

For example:

```
debugtool save NAME="USBSmartCable"
```

debugtool set

The `debugtool set` command sets the given data item to the given data value for the active debug tool or activates a particular debug tool. The syntax can take one of two forms:

- `debugtool set "<data item>" "<new value>"` sets *data item* to *new value* for the active debug tool. Use `debugtool setup` to view available data items and current values.

For example:

```
debugtool set "ipAddress" "123.456.7.89"
```

- `debugtool set "<debug tool name>"` activates the debug tool with the given name. Use `debugtool list` to view available debug tools.

debugtool setup

The `debugtool setup` command displays the current configuration of the active debug tool. The following example presents the syntax of the `debugtool setup` command:

```
debugtool setup
```

defines

The `defines` command provides a mechanism to add to, remove from, or replace define strings in the compiler preprocessor defines and assembler defines options. This command provides a more flexible method to modify the defines options than the `option` command, which requires that the entire defines string be set with each use. Each `defines` parameter is a string containing a *single* define symbol, such as `"TRACE"` or `"_SIMULATE=1"`. The `defines` command can take one of three forms:

- `defines <compiler|assembler> add "<new define>"` adds the given define to the compiler or assembler defines, as indicated by the first parameter.
- `defines <compiler|assembler> replace "<new define>" "<old define>"` replaces <*old define*> with <*new define*> for the compiler or assembler defines, as indicated by the first parameter. If <*old define*> is not found, no change is made.
- `defines <compiler|assembler> remove "<define to be removed>"` removes the given define from the compiler or assembler defines, as indicated by the first parameter.

For example:

```
defines compiler add "_TRACE"
defines assembler add "_TRACE=1"
defines assembler replace "_TRACE" "_NOTRACE"
defines assembler replace "_TRACE=1" "_TRACE=0"
defines compiler remove "_NOTRACE"
```

delete config

The `delete config` command deletes the given existing project build configuration. The following example presents the syntax of the `delete config` command:

```
delete config "<config_name>"
```

If <*config_name*> is active, the first remaining build configuration, if any, is made active. If <*config_name*> does not exist, no action is taken.

For example:

```
delete config "MyDebug"
```

examine (?) for Expressions

The examine command evaluates the given expression and displays the result. It accepts any legal expression made up of constants, program variables, and C operators. The syntax takes the following form:

```
? [<data_type>] [<radix>] <expr> [:<count>]
```

<*data_type*> can consist of one of the following types:

```
short
int[eger]
long
ascii
asciz
```

<*radix*> can consist of one of the following types:

```
dec[imal]
hex[adecimal]
oct[al]
bin[ary]
```

Omitting a <*data_type*> or <*radix*> results in using the `$data_type` or `$radix` pseudo-variable, respectively.

[:<*count*>] represents the number of items to display.

The following list presents examples.

`? x`	shows the value of x using `$data_type` and `$radix`.
`? ascii STR`	shows the ASCII string representation of STR.
`? 0x1000`	shows the value of 0x1000 in the `$data_type` and `$radix`.
`? *0x1000`	shows the byte at address 0x1000.
`? *0x1000 :25`	shows 25 bytes at address 0x1000.
`? L0`	shows the value of register D0:0 using `$data_type` and `$radix`.
`? asciz D0:0`	shows the null-terminated string pointed to by the contents of register D0:0.

examine (?) for Variables

The examine command displays the values of variables. This command works for values of any type, including arrays and structures. The following example presents the syntax:

? <*expression*>

The following list presents examples:

To see the value of z, enter:	`?z`
To see the nth value of array x, enter:	`? x[n]`
To see all values of array x, enter:	`?x`

To see the nth through the n+5th values of array x, enter:	`?x[n]:5`
If x is an array of pointers to strings, enter:	`? asciz *x[n]`

Note: When displaying a structure's value, the examine command also displays the names of each of the structure's elements.

exit

The `exit` command exits the IDE. The following example presents the syntax of the `exit` command:

```
exit
```

fillmem

The `fillmem` command fills a block of a specified memory space with the specified value. The functionality is similar to the Fill Memory command available from the context menu in the Memory window (see the Fill Memory section on page 334). The syntax of the `fillmem` command is:

```
fillmem SPACE="<displayed spacename>" FILLVALUE="<hexcadecimal
value>"
                     [STARTADDRESS="<hexadecimal address>"]
[ENDADDRESS="<hexadecimal address>"]
```

If `STARTADDRESS` and `ENDADDRESS` are not specified, all of the memory contents of a specified space are filled.

For example:

```
fillmem SPACE="ROM" VALUE="AA"
fillmem SPACE="ROM" VALUE="AA" STARTADDRESS="1000"
ENDADDRESS="2FFF"
```

go

The `go` command executes the program code from the current program counter until a breakpoint or, optionally, a symbol is encountered. This command starts a debug session if one has not been started. The `go` command can take one of the following forms:

- `go` resumes execution from the current location.
- `go` <*symbol*> resumes execution at the function identified by <*symbol*>. This version of the `go` command can only be used during a debug session.

The following list presents examples:

```
go
go myfunc
```

list bp

The `list bp` command displays a list of all of the current breakpoints of the active file. The following example presents the syntax of the `list bp` command:

```
list bp
```

loadmem

The `loadmem` command loads the data of an Intel hex file, a binary file, or a text file to a specified memory space at a specified address. The functionality is similar to the Load from File command available from the context menu in the Memory window (see Load a File into Memory – see page 335). The following example presents the syntax of the `loadmem` command:

```
loadmem SPACE="<displayed spacename>" FORMAT=<HEX | BIN |TEXT>
"<[PATH\]name>"
 [STARTADDRESS="<hexadecimal address>"]
```

If `STARTADDRESS` is not specified, the data is loaded at the memory lower address.

For example:

```
loadmem SPACE="RDATA" FORMAT=BIN "c:\temp\file.bin"
STARTADDRESS="20"
loadmem SPACE="ROM" FORMAT=HEX "c:\temp\file.hex"
loadmem SPACE="ROM" FORMAT=TEXT "c:\temp\file.txt"
STARTADDRESS="1000"
```

log

The `log` command manages the IDE's logging feature. The `log` command can take one of three forms:

- `log "<[path\]filename>" [APPEND]`

  sets the path and file name for the log file. If `APPEND` is not provided, an existing log file with the same name is truncated when the log is next activated.
- `log on` activates the logging of data
- `log off` deactivates the logging of data

For example:

```
log "buildall.log"
log on
log off
```

makfile or makefile

The `makfile` and `makefile` commands export a make file for the current project. The syntax can take one of two forms:

- `makfile "<[path\]file name>"`
- `makefile "<[path\]file name>"`

If *path* is not provided, the current working directory is used.

For example:

```
makfile "myproject.mak"
makefile "c:\projects\test.mak"
```

new project

The `new project` command creates a new project designated by *project_name*, *target*, and the type supplied. If the full path is not supplied, the current working directory is used. By default, existing projects with the same name are replaced. Use `NOREPLACE` to prevent the overwriting of existing projects. The syntax can take one of the following forms:

- `new project "<[path\]name>" "<target>" "<exe|lib>" ["<cpu>"] [NOREPLACE]`
- `new project "<[path\]name>" "<target>" "<project type>" "<exe|lib>" "<cpu>" [NOREPLACE]`

where:

- <`name`> is the path and name of the new project. If the path is not provided, the current working directory is assumed. Any file extension can be used, but none is required. If not provided, the default extension of `.zdsproj` is used.
- <`target`> *must* match that of the IDE (that is, the ZNEO IDE can only create ZNEO-based projects).
- <`exe`|`lib`> must be either `exe` (Executable) or `lib` (Static Library).
- ["<`cpu`>"] is the name of the CPU to configure for the new project.
- "<`project type`"> can be `"Standard"` or `"Assembly Only"`. `Standard` is the default.

- `NOREPLACE` is an optional parameter to use to prevent the overwriting of existing projects.

For example:

```
new project "test1.zdsproj" "ZNEO" "exe"
new project "test1.zdsproj" "ZNEO" "exe" NOREPLACE
```

open project

The `open project` command opens the project designated by *project_name*. If the full path is not supplied, the current working directory is used. The command fails if the specified project does not exist. The following example presents the syntax of the `open project` command:

```
open project "<project_name>"
```

For example:

```
open project "test1.zdsproj"
open project "c:\projects\test1.zdsproj"
```

option

The `option` command manipulates project settings for the active build configuration of the currently open project. Each call to `option` applies to a single tool but can set multiple options for the given tool. The following example presents the syntax for the `option` command:

```
option <tool_name> expr1 expr2 . . . exprN,
```

where:

expr is (<*option name*> = <*option value*>)

For example:

```
option assembler debug = TRUE
option compiler debug = TRUE keeplst = TRUE
option debugger readmem = TRUE
option linker igcase = "FALSE"
option linker code = 0000-FFFF
option general cpu=z8f64
```

Note: Many of these script file options are also available from the command line. For more details, see Running ZDSII from the Command Line on Running ZDSII from the Command Line – see page 349.

Table 28 lists some command line examples and the corresponding script file commands.

Table 28. Command Line Examples

Script File Command Examples	Corresponding Command Line Examples
option compiler keepasm = TRUE	eZ8cc -keepasm
`option compiler keepasm = FALSE`	`eZ8cc -nokeepasm`
`option compiler const = RAM`	`eZ8cc -const:RAM`
`option assembler debug = TRUE`	`eZ8asm -debug`
`option linker igcase = "FALSE"`	`eZ8link -NOigcase`
`option librarian warn = FALSE`	`eZ8lib -nowarn`

The following script file options are available:

- Assembler Options – see page 376
- Compiler Options – see page 377
- Debugger Options – see page 378
- General Options – see page 378
- Librarian Options – see page 378
- Linker Options – see page 379

Assembler Options

Table 29. Assembler Options

Option Name	Description or Corresponding Option in Project Settings Dialog Box	Acceptable Values
`define`	Assembler page, Defines field	string (separate multiple defines with semicolons)
`include`	Assembler page, Includes field	string (separate multiple paths with semicolons)
`list`	Assembler page, Generate Assembler Listing Files (.lst) checkbox	TRUE, FALSE
`listmac`	Assembler page, Expand Macros checkbox	TRUE, FALSE
`pagelen`	Assembler page, Page Length field	integer
`pagewidth`	Assembler page, Page Width field	integer

Table 29. Assembler Options (Continued)

Option Name	Description or Corresponding Option in Project Settings Dialog Box	Acceptable Values
`quiet`	Toggles quiet assemble.	TRUE, FALSE
sdiopt	Toggles Jump Optimization.	TRUE, FALSE

Compiler Options

Table 30. Compiler Options

Option Name	Description or Corresponding Option in Project Settings Dialog Box	Acceptable Values
`chartype`	Advanced page, Default Type of Char drop-down menu.	string (“unsigned” or “signed”)
`define`	Preprocessor page, Preprocessor Definitions field.	string (separate multiple defines with semicolons)
`genprintf`	Advanced page, Generate Assembly Source Code checkbox.	TRUE, FALSE
keepasm	Listing Files page, Preprocessor Definitions field.	
`keeplst`	Listing Files page, Generate Assembly Listing Files (.lst) checkbox.	TRUE, FALSE
`list`	Listing Files page, Generate C Listing Files (.lis) checkbox.	TRUE, FALSE
`listinc`	Listing Files page, With Include Files checkbox. Only applies if `list` option is currently true.	TRUE, FALSE
`model`	Code Generation page, Memory Model drop-down menu.	string (“large” or “small”)
`modsect`	Advanced page, Distinct Code Segment for Each Module checkbox.	TRUE, FALSE
`optspeed`	Toggles optimizing for speed.	TRUE (optimize for speed), FALSE (optimize for size)
`reduceopt`	Code Generation page, Limit Optimizations for Easier Debugging checkbox.	TRUE, FALSE
`regvar`	Advanced page, Use Register Variables checkbox.	TRUE, FALSE

Table 30. Compiler Options (Continued)

Option Name	Description or Corresponding Option in Project Settings Dialog Box	Acceptable Values
`stdinc`	Preprocessor page, Standard Include Path field.	string (separate multiple paths with semicolons)
`usrinc`	Preprocessor page, User Include Path field.	string (separate multiple paths with semicolons)

Debugger Options

For debugger options, use the `target help` and `debugtool help` commands.

General Options

Table 31. General Options

Option Name	Corresponding Option in Project Settings Dialog Box	Acceptable Values
`cpu`	General page, CPU drop-down field.	string (valid CPU name)
debug	General page, Generate Debug Information checkbox.	TRUE, FALSE
igcase	General page, Ignore Case of Symbols checkbox.	TRUE, FALSE
`outputdir`	General page, Intermediate Files Directory field.	string (path)
warn	General page, Show Warnings checkbox.	TRUE, FALSE

Librarian Options

Table 32. Librarian Options

Option Name	Corresponding Option in Project Settings Dialog Box	Acceptable Values
`outfile`	Librarian page, Output File Name field	string (library file name with option path)

Linker Options

Table 33. Linker Options

Option Name	Description or Corresponding Option in Project Settings Dialog Box	Acceptable Values
`createnew`	Commands page, Always Generate from Settings button.	TRUE, FALSE
`directives`	Commands page, Edit button, Additional Linker Directives dialog box. Contains the text for additional directives.	string
eram	Address Spaces page, Extended RAM (ERAM) field.	string (address range in the format “*<low>*-*<high>*”)
erom	Address Spaces page, Program Space (EROM).	string (address range in the format “*<low>*-*<high>*”)
`exeform`	Output page, Executable Formats area.	string (one or more of “IEEE 695” or “Intel Hex32”)
fplib	Objects and Libraries page, Floating Point Library drop-down menu.	string (“real”, “dummy”, or “none”)
`iodata`	Address Spaces page, SFRs and IO (IOData) field.	string (address range in the format “*<low>*-*<high>*”)
linkconfig	Commands page, Use Existing button, Select Linker Command File dialog box.	string (“All Internal” or “External Included”)
`linkctlfile`	Sets the linker command file (path and) name. The value is only used when `createnew` is set to `1`.	string
`map`	Output page, Generate Map File checkbox.	TRUE, FALSE
maxhexlen	Output page, Maximum Bytes per Hex File Line drop-down menu.	integer (16, 32, 64, or 128)
`objlibmods`	Objects and Libraries page, Additional Object/ Library Modules field.	string (separate multiple modules names with commas)
`of`	Output page, Output File Name field.	string (path and file name, excluding file extension)
padhex	Output page, Fill Unused Hex File Bytes with 0xFF checkbox.	TRUE, FALSE
ram	Address Spaces page, Internal RAM (RAM) field.	string (address range in the format “*<low>*-*<high>*”)
relist	Output page, Show Absolute Addresses in Assembly checkbox.	TRUE, FALSE

Table 33. Linker Options (Continued)

Option Name	Description or Corresponding Option in Project Settings Dialog Box	Acceptable Values
rom	Address Spaces page, Constant Data (ROM) field.	string (address range in the format "*<low>-<high>*")
sort	Output page, Sort Symbols By buttons.	string
startuptype	Objects and Libraries page, C Start-up Module area.	string ("standard" or "included")
undefisfatal	Warnings page, Treat Undefined Symbols as Fatal checkbox.	TRUE, FALSE
`usecrun`	Objects and Libraries page, Use C Runtime Library checkbox.	TRUE, FALSE
warnisfatal	Warnings page, Treat All Warnings as Fatal check-box.	TRUE, FALSE
`warnoverlap`	Warnings page, Warn on Segment Overlap check-box.	TRUE, FALSE

print

The `print` command writes formatted data to the Command Output window and the log (if the log is enabled). Each expression is evaluated, and the value is inserted into the *format_string*, which is equivalent to that supported by a C language `printf`. The following example presents the syntax of the `print` command:

```
print "<format_string>" expression1 expression2 ... expressionN
```

For example:

```
PRINT "the pc is %x" REG PC
print "pc: %x, sp: %x" REG PC REG SP
```

pwd

The pwd command retrieves the current working directory. The following example presents the syntax of the pwd command:

```
pwd
```

quit

The `quit` command ends the current debug session. The following example presents the syntax of the `quit` command:

```
quit
```

rebuild

The `rebuild` command rebuilds the currently open project. This command blocks the execution of other commands until the build process is complete. The following example presents the syntax of the `rebuild` command:

```
rebuild
```

reset

The `reset` command resets execution of program code to the beginning of the program. This command starts a debug session if one has not been started. The following example presents the syntax of the `reset` command:

```
reset
```

By default, the `reset` command resets the PC to symbol 'main'. If you deselect the **Reset to Symbol 'main' (Where Applicable)** checkbox on the **Debugger** tab of the **Options** dialog box (see Options—Debugger Tab – see page 102), the PC resets to the first line of the program.

savemem

The `savemem` command saves the memory content of the specified range into an Intel hex file, a binary file, or a text file. The functionality is similar to the Save to File command available from the context menu in the Memory window (see Save Memory to a File – see page 335). The following example presents the syntax of the `savemem` command:

```
savemem SPACE="<displayed spacename>" FORMAT=<HEX | BIN |TEXT>
"<[PATH\]name>"
 [STARTADDRESS="<hexadecimal address>"] [ENDADDRESS="<hexadecimal
address>"]
```

If `STARTADDRESS` and `ENDADDRESS` are not specified, all of the memory contents of a specified space are saved.

For example:

```
savemem SPACE="RDATA" FORMAT=BIN "c:\temp\file.bin"
STARTADDRESS="20" ENDADDRESS="100"
savemem SPACE="ROM" FORMAT=HEX "c:\temp\file.hex"
savemem SPACE="ROM" FORMAT=TEXT "c:\temp\file.txt"
STARTADDRESS="1000" ENDADDRESS="2FFF"
```

set config

The `set config` command activates an existing build configuration for or creates a new build configuration in the currently loaded project. The following example presents the syntax of the `set config` command:

`set config "`*`config_name`*`" ["`*`copy_from_config_name`*`"]`

The `set config` command performs the following functions:

- Activates *config_name* if it exists.
- Creates a new configuration named *config_name* if it does not yet exist. When complete, the new configuration is made active. When creating a new configuration, the Command Processor copies the initial settings from the *copy_from_config_name* parameter, if provided. If not provided, the active build configuration is used as the copy source. If *config_name* exists, the *copy_from_config_name* parameter is ignored.

Note: The active/selected configuration is used with commands such as `option tool name="value"` and `build`.

step

The `step` command performs a single step (stepover) from the current location of the program counter. If the count is not provided, a single step is performed. This command starts a debug session if one has not been started. The following example presents the syntax of the `step` command:

```
step
```

stepin

The `stepin` command steps into the function at the PC. If there is no function at the current PC, this command is equivalent to `step`. This command starts a debug session if one has not been started. The following example presents the syntax of the `stepin` command:

```
stepin
```

stepout

The `stepout` command steps out of the function. This command starts a debug session if one has not been started. The following example presents the syntax of the `stepout` command:

```
stepout
```

stop

The `stop` command stops the execution of program code. The following example presents the syntax of the `stop` command:

```
stop
```

target copy

The `target copy` command creates a copy of the existing target with a given name with the given new name. The syntax can take one of two forms:

- `target copy NAME="<new target name>"` creates a copy of the active target named the value given for `NAME`.
- `target copy NAME="<new target name>" SOURCE="<existing target name>"` creates a copy of the `SOURCE` target named the value given for `NAME`.

For example:

```
target copy NAME="mytarget" SOURCE="Sim3"
```

target create

The `target create` command creates a new target with the given name and using the given CPU. The following example presents the syntax of the `target create` command:

```
target create NAME="<target name>" CPU="<cpu name>"
```

For example:

```
target create NAME="mytarget" CPU="Z16F2811AL"
```

target get

The `target get` command displays the current value for the given data item for the active target. The following example presents the syntax of the `target get` command:

```
target get "<data item>"
```

Use the `target setup` command to view available data items and current values.

For example:

```
target get "cpu"
```

target help

The `target help` command displays all target commands. The following example presents the syntax of the `target help` command:

```
target help
```

target list

The `target list` command lists all available targets. The syntax can take one of three forms:

- `target list` displays the names of all available targets (restricted to the currently configured CPU family).
- `target list CPU="<cpu name>"` displays the names of all available targets associated with the given CPU name.
- `target list FAMILY="<family name>"` displays the names of all available targets associated with the given CPU family name.

For example:

```
target list FAMILY="ZNEO"
```

target options

Note: See a target in the following directory for a list of categories and options:

<*ZDS Installation Directory*>`\targets`

where <*ZDS Installation Directory*> is the directory in which Zilog Developer Studio was installed. By default, this filepath is:

`C:\Program Files\Zilog\ZDSII_ZNEO_<version>`,

where <*version*> can be `4.11.0` or `5.0.0`.

To set a target value, use one of the following syntaxes:

```
target options CATEGORY="<Category>" OPTION="<Option>" "<token
name>"="<value to set>"
target options CATEGORY="<Category>" "<token name>"="<value to
set>"
target options "<token name>"="<value to set>"
```

To select a target, use the following syntax:

```
target options NAME ="<Target Name>"
```

target save

The `target save` command saves a target. To save the selected target, use the following syntax:

```
target save
```

To save a specified target, use the following syntax:

```
target save NAME="<Target Name>"
```

For example:

```
target save Name="Sim3"
```

target set

The `target set` command sets the given data item to the given data value for the active target or activates a particular target. The syntax can take one of two forms:

- `target set "<data item>" "<new value>"` sets data item to new value for the active debug tool. Use `target setup` to view available data items and current values.

For example:

```
target set "frequency" "20000000"
```

- `target set "<target name>"` activates the target with the given name. Use `target list` to view available targets.

target setup

The `target setup` command displays the current configuration. The following example presents the syntax of the `target setup` command:

```
target setup
```

wait

The `wait` command instructs the Command Processor to wait the specified milliseconds before executing the next command. The following example presents the syntax of the `wait` command:

```
wait <milliseconds>
```

For example:

```
wait 5000
```

wait bp

The `wait bp` command instructs the Command Processor to wait until the debugger stops executing. The optional *max_milliseconds* parameter provides a method to limit the amount of time a wait takes (that is, wait until the debugger stops or *max_milliseconds* passes). The following example presents the syntax of the `wait bp` command:

```
wait bp [max_milliseconds]
```

For example:

```
wait bp
wait bp 2000
```

Running the Flash Loader from the Command Processor

You can run the Flash Loader from the Command field. Command Processor keywords have been added to allow for easy scripting of the Flash loading process. Each of the parameters is persistent, which allows for the repetition of the Flash and verification processes with a minimum amount of repeated key strokes.

Observe the following procedure to run the Flash Loader:

1. Create a project or open a project with a ZNEO microcontroller selected in the **CPU Family** and **CPU** fields of the **General** page of the **Project Settings** dialog box (see Project Settings—General Page – see page 47).
2. Set up the USB communication in the **Configure Target** dialog box (see Figure 61 on page 76).
3. In the **Command** field (in the **Command Processor** toolbar), enter one of the following command sequences to use the Flash Loader.

Displaying Flash Help

Flash Setup	Displays the Flash setup in the Command Output window
Flash Help	Displays the Flash command format in the Command Output window

Setting Up Flash Options

Flash Options "<*File Name*>"	File to be flashed
Flash Options OFFSET = "<*address*>"	Offset address in hex file
Flash Options NAUTO	Do not automatically select external Flash device
Flash Options AUTO	Automatically select external Flash device
Flash Options INTMEM	Set to internal memory
Flash Options EXTMEM	Set to external memory
Flash Options BOTHMEM	Set to both internal and external memory
Flash Options NEBF	Do not erase before flash
Flash Options EBF	Erase before flash
Flash Options NISN	Do not include serial number

Flash Options ISN	Include a serial number
Flash Options NPBF	Do not page-erase Flash memory; use mass erase
Flash Options PBF	Page-erase Flash memory
Flash Options SERIALADDRESS = "<*address*>"	Serial number address
Flash Options SERIALNUMBER = "<*Number in Hex*>"	Initial serial number value
Flash Options SERIALSIZE = <1-8>	Number of bytes in serial number
Flash Options INCREMENT = "<*Decimal value*>"	Increment value for serial number

Executing Flash Commands

Flash READSERIAL	Read the serial number
Flash READSERIAL REPEAT	Read the serial number and repeat
Flash BURNSERIAL	Program the serial number
Flash BURNSERIAL REPEAT	Program the serial number and repeat
Flash ERASE	Erase Flash memory
Flash ERASE REPEAT	Erase Flash memory and repeat
Flash BURN	Program Flash memory
Flash BURN REPEAT	Program Flash memory and repeat
Flash BURNVERIFY	Program and verify Flash memory
Flash BURNVERIFY REPEAT	Program and verify Flash memory and repeat
Flash VERIFY	Verify Flash memory
Flash VERIFY REPEAT	Verify Flash memory and repeat

Caution: The Flash Loader dialog box and the Command Processor interface use the same parameters. If an option is not specified with the Command Processor interface, the current setting in the Flash Loader dialog box is used. If a setting is changed in the Command Processor interface, the Flash Loader dialog box settings are changed.

Examples

The following are valid examples:

```
FLASH Options INTMEM
FLASH Options "c:\testing\test.hex"
FLASH BURN REPEAT
```

or

```
flash options intmem
flash options "c:\testing\test.hex"
flash burn repeat
```

The file `test.hex` is loaded into internal Flash memory. After the Flashing is completed, you are prompted to program an additional unit.

```
FLASH VERIFY
```

The file `test.hex` is verified against internal Flash memory.

```
FLASH SETUP
```

The current Flash Loader parameters settings are displayed in the Command Output window.

```
FLASH HELP
```

The current Flash Loader command options are displayed in the Command Output window.

```
Flash Options NAUTO
```

The Flash Loader does not automatically select the external Flash device.

```
Flash Options PBF
```

Page erase is enabled instead of mass erase for internal and external Flash programming.

Appendix C. C Standard Library

As described in the Run-Time Library section on page 177, the ZNEO C-Compiler provides a collection of run-time libraries. The largest section of these libraries consists of an implementation of much of the C Standard Library.

The ZNEO C-Compiler is a conforming freestanding 1989 ANSI C implementation with some exceptions. In accordance with the definition of a freestanding implementation, the compiler supports the required standard header files <`float.h`>, <`limits.h`>, <`stdarg.h`>, and <`stddef.h`>. It also supports additional standard header files and Zilog-specific nonstandard header files. The latter are described in the Run-Time Library section on page 177.

The standard header files and functions are, with minor exceptions, fully compliant with the ANSI C Standard. The deviations from the ANSI Standard in these files are summarized in Library Files Not Required for Freestanding Implementation – see page 196. The standard header files provided with the compiler are listed in Table 34 and described in detail in the Standard Header Files section on page 392. The following sections describe the use and format of the standard portions of the run-time libraries:

- Standard Header Files – see page 392
- Standard Functions – see page 407

Table 34. Standard Headers

Header	Description	Page
`<assert.h>`	Diagnostics	394
`<ctype.h>`	Character-handling functions	394
`<errno.h>`	Error numbers	393
`<float.h>`	Floating-point limits	396
`<limits.h>`	Integer limits	395
`<math.h>`	Math functions	398
`<setjmp.h>`	Nonlocal jump functions	401
`<stdarg.h>`	Variable arguments functions	401
`<stddef.h>`	Standard defines	393
`<stdio.h>`	Standard input/output functions	402
`<stdlib.h>`	General utilities functions	403
`<string.h>`	String-handling functions	405

Note: The standard include header files are located in the following directory:

<*ZDS Installation Directory*>`\include\std`

where <*ZDS Installation Directory*> is the directory in which Zilog Developer Studio was installed. By default, this would be `C:\Program Files\Zilog\ZDSII_ZNEO_<version>`, where <*version*> might be `4.11.0` or `5.0.0`.

Standard Header Files

Each library function is declared in a header file. The header files can be included in the source files using the `#include` preprocessor directive. The header file declares a set of related functions, any necessary types, and additional macros required to facilitate their use.

Header files can be included in any order; each can be included more than once in a given scope with no adverse effect. Header files must be included in the code before the first reference to any of the functions they declare or types and macros they define.

The following sections describe the standard header files:

- Errors <errno.h> – see page 393
- Standard Definitions <stddef.h> – see page 393
- Diagnostics <assert.h> – see page 394
- Character Handling <ctype.h> – see page 394
- Limits <limits.h> – see page 395
- Floating Point <float.h> – see page 396
- Mathematics <math.h> – see page 398
- Nonlocal Jumps <setjmp.h> – see page 401
- Variable Arguments <stdarg.h> – see page 401
- Input/Output <stdio.h> – see page 402
- General Utilities <stdlib.h> – see page 403
- String Handling <string.h> – see page 405

Errors <errno.h>

The <`errno.h`> header defines macros relating to the reporting of error conditions.

Macros

EDOM	Expands to a distinct nonzero integral constant expression.
ERANGE	Expands to a distinct nonzero integral constant expression.
errno	A modifiable value that has type int. Several libraries set errno to a positive value to indicate an error. errno is initialized to zero at program startup, but it is never set to zero by any library function. The value of errno can be set to nonzero by a library function even if there is no error, depending on the behavior specified for the library function in the ANSI Standard.

Additional macro definitions, beginning with E and an uppercase letter, can also be specified by the implementation.

Standard Definitions <stddef.h>

The following types and macros are defined in several headers referred to in the descriptions of the functions declared in that header, as well as the common <`stddef.h`> standard header.

Macros

NULL	Expands to a null pointer constant.
offsetof (type, identifier)	Expands to an integral constant expression that has type size_t and provides the offset in bytes, from the beginning of a structure designated by type to the member designated by identifier.

Types

ptrdiff_t	Signed integral type of the result of subtracting two pointers.
size_t	Unsigned integral type of the result of the sizeof operator.
wchar_t	Integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

Diagnostics <assert.h>

The <`assert.h`> header declares two macros.

Macros

NDEBUG	The <assert.h> header defines the assert() macro. It refers to the NDEBUG macro that is not defined in the header. If NDEBUG is defined as a macro name before the inclusion of this header, the assert() macro is defined simply as: #define assert(ignore)((void) 0)
assert(expression);	Tests the expression and, if false, prints the diagnostics including the expression, file name, and line number. Also calls exit with non-zero exit code if the expression is false.

Character Handling <ctype.h>

The <`ctype.h`> header declares several macros and functions useful for testing and mapping characters. In all cases, the argument is an `int`, the value of which is represented as an `unsigned char` or equals the value of the EOF macro. If the argument has any other value, the behavior is undefined.

Macros

TRUE	Expands to a constant 1.
FALSE	Expands to a constant 0.

Note: The above character-handling macros are nonstandard macros.

Functions

The functions in this section return nonzero (true) if, and only if, the value of the argument c conforms to that in the description of the function. The term *printing character* refers to a member of a set of characters, each of which occupies one printing position on a display device. The term *control character* refers to a member of a set of characters that are not printing characters.

Character Testing

int isalnum(int c);	Tests for alphanumeric character.
int isalpha(int c);	Tests for alphabetic character.
int iscntrl(int c);	Tests for control character.
int isdigit(int c);	Tests for decimal digit.
int isgraph(int c);	Tests for printable character except space.
int islower(int c);	Tests for lowercase character.
int isprint(int c);	Tests for printable character.
int ispunct(int c);	Tests for punctuation character.
int isspace(int c);	Tests for white-space character.
int isupper(int c);	Tests for uppercase character.
int isxdigit(int c);	Tests for hexadecimal digit.

Character Case Mapping

int tolower(int c);	Tests character and converts to lowercase if uppercase.
int toupper(int c);	Tests character and converts to uppercase if lowercase.

Limits <limits.h>

The <`limits.h`> header defines macros that expand to various limits and parameters.

Macros

CHAR_BIT	Maximum number of bits for smallest object that is not a bit-field (byte).
CHAR_MAX	Maximum value for an object of type `char`.
CHAR_MIN	Minimum value for an object of type `char`.
INT_MAX	Maximum value for an object of type `int`.
INT_MIN	Minimum value for an object of type `int`.
LONG_MAX	Maximum value for an object of type long `int`.
LONG_MIN	Minimum value for an object of type long `int`.
SCHAR_MAX	Maximum value for an object of type `signed char`.
SCHAR_MIN	Minimum value for an object of type `signed char`.

SHRT_MAX	Maximum value for an object of type `short int`.
SHRT_MIN	Minimum value for an object of type `short int`.
UCHAR_MAX	Maximum value for an object of type `unsigned char`.
UINT_MAX	Maximum value for an object of type `unsigned int`.
ULONG_MAX	Maximum value for an object of type `unsigned long int`.
USHRT_MAX	Maximum value for an object of type `unsigned short int`.
MB_LEN_MAX	Maximum number of bytes in a multibyte character.

If the value of an object of type char sign-extends when used in an expression, the value of CHAR_MIN is the same as that of SCHAR_MIN, and the value of CHAR_MAX is the same as that of SCHAR_MAX. If the value of an object of type char does not sign-extend when used in an expression, the value of CHAR_MIN is 0, and the value of CHAR_MAX is the same as that of UCHAR_MAX.

Floating Point <float.h>

The <`float.h`> header defines macros that expand to various limits and parameters.

Macros

DBL_DIG	Number of decimal digits of precision.
DBL_MANT_DIG	Number of base-FLT_RADIX digits in the floating-point mantissa.
DBL_MAX	Maximum represented floating-point numbers.
DBL_MAX_EXP	Maximum integer such that FLT_RADIX raised to that power approximates a floating-point number in the range of represented numbers.
DBL_MAX_10_EXP	Maximum integer such that 10 raised to that power approximates a floating-point number in the range of represented value ((int)log10(DBL_MAX), and so on).
DBL_MIN	Minimum represented positive floating-point numbers.
DBL_MIN_EXP	Minimum negative integer such that FLT_RADIX raised to that power approximates a positive floating-point number in the range of represented numbers.
DBL_MIN_10_EXP	Minimum negative integer such that 10 raised to that power approximates a positive floating-point number in the range of represented values `((int)log10(DBL_MIN)`, and so on).
FLT_DIG	Number of decimal digits of precision.
FLT_MANT_DIG	Number of base-FLT_RADIX digits in the floating-point mantissa.
FLT_MAX	Maximum represented floating-point numbers.

FLT_MAX_EXP	Maximum integer such that FLT_RADIX raised to that power approximates a floating-point number in the range of represented numbers.
FLT_MAX_10_EXP	Maximum integer such that 10 raised to that power approximates a floating-point number in the range of represented value `((int)log10(FLT_MAX)`, and so on).
FLT_MIN	Minimum represented positive floating-point numbers.
FLT_MIN_EXP	Minimum negative integer such that FLT_RADIX raised to that power approximates a positive floating-point number in the range of represented numbers
FLT_MIN_10_EXP	Minimum negative integer such that 10 raised to that power approximates a positive floating-point number in the range of represented values `((int)log10(FLT_MIN)`, and so on).
FLT_RADIX	Radix of exponent representation.
FLT_ROUND	Rounding mode for floating-point addition. -1 indeterminable 0 toward zero 1 to nearest 2 toward positive infinity 3 toward negative infinity
LDBL_DIG	Number of decimal digits of precision.
LDBL_MANT_DIG	Number of base-FLT_RADIX digits in the floating-point mantissa.
LDBL_MAX	Maximum represented floating-point numbers.
LDBL_MAX_EXP	Maximum integer such that FLT_RADIX raised to that power approximates a floating-point number in the range of represented numbers.
LDBL_MAX_10_EXP	Maximum integer such that 10 raised to that power approximates a floating-point number in the range of represented value `((int)log10(LDBL_MAX)`, and so on).
LDBL_MIN	Minimum represented positive floating-point numbers.
LDBL_MIN_EXP	Minimum negative integer such that FLT_RADIX raised to that power approximates a positive floating-point number in the range of represented numbers.
LDBL_MIN_10_EXP	Minimum negative integer such that 10 raised to that power approximates a positive floating-point number in the range of represented values `((int)log10(LDBL_MIN)`, and so on).

Note: The limits for the double and long double data types are the same as that for the float data type for the ZNEO C-Compiler.

Mathematics <math.h>

The <`math.h`> header declares several mathematical functions and defines one macro. The functions take double-precision arguments and return double-precision values. Integer arithmetic functions and conversion functions are discussed later.

Note: The double data type is implemented as float in the ZNEO C-Compiler.

Macro

HUGE_VAL	Expands to a positive double expression, not necessarily represented as a float.

Treatment of Error Conditions

The behavior of each of these functions is defined for all values of its arguments. Each function must return as if it were a single operation, without generating any externally visible exceptions.

For all functions, a domain error occurs if an input argument to the function is outside the domain over which the function is defined. On a domain error, the function returns a specified value; the integer expression errno acquires the value of the EDOM macro.

Similarly, a range error occurs if the result of the function cannot be represented as a double value. If the result overflows (the magnitude of the result is so large that it cannot be represented in an object of the specified type), the function returns the value of the HUGE_VAL macro, with the same sign as the correct value of the function; the integer expression errno acquires the value of the ERANGE macro. If the result underflows (the magnitude of the result is so small that it cannot be represented in an object of the specified type), the function returns zero.

Functions

Trigonometric

double acos(double x);	Calculates arc cosine of x.
double asin(double x)	Calculates arc sine of x.
double atan(double x);	Calculates arc tangent of x.
double atan2(double y, double x);	Calculates arc tangent of y/x.
double cos(double x);	Calculates cosine of x.

double sin(double x);	Calculates sine of x.
double tan(double x);	Calculates tangent of x.

The following additional trigonometric functions are provided:

float acosf(float x);	Calculates arc cosine of x.
float asinf(float x);	Calculates arc sine of x.
float atanf(float x);	Calculates arc tangent of x.
float atan2f(float y, float x);	Calculates arc tangent of y/x.
float cosf(float x);	Calculates cosine of x.
float sinf(float x);	Calculates sine of x.
float tanf(float x);	Calculates tangent of x.

Hyperbolic

double cosh(double x);	Calculates hyperbolic cosine of x.
double sinh(double x);	Calculates hyperbolic sine of x.
double tanh(double x);	Calculates hyperbolic tangent of x.

The following additional hyperbolic functions are provided:

float coshf(float x);	Calculates hyperbolic cosine of x.
float sinhf(float x);	Calculates hyperbolic sine of x.
float tanhf(float x);	Calculates hyperbolic tangent of x.

Exponential

double exp(double x);	Calculates exponential function of x.
double frexp(double value, int *exp);	Shows x as product of mantissa (the value returned by frexp) and 2 to the n.
double ldexp(double x, int exp);	Calculates x times 2 to the exp.

The following additional exponential functions are provided:

float expf(float x);	Calculates exponential function of x.
float frexpf(float value, int *exp);	Shows x as product of mantissa (the value returned by frexp) and 2 to the n.
float ldexpf(float x, int exp);	Calculates x times 2 to the exp.

Logarithmic

double log(double x);	Calculates natural logarithm of x.
double log10(double x);	Calculates base 10 logarithm of x.
double modf(double value, double *iptr);	Breaks down x into integer (the value returned by modf) and fractional (n) parts.

The following additional logarithmic functions are provided:

float logf(float x);	Calculates natural logarithm of x.
float log10f(float x);	Calculates base 10 logarithm of x.
float modff(float value, float *iptr);	Breaks down x into integer (the value returned by modf) and fractional (n) parts.

Power

double pow(double x, double y);	Calculates x to the y.
double sqrt(double x);	Finds square root of x.

The following additional power functions are provided:

float powf(float x, float y);	Calculates x to the y.
float sqrtf(float x);	Finds square root of x.

Nearest Integer

double ceil(double x);	Finds integer ceiling of x.
double fabs(double x);	Finds absolute value of x.
double floor(double x);	Finds largest integer less than or equal to x.
double fmod(double x,double y);	Finds floating-point remainder of x/y.

The following additional nearest integer functions are provided:

float ceilf(float x);	Finds integer ceiling of x.
float fabsf(float x);	Finds absolute value of x.
float floorf(float x);	Finds largest integer less than or equal to x.
float fmodf(float x,float y);	Finds floating-point remainder of x/y.

Nonlocal Jumps <setjmp.h>

The <`setjmp.h`> header declares two functions and one type for bypassing the normal function call and return discipline.

Type

jmp_buf	An array type suitable for holding the information required to restore a calling environment.

Functions

int setjmp(jmp_buf env);	Saves a stack environment.
void longjmp(jmp_buf env, int val);	Restores a saved stack environment.

Variable Arguments <stdarg.h>

The <`stdarg.h`> header declares a type and a function and defines two macros for advancing through a list of arguments whose number and types are not known to the called function when it is translated.

A function can be called with a variable number of arguments of varying types. A *Function Definitions* parameter list contains one or more parameters. The rightmost parameter plays a special role in the access mechanism and is designated parmN in this description.

Type

va_list	An array type suitable for holding information required by the macro va_arg and the function va_end. The called function declares a variable (referred to as ap in this section) having type va_list. The variable ap can be passed as an argument to another function.

Variable Argument List Access Macros and Function

The va_start and va_arg macros described in this section are implemented as macros, not as real functions. If `#undef` is used to remove a macro definition and obtain access to a real function, the behavior is undefined.

Functions

void va_start(va_list ap, parmN);	Sets pointer to beginning of argument list.
type va_arg (va_list ap, type);	Retrieves argument from list.
void va_end(va_list ap);	Resets pointer.

Input/Output <stdio.h>

The `<stdio.h>` header declares input and output functions.

Macro

EOF	Expands to a negative integral constant. Returned by functions to indicate end of file.

Functions

Formatted Input/Output

int printf(const char *format, ...);	Writes formatted data to stdout.
int scanf(const char *format, ...);	Reads formatted data from stdin.
int sprintf(char *s, const char *format, ...);	Writes formatted data to string.
int sscanf(const char *s, const char *format, ...);	Reads formatted data from string.
int vprintf(const char *format, va_list arg);	Writes formatted data to a stdout.
int vsprintf(char *s, const char *format, va_list arg);	Writes formatted data to a string.

Character Input/Output

int getchar(void);	Reads a character from stdin.
char *gets(char *s);	Reads a line from stdin.
int putchar(int c);	Writes a character to stdout.
int puts(const char *s);	Writes a line to stdout.

General Utilities <stdlib.h>

The `<stdlib.h>` header declares several types, functions of general utility, and macros.

Types

div_t	Structure type that is the type of the value returned by the div function.
ldiv_t	Structure type that is the type of the value returned by the ldiv function.
size_t	Unsigned integral type of the result of the sizeof operator.
wchar_t	Integral type whose range of values can represent distinct codes for all members of the largest extended character set specified among the supported locales.

Macros

EDOM	Expands to distinct nonzero integral constant expressions.
ERANGE	Expands to distinct nonzero integral constant expressions.
EXIT_SUCCESS	Expands to integral expression which indicates successful termination status.
EXIT_FAILURE	Expands to integral expression which indicates unsuccessful termination status.
HUGE_VAL	Expands to a positive double expression, not necessarily represented as a float.
NULL	Expands to a null pointer constant.
RAND_MAX	Expands to an integral constant expression, the value of which is the maximum value returned by the rand function.

Functions

String Conversion

The `atof`, `atoi`, and `atol` functions do not affect the value of the errno macro on an error. If the result cannot be represented, the behavior is undefined.

double atof(const char *nptr);	Converts string to double.
int atoi(const char *nptr);	Converts string to int.
long int atol(const char *nptr);	Converts string to long.

double strtod(const char *nptr, char **endptr);	Converts string pointed to by nptr to a double.
long int strtol(const char *nptr, char **endptr, int base);	Converts string to a long decimal integer that is equal to a number with the specified radix.

The following additional string conversion functions are provided:

float atoff(const char *nptr);	Converts string to float.
float strtof(const char *nptr, char **endptr);	Converts string pointed to by nptr to a double.

Pseudorandom Sequence Generation

int rand(void)	Gets a pseudorandom number.
void srand(unsigned int seed);	Initializes pseudorandom series.

Memory Management

The order and contiguity of storage allocated by successive calls to the `calloc`, `malloc`, and `realloc` functions are unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it can be assigned to a pointer to any type of object and then used to access such an object in the space allocated (until the space is explicitly freed or reallocated).

void *calloc(size_t nmemb, size_t size);	Allocates storage for array.
void free(void *ptr);	Frees a block allocated with calloc, malloc, or realloc.
void *malloc(size_t size);	Allocates a block.
void *realloc(void *ptr, size_t size);	Reallocates a block.

Searching and Sorting Utilities

void *bsearch(void *key, void *base, size_t nmemb, size_t size, int (*compar)(void *, void *));	Performs binary search.
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(void *, void *));	Performs a quick sort.

Integer Arithmetic

int abs(int j);	Finds absolute value of integer value.
div_t div(int numer, int denom);	Computes integer quotient and remainder.
long int labs(long int j);	Finds absolute value of long integer value.
ldiv_t ldiv(long int numer, long int denom);	Computes long quotient and remainder.

String Handling <string.h>

The `<string.h>` header declares several functions useful for manipulating character arrays and other objects treated as character arrays. Various methods are used for determining the lengths of arrays, but in all cases a char* or void* argument points to the initial (lowest addressed) character of the array. If an array is written beyond the end of an object, the behavior is undefined.

Type

size_t	Unsigned integral type of the result of the sizeof operator.

Macro

NULL	Expands to a null pointer constant.

Functions

Copying

void *memcpy(void *s1, const void *s2, size_t n);	Copies a specified number of characters from one buffer to another.
void *memmove(void *s1, const void *s2, size_t n);	Moves a specified number of characters from one buffer to another.
char *strcpy(char *s1, const char *s2);	Copies one string to another.
char *strncpy(char *s1, const char *s2, size_t n);	Copies n characters of one string to another.

Concatenation

char *strcat(char *s1, const char *s2);	Appends a string.
char *strncat(char *s1, const char *s2, size_t n);	Appends n characters of string.

Comparison

The sign of the value returned by the comparison functions is determined by the sign of the difference between the values of the first pair of characters that differ in the objects being compared.

int memcmp(const void *s1, const void *s2, size_t n);	Compares the first n characters.
int strcmp(const char *s1, const char *s2);	Compares two strings.
int strncmp(const char *s1, const char *s2, size_t n);	Compares n characters of two strings.

Search

void *memchr(const void *s, int c, size_t n);	Returns a pointer to the first occurrence, within a specified number of characters, of a given character in the buffer.
char *strchr(const char *s, int c);	Finds first occurrence of a given character in string.
size_t strcspn(const char *s1, const char *s2);	Finds first occurrence of a character from a given character in string.
char *strpbrk(const char *s1, const char *s2);	Finds first occurrence of a character from one string to another.
char *strrchr(const char *s, int c);	Finds last occurrence of a given character in string.
size_t strspn(const char *s1, const char *s2);	Finds first substring from a given character set in string.
char *strstr(const char *s1, const char *s2);	Finds first occurrence of a given string in another string.
char *strtok(char *s1, const char *s2);	Finds next token in string.

Miscellaneous

void *memset(void *s, int c, size_t n);	Uses a given character to initialize a specified number of bytes in the buffer.
size_t strlen(const char *s);	Finds length of string.

Standard Functions

The following functions are standard functions:

abs	acos, acosf	asin, asinf	assert	atan, atanf
atan2, atan2f	atof, atoff	atoi	atol	bsearch
calloc	ceil, ceilf	cos, cosf	cosh, coshf	div
exp, expf	fabs, fabsf	floor, floorf	fmod, fmodf	free
frexp, frexpf	getchar	gets	isalnum	isalpha
iscntrl	isdigit	isgraph	islower	isprint
ispunct	isspace	isupper	isxdigit	labs
ldexp, ldexpf	ldiv	log, logf	log10, log10f	longjmp
malloc	memchr	memcmp	memcpy	memmove
memset	modf, modff	pow, powf	printf	putchar
puts	qsort	rand	realloc	scanf
setjmp	sin, sinf	sinh, sinhf	sprintf	sqrt, sqrtf
srand	sscanf	strcat	strchr	strcmp
strcpy	strcspn	strlen	strncat	strncmp
strncpy	strpbrk	strrchr	strspn	strstr
strtod, strtof	strtok	strtol	tan, tanf	tanh, tanhf
tolower	toupper	va_arg	va_end	va_start
vprintf	vsprintf			

abs

Computes the absolute value of an integer `j`. If the result cannot be represented, the behavior is undefined.

Synopsis

```
#include <stdlib.h>
int abs(int j);
```

Returns

The absolute value.

Example

```
int I=-5632;
int j;
j=abs(I);
```

acos, acosf

Computes the principal value of the arc cosine of `x`. A domain error occurs for arguments not in the range [-1,+1].

Synopsis

```
#include <math.h>
double acos(double x);
float acosf(float x);
```

Returns

The arc cosine in the range [0, pi].

Example

```
double y=0.5635;
double x;
x=acos(y)
```

asin, asinf

Computes the principal value of the arc sine of `x`. A domain error occurs for arguments not in the range [-1,+1].

Synopsis

```
#include <math.h>
double asin(double x);
float asinf(float x);
```

Returns

The arc sine in the range [-pi/2,+pi/2].

Example

```
double y=.1234;
double x;
x = asin(y);
```

assert

Puts diagnostics into programs. When it is executed, if `expression` is false (that is, evaluates to zero), the `assert` macro writes information about the particular call that failed (including the text of the argument, the name of the source file, and the source line number—the latter are respectively the values of the preprocessing macros `__FILE__` and `__LINE__`) on the serial port using the `printf`() function. It then loops forever.

Synopsis

```
#include <assert.h>
void assert(int expression);
```

Returns

If *expression* is true (that is, evaluates to nonzero), the `assert` macro returns no value.

Example

```
#include <assert.h>

char str[] = "COMPASS";

void main(void)
{
 assert(str[0] == 'B');
}
```

atan, atanf

Computes the principal value of the arc tangent of *x*.

Synopsis

```
#include <math.h>
double atan(double x);
float atanf(float x);
```

Returns

The arc tangent in the range (-pi/2, +pi/2).

Example

```
double y=.1234;
double x;
x=atan(y);
```

atan2, atan2f

Computes the principal value of the arc tangent of *y*/*x*, using the signs of both arguments to determine the quadrant of the return value. A domain error occurs if both arguments are zero.

Synopsis

```
#include <math.h>
double atan2(double y, double x);
float atan2f(float y, float x);
```

Returns

The arc tangent of *y*/*x*, in the range [-pi, +pi].

Example

```
double y=.1234;
double x=.4321;
double z;
z=atan2(y,x);
```

atof, atoff

Converts the string pointed to by nptr to double representation. Except for the behavior on error, `atof` is equivalent to `strtod (nptr, (char **)NULL)`, and `atoff` is equivalent to `strtof (nptr, (char **)NULL)`.

Synopsis

```
#include <stdlib.h>
double atof(const char *nptr);
float atoff(const char *nptr);
```

Returns

The converted value.

Example

```
char str []="1.234";
double x;
x= atof(str);
```

atoi

Converts the string pointed to by nptr to int representation. Except for the behavior on error, it is equivalent to `(int)strtol(nptr, (char **)NULL, 10)`.

Synopsis

```
#include <stdlib.h>
int atoi(const char *nptr);
```

Returns

The converted value.

Example

```
char str []="50";
int x;
x=atoi(str);
```

atol

Converts the string pointed to by nptr to long int representation. Except for the behavior on error, it is equivalent to `strtol(nptr, (char **)NULL, 10)`.

Synopsis

```
#include <stdlib.h>
long int atol(const char *nptr);
```

Returns

The converted value.

Example

```
char str[]="1234567";
long int x;
x=atol(str);
```

bsearch

Searches an array of nmemb objects, the initial member of which is pointed to by base, for a member that matches the object pointed to by key. The size of each object is specified by size.

The array has been previously sorted in ascending order according to a comparison function pointed to by `compar`, which is called with two arguments that point to the objects being compared. The `compar` function returns an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

Synopsis

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base, size_t nmemb,
size_t size, int (*compar)(const void *, const void *));
```

Returns

A pointer to the matching member of the array or a null pointer, if no match is found.

Example

```
#include <stdlib.h>
int list[]={2,5,8,9};
int k=8;

int compare (const void * x, const void * y);
int main(void)
{
 int *result;
 result = bsearch(&k, list, 4, sizeof(int), compare);
}

int compare (const void * x, const void * y)
{
 int a = *(int *) x;
 int b = *(int *) y;
 if (a < b) return -1;
 if (a == b)return 0;
 return 1;
}
```

The `compare` function prototype is, as shown in the preceding example:

```
int compare (const void * x, const void * y);
```

calloc

Allocates space for an array of nmemb objects, each of whose size is `size`. The space is initialized to all bits zero.

Synopsis

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

Returns

A pointer to the start (lowest byte address) of the allocated space. If the space cannot be allocated, or if nmemb or `size` is zero, the `calloc` function returns a null pointer.

Example

```
char *buf;
buf = (char*)calloc(40, sizeof(char));
if (buf != NULL)
/*success*/
else
/*fail*/
```

ceil, ceilf

Computes the smallest integer not less than x.

Synopsis

```
#include <math.h>
double ceil(double x);
float ceilf(float x);
```

Returns

The smallest integer not less than x, expressed as a `double` for `ceil` and expressed as a `float` for `ceilf`.

Example

```
double y=1.45;
double x;
x=ceil(y);
```

cos, cosf

Computes the cosine of x (measured in radians). A large magnitude argument can yield a result with little or no significance.

Synopsis

```
#include <math.h>
double cos(double x);
float cosf(float x);
```

Returns

The cosine value.

Example

```
double y=.1234;
double x;
x=cos(y);
```

cosh, coshf

Computes the hyperbolic cosine of x. A range error occurs if the magnitude of x is too large.

Synopsis

```
#include <math.h>
double cosh(double x);
float coshf(float x);
```

Returns

The hyperbolic cosine value.

Example

```
double y=.1234;
double x
x=cosh(y);
```

div

Computes the quotient and remainder of the division of the numerator `numer` by the denominator `denom`. If the division is inexact, the sign of the quotient is that of the mathematical quotient, and the magnitude of the quotient is the largest integer less than the magnitude of the mathematical quotient.

Synopsis

```
#include <stdlib.h>
div_t div(int numer, int denom);
```

Returns

A structure of type div_t, comprising both the quotient and the remainder. The structure contains the following members, in either order:

```
int quot;           /* quotient */
int rem;            /* remainder */
```

Example

```
int x=25;
int y=3;
div_t t;
int q;
int r;
t=div (x,y);
q=t.quot;
r=t.rem;
```

exp, expf

Computes the exponential function of x. A range error occurs if the magnitude of x is too large.

Synopsis

```
#include <math.h>
double exp(double x);
float expf(float x);
```

Returns

The exponential value.

Example

```
double y=.1234;
double x;
x=exp(y);
```

fabs, fabsf

Computes the absolute value of a floating-point number `x`.

Synopsis

```
#include <math.h>
double fabs(double x);
float fabsf(float x);
```

Returns

The absolute value of x.

Example

```
double y=6.23;
double x;
x=fabs(y);
```

floor, floorf

Computes the largest integer not greater than x.

Synopsis

```
#include <math.h>
double floor(double x);
float floorf(float x);
```

Returns

The largest integer not greater than x, expressed as a `double` for `floor` and expressed as a `float` for `floorf`.

Example

```
double y=6.23;
double x;
x=floor(y);
```

fmod, fmodf

Computes the floating-point remainder of `x/y`. If the quotient of `x/y` cannot be represented, the behavior is undefined.

Synopsis

```
#include <math.h>
double fmod(double x, double y);
float fmodf(float x, float y);
```

Returns

The value of x if y is zero. Otherwise, it returns the value f, which has the same sign as x, such that x - i * y + f for some integer i, where the magnitude of f is less than the magnitude of y.

Example

```
double y=7.23;
double x=2.31;
double z;
z=fmod(y,x);
```

free

Causes the space pointed to by ptr to be deallocated; that is, made available for further allocation. If ptr is a null pointer, no action occurs. Otherwise, if the argument does not match a pointer earlier returned by the `calloc`, `malloc`, or `realloc` function, or if the space has been deallocated by a call to `free` or `realloc`, the behavior is undefined. If freed space is referenced, the behavior is undefined.

Synopsis

```
#include <stdlib.h>
void free(void *ptr);
```

Example

```
char *buf;
buf=(char*) calloc(40, sizeof(char));
free(buf);
```

frexp, frexpf

Breaks a floating-point number into a normalized fraction and an integral power of 2. It stores the integer in the `int` object pointed to by `exp`.

Synopsis

```
#include <math.h>
double frexp(double value, int *exp);
float frexpf(float value, int *exp);
```

Returns

The value x, such that x is a `double` (`frexp`) or `float` (`frexpf`) with magnitude in the interval [1/2, 1] or zero, and value equals x times 2 raised to the power *exp. If value is zero, both parts of the result are zero.

Example

```
double y, x=16.4;
int n;
y=frexp(x,&n);
```

getchar

Waits for the next character to appear at the serial port and return its value.

Synopsis

```
#include <stdio.h>
int getchar(void);
```

Returns

The next character from the input stream pointed to by stdin. If the stream is at end-of-file, the end-of-file indicator for the stream is set, and `getchar` returns EOF. If a read error occurs, the error indicator for the stream is set, and `getchar` returns EOF.

Example

```
int i;
i=getchar();
```

Note: The UART must be initialized using the Zilog `init_uart()` function. See the init_uart command on page 182.

gets

Reads characters from the input stream into the array pointed to by `s`, until end-of-file is encountered or a new-line character is read. The new-line character is discarded, and a null character is written immediately after the last character read into the array.

Synopsis

```
#include <stdio.h>
char *gets(char *s);
```

Returns

The value of `s`, if successful. If a read error occurs during the operation, the array contents are indeterminate, and a null pointer is returned.

Example

```
char *r;
char buf [80];
r=gets(buf);
if (r==NULL)
 /*No input*/
```

Note: The UART must be initialized using the Zilog `init_uart()` function. See the init_uart command on page 182.

isalnum

Tests for any character for which `isalpha` or `isdigit` is true.

Synopsis

```
include <ctype.h>
int isalnum(int c);
```

Example

```
int r;
char c='a';
r=isalnum(c);
```

isalpha

Tests for any character for which `isupper` or `islower` is true.

Synopsis

```
#include <ctype.h>
int isalpha(int c);
```

Example

```
int r;
char c='a';
r=isalpha(c);
```

iscntrl

Tests for any control character.

Synopsis

```
#include <ctype.h>
int iscntrl(int c);
```

Example

```
int r;
char c=NULL;
r=iscntrl(c);
```

isdigit

Tests for any decimal digit.

Synopsis

```
#include <ctype.h>
int isdigit(int c);
```

Example

```
int r;
char c='4';
r=isdigit(c);
```

isgraph

Tests for any printing character except space (' ').

Synopsis

```
#include <ctype.h>
int isgraph(int c);
```

Example

```
int r;
char c='';
r=isgraph(c);
```

islower

Tests for any lowercase letter 'a' to 'z'.

Synopsis

```
#include <ctype.h>
int islower(int c);
```

Example

```
int r;
char c='a';
r=islower(c);
```

isprint

Tests for any printing character including space (' ').

Synopsis

```
#include <ctype.h>
int isprint(int c);
```

Example

```
int r;
char c='1';
r=isprint(c);
```

ispunct

Tests for any printing character except space (' ') or a character for which `isalnum` is true.

Synopsis

```
#include <ctype.h>
int ispunct(int c);
```

Example

```
int r;
char c='a';
r=ispunct(c);
```

isspace

Tests for the following white-space characters: space (' '), form feed ('\f'), new line ('\n'), carriage return ('\r'), horizontal tab ('\t'), or vertical tab ('\v').

Synopsis

```
#include <ctype.h>
int isspace(int c);
```

Example

```
int r;
char c='';
r=isspace(c);
```

isupper

Tests for any uppercase letter 'A' to 'Z'.

Synopsis

```
#include <ctype.h>
int isupper(int c);
```

Example

```
int r;
char c='a';
r=isupper(c);
```

isxdigit

Tests for any hexadecimal digit '0' to '9' and 'A' to 'F'.

Synopsis

```
#include <ctype.h>
int isxdigit(int c);
```

Example

```
int r;
char c='f';
r=isxdigit(c);
```

labs

Computes the absolute value of a long int j.

Synopsis

```
#include <stdlib.h>
long labs(long j);
```

Example

```
long i=-193250;
long j
j=labs(i);
```

ldexp, ldexpf

Multiplies a floating-point number by an integral power of 2. A range error can occur.

Synopsis

```
#include <math.h>
double ldexp(double x, int exp);
float ldexpf(float x, int exp);
```

Returns

The value of x times 2 raised to the power of exp.

Example

```
double x=1.235
int exp=2;
double y;
y=ldexp(x,exp);
```

ldiv

Computes the quotient and remainder of the division of the numerator `numer` by the denominator `denom`. If the division is inexact, the sign of the quotient is that of the mathematical quotient, and the magnitude of the quotient is the largest integer less than the magnitude of the mathematical quotient.

Synopsis

```
#include <stdlib.h>
ldiv_t ldiv(long numer, long denom);
```

Example

```
long x=25000;
long y=300;
ldiv_t t;
long q;
long r;
t=ldiv(x,y);
q=t.quot;
r=t.rem;
```

log, logf

Computes the natural logarithm of x. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

Synopsis

```
#include <math.h>
double log(double x);
float logf(float x);
```

Returns

The natural logarithm.

Example

```
double x=2.56;
double y;
y=log(x);
```

log10, log10f

Computes the base-ten logarithm of x. A domain error occurs if the argument is negative. A range error occurs if the argument is zero.

Synopsis

```
#include <math.h>
double log10(double x);
float log10f(float x);
```

Returns

The base-ten logarithm.

Example

```
double x=2.56;
double y;
y=log10(x);
```

longjmp

Restores the environment saved by the most recent call to `setjmp` in the same invocation of the program, with the corresponding jmp_buf argument. If there has been no such call, or if the function containing the call to `setjmp` has executed a `return` statement in the interim, the behavior is undefined.

All accessible objects have values as of the time `longjmp` was called, except that the values of objects of automatic storage class that do not have `volatile` type and have been changed between the `setjmp` and `longjmp` call are indeterminate.

As it bypasses the usual function call and returns mechanisms, the `longjmp` function executes correctly in contexts of interrupts, signals, and any of their associated functions. However, if the `longjmp` function is invoked from a nested signal handler (that is, from a function invoked as a result of a signal raised during the handling of another signal), the behavior is undefined.

Synopsis

```
#include <setjmp.h>
void longjmp(jmp_buf env, int val);
```

Returns

After `longjmp` is completed, program execution continues as if the corresponding call to `setjmp` had just returned the value specified by `val`. The `longjmp` function cannot cause `setjmp` to return the value 0; if val is 0, `setjmp` returns the value 1.

Example

```
int i;
jmp_buf env;
i=setjmp(env);
longjmp(env,i);
```

malloc

Allocates space for an object whose size is specified by size.

Note: The existing implementation of `malloc()` depends on the heap area being located from the bottom of the heap (referred to by the symbol __heapbot) to the top of the stack (SP). Care must be taken to avoid holes in this memory range. Otherwise, the `malloc()` function might not be able to allocate a valid memory object.

Synopsis

```
#include <stdlib.h>
void *malloc(size_t size);
```

Returns

A pointer to the start (lowest byte address) of the allocated space. If the space cannot be allocated, or if size is zero, the `malloc` function returns a null pointer.

Example

```
char *buf;
buf=(char *) malloc(40*sizeof(char));
if(buf !=NULL)
 /*success*/
else
 /*fail*/
```

memchr

Locates the first occurrence of c (converted to an `unsigned char`) in the initial n characters of the object pointed to by s.

Synopsis

```
#include <string.h>
void *memchr(const void *s, int c, size_t n);
```

Returns

A pointer to the located character or a null pointer if the character does not occur in the object.

Example

```
char *p1;
char str[]="COMPASS";
c='p';
p1=memchr(str,c,sizeof(char));
```

memcmp

Compares the first n characters of the object pointed to by s2 to the object pointed to by s1.

Synopsis

```
#include <string.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Returns

An integer greater than, equal to, or less than zero, according as the object pointed to by s1 is greater than, equal to, or less than the object pointed to by s2.

Example

```
char s1[]="COMPASS";
char s2[]="IDE";
int res;
res=memcmp(s1, s2, sizeof (char));
```

memcpy

Copies n characters from the object pointed to by s2 into the object pointed to by s1. If the two regions overlap, the behavior is undefined.

Synopsis

```
#include <string.h>
void *memcpy(void *s1, const void *s2, size_t n);
```

Returns

The value of s1.

Example

```
char s1[10];
char s2[10] = "COMPASS";
memcpy(s1, s2, 8);
```

memmove

Moves n characters from the object pointed to by s2 into the object pointed to by s1. Copying between objects that overlap takes place correctly.

Synopsis

```
#include <string.h>
void *memmove(void *s1, const void *s2, size_t n);
```

Returns

The value of s1.

Example

```
char s1[10];
char s2[]="COMPASS";
memmove(s1, s2, 8*sizeof(char));
```

memset

Copies the value of c (converted to an `unsigned char`) into each of the first n characters of the object pointed to by s.

Synopsis

```
#include <string.h>
void *memset(void *s, int c, size_t n);
```

Returns

The value of s.

Example

```
char str[20];
char c='a';
memset(str, c, 10*sizeof(char));
```

modf, modff

Breaks the argument value into integral and fractional parts, each of which has the same sign as the argument. It stores the integral part as a `double` (`modf`) or `float` (`modff`) in the object pointed to by iptr.

Synopsis

```
#include <math.h>
double modf(double value, double *iptr);
float modff(float value, float *iptr);
```

Returns

The signed fractional part of value.

Example

```
double x=1.235;
double f;
double i;
i=modf(x, &f);
```

pow, powf

Computes the x raised to the power of y. A domain error occurs if x is zero and y is less than or equal to zero, or if x is negative and y is not an integer. A range error can occur.

Synopsis

```
#include <math.h>
double pow(double x, double y);
float powf(float x, float y);
```

Returns

The value of x raised to the power y.

Example

```
double x=2.0;
double y=3.0;
double res;
res=pow(x,y);
```

printf

Writes output to the stream pointed to by stdout, under control of the string pointed to by format that specifies how subsequent arguments are converted for output.

A format string contains two types of objects: plain characters, which are copied unchanged to stdout, and conversion specifications, each of which fetch zero or more subsequent arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored. The `printf` function returns when the end of the format string is encountered.

Each conversion specification is introduced by the character "`%`". After this `%` character, the following events occur in sequence:

1. Zero or more flags that modify the meaning of the conversion specification.
2. An optional decimal integer specifying a minimum field width. If the converted value has fewer characters than the field width, it is padded on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The padding is with spaces unless the field width integer starts with a zero, in which case the padding is with zeros.
3. An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x, and X conversions, the number of digits to appear after the decimal point for e, E, and f conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of characters to be written from a string in s conversion. The precision takes the form of a period (.) followed by an optional decimal integer; if the integer is omitted, it is treated as zero. The amount of padding specified by the precision overrides that specified by the field width.
4. An optional h specifies that a following d, i, o, u, x, or X conversion character applies to a `short_int` or `unsigned_short_int` argument (the argument has been promoted according to the integral promotions, and its value is converted to `short_int` or `unsigned_short_int` before printing). An optional l (ell) specifies that a following d, i, o, u, x or X conversion character applies to a `long_int` or `unsigned_long_int` argument. An optional L specifies that a following e, E, f, g, or

G conversion character applies to a `long_double` argument. If an h, l, or L appears with any other conversion character, it is ignored.

5. A character that specifies the type of conversion to be applied.
6. A field width or precision, or both, can be indicated by an asterisk (`*`) instead of a digit string. In this case, an int argument supplies the files width or precision. The arguments specifying field width or precision displays before the argument (if any) to be converted. A negative field width argument is taken as a - flag followed by a positive field width. A negative precision argument is taken as if it were missing.

Note: For more specific information about the flag characters and conversion characters for the `printf` function, see the printf Flag Characters section on page 431.

Synopsis

```
#include <stdio.h>
int printf(const char *format, ...);
```

Returns

The number of characters transmitted or a negative value if an output error occurred.

Example

```
int i=10;
printf("This is %d",i);
```

Note: The UART must be initialized using the Zilog `init_uart()` function. See the init_uart command on page 182.

printf Flag Characters

–	The result of the conversion is left-justified within the field.
+	The result of a signed conversion always begins with a plus or a minus sign.

space	If the first character of a signed conversion is not a sign, a space is added before the result. If the space and + flags both appear, the space flag is ignored
#	The result is to be converted to an "alternate form". For c, d, i, s, and u conversions, the flag has no effect. For o conversion, it increases the precision to force the first digit of the result to be a zero. For x (or X) conversion, a nonzero result always contains a decimal point, even if no digits follow the point (normally, a decimal point appears in the result of these conversions only if a digit follows it). For g and G conversions, trailing zeros are not removed from the result, as they normally are.

printf Conversion Characters

d,i,o,u,x,X	The int argument is converted to signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X); the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.
f	The double argument is converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is explicitly zero, no decimal point appears. If a decimal point appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.
e,E	The double argument is converted in the style [-]d.ddde+dd, where there is one digit before the decimal point and the number of digits after it is equal to the precision; when the precision is missing, six digits are produced; if the precision is zero, no decimal point appears. The value is rounded to the appropriate number of digits. The E conversion character produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits. However, if the magnitude to be converted is greater than or equal to lE+100, additional exponent digits are written as necessary.
g,G	The double argument is converted in style f or e (or in style E in the case of a G conversion character), with the precision specifying the number of significant digits. The style used depends on the value converted; style e is used only if the exponent resulting from the conversion is less than -4 or greater than the precision. Trailing zeros are removed from the result; a decimal point appears only if it is followed by a digit.
c	The int argument is converted to an unsigned char, and the resulting character is written.

s	The argument is taken to be a (const char *) pointer to a string. Characters from the string are written up to, but not including, the terminating null character, or until the number of characters indicated by the precision are written. If the precision is missing it is taken to be arbitrarily large, so all characters before the first null character are written.
p	The argument is taken to be a (const void) pointer to an object. The value of the pointer is converted to a sequence of hex digits.
n	The argument is taken to be an (int) pointer to an integer into which is written the number of characters written to the output stream so far by this call to `printf`. No argument is converted.
%	A % is written. No argument is converted.

In no case does a nonexistent or small field width cause truncation of a field. If the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

putchar

Writes a character to the serial port.

Synopsis

```
#include <stdio.h>
int putchar(int c);
```

Returns

The character written. If a write error occurs, `putchar` returns EOF.

Example

```
int i;
charc='a';
i=putchar(c);
```

Note: The UART must be initialized using the Zilog `init_uart()` function. See the init_uart command on page 182.

puts

Writes the string pointed to by s to the serial port and appends a new-line character to the output. The terminating null character is not written.

Synopsis

```
#include <stdio.h>
int puts(char *s);
```

Returns

EOF if an error occurs; otherwise, it is a non-negative value.

Example

```
int i;
char strp[]="COMPASS";
i=puts(str);
```

Note: The UART must be initialized using the Zilog `init_uart()` function. See init_uart – see page 182.

qsort

Sorts an array of nmemb objects, the initial member of which is pointed to by any base. The size of each object is specified by size.

The array is sorted in ascending order according to a comparison function pointed to by `compar`, which is called with two arguments that point to the objects being compared. The `compar` function returns an integer less than, equal to, or greater than zero if the first argument is considered to be respectively less than, equal to, or greater than the second.

If two members in the array compare as equal, their order in the sorted array is unspecified.

Synopsis

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size, int
(*compar)(const void *, const void *));
```

Example

```
int lst[]={5,8,2,9};
int compare (const void * x, const void * y);
qsort (lst, sizeof(int), 4, compare);

int compare (const void * x, const void * y)
{
 int a = *(int *) x;
 int b = *(int *) y;
 if (a < b) return -1;
 if (a == b)return 0;
 return 1;
}
```

The `compare` function prototype is, as shown in the preceding example:

```
int compare (const void * x, const void * y);
```

rand

Computes a sequence of pseudorandom integers in the range 0 to RAND_MAX.

Synopsis

```
#include <stdlib.h>
int rand(void);
```

Returns

A pseudorandom integer.

Example

```
int i;
srand(1001);
i=rand();
```

realloc

Changes the size of the object pointed to by ptr to the size specified by size. The contents of the object are unchanged up to the lesser of the new and old sizes. If ptr is a null pointer, the `realloc` function behaves in a similar fashion to the `malloc` function for the specified size. Otherwise, if ptr does not match a pointer earlier returned by the `calloc`, `malloc`, or `realloc` function, or if the space has been deallocated by a call to the `free` or `realloc` function, the behavior is undefined. If the space cannot be allocated, the `realloc` function returns a null pointer and the object pointed to by ptr is unchanged. If size is zero, the `realloc` function returns a null pointer and, if ptr is not a null pointer, the object it points to is freed.

Synopsis

```
#include <stdlib.h>
void *realloc(void *ptr, size_t size);
```

Returns

Returns a pointer to the start (lowest byte address) of the possibly moved object.

Example

```
char *buf;
buf=(char *) malloc(40*sizeof(char));
buf=(char *) realloc(buf, 80*sizeof(char));
if(buf !=NULL)
 /*success*/
else
 /*fail*/
```

scanf

Reads input from the stream pointed to by stdin, under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the object to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but otherwise ignored.

The format is composed of zero or more directives from the following list:

- one or more white-space characters
- an ordinary character (not `%`)
- a conversion specification

Each conversion specification is introduced by the “%” character. After this % character, the following items appear in sequence:

1. An optional assignment-suppressing character (`*`).
2. An optional decimal integer that specifies the maximum field width.
3. An optional h, l or L indicating the size of the receiving object. The conversion characters d, l, n, o, and x can be preceded by h to indicate that the corresponding argument is a pointer to `short_int` rather than a pointer to int, or by l to indicate that it is

a pointer to `long_int`. Similarly, the conversion character u can be preceded by h to indicate that the corresponding argument is a pointer to `unsigned_short_int` rather than a pointer to `unsigned_int`, or by l to indicate that it is a pointer to `unsigned_long_int`. Finally, the conversion character e, f, and g can be preceded by l to indicate that the corresponding argument is a pointer to double rather than a pointer to float, or by L to indicate a pointer to `long_double`. If an h, l, or L appears with any other conversion character, it is ignored.

4. A character that specifies the type of conversion to be applied. The valid conversion characters are described in the following paragraphs.

The `scanf` function executes each directive of the format in turn. If a directive fails, as detailed below, the `scanf` function returns. Failures are described as input failures (due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white space is executed by reading input up to the first non-white-space character (which remains unread), or until no more characters can be read. A white-space directive fails if no white-space character can be found.

A directive that is an ordinary character is executed by reading the next character of the stream. If the character differs from the one comprising the directive, the directive fails, and the character remains unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each character. A conversion specification is executed in the following steps:

1. Input white-space characters (as specified by the `isspace` function) are skipped, unless the specification includes a ’[’, ’c,’ or ’n’ character.
2. An input item is read from the stream, unless the specification includes an n character. An input item is defined as the longest sequence of input characters (up to any specified maximum field width) which is an initial subsequence of a matching sequence. The first character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless an error prevented input from the stream, in which case it is an input failure.
3. Except in the case of a % character, the input item (or, in the case of a `%n` directive, the count of input characters) is converted to a type appropriate to the conversion character. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behavior is undefined.

Note: See the next section, *scanf Conversion Characters*, for valid input information.

Synopsis

```
#include <stdio.h>
int scanf(const char *format, ...);
```

Returns

The value of the macro EOF if an input failure occurs before any conversion. Otherwise, the `scanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early conflict between an input character and the format.

Examples

```
int i
scanf("%d", &i);
```

The following example reads in two values. `var1` is an `unsigned char` with two decimal digits, and `var2` is a `float` with three decimal place precision.

```
scanf("%2d,%f",&var1,&var2);
```

scanf Conversion Characters

d	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the strtol function with the value 10 for the base argument. The corresponding argument is a pointer to integer.
i	Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the strtol function with the value 0 for the base argument. The corresponding argument is a pointer to integer.
o	Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the strtol function with the value 8 for the base argument. The corresponding argument is a pointer to integer.
u	Matches an unsigned decimal integer, whose format is the same as expected for the subject sequence of the strtol function with the value 10 for the base argument. The corresponding argument is a pointer to unsigned integer.
x	Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the strtol function with the value of 16 for the base argument. The corresponding argument is a pointer to integer.
e,f,g	Matches an optionally signed floating-point number, whose format is the same as expected for the subject string of the strtod function. The corresponding argument is a pointer to floating.

s	Matches a sequence of non-white-space characters. The corresponding argument is a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which is added automatically.
[	Matches a sequence of expected characters (the scanset). The corresponding argument is a pointer to the initial character of an array large enough to accept the sequence and a terminating null character, which is added automatically. The conversion character includes all subsequent characters is the format string, up to and including the matching right bracket ( ]). The characters between the brackets (the scanlist) comprise the scanset, unless the character after the left bracket is a circumflex (^), in which case the scanset contains all characters that do not appear in the scanlist between the circumflex and the right bracket. As a special case, if the conversion character begins with [] or [^], the right bracket character is in the scanlist and next right bracket character is the matching right bracket that ends the specification. If a - character is in the scanlist and is neither the first nor the last character, the behavior is indeterminate.
c	Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). The corresponding argument is a pointer to the initial character of an array large enough to accept the sequence. No null character is added.
p	Matches a hexadecimal number. The corresponding argument is a pointer to a pointer to void.
n	No input is consumed. The corresponding argument is a pointer to integer into which is to be written the number of characters read from the input stream so far by this call to the scanf function. Execution of a %n directive does not increment the assignment count returned at the completion of execution of the `scanf` function.
%	Matches a single %; no conversion or assignment occurs.

If a conversion specification is invalid, the behavior is undefined.

The conversion characters `e`, `g` and `x` can be capitalized. However, the use of upper case is ignored.

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (if any) is terminated with an input failure.

If conversion terminates on a conflicting input character, the offending input character is left unread in the input stream. Trailing white space (including new-line characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than using the `%n` directive.

setjmp

Saves its calling environment in its jmp_buf argument, for later use by the `longjmp` function.

Synopsis

```
#include<setjmp.h>
int setjmp(jmp_buf env);
```

Returns

If the return is from a direct invocation, the `setjmp` function returns the value zero. If the return is from a call to the `longjmp` function, the `setjmp` function returns a nonzero value.

Example

```
int i;
jmp_buf env;
i=setjmp(env);
longjmp(env, i);
```

sin, sinf

Computes the sine of x (measured in radians). A large magnitude argument can yield a result with little or no significance.

Synopsis

```
#include <math.h>
double sin(double x);
float sinf(float x);
```

Returns

The sine value.

Example

```
double x=1.24;
double y;
y=sin(x);
```

sinh, sinhf

Computes the hyperbolic sine of x. A range error occurs if the magnitude of x is too large.

Synopsis

```
#include <math.h>
double sinh(double x);
float sinhf(float x);
```

Returns

The hyperbolic sine value.

Example

```
double x=1.24;
double y;
y=sinh(x);
```

sprintf

The `sprintf` function is equivalent to `printf`, except that the argument s specifies an array into which the generated output is to be written, rather than to a stream. A null character is written at the end of the characters written; it is not counted as part of the returned sum.

Synopsis

```
#include <stdio.h>
int sprintf(char *s, const char *format, ...);
```

Returns

The number of characters written in the array, not counting the terminating null character.

Example

```
int d=51;
char buf [40];
sprintf(buf,"COMPASS/%d",d);
```

sqrt, sqrtf

Computes the non-negative square root of x. A domain error occurs if the argument is negative.

Synopsis

```
#include <math.h>
double sqrt(double x);
float sqrtf(float x);
```

Returns

The value of the square root.

Example

```
double x=25.0;
double y;
y=sqrt(x);
```

srand

Uses the argument as a seed for a new sequence of pseudorandom numbers to be returned by subsequent calls to `rand`. If `srand` is then called with the same seed value, the sequence of pseudorandom numbers is repeated. If `rand` is called before any calls to `srand` have been made, the same sequence is generated as when `srand` is first called with a seed value of 1.

Synopsis

```
#include <stdlib.h>
void srand(unsigned int seed);
```

Example

```
int i;
srand(1001);
i=rand();
```

sscanf

Reads formatted data from a string.

Synopsis

```
#include <stdio.h>
int sscanf(const char *s, const char *format, ...);
```

Returns

The value of the macro EOF if an input failure occurs before any conversion. Otherwise, the `sscanf` function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early conflict between an input character and the format.

Example

```
char buf [80];
int i;
sscanf(buf,"%d",&i);
```

strcat

Appends a copy of the string pointed to by s2 (including the terminating null character) to the end of the string pointed to by s1. The initial character of s2 overwrites the null character at the end of s1.

Synopsis

```
#include <string.h>
char *strcat(char *s1, const char *s2);
```

Returns

The value of s1.

Example

```
char *ptr;
char s1[80]="Production";
char s2[]="Languages";
ptr=strcat(s1,s2);
```

strchr

Locates the first occurrence of c (converted to a `char`) in the string pointed to by s. The terminating null character is considered to be part of the string.

Synopsis

```
#include <string.h>
char *strchr(const char *s, int c);
```

Returns

A pointer to the located character or a null pointer if the character does not occur in the string.

Example

```
char *ptr;
char str[]="COMPASS";
ptr=strchr(str,'p');
```

strcmp

Compares the string pointed to by s1 to the string pointed to by s2.

Synopsis

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Returns

An integer greater than, equal to, or less than zero, according as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.

Example

```
char s1[]="Production";
char s2[]="Programming";
int res;
res=strcmp(s1,s2);
```

strcpy

Copies the string pointed to by s2 (including the terminating null character) into the array pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.

Synopsis

```
#include <string.h>
char *strcpy(char *s1, const char *s2);
```

Returns

The value of s1.

Example

```
char s1[80], *s2;
s2=strcpy(s1,"Production");
```

strcspn

Computes the length of the initial segment of the string pointed to by s1 that consists entirely of characters not from the string pointed to by s2. The terminating null character is not considered part of s2.

Synopsis

```
#include <string.h>
size_t strcspn(const char *s1, const char *s2);
```

Returns

The length of the segment.

Example

```
size_t pos;
char s1[]="xyzabc";
char s2[]="abc";
pos=strcspn(s1,s2);
```

strlen

Computes the length of the string pointed to by s.

Synopsis

```
#include <string.h>
size_t strlen(const char *s);
```

Returns

The number of characters that precede the terminating null character.

Example

```
char s1[]="COMPASS";
size_t i;
i=strlen(s1);
```

strncat

Appends no more than n characters of the string pointed to by s2 (not including the terminating null character) to the end of the string pointed to by s1. The initial character of s2 overwrites the null character at the end of s1. A terminating null character is always appended to the result.

Synopsis

```
#include <string.h>
char *strncat(char *s1, const char *s2, size_t n);
```

Returns

The value of s1.

Example

```
char *ptr;
char strl[80]="Production";
char str2[]="Languages";
ptr=strncat(str1,str2,4);
```

strncmp

Compares no more than n characters from the string pointed to by s1 to the string pointed to by s2.

Synopsis

```
#include <string.h>
int strncmp(const char *s1, const char *s2, size_t n);
```

Returns

An integer greater than, equal to, or less than zero, according as the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2.

Example

```
char s1[]="Production";
char s2[]="Programming";
int res;
res=strncmp(s1,s2,3);
```

strncpy

Copies not more than n characters from the string pointed to by s2 to the array pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined.

If the string pointed to by s2 is shorter than n characters, null characters are appended to the copy in the array pointed to by s1, until n characters in all have been written.

Synopsis

```
#include <string.h>
char *strncpy(char *s1, const char *s2, size_t n);
```

Returns

The value of s1.

Example

```
char *ptr;
char s1[40]="Production";
char s2[]="Languages";
ptr=strncpy(s1,s2,4);
```

strpbrk

Locates the first occurrence in the string pointed to by s1 of any character from the string pointed to by s2.

Synopsis

```
#include <string.h>
char *strpbrk(const char *s1, const char *s2);
```

Returns

A pointer to the character, or a null pointer if no character from s2 occurs in s1.

Example

```
char *ptr;
char s1[]="COMPASS";
char s2[]="PASS";
ptr=strpbrk(s1,s2);
```

strrchr

Locates the last occurrence of c (converted to a `char`) in the string pointed to by s. The terminating null character is considered to be part of the string.

Synopsis

```
#include <string.h>
char *strrchr(const char *s, int c);
```

Returns

A pointer to the character, or a null pointer if c does not occur in the string.

Example

```
char *ptr;
char s1[]="COMPASS";
ptr=strrchr(s1,'p');
```

strspn

Finds the first substring from a given character set in a string.

Synopsis

```
#include <string.h>
size_t strspn(const char *s1, const char *s2);
```

Returns

The length of the segment.

Example

```
char s1[]="cabbage";
char s2[]="abc";
size_t res,
res=strspn(s1,s2);
```

strstr

Locates the first occurrence of the string pointed to by s2 in the string pointed to by s1.

Synopsis

```
#include <string.h>
char *strstr(const char *s1, const char *s2);
```

Returns

A pointer to the located string or a null pointer if the string is not found.

Example

```
char *ptr;
char s1[]="Production Languages";
char s2[]="Lang";
ptr=strstr(s1,s2);
```

strtod, strtof

Converts the string pointed to by nptr to `double` (`strtod`) or `float` (`strtof`) representation. The function recognizes an optional leading sequence of white-space characters (as specified by the `isspace` function), then an optional plus or minus sign, then a sequence of digits optionally containing a decimal point, then an optional letter e or E followed by an optionally signed integer, then an optional floating suffix. If an inappropriate character occurs before the first digit following the e or E, the exponent is taken to be zero.

The first inappropriate character ends the conversion. If endptr is not a null pointer, a pointer to that character is stored in the object endptr points to; if an inappropriate character occurs before any digit, the value of nptr is stored.

The sequence of characters from the first digit or the decimal point (whichever occurs first) to the character before the first inappropriate character is interpreted as a floating constant according to the rules of this section, except that if neither an exponent part or a decimal point appears, a decimal point is assumed to follow the last digit in the string. If a minus sign appears immediately before the first digit, the value resulting from the conversion is negated.

Synopsis

```
#include <stdlib.h>
double strtod(const char *nptr, char **endptr);
float strtof(const char *nptr, char **endptr);
```

Returns

The converted value, or zero if an inappropriate character occurs before any digit. If the correct value would cause overflow, plus or minus HUGE_VAL is returned (according to the sign of the value), and the macro `errno` acquires the value ERANGE. If the correct value causes underflow, zero is returned and the macro `errno` acquires the value ERANGE.

Example

```
char *ptr;
char s[]="0.1456";
double res;
res=strtod(s,&ptr);
```

strtok

A sequence of calls to the `strtok` function breaks the string pointed to by s1 into a sequence of tokens, each of which is delimited by a character from the string pointed to by s2. The first call in the sequence has s1 as its first argument, and is followed by calls with a null pointer as their first argument. The separator string pointed to by s2 can be different from call to call.

The first call in the sequence searches s1 for the first character that is not contained in the current separator string s2. If no such character is found, there are no tokens in s1, and the `strtok` function returns a null pointer. If such a character is found, it is the start of the first token.

The `strtok` function then searches from there for a character that is contained in the current separator string. If no such character is found, the current token extends to the end of the string pointed to by s1, and subsequent searches for a token fail. If such a character is found, it is overwritten by a null character, which terminates the current token. The `strtok` function saves a pointer to the following character, from which the next search for a token starts.

Each subsequent call, with a null pointer as the value of the first argument, starts searching from the saved pointer and behaves as described in the preceding paragraphs.

Synopsis

```
#include <string.h>
char *strtok(char *s1, const char *s2);
```

Returns

A pointer to the first character of a token or a null pointer if there is no token.

Example

```
#include <string.h>
static char str[] = "?a???b, , ,#c";
char *t;
```

```
t = strtok(str,"?"); /* t points to the token "a" */
t = strtok(NULL,","); /* t points to the token   "??b " */
t = strtok(NULL,"#,"); /* t points to the token "c" */
t = strtok(NULL,"?"); /* t is a null pointer */
```

strtol

Converts the string pointed to by nptr to long int representation. The function recognizes an optional leading sequence of white-space characters (as specified by the isspace function), then an optional plus or minus sign, then a sequence of digits and letters, then an optional integer suffix.

The first inappropriate character ends the conversion. If endptr is not a null pointer, a pointer to that character is stored in the object endptr points to; if an inappropriate character occurs before the first digit or recognized letter, the value of nptr is stored.

If the value of base is 0, the sequence of characters from the first digit to the character before the first inappropriate character is interpreted as an integer constant according to the rules of this section. If a minus sign appears immediately before the first digit, the value resulting from the conversion is negated.

If the value of base is between 2 and 36, it is used as the base for conversion. Letters from a (or A) through z (or Z) are ascribed the values 10 to 35; a letter whose value is greater than or equal to the value of base ends the conversion. Leading zeros after the optional sign are ignored, and leading 0x or 0X is ignored if the value of base is 16. If a minus sign appears immediately before the first digit or letter, the value resulting from the conversion is negated.

Synopsis

```
#include <stdlib.h>
long strtol(const char *nptr, char **endptr, int base);
```

Returns

The converted value, or zero if an inappropriate character occurs before the first digit or recognized letter. If the correct value would cause overflow, LONG_MAX or LONG_MIN is returned (according to the sign of the value), and the macro `errno` acquires the value ERANGE.

Example

```
char *ptr;
char s[]="12345";
long res;
res=strtol(s,&ptr,10);
```

tan, tanf

The tangent of x (measured in radians). A large magnitude argument can yield a result with little or no significance.

Synopsis

```
#include <math.h>
double tan(double x);
float tanf(float x);
```

Returns

The tangent value.

Example

```
double x=2.22;
double y;
y=tan(x);
```

tanh, tanhf

Computes the hyperbolic tangent of x.

Synopsis

```
#include <math.h>
double tanh(double x);
float tanhf(float x);
```

Returns

The hyperbolic tangent of x.

Example

```
double x=2.22;
double y;
y=tanh(x);
```

tolower

Converts an uppercase letter to the corresponding lowercase letter.

Synopsis

```
#include <ctype.h>
int tolower(int c);
```

Returns

If the argument is an uppercase letter, the `tolower` function returns the corresponding lowercase letter, if any; otherwise, the argument is returned unchanged.

Example

```
char c='A';
int i;
i=tolower(c);
```

toupper

Converts a lowercase letter to the corresponding uppercase letter.

Synopsis

```
#include <ctype.h>
int toupper(int c);
```

Returns

If the argument is a lowercase letter, the `toupper` function returns the corresponding uppercase letter, if any; otherwise, the argument is returned unchanged.

Example

```
char c='a';
int i;
i=toupper(c);
```

va_arg

Expands to an expression that has the type and value of the next argument in the call. The parameter `ap` is the same as the va_list ap initialized by `va_start`. Each invocation of `va_arg` modifies `ap` so that successive arguments are returned in turn. The parameter type is a type name such that the type of a pointer to an object that has the specified type can be obtained simply by fixing a * to type. If type disagrees with the type of the actual next argument (as promoted, according to the default argument conversions, into `int`, unsigned int, or double), the behavior is undefined.

Synopsis

```
#include <stdarg.h>
type va_arg(va_list ap, type);
```

Returns

The first invocation of the `va_arg` macro after that of the `va_start` macro returns the value of the argument after that specified by parmN. Successive invocations return the values of the remaining arguments in succession.

Example

The function f1 gathers into an array a list of arguments that are pointers to strings (but not more than MAXARGS arguments), then passes the array as a single argument to function f2. The number of pointers is specified by the first argument to f1.

```
#include <stdarg.h>
extern void f2(int n, char *array[]);
#define MAXARGS 31
void f1(int n_ptrs,...) {
 va_list ap;
 char *array[MAXARGS];
 int ptr_no = 0;

 if (n_ptrs > MAXARGS)
                   n_ptrs = MAXARGS;
 va_start(ap, n_ptrs);
 while (ptr_no < n_ptrs)
  array[ptr_no++] = va_arg(ap, char *);
 va_end(ap);
 f2(n_ptrs, array);
}
```

Each call to f1 has in scope the definition of the function of a declaration such as `void f1(int, ...);`

va_end

Facilitates a normal return from the function whose variable argument list was referenced by the expansion of `va_start` that initialized the `va_list ap`. The `va_end` function can modify `ap` so that it is no longer usable (without an intervening invocation of `va_start`). If the `va_end` function is not invoked before the return, the behavior is undefined.

Synopsis

```
#include <stdarg.h>
void va_end(va_list ap);
```

Example

The function f1 gathers into an array a list of arguments that are pointers to strings (but not more than MAXARGS arguments), then passes the array as a single argument to function f2. The number of pointers is specified by the first argument to f1.

```
#include <stdarg.h>
extern void f2(int n, char *array[]);
#define MAXARGS 31
void f1(int n_ptrs,...) {
 va_list ap;
 char *array[MAXARGS];
 int ptr_no = 0;

 if (n_ptrs > MAXARGS)
                   n_ptrs = MAXARGS;
 va_start(ap, n_ptrs);
 while (ptr_no < n_ptrs)
  array[ptr_no++] = va_arg(ap, char *);
 va_end(ap);
 f2(n_ptrs, array);
}
```

Each call to f1 has in scope the definition of the function of a declaration such as `void f1(int, ...);`

va_start

Is executed before any access to the unnamed arguments.

The parameter `ap` points to an object that has type `va_list`. The parameter `parmN` is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the , ...). The `va_start` macro initializes `ap` for subsequent use by `va_arg` and `va_end`.

Synopsis

```
#include <stdarg.h>
void va_start(va_list ap, parmN);
```

Example

The function f1 gathers into an array a list of arguments that are pointers to strings (but not more than MAXARGS arguments), then passes the array as a single argument to function f2. The number of pointers is specified by the first argument to f1.

```
#include <stdarg.h>
extern void f2(int n, char *array[]);
#define MAXARGS 31
void f1(int n_ptrs,...) {
 va_list ap;
 char *array[MAXARGS];
 int ptr_no = 0;

 if (n_ptrs > MAXARGS)
  n_ptrs = MAXARGS;
 va_start(ap, n_ptrs);
 while (ptr_no < n_ptrs)
  array[ptr_no++] = va_arg(ap, char *);
 va_end(ap);
 f2(n_ptrs, array);
}
```

Each call to f1 has in scope the definition of the function of a declaration such as `void f1(int, ...);`

vprintf

Equivalent to `printf`, with the variable argument list replaced by arg, which has been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vprintf` function does not invoke the `va_end` function.

Synopsis

```
#include <stdarg.h>
#include <stdio.h>
int vprintf(const char *format, va_list arg);
```

Returns

The number of characters transmitted or a negative value if an output error occurred.

Example

```
va_list va;
/* initialize the variable argument va here */
vprintf("%d %d %d",va);
```

vsprintf

Equivalent to `sprintf`, with the variable argument list replaced by arg, which has been initialized by the `va_start` macro (and possibly subsequent `va_arg` calls). The `vsprintf` function does not invoke the va_end function.

Synopsis

```
#include <stdarg.h>
#include <stdio.h>
int vsprintf(char *s, const char *format, va_list arg);
```

Returns

The number of characters written in the array, not counting the terminating null character.

Example

```
va_list va;
char buf[80];
/*initialize the variable argument va here*/
vsprint(buf, "%d %d %d",va);
```

Glossary

A

ABS. Absolute Value.

A/D. Analog-to-Digital—the conversion of an analog signal, such as a waveform, to a digital signal, represented by binary data. See ADC.

ADC. Analog-to-Digital Converter—a circuit that converts an analog signal to a digital bit stream. See A/D.

address space. The physical or logical area of the target system's memory map. The memory map could be physically partitioned into ROM to store code, and RAM for data. The memory can also be divided logically to form separate areas for code and data storage.

ALU. See Arithmetic Logical Unit.

American National Standards Institute (ANSI). The U.S. standards organization that establishes procedures for the development and coordination of voluntary American National Standards.

analog. From the word *analogous*, meaning *similar to*. The signal being transmitted can be represented in a way similar to the original signal. For example, a telephone signal can be seen on an oscilloscope as a sine wave similar to the voice signal being carried through the phone line.

analog signal. A signal that exhibits a continuous nature rather than a pulsed or discrete nature.

AND. A bitwise AND instruction.

ANSI. American National Standards Institute.

application program interface (API). A formalized set of software calls and routines that can be referenced by an application program to access supporting network services.

architecture. Of a computer, the physical configuration, logical structure, formats, protocols, and operational sequences for processing data, controlling the configuration, and controlling the operations. Computer architecture may also include word lengths, instruction codes, and the interrelationships among the main parts of a computer or group of computers.

Arithmetic Logical Unit (ALU). the element that can perform the basic data manipulations in the central processor. Usually, the ALU can add, subtract, complement, negate, rotate, AND, and OR.

array. 1. An arrangement of elements in one or more dimensions. 2. In a programming language, an aggregate that consists of data objects with identical attributes, each of which may be uniquely referenced by subscription.

ASCII. Acronym for American Standard Code for Information Interchange. The standard code used for information interchange among data processing systems, data communications systems, and associated equipment in the United States.

ASM. Assembler File.

assembly. 1. The manufacturing process that converts circuits in wafer form into finished packaged parts. 2. A short term for assembly language.

B

baud. A unit of measure of transmission capacity. The speed at which a modem can transmit data. The number of events or signal changes that occur in one second. Because one event can encode more than one bit in high-speed digital communications, baud rate and bits per second are not always synonymous, especially at speeds above 2400 bps.

baud rate. A unit of measure of the number of state changes (from 0 to 1 or 1 to 0) per second on an asynchronous communications channel.

binary (b). A number system based on 2. A binary digit is a bit.

bit. *binary digit*—a digit of a binary system. It contains only two possible values: 0 or 1.

block diagram. A diagram of a system, a computer, or a device in which the principal parts are represented by suitably annotated geometrical figures to show both the basic functions of the parts and their functional relationships.

buffer. 1. In hardware, a device that restores logic drive signal levels to drive a bus or a large number of inputs. In software, any memory structure allocated to the temporary storage of data. 2. A routine or storage medium used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transferring data from one device to another.

bus. In electronics, a parallel interconnection of the internal units of a system that enables data transfer and control information. One or more conductors or optical fibers that serve as a common connection for a group of related devices.

byte (B). A sequence of adjacent bits (usually 8) considered as a unit. A collection of four sequential bits of memory. Two sequential bytes (8 bits) comprise one word.

C

CALL. This command invokes a subroutine.

CCF. Clear Carry Flag.

character set. A finite set of different characters that is complete for a given purpose. A character set might include punctuation marks or other symbols.

CIEF. Clear IE Flag.

clock. A specific cycle designed to time events, used to synchronize events in a system.

CLR. Clear.

CMOS. Complementary Metal Oxide Semiconductor. A type of integrated circuit used in processors and for memory.

compile. 1. To translate a computer program expressed in a high-level language into a program expressed in a lower level language, such as an intermediate language, assembly language, or a machine language. 2. To prepare a machine language program from a computer program written in another programming language by making use of the overall logic structure of the program or by generating more than one computer instruction for each symbolic statement as well as performing the function of an assembler.

compiler. A computer program for compiling.

COPF. Clear Overflow Protection Flag.

CPU. Abbreviation for Central Processing Unit. 1. The portion of a computer that includes circuits controlling the interpretation and execution of instructions. 2. The portion of a computer that executes programmed instructions, performs arithmetic and logical operations on data, and controls input/output functions.

D

debug. To detect, trace, and eliminate mistakes.

DI. Disable interrupt.

E

EI. Enable interrupt.

emulation. The process of duplicating the characteristics of one product or part using another medium. For example, an In-Circuit Emulator (ICE) module duplicates the behavior of the chip it emulates, in the circuit being tested.

emulator. An emulation device.

EOF. End of file—when all records in a file are processed, the computer encounters an end-of-file condition.

EPROM. Erasable Programmable Read-Only Memory. An EPROM can be erased by exposure to ultraviolet light.

EQ. A Boolean operator meaning Equal to.

escape sequence. A special escape command is entered as three *plus* symbols (+++). placing the modem in command mode, and interrupting user data transmission. However, the escape sequence does not terminate the data connection. Command mode allows the entering of commands while the connection is maintained.

F

F. Falling Edge.

Fast Fourier Transform. An algorithm for computing the Fourier transform of a set of discrete data values. Given a finite set of data points—for example, a periodic sampling taken from a real-world signal—the FFT expresses the data in terms of its component frequencies. It also solves the essentially identical inverse problem of reconstructing a signal from the frequency data.

FFT. See Fast Fourier Transform.

filter. A process for removing information content, such as high or low frequencies.

flag. In data transmission or processing, an indicator, such as a signal, symbol, character, or digit, used for identification. A flag may be a byte, word, mark, group mark, or letter that signals the occurrence of some condition or event, such as the end of a word, block, or message.

frequency. For a periodic function, the number of cycles or events per unit time.

G

graphical user interface (GUI). 1. A graphics-based user interface that enables users to select files, programs or commands by pointing to pictorial representations (icons) on the screen, rather than by typing long, complex commands from a command prompt. 2. The windows and incorporated text displayed on a computer screen.

groups. Collections of logical address spaces typically used for convenience of locating a set of address spaces.

GUI. See graphical user interface.

H

h. See hexadecimal.

hardware. The boards, wires, and devices that comprise the physical components of a system.

Hertz. Abbreviated Hz. A measurement of frequency in cycles per second. A hertz is one cycle per second. A kilohertz (KHz) is one thousand cycles per second. A megahertz (MHz) is one million cycles per second. A gigahertz (GHz) is a billion cycles per second.

hexadecimal. A base-16 number system. Hex values are often substituted for harder-to-read binary numbers.

I

ICE. In-Circuit Emulator. A Zilog product that supports the application design process.

icon. A small screen image representing a specific element such as a document, embedded and linked objects, or a collection of programs gathered together in a group.

ID. Identifier.

IE. Interrupt Enable.

initialize. To establish start-up parameters, typically involving clearing all of some part of the device's memory space.

instruction. Command.

interface (I/F). 1. In a system, a shared boundary, i.e., the boundary between two subsystems or two devices. 2. A shared boundary between two functional units, defined by specific attributes, such as functional characteristics, common physical interconnection characteristics, and signal characteristics. 3. A point of communication between two or more processes, persons, or other physical entities.

interleaving. The transmission of pulses from two or more digital sources in time-division sequence over a single path.

interrupt. A suspension of a process, such as the execution of a computer program, caused by an event external to that process, and performed in such a way that the process can be resumed. The three types of interrupts include: internal hardware, external hardware, and software.

I/O. Input/Output. In computers, the part of the system that deals with interfacing to external devices for input or output, such as keyboards or printers.

IPR. Interrupt Priority Register.

J

JP. Jump.

K

K. Thousands. May indicate 1000 or 1024 to differentiate between decimal and binary values. Abbreviation for the Latin root *kilo*.

kHz. See kilohertz.

kilohertz (kHz). A unit of frequency denoting one thousand (103) Hz.

L

LD. Load.

library. A file that contains a collection of object modules that were created by an assembler or directly by a C compiler.

LSB. Least significant bit.

M

MAC. An acronym for Media Access Control, the method a computer uses to transmit or receive data across a LAN.

Megahertz (MHz). A unit of frequency denoting one million (10^6) Hz.

memory. 1. All of the addressable storage space in a processing unit and other internal memory that is used to execute instructions. 2. The volatile, main storage in computers.

MHz. See Megahertz; also Hertz.

MI. Minus.

MLD. Multiply and Load.

MPYA. Multiply and Add.

MPYS. Multiply and Subtract.

MSB. Most significant bit.

N

*n***.** Number. This letter is used as place holder notation.

NC. No Connection.

NE. Not Equal.

NEG. Negate.

NMI. Nonmaskable interrupt.

NOP. An acronym for No Operation, an instruction whose sole function is to increment the program counter, but that does not affect any changes to registers or memory.

O

Op Code. Operation Code, a quantity that is altered by a microprocessor’s instruction. Also abbreviated OPC.

OR. Bitwise OR.

OV. Overflow.

P

PC. Program counter.

pipeline. The act of initiating a bus cycle while another bus cycle is in progress. Thus, the bus can have multiple bus cycles pending at a time.

POP. Retrieve a value from the stack.

port. The point at which a communications circuit terminates at a network, serial, or parallel interface card.

power. The rate of transfer or absorption of energy per unit time in a system.

push. To store a value in the stack.

R

RAM. Random-access memory. A memory that can be written to or read at random. The device is usually volatile, which means the data is lost without power.

random-access memory (RAM). A read/write, nonsequential-access memory used for the storage of instructions and data.

read-only memory (ROM). A type of memory in which data, under normal conditions, can only be read. Nonvolatile computer memory that contains instructions that do not require change, such as permanent parts of an operating system. A computer can read instructions from ROM, but cannot store new data in ROM. Also called *nonerasable storage*.

register. A device, accessible to one or more input circuits, that accepts and stores data. A register is most often used only as a device for temporary storage of data.

ROM. See read-only memory.

RR. Rotate Right.

S

SCF. Set C Flag.

SL. Shift Left.

SLL. Shift Left Logical.

SP. Stack Pointer.

SR. Shift Right.

SRA. Shift Right Arithmetic.

Static. Characteristic of random-access memory that enables it to operate without clocking signals.

SUB. Subtract.

T

tristate. A form of transistor-to-transistor logic in which output stages, or input and output stages, can assume three states. Two are normal low-impedance 1 and 0 states; the third is a high-impedance state that allows many tristate devices to time-share bus lines. This industry term is not trademarked, and is available for Zilog use. Do not use *3-state* or *three-state*.

U

ULT. Unsigned Less Than.

W

wait state. A clock cycle during which no instructions are executed because the processor is waiting for data from memory.

word. Amount of data a processor can hold in its registers and process at one time.

write. To make a permanent or transient recording of data in a storage device or on a data medium.

X

X. 1. Indexed Address. 2. An undefined or indeterminate variable.

XOR. Bitwise exclusive OR.

Z

Z. 1. Zero. 2. Zero Flag.

ZDS. Zilog Developer Studio. Zilog’s program development environment for Microsoft Windows.

Index

Customer Support

To share comments, get your technical questions answered, or report issues you may be experiencing with our products, please visit Zilog's Technical Support page at http://support.zilog.com.

To learn more about this product, find additional documentation, or to discover other facets about Zilog product offerings, please visit the Zilog Knowledge Base at http://zilog.com/kb or consider participating in the Zilog Forum at http://zilog.com/forum.

This publication is subject to replacement by a later edition. To determine whether a later edition exists, please visit the Zilog website at http://www.zilog.com.